Communication-Efficient Parallel Dense LU Using a 3-Dimensional Approach*

Dror Irony † ***and Sivan Toledo*** ‡

1 Introduction

We present new communication-efficient parallel dense linear solvers: An *LU* factorization algorithm and a triangular linear solver. The new algorithms perform asymptotically a factor of $P^{1/6}$ less communication than existing algorithms, where P is the number of processors . The new algorithms employ a 3-dimensional (3D) approach, which has been previously applied only to matrix multiplication. We have implemented and tested the new algorithms. Our *LU* factorization algorithm is competitive with ScaLAPACK and scales better with the number of processors.

The new algorithms employ a 3D approach that reduces communication using replication. The algorithms perform less communication but use more temporary storage than existing algorithms, which all use a 2-dimensional (2D) approach. Until now, the 3D approach has only been used for parallel matrix multiplication in algorithms that were proposed by Berntsen [3], by Aggarwal, Chandra, and Snir [2], by Gupta and Kumar [5], by Johnsson [7], and by Agarwal, Balle, Gustavson, Joshi, and Palkar [1].

3D algorithms work by distributing the 3D iteration space of the computation among processors. Matrix-matrix computations that can be implemented using three nested loops have a natural representation on a 3D grid in which every grid

*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. Access to the SGI Origin 2000 was provided by Israel's High-Performance Computing Unit.

†School of Computer Science, Tel-Aviv University. Email: irony@tau.ac.il.

‡School of Computer Science, Tel-Aviv University. Email: stoledo@tau.ac.il

point represents one ijk loop-index triplet, which in tern represents one elementary computation (e.g., a scalar multiply-add). 3D algorithms distributes this space so that every processor is responsible for a 3D subcube. In contrast, conventional 2D algorithms distributed the matrices among processors and employ an "owner-computes" rule to assign work to processors (usually the owner of an output element performs all the numerical computations that contribute to that output).

Our new algorithms transfer a total of $\Theta(P^{1/3}n^2)$ words between processors, a factor of $P^{1/6}$ better than the $\Theta(P^{1/2}n^2)$ words that 2D solvers transfer (n is the order of the matrix). These amounts are asymptotically the same as the amounts that 3D and 2D matrix-multiplication algorithms transfer, respectively.

We also analyze the constants involved and show that they are small.

Our algorithms are formulated recursively and use a cyclic or block-cyclic data layout.

We have implemented our new algorithm in C using MPI. The implementation is general, in the sense that it allows any matrix order, any 3D processor grid, and any block size for the block-cyclic data layout.

We have so far tested the algorithm on a network of workstations and on an SGI Origin 2000 parallel computer. On 8 Linux workstations, the new LU factorization is only slightly slower than ScaLAPACK's 2D algorithm (around 36% slower on large matrices). On 64 processors of an Origin 2000, our algorithm is faster than ScaLAPACK on small matrices but slower by a factor of around 4 on large matrices. We have not yet determined whether our algorithm is slower because our algorithm inherently performs more communication when P is small, or because our code is not well optimized. In any case, the analysis of 3D and 2D algorithms shows that our algorithm scales better as the number of processors grows, so we expect our algorithm to outperform 2D algorithms when many processors are used.

2 Communication-Efficient Triangular Solvers

This section describes a new parallel algorithm for solving triangular systems with multiple right-hand sides and analyzes its efficiency. The main advantage of the new algorithm over existing algorithms is that the new algorithm performs less communication, at the expense of using more temporary storage. The algorithm is essentially a schedule of the substitution algorithm, and is therefore numerically stable. The algorithm is based on an 3D assignment of processors to the 3D computation structure of the substitution algorithm. In this abstract we limit the discussion to a simple case in which the number of processors is $P = p^3$ for an integer p, and the matrices are order $n = p \times 2^l$ for some integer l. The general case, in which $P = p_1 \times p_2 \times p_3$ for any integers p_1, p_2, and p_3, and in which the matrices can have any order, is not much harder and is described in the full version of the paper and in [6].

The simple case of the algorithm solves $AX = B$ where all three matrices are n-by-n and A is triangular.

We refer to processors using their position in the 3D processor grid. For example, processor ijk is the processor in position (i, j, k) in the grid. We also

need to refer to entire 2D and 1-dimensional subsets of the grid. We define the rth *ij layer* of the grid as all processors with indices (i, j, r) for some constant r. We similarly define ij and jk layers. We define the sr'th i *line* as all processors with indices (i, s, r) for some constant s and r.

Processor i, j, k in a 3D grid of processors G, of l-by-m-by-n processors, is written as G_{ijk}, where $1 \leq i \leq l$, $1 \leq j \leq m$, $1 \leq k \leq n$.

We now describe our algorithm for solving $AX = B$ using p^3 processors, where all matrices are n-by-n.

We arrange the processors in a p-by-p-by-p 3D grid, and distribute the matrices A and B in a block-cyclic distribution, as described in the Introduction, each on a different face of the grid. More specifically, the matrix A is block-cyclically distributed on the first jk layer and B is block-cyclically distributed on the first ij layer. At the end of the execution, X will be block-cyclically distributed on the first ik layer.

In the first phase of the algorithm, each processor $1jk$, which stores a (non-contiguous) block of A, broadcasts its block of A to the jk'th i line. The effect of this step, viewed over the entire grid, is to broadcast all of A from the first jk layer to all the jk layers. The algorithm broadcasts B similarly.

The second phase of the algorithm solves the linear system recursively. The dimensions of all the matrices remain equal throughout the recursion. The algorithm works as follows:

1. If A is p-by-p, then each layer of the processor cube solves one triangular linear system with a single right-hand side. This is the base case of the recursion. This step inefficient, because the p^2 processors in each layer work for p steps to solve a linear system that one processor can solve in p^2 steps. According to our assumptions, B and X have p columns, so there is exactly one layer of processors per column. Each ik layer of the grid contains a copy of X. The algorithm returns.

2. If the algorithm did not yet return, then A is at least $2p$-by-$2p$ and we split A, B, and X into 2-by-2 block matrices

$$\begin{bmatrix} A_{11} & \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} .$$

   The splitting is purely conceptual—no data movement occurs.

3. Solve $A_{11}X_{11} = B_{11}$ recursively using the entire processor grid. Since the matrices are laid out on the processor grid in a cyclic distribution, these blocks are distributed on the entire processor grid.

4. Solve $A_{11}X_{12} = B_{12}$ recursively.

5. Multiply and subtract $B_{21} = B_{21} - A_{21}X_{11}$. The entire processor grid performs this matrix multiply-add using the 3D algorithm. Note however, that since every jk layer already contains a copy of A and every ik layer contains a copy of X, there is no need for broadcasting in the 3D matrix-multiplication algorithm.

6. Multiply and subtract $B_{22} = B_{22} - A_{21}X_{12}$.
7. Solve $A_{22}X_{21} = B_{21}$ recursively.
8. Solve $A_{22}X_{22} = B_{22}$ recursively.

We stress that steps 2–8 of the algorithm are performed sequentially, not in parallel. All the parallelism in the algorithm lies in the leaves of the recursion, both matrix multiplication and triangular solves.

Before we analyze the performance of the algorithm is it worth pointing out that the algorithm can also be understood as a collection of 2D algorithms, each performed by one ij layer in the processor grid. Each such layer solves a triangular linear system with multiple right-hand sides.

Theorem 1. *In a synchronous model of parallel computation, in which a processor can perform one operation per step and where communication is instantaneous, the number of computational steps that the algorithm requires is* $O(n^3/p^3 + n^2/p)$.

Proof. We denote the number of steps for matrices of order n by T_n and hold p fixed. The base case of the recursion is $T_p = p$. The number of steps to perform a matrix multiply and subtract on matrices of order n is at most $2n^3/p^3$. We therefore have the following recurrence for the number of steps,

$$T_n \leq \begin{cases} 4T_{n/2} + 2(2(n/2)^3/p^3) & \text{when } n > p \\ p & \text{when } n = p \end{cases}$$

It's not hard to verify by substitution that the solution of this recurrence is $T_n = O(n^3/p^3 + n^2/p)$. □

Corollary 2. *The algorithm is asymptotically work efficient when* $p \leq \sqrt{n}$. *That is, when* $p \leq \sqrt{n}$ *the number of steps during which a processor works is proportional to the total number of steps.*

Proof. When $p \leq \sqrt{n}$,

$$p^3 \times (n^3/p^3 + n^2/p) = n^3 + n^2p^2 \leq 2n^3$$

□

Theorem 3. *The amount of communication in the algorithm is* $O(n^2 p \lg(n/p))$.

Proof. We denote the amount of communication for matrices of order n by C_n and hold p fixed. The base case of the recursion is $C_p = p^3$. The amount of communication needed to perform a matrix multiply and subtract on matrices of order n is at most pn^2. We therefore have the following recurrence for the number of steps,

$$C_n \leq \begin{cases} 4C_{n/2} + 2(p(n/2)^2) & \text{when } n > p \\ p^3 & \text{when } n = p \end{cases}$$

It's not hard to verify by substitution that the solution of this recurrence is $C_n = O(n^2 p \lg(n/p))$. □

2.1 Eliminating Redundant Communication

We now present a more efficient variant of the algorithm from the previous section. The improved algorithm performs less communication. The amount of communication that it performs is $O(n^2 p)$. It is asymptotically the same as the amount that the parallel 3D matrix multiplication performs. The key to the improvement of the algorithm is the elimination of redundant communication.

The redundancy in the previous algorithm lies in the calls to the 3D matrix multiplication routine. This routine performs communication that, in the context of the triangular solver, is unnecessary. The 3D matrix multiplication routine broadcasts its inputs throughout the processor cube, and it performs a distributed summation to compute the product. It turns out that in the context of our triangular solver, the inputs to matrix multiplications are already duplicated throughout the cube, and the output can be left unsummed. The summation can be delayed until elements of the product are actually needed by the triangular solver.

The key idea is to generalize the algorithm so that it solves triangular linear systems of the form $AX = B - Y$, where Y is represented in a special way. Specifically, Y is represented by a sum of matrices, one on each ij layer. A, B, and X are fully summed are replicated. The role of Y is to collect partial updates to the right-hand sides. To solve a linear system $AX = B$, we simply start the algorithm with $Y = 0$. The role of Y in this case is to store partial updates to B when the algorithm descends into the recursion.

The improved algorithm maintains the following data-layout invariants:

- The coefficient matrix A is replicated in each jk layer. The matrix does not change during the algorithm, so we can perform the replication before the algorithm begins at a communication cost of pn^2.
- The original right-hand sides B are replicated in each ij layer.
- Y is kept unsummed. Each ij layer contains one term of the matrix sum. The data layout in all the layers is the same and conforms to the layout of B (that is, a processor that stores b_{ij} also stores a term of y_{ij}).
- The values of X that have already been computed are replicated in each ik layer.

The algorithm is again recursive, but before the recursion begins, we replicate A and B as required and initialize $Y = 0$. This transfers $2pn^2$ words between processors.

Since no values of X have been computed, the invariant concerning it holds. The algorithm uses the same framework as the previous algorithm:

1. If A is p-by-p, then each layer of the processor cube solves one triangular linear system with a single right-hand side $B - Y$. We first sum the right-hand sides: each k line sums a single element of a single right-hand-side. We now perform the solution of the p triangular linear systems. Finally, we replicate the elements of X that have been computed in each ik layer. The total amount of communication in this step is $\Theta(p^3)$. The algorithm returns.

2. If the algorithm did not yet return, split A, B, X and Y into 2-by-2 block matrices

$$\begin{bmatrix} A_{11} & \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} - \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix} .$$

3. Solve $A_{11}X_{11} = B_{11} - Y_{11}$ recursively using the entire processor grid.

4. Solve $A_{11}X_{12} = B_{12} - Y_{12}$ recursively.

5. Multiply $\widetilde{Y_{21}} = Y_{21} + A_{21}X_{11}$ but leave Y only partially summed—intraprocessor summations are performed but intraprocessor summations are not. Our invariants imply that A_{21} and X_{11} are already replicated, and we leave $\widetilde{Y_{21}}$ partially summed, so this step is equivalent to a 3D multiply-add, but with no communication at all. We can overwrite Y_{21} with $\widetilde{Y_{21}}$.

6. Multiply and subtract $\widetilde{Y_{22}} = Y_{22} + A_{21}X_{12}$ in the same way.

7. Solve $A_{22}X_{21} = B_{21} - \widetilde{Y_{21}}$ recursively.

8. Solve $A_{22}X_{22} = B_{22} - \widetilde{Y_{22}}$ recursively.

Theorem 4. *The triangular solver algorithm presented above is correct.*

Proof. We prove the theorem by induction. The base case is clearly correct—we sum the right-hand side and solve using a correct algorithm.

Assume that the algorithm is correct for inputs of order $n/2$. Given an input of size n, the X_{11} and X_{12} outputs are correct because they are computed recursively. The splitting into blocks gives us $A_{21}X_{11} + A_{22}X_{21} = B_{21} - Y_{21}$, or $A_{22}X_{21} = B_{21} - (Y_{21} + A_{21}X_{11}) = B_{21} - \widetilde{Y_{21}}$, so X_{21} is also correct . The same argument works for X_{22}. □

Theorem 5. *The number of computational steps that the algorithm requires is* $O(n^3/p^3 + n^2/p)$.

Proof. The proof of Theorem 1 applies here as well. □

Corollary 6. *The algorithm is asymptotically work efficient when* $p \leq \sqrt{n}$.

Theorem 7. *The amount of communication in the algorithm is* $O(pn^2)$.

Proof. The algorithm performs communication only before the recursion starts and in the leaves of the recursion. The amount of communication required before the recursion is $\Theta(pn^2)$, to replicate A and B, p times each. The amount of communication at the leaves of the recursion is also $\Theta(pn^2)$, since the recursion has $(n/p)^2$ leaves and each requires $O(p^3)$ communication. □

2.2 A More Detailed Analysis of Communication

The previous sections discussed the asymptotic behavior of the algorithm. Since the reduction in communication over conventional algorithms is only a factor of $\Theta(P^{1/6})$, it is important to perform a more detailed analysis of the constants involved.

The algorithm replicates two n-by-n distributed matrices, A and B, p times each, using broadcasts when the algorithm starts. The algorithm also replicates the output matrix X p times using broadcasts. These broadcasts occur at the bottom level of the recursion, whenever a block of X is computed. Finally, the algorithm sums a fourth matrix, Y. The summation has p terms, each distributed. The amount of communication is roughly the same in each of these 4 operations.

2.3 The General Case

The full version of the paper, as well as [6] describes extensions of the simple case to arbitrary 3D processor grids, to odd n, to n not divisible by p, and to block-cyclic data layouts (as opposed to pure cyclic layouts). These extensions are omitted from this abstract due to lack of space.

3 Communication-Efficient Triangular Factorization

The new 3D LU factorization algorithm is similar in many ways to the 3D triangular solver. Due to lack of space in this abstract and to the similarity to the triangular solver, we do not describe the details of the LU algorithm. Instead, we present the basic recursive algorithm and the analysis of its performance. It uses the same data layout on a 3D processor grid, and it also employs and inefficient base case.

Here is the recursive LU factorization algorithm for factoring an n-by-n matrix on a p-by-p-by-p grid of processors.

1. If A is p-by-p, factor it using a nonrecursive algorithm that is omitted from this abstract and return.

2. If the algorithm did not yet return, split A into a 2-by-2 block matrix

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

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 2.1.
5. Solve $L_{11}U_{12} = A_{12}$ for U_{12} using the 3D triangular solver.
6. Update $\widetilde{A_{22}} = A_{22} - L_{21}U_{12}$ using a 3D matrix multiply-add subroutine.
7. Factor $\widetilde{A_{22}} = L_{22}U_{22}$.

We present the results of our theoretical analysis of the algorithm without proofs. It is, however, worth noting that unlike the triangular solver, the proof of the communication bound in Theorem 9 is simple and does not require delayed summation. The reason is that the recursive LU algorithm makes only two calls to itself on matrices of half the size, whereas the triangular solver makes 4 such calls.

Theorem 8. *The number of computational steps that the algorithm requires is* $O(n^3/p^3 + n^2/p)$.

Theorem 9. *The amount of communication in the algorithm is* $O(n^2p)$.

4 Experimental Results

This section summarizes our experimental results. The experiments compare our C-and-MPI implementation of our 3D algorithms to ScaLAPACK's [4] 2D algorithms.

We conducted the experiments reported here on a 400MHz, 112-processor SGI Origin 2000 that was only lightly loaded at the time of the experiments. The machine has 14GB of main memory and no significant paging occurred during our experiments.

We report experiments on 64 processors. We used ScaLAPACK to compute the LL^T (Cholesky) factorization on 1 and on 64 processors, and we used our algorithm to compute the LU factorization using 64 processors. We compare ScaLAPACK's Cholesky to our LU since our LU does not perform partial pivoting, so from the performance characteristics it is more similar to a Cholesky factorization than to a pivoting LU factorization, except that it performs twice the amount of work and communication. Hence, in table 1 we multiply the Cholesky running times by 2 to simplify comparisons.

Our results, which are summarized in Table 1, validates our theoretical analyses but also indicate that our implementation is probably not as well optimized as ScaLAPACK's.

Specifically, on small matrices, where communication represents a huge overhead and both algorithms show little or no speedups, our algorithm outperforms ScaLAPACK. On matrices larger than about $n = 300$, our algorithm becomes slower than ScaLAPACK. We have not researched this extensively but several explanations are possible. The most likely one is that MPI's broadcast and reduction primitives, which our code uses extensively, are implemented inefficiently, and that the point-to-point primitives that ScaLAPACK uses perform better. Another reason might be that our algorithm spends more time in the inefficient bottom layer of the recursion.

Table 1. *The running times in seconds of ScaLAPACK's 2D Cholesky factorization algorithm and of our 3D LU factorization algorithm, all in double precision. Column 1 shows the size of the matrix. Columns 2–4 show the time it takes ScaLAPACK to factor the matrix on 1 processor (multiplied by 2 for easy comparisons), the time it take ScaLAPACK to factor the matrix on 64 processors (multiplied by 2), and the time it takes our algorithm to factor the matrix.*

n	$2 * T_1^{LL^T}$	$2 \times T_{64}^{LL^T,2D}$	$T_{64}^{LU,3D}$
200	0.02	0.02	0.02
250	0.02	0.04	0.03
300	0.04	0.04	0.03
400	0.10	0.06	0.08
500	0.18	0.06	0.14
800	0.68	0.14	0.44
1000	1.30	0.16	0.69
1500	4.66	0.42	1.62
2000	11.06	0.72	3.02
3000	38.86	1.64	7.22
4000	87.80	3.22	12.06

We also conducted experiments on a cluster of 8 Linux workstations with even better results. On small matrices our algorithm is slightly faster than ScaLAPACK and on large matrices it is only 36% slower. Due to lack of space, we omit further details from this abstract.

5 Conclusions And Discussion

We have presented new communication efficient algorithms for LU decomposition and solution of triangular linear systems. The algorithm perform asymptotically a factor of $P^{1/6}$ less communication than all existing algorithms.

While generally slower than ScaLAPACK's [4] 2D algorithm, our implementation of these algorithms is faster than ScaLAPACK's 2D algorithms on small matrices and large numbers of processors, where communication matters most. These findings are consistent with our theoretical analyses of the algorithms.

In another experiment not reported here, which was conducted on a cluster of Linux workstations, the gap between the performance of our algorithm and ScaLAPACK's was smaller.

Clearly, much more work is required to bring our implementation to the performance level of a mature code like ScaLAPACK's. In particular, we conjecture that the main reason that our algorithm is sometimes slower than ScaLAPACK is our extensive use of MPI's broadcast and reduction primitives.

Obvious extensions to these algorithms would be a 3D Cholesky algorithm and 3D algorithms for LU factorization with partial pivoting and for the QR decomposition. The extension to Cholesky is easy but the other two may prove challenging.

Bibliography

[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] 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.

[5] 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.

[6] Dror Irony. A 3D parallel communication-efficient dense linear solver. Master's thesis, School of Computer Science, Tel-Aviv University, 2000.

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