TEL-AVIV UNIVERSITY
RAYMOND AND BEVERLY SACKLER FACULTY OF EXACT SCIENCES
SCHOOL OF MATHEMATICAL SCIENCES

A 3D Parallel Communication-Efficient Dense Linear Solver

Thesis submitted in partial fulfillment of the requirements for the M.Sc. degree of Tel-Aviv University by

Dror Irony

The research work for this thesis has been carried out at Tel-Aviv University under the direction of Dr. Sivan Toledo

August 2000

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 asymtotically work efficient and they perform a factor of $P^{1/6}$ less communication than any existing algorithm, where P is number of processors. In other words, these solvers are likely to run faster than any other solvers on parallel computers with a large number of processors. These provably-efficient algorithms are the main contribution of this thesis.

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. These so-called three-dimensional matrix-multiplication algorithms use a three-dimensional grid of processors and they replicate matrices on each two-dimensional "layer" of the 3D processor grid. (the processor grid can be a virtual grid embeded in any network topology, however.)

Our new algorithm are also three-dimensional, in the sense that they arrange processors in a 3D grid and replicate matrices on each layer.

Our algorithm are recursive, elegant, and relatively simple to program. We have implemnted them using MPI, a message-passing libray, and tested them on both a cluster of workstations and on a large parallel computer.

The algorithms perform well even on a small number of processors. On 8 processors, they are slower than the state-of-the-art implementation of the conventional algorithm, but not by much. Since theory predicts that they scale better as the number of processors grows, we expect them to be faster for large numbers of processors.

We also present new lower bounds on the amount of communication in matrix-multiplication algorithms. These lower bounds show that so-called three-dimentional algorithms are asymptotically optimal in terms of the amount of communication that they consume, and that conventional two-dimensional algorithms are optimal when little or no replication is allowed.

Contents

CHAPTER 1

Introduction

In a distributed-memory parallel computer each processor has its own memory, which it can address directly and quickly. A processor may or may not be able to address the memory of other processors directly, and in many cases, accesing remote memories is slower than accessing its own local memory. Designing distributed-memory parallel algorithms is, therefore, challenging: we have to reduce communication between processors while maintaining a good balance of the computational load among processors.

In this thesis we introduce **two new distributed-memory parallel algorithms** in the field of linear algebra computations, and also give **two new lower bounds** on the amount of communication in matrix-multiplication algorithms.

This chapter explains the significance of this thesis, puts it into context by providing background information, and introduces our new algorithms and lower bounds. Some of the background material in this chapter is taken, with permission, from lecture notes written by Sivan Toledo.

1.1. Significance

The significance of this work stems from two factors: (1) the importance of parallel, distributed-memory dense linear solvers, and (2) the need for reducing communication in such solvers.

Dense linear systems arise in many applications. In some applications, a discretization of a continuous system yields a dense linear system of equations. For example, boundary-element discretizations, which discretize an integral equation, result in large dense linear systems. In radar cross-section simulations, one application area in which boundary-element discretizations are widely used, factoring complex dense matrices of order $100,000$ or more is not uncommon (to appreciate the size, note that a matrix of this size requires 160Gbytes of storage in a 64-bit floating-point representation).

In other applications, dense linear systems arise inside a sparse linear solver. That is, the mathematical model yields a sparse linear system that must be solved, but the sparse solver itself uses a dense solver as a subroutine. Multifrontal supernodal sparse direct solvers are the best example of such applications. These solvers, usually for sparse symmetric positive-definite linear systems, solve the system by factoring a sparse matrix A into a product of a sparse triangular linear system and its transpose $A = LL^T$. As the factorization proceeds, the trailing submatrix (the yet-unfactored equations) fills, eventually completely. Consider, for example, a linear system defined on a 100-by-100-by-100 three-dimensional grid, and assume that matrix is ordered with a nested-dissection ordering. At the end of the factorization of the matrix, the factorization algorithm must factor a *dense* matrix of order $10,000$, which represents the separator once all the other grid points have been eliminated. Clearly, an efficient dense linear solver is required for a sparse direct solver.

Communication between processors is the bottleneck in many parallel algorithms. Communication between processors may significantly slow down parallel computations, even if the work and communication is equally distributed between processors.

Well load-balanced parallel dense linear solvers require large amount of data to be transferred among processors when solving systems of high order. Therefore, reducing the amount of communication is challenging and may significantly enhance performance.

1.2. New Three-Dimensional Algorithms

The main contribution of this work consists of two new three-dimensional parallel algorithms, one for solving triangular linear systems with multiple right-hand sides, and the other for LU factorization without pivoting.

The main advantage of both algorithms is that they perform less communication than any other known algorithms by a factor of $P^{1/6}$ where P is the number of processors. This improvement comes at the expense of using more temporary storage. In other words, we trade communication for replication.

The algorithms are schedules of existing numerically-stable algorithms, and therefore stability is preserved. They both arrange the processors in a p-by-p-by-p grid. Matrices are laid out using a block-cyclic distribution on 2-dimensional faces of the processor grid. This layout ensures good load balance between processors. Both of the our algorithms are also recursive.

The recursive triangular solver solves the system $AX = B$ where both A, B, and X are n-by-n. The base case of the recursion is $n = p$. If $n > p$, it splits all three matrices into 2-by-2 block matrices and performs block substitution. Since the matrices are laid out cyclically (or block cyclically), each block is still laid out on the entire 3D processor cube, so each recursive call is performed by all the processors in parallel. The algorithm can also be viewed as p separate recursive 2D triangular solvers, each solving for n/p right-hand sides. Our algorithm is asymptotically work efficient as long as $p \leq \sqrt{n}$ or $P \leq n^{3/2}$. Thus, it cannot use efficiently as many processors as the 2D algorithm, which works efficiently for $P \leq n$. This again can be easily explained and justified by considering the solver to be as p separate triangular solvers.

The recursive 3D LU factorization algorithm also works efficiently when $P \leq n^{3/2}$. Like the triangular solver, the base case of the recursion is used when $n = p$, otherwise the input matrix is split into a 2-by-2 block matrix.

When will the new algorithms outperform existing 2D algorithms in practice? $P^{1/6}$ is not a large factor even for P in the 100's ($1000^{1/6} \approx 3.2$). Therefore, it is fair to say that the new algorithm is likely to outperform a 2D algorithm only if the constants associated with the amount of communication are small, and when we have a very large number of processors. We show below that the constants are indeed small. Also, the number of processors that our algorithm can use efficiently for a matrix of a given size is smaller than the number that the 2D algorithm can. Therefore, our algorithm should outperform 2D algorithms in a particular regime of matrix sizes: when the matrices are large enough relative to the number of processors so that our algorithm runs efficiently, but not so large that computation becomes dominant even in the 2D algorithm. For larger matrices, both algorithms should deliver similar performance.

The algorithms were implemented using MPI. The implementation was tested and evaluated on 2 different distributed-memory parallel computers.

1.3. New Lower Bounds on Communication in Parallel Matrix Algorithms

Another contribution of this thesis consists of two new lower bounds on the amount of communication in distributed-memory parallel matrix multiplication algorithm. The lower bounds show, essentially, that the 2D algorithm is optimal in terms of communication when no replication is allowed, and that the 3D algorithm is optimal in terms of communication even when the amount of memory per processor is unlimited.

The significance of these bounds is that they show that the matrix-multiplication algorithms that we use are both optimal (under different assumptions on the amount of memory). A search for more efficient algorithm would be futile.

The proof of these lower bounds are based on the proof technique of Toledo [**21**].

1.4. Outline of the Thesis

The rest of the thesis is organized as follows. The next chapter surveys the literature and presents important existing algorithms. Chapter 3 presents our new 3D triangular solver. Chapter 4 presents our new 3D *LU* factorization algorithm.Chapter 5 describes our implementation and experimental results.Chapter 6 presents our lower bounds on communication in parallel matrix-multiplication algorithms and shows that both the 2D and 3D algorithms are asymptotically optimal under appropriate assumptions on the amount of memory. Chapter 7 presents our conclusions, plans for future work, and an open problem.

CHAPTER 2

Background

This section survey the literature and provides background on existing algorithms. In particular, we present 2D and 3D matrix-multiplication algorithms, 2D triangular solvers, and 2D *LU* factorization algorithms.

2.1. 2D and 3D Matrix-Multiplication Algorithm

Most of the distributed-memory parallel dense-matrix algorithms in use today use a so-called *two dimensional block distribution* of the matrices. The processors are arranged conceptually in a p-by-p two-dimensional grid, the matrices are viewed as p-by-p block matrices, and the (i, j) block of the matrix is stored on processor with index (i, j) in the grid. There are several widely-used generalizations to this layout, including rectangular p-by-q processor grids and noncontiguous blocks in the matrices, as in cyclic and block-cyclic distribution. (In a cyclic distribution, the first row of the matrix is distributed on the first row of processors, the second on the second row of processors, and so on; row $p + 1$ is distributed again on the first row of processors, etc.; in a block-cyclic distribution, blocks of rows are laid out cyclically.)

Cannon [**5**] was the first to introduce an efficient distributed-memory parallel matrix multiplication algorithm using a two-dimensional blocks layout of matrices. Cannon originally proposed an algorithm for the case $P = n^2$ (where P is the number of processors and n is the order of the matrix) and for a parallel computer whose processors are physically connected as a two-dimensional mesh. But the algorithm is easily generalized to handle larger matrices and to any network topology (using an embedding of the 2D processor grid onto the physical network).

The basic idea of Cannon's algorithm is to assign the processor (i, j) the computation of $C_{ij} = \sum_{k=0}^{\sqrt{P}-1} A_{ik}B_{kj}$. The algorithm consists of $\sqrt(P)$ main phases. In every phase the processor receives a message containing a submatrix of A and a message containing a submatrix of B, multiplies them and adds the product to the submatrix C that it computes, and then sends the submatrices of A and B to other processors. The trick is to ensure that each processor gets the right submatrices, and that corresponding submatrices of A and B arrive together in the same phase.

The generalization of Cannon's algorithm to larger block-distributed matrices is due to Dekel, Nassimi and Sahni [**9**]. Gentleman [**12**] showed

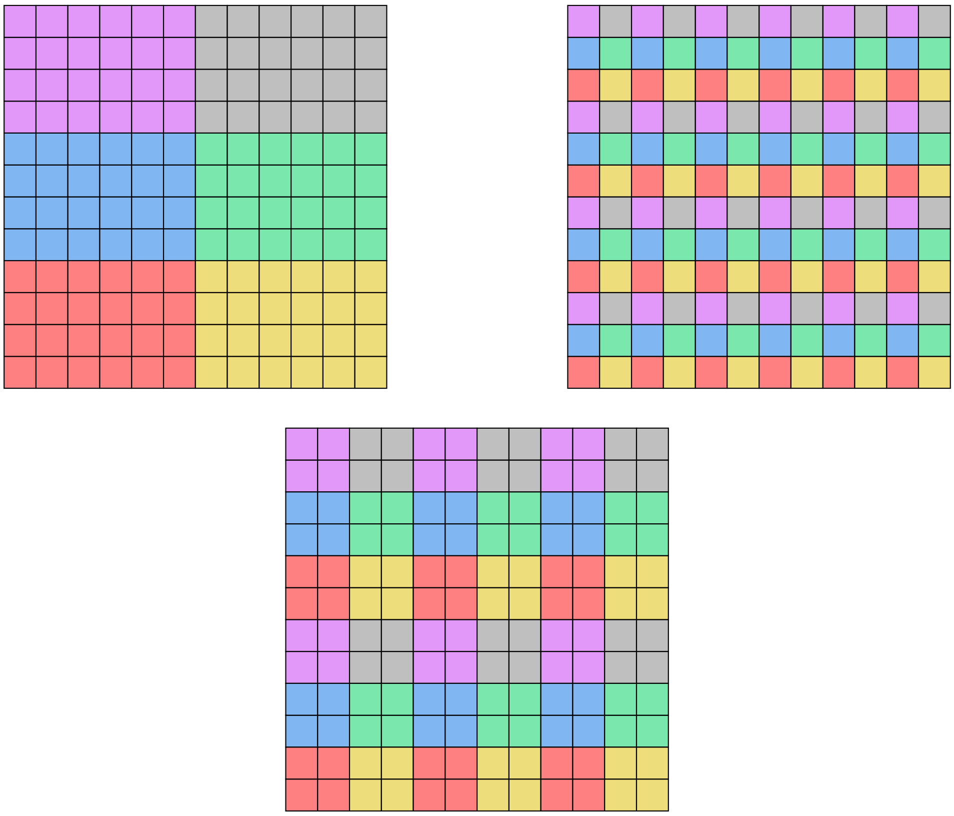

FIGURE 2.1.1. Two-dimensional data distributions of a 12-by-12 matrix on a 2-by-3 processor grid. Each square represents one matrix element and each color represents one processor. That is, all the red elements are stored on one processor, the one with index $(0, 0)$ in the processor grid, the gray elements are stored on processor $(1, 2)$, and so on. The top-left figure shows a block distribution, the top-right figure shows a cyclic distribution, and the bottom figure shows a block-cyclic distribution with 2-by-2 blocks.

that Cannon's algorithms is asymptotically optimal for mesh-connected parallel computers, in the sense that if each processor in such a computer is allowed to store initially only one element of A and one element of B, then the multiplication must take $\Omega(n)$ steps. Several other 2D algorithms have also been proposed. These algorithms, like Cannon's algorithm, arrange the processors on a two dimensional mesh. Fox, Otto and Hey [**11**] describe an algorithm which unlike Cannon's algorithm, use broadcasts. In every step, their algorithm broadcasts a block of A along each process row and cyclically shifts all the blocks of B one row up. Because of the broadcasts that are required in every step, this algorithm is usually less efficient than Cannon's algorithm. Another two-dimensional algorithm was proposed by van de Geijn and

Watts [**22**]. Their algorithm broadcasts elements of both A and B. The algorithm uses many broadcasts of small pieces of a matrix (smaller than the block stored on a processor) which allows the broadcasts to be pipelined and to occur concurrently with computation.

Analysis of these 2D algorithms shows that they perform $\Theta(\sqrt{P} \times n^2)$ communication, and are perfectly load balanced (when all the processors fit into a square grid). They can efficiently exploit up to n^2 processors. While it is possible to use more processors in a work-efficient matrix multiplication algorithm, the n^2 processor limit is not a significant disadvantage, since it usually does not make sense to use large numbers of processors to multiply small matrices, because of the communication overhead.

Three-dimensional matrix-multiplication algorithms reduce communication over 2D algorithms using replication. Such algorithms were first proposed by Berntsen [**3**] and by Aggarwal, Chandra and Snir [**2**] at about the same time. Essentially the same algorithm that Aggarwal et al. proposed was later proposed, independently, by Gupta and Kumar [**13**] and by Johnsson [**17**]. Berntsen described a somewhat more complex algorithm, while Aggarwal, Chandra and Snir also prove that the 3D algorithm is optimal in terms of the amount of communication that it performs. In this work, we give a different proof for this statement, based on a proof technique of Toledo[**21**].

Dekel, Nassimi and Sahni [**9**] were perhaps the first to propose 3D algorithms, but their algorithms are not communication efficient. They proposed algorithms for hypercube connected computers with $P = n^3$ processors, as well as for hypercubes with $n^2 < P = mn^2 < n^3$ processors. Both algorithms arrange the processors or groups of processors in a 3D mesh, both use replication, but in both the total number of words that are communicated is $\Theta(n^3)$. For $P <= n^2$ processors, Dekel, Nissimi and Sahni propose to block distribute the matrices and to use a variant of Cannon's algorithm as mentioned above.

The basic idea of the 3D algorithm is to arrange the processors in a p-by-p-by- p 3D grid (the number of processor must fit into this 3D grid), where $p = P^{1/3}$ and to split the matrices into p-by-p block matrices. The first phase distributes the matrix so that processor (i, j, k) stores A_{ik} and B_{kj}. This essentially replicates each matrix p times, since the ik block of A, say, is stored on the p processors with index (i, j, k) for all j. In the next phase each processor computes the product $A_{ik}B_{kj}$. In the third and last phase of the algorithm the processors sum up the products $A_{ik}B_{kj}$ to produce the blocks C_{ij} of the output.

The 3D algorithm performs $\Theta(P^{1/3} \times n^2)$ communication, better by a factor of $P^{1/6}$ than the 2D algorithms. The computational load is almost perfectly balanced: Each processor multiplies two blocks and

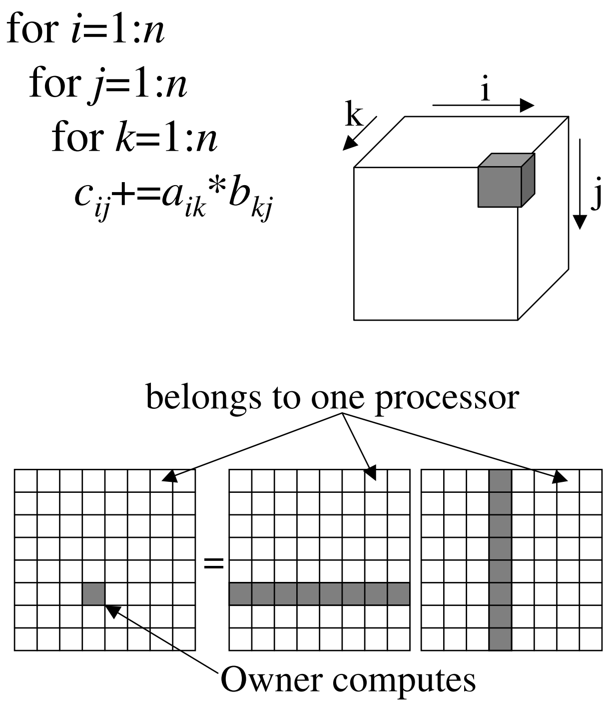

FIGURE 2.1.2. An illustration of 2D versus 3D algorithms. The top figure shows the 3D iteration space created by the triply nested loops. Each processor is responsible for performing computations in a subcube of the iteration space. In a 2D algorithm, shown below, we allocate allocate a block of each matrix to each processor. The processor that stores a block of the product is responsible for computing it, which requires a block of rows from one factor and a block of columns from the other.

adds at most two, but some processors add none (multiplying blocks requires $\Theta((n/p)^3)$ work and adding blocks only $\Theta((n/p)^2)$, so balancing multiplications balances the load almost perfectly when n/p is large).

Both 2D and 3D algorithms can be adapted to many interconnection networks. Much of the research on distributed-memory parallel matrix multiplication was aimed at parallel computers whose processors are connected as a hypercube. In such a computer, each processor is directly connected to $\lg P$ other processors. Most of the algorithms that have been adapted to hypercubes simply embed a 2D or 3D mesh in the hypercube. In many cases a collection of binary trees is also embedded to perform fast broadcasts or summations. Algorithms that use such embedding, however, use only a fixed number of communication channels per processor in every step. Ho, Johnsson and Edelman [**14**] developed a 2D algorithm that can use all the $\lg P$ communication channels of each processor simultaneously, thereby reducing the running time. No current parallel computer use a hypercube interconnection network with multiple communication channels per processor, although in the late 1980's and early 1990's several such computers were built (for example, the experimental Caltech Hypercube and the commercial Thinking Machines CM-2 and MasPar MP-1).

Much of the research on parallel matrix multiplication since the late 1980's was focused on robust and complete implementations. Such implementations must support not only products of the form $C = AB$, but also matrix multiply-add, multiplications with either A^T or B^T or both (almost always without actually transposing them), adding AA^T to a symmetric C, and so on. Complete implementations must also be able to handle rectangular matrices and block-distributed matrices where the blocks are discontiguous. One of the first complete implementations was produced by Johnsson and his colleagues for the Thinking Machines CM-2 and CM-5 computers. A series of papers describe the algorithmic issues involved [**14, 15, 16, 18, 19**]. Implementing the matrix-multiplication routine for the CM-2 alone took about 10 person-years, according to Johnsson. Another complete implementation, for the Intel Touchstone Delta computer, is described by Choi, Dongarra, and Walker [**8**]. van de Geijn and Watts [**22**] describe a portable complete implementation that uses MPI, along with a new 2D algorithm. Agarwal, Balle, Gustavson, Joshi, and Palkar [**1**] describe a complete 3-dimensional algorithm with reduced communication.

The *Parallel Basic Linear Algebra Subprograms (PBLAS)* [**6**] is a standard for distributed-memory parallel library that implements matrix multiplication, triangular solvers, and more. The PBLAS were designed as the distributed-memory analog of the BLAS. Although it was originally envisioned that computer vendors would produce optimized implementations of the PBLAS, there are virtually no such implementation. The portable reference implementation of the PBLAS (available from www.netlib.org/scalapack), on the other hand, has been carefully optimized and it is widely used. Most of the parallel matrix libraries that are sold include the PBLAS. Another portable library that includes matrix multiplication algorithms is *PLAPACK* [**23**] (available from www.cs.utexas.edu/users/plapack). The routines in PLAPACK are equivalent in functionality to the PBLAS, but the interface is different.

2.2. Two-Dimensional Triangular Linear Solvers

Perhaps the easiest linear systems to solve are triangular linear systems, in which the coefficient matrix is zero either above or below the main diagonal. Because triangular systems are easy to solve, many solvers for general linear systems work by factoring the coefficient matrix into a product of triangular linear systems.

Let's introduce a bit of notation. We write a system of n linear equations with unknowns x_1, x_2, …, x_n

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

as a matrix equation $Ax = b$, where A is an n-by-n matrix and x and b are n-vectors. When we have multiple systems all with the same coefficient matrix A but with different right hand sides b_1, b_2, …, b_m, we write all the systems as a matrix equation $AX = B$ where X and B are n-by-m matrices.

When A is triangular, we can solve the system by substitution. Suppose that A is lower triangular. Then the first equation involves only x_1, so we can solve for $x_1 = b_1/a_{11}$. We can now eliminate x_1 from the rest of the equations. Now the second equation involves only one unknown, so we can solve it directly, and so on.

Let's begin our discussion of distributed-memory parallel triangular solvers by analyzing a flawed but simple algorithm that uses the same data distribution as the 2D matrix-multiplication algorithm. We split the matrix A into square blocks and distribute one block to each processor in a p-by-p processor grid. Zero blocks are assigned to $p(p-1)/2$ of the processors, so we don't have to assign processors to them. The right-hand-side b is split into blocks as well and distributed to processors on the diagonal of the grid.

In the first phase of the algorithm, the processor that stores A_{11} solves the triangular system $A_{11}x_1 = b_1$ for the first block x_1 of the unknown vector x. The other processors wait. In the second phase, x_1 is broadcast down the column of processors. (That is, x_1 is broadcast to all the processors in that column in the processor grid; not to all the processors in the computation.) Each processor eliminate these variables from a block of equations and send the results to the next column of processors to the right. The processor that stores A_{22} can now solve for x_2 and the process continues. All the phases take about the same amount of time (actually, odd phases perform half as many operations as even phases, but we ignore the difference). Here is the pseudocode of the algorithm:

```
Every processor (i, 1) sets a length-n/p vector y_{i,1} to 0
for j = 1 : p
    processor (j, j) solves A_{jj} x_j = b_j − y_{jj}
    processor (j, j) broadcasts x_j to processors (:,j)
    for all k > j in parallel (but do nothing if j = n/p)
        processor (k, j) computes y_{kj} = y_{kj} − A_{kj} x_j
        processor (k, j) sends y_{kj} to (k, j + 1)
        processor (k, j + 1) sets y_{k,j+1} = y_{k,j}
    end
end
```

The flaw in the algorithm is that each processor only computes during one phase! We use $p(p+1)/2$ processors, but each processor is active only during about $1/p$ of the algorithm's execution time. This is not a very good parallel algorithm: the more processors we use, the less efficient the algorithm becomes.

One way to fix the problem is to split only the rows of the matrix among processors: the first processor stores the first n/p rows of A, the second processor the next n/p rows, and so on. We can design an algorithm similar to the first for this data distribution, called a *1-dimensional block* distribution (since only one dimension of the matrix is distributed among processors). In the first phase, the first processor solves the first block of equations. This block itself is triangular. In the second phase, it broadcasts x_1 to all the other processors, which eliminate the variables in x_1 from the equations that they store. Now the second processor solves for x_2 and so on. All the phases take about the same amount of time. The computational load is imbalanced, as opposed to the balanced load in the first algorithm, since the ith processor works only during the first $2i$ phases. But this algorithm is better than the first since many processors work for many phases. Half the processors work for at least half the phases, so the algorithm does not become more wasteful with more processors.

The 1-dimensional algorithm uses at the limit n processors, one storing each row, and it runs in $2n$ steps, so it is asymptotically work efficient and it cannot use asymptotically more processors, according to Brent's theorem.

Another way to fix the problem in the 2-dimensional block distribution is to use a 2D *cyclic* or a 2D *block-cyclic* distribution. In a 2D cyclic distribution, we use a p-by-p processor grid and split both the rows and columns of the matrix among processors. The first row of processors in the grid stores the first row of A, the second row in the grid stores the second row of A, and so on. The $p+1$ row of A is stored again on the first row of the grid, row $p+2$ is stored on the second row in the grid, and so on cyclically. The way a row of A is stored on a row of processors is similar. The first element in the row is stored on the first processor, the next element on the second processor, and so on. The $p+1$ element is stored again on processor 1 and so on cyclically. A 2D block-cyclic distribution with block size $\mathcal{B}$ is similar, except that we store a block of $\mathcal{B}$ consecutive rows of A on a row of processors before we move to the next row of processors, and within a block of rows we store a block of $\mathcal{B}$ columns on a processor before moving to the next processor in the grid.

The 2D block, cyclic, and block-cyclic distributions are similar in that all store on a processor the elements of A that lie in the intersection of a set of rows and a set of columns. In a block distribution, the set of rows and the set of columns are contiguous. That is, the first processor stores elements that lie in rows $\{1, 2, \ldots, \mathcal{B}\}$ and columns $\{1, 2, \ldots, \mathcal{B}\}$. In a cyclic or block cyclic distribution, the sets are not contiguous. In a cyclic distribution, the first processor stores elements that lie in rows and columns $\{1, p+1, 2p+2, \ldots\}$. In a block-cyclic distribution, the sets are $\{1, 2, \ldots, \mathcal{B}, p\mathcal{B}+1, p\mathcal{B}+2, \ldots\}$.

We now describe an algorithm for the more general block-cyclic 2D distribution. The matrix is split into $\mathcal{B}$-by-$\mathcal{B}$ blocks. We also view the matrix as being split into $p\mathcal{B}$-by-$p\mathcal{B}$ *superblocks.* We describe the algorithm in terms of superphases, in which entire superblocks are operated upon, and which consist of phases. Each superblock is distributed among the processors in a 2D block distribution.

In the first superphase the processors use the 2D-block algorithm described above to solve for the first superblock of x. This takes p phases, each of which involves $O((n/p\mathcal{B})^2)$ work per processor and the transfer of $O(n/p\mathcal{B})$ words per processor. Once this superphase is over, every processor `(k,j)` computes $y_{k'j} = y_{k'j} - A_{k'j}x_j$ for $k' = p+k, 2p+k, \ldots$. That is, every processor eliminates the elements of x in the block of columns that it stores from all the equations that it stores. The processors do not, however, send all the y vectors to the right to sum the updates, as the do in the 2D-block algorithm. The other equations will be updated again anyway in the next superphase, so there is no point in summing updates now.

In the second superphase the processors first solve for another superblock of x and then eliminate this block from the remaining equations, and so on.

Each superphase consist of an inefficient application of the 2D-block algorithm to the diagonal block, followed by the elimination of a superblock of x from the remaining equations. The elimination from the remaining equations is perfectly load balanced and processors do not wait for each other. Since by Brent's theorem it does not make sense to use more than n processors, we have $p^2 = O(n)$, which means that we have $\Omega(p)$ superblocks of equations. Hence, at least a constant fraction of the superphases are asymptotically work efficient: processing the diagonal superblock takes p phases, where each processor is busy only during one, but the elimination of the superblock of x from the remaining equations takes $\Omega(p)$ phases, and each processor is busy during all of them. Thus, the entire algorithm is work-efficient.

The algorithm is also communication-efficient. A processor transfers $O(\mathcal{B})$ words per superphase, but performs, on average, $\Theta(\mathcal{B} \times (n/p)) = \Theta(n\mathcal{B}/p)$ work. The asymptotic computation-to-communication ratio is the same as in the 2D-block matrix-multiplication algorithm.

The choice of $\mathcal{B}$ involves a tradeoff. A small $\mathcal{B}$ yields high efficiency, because less work is done during the inefficient processing of diagonal superblocks and more work is done during the elimination of variables from equations. On the other hand, a small $\mathcal{B}$ causes the algorithm to transfer data in many small messages rather than few large messages. If the overhead per message is large, it does not make sense to use a small $\mathcal{B}$. (The total volume of communication is independent of $\mathcal{B}$.) In addition, a small $\mathcal{B}$ causes the algorithm to invoke sequential triangular-solve and matrix-multiplication routines on small matrices, which due to cache and other low-level issues are usually less efficient than routines operating on large matrices. In practice, people have found that the optimal value of $\mathcal{B}$ often lies in the range 30–100.

When there are multiple right-hand sides, we can improve the work efficiency of the algorithm and increase the size of messages. We can improve the work efficiency for a fixed $\mathcal{B}$ by *pipelining* the solution of p right-hand sides. During the processing of a diagonal superblock and a single right-hand side, each processor is busy for only one phase. So once a processor finishes processing one right-hand side, it can immediately start working on another. Processing p right-hand sides will take $p + (p - 1) = 2p - 1$ phases rather than p, but each processor is busy during p of them. Thus, even the processing of diagonal superblocks is asymptotically work-efficient in this case.

We can also *cluster* the processing of multiple right-hand sides. This technique involves processing q right-hand sides together (without pipelining), so that each phase that a processor performs is applied to a cluster of q right-hand sides before proceeding to the next step. This has two effects. First, each message is now q times larger but there is a factor of q fewer messages. This leads to improved performance. Second, matrix operations that are applied by a processor are now applied to q-column matrices, which yields fewer cache misses than applying the operation q times to single vectors.

The best performance, of course, is achieved when the algorithm pipelines the solution of whole clusters of right-hand sides.

2.3. Two-Dimensional LU Factorizations

Suppose that we factor A into $A = LU$ where L is lower triangular and U is upper triangular. We can then solve $Ax = LUx = b$ by first solving the triangular system $Ly = b$ and then another triangular system $Ux = y$. We discuss here two such decompositions, the LU decomposition and the LL^T decomposition, which decomposes A into a product of a lower triangular matrix and its transpose, and is called also Cholesky decomposition.

The basic algorithm to compute an LU decomposition is simple.

1. If A is 1-by-1, then $L = 1$ and $U = A$ is a triangular decomposition.
2. Split A into a 2-by-2 block matrix, where the $(1, 1)$ block is $\mathcal{B}$-by-$\mathcal{B}$, the $(1, 2)$ block is $\mathcal{B}$-by-$(n-\mathcal{B})$, the $(2, 1)$ block is $(n-\mathcal{B})$-by-$\mathcal{B}$, and the $(2, 2)$ block is $(n - \mathcal{B})$-by-$(n - \mathcal{B})$. The parameter $\mathcal{B}$ can take any integer value between 1 and $n - 1$.
3. Recursively factor the leading diagonal block $A_{11} = L_{11}U_{11}$.
4. Solve $L_{21}U_{11} = A_{21}$ for L_{21} (This is a triangular system, equivalent to $U_{11}^T L_{21}^T = A_{21}^T$).
5. Solve $L_{11}U_{12} = A_{12}$ for U_{12}.
6. Update $A'_{22} \leftarrow A_{22} - L_{21}U_{12}$.
7. Recursively factor the updated trailing submatrix $A'_{22} = L_{22}U_{22}$.

The algorithm may break in Steps 4 or 5 if the matrices L_{11} or U_{11} are singular. But it's not hard to see that if it does not break down, it produces correct results.

$$
\begin{aligned}
LU &= \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix} \\
&= \begin{pmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{pmatrix} \\
&= \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \\
&= A\,.
\end{aligned}
$$

We shall see later that there are classes of matrices for which the algorithm never breaks down, and that there are ways to prevent breakdown for other matrices. But for now, we ignore the issue of breakdown. The algorithm performs about $(2/3)n^3$ floating-point operations.

We distribute A using a 2D block-cyclic data layout with block size $\mathcal{B}$ on a p-by-p processor grid. We view A as a block matrix with $\mathcal{B}$-by-$\mathcal{B}$ blocks; the (i, j) block is denoted A_{ij}. The algorithm uses the recursive framework that was presented above in the following way. In each phase, one processor factors the diagonal block, then broadcasts the L factor of the diagonal block to processors in its grid row and the U factor to processors in its grid column. The receivers solve the triangular linear systems that give the entire block row and block column of the factors. The block column of L is then broadcast down the matrix, the block row of U is broadcast to the right, where they are multiplied and used to update the trailing submatrix.

for $k = 1 : n/\mathcal{B}$
  set $K = k \bmod p$
  processor (K, K) factors $L_{KK}U_{KK} = A_{KK}$
  processor (K, K) broadcasts L_{KK} to processors $(K, :)$
  processor (K, K) broadcasts U_{KK} to processors $(:, K)$
  for all $J = 1 : p$ in parallel
    set $J' = \{\ j' > i, J = j' \bmod p\}$
    processor (K, J) solves $L_{KK}U_{Kj'} = A_{Kj'}$ for all $j' \in J'$
    processor (K, J) broadcasts $U_{Kj'}$ to processors $(:, J)$ for $j' \in J'$ (all in one message)
  end
  for all $I = 1 : p$ in parallel
    set $I' = \{\ i' > i, I = i' \bmod p\}$
    processor (I, K) solves $L_{i'K}U_{KK} = A_{i'K}$ for all $i' \in I'$
    processor (I, K) broadcasts $L_{i'K}$ to processors $(J, :)$ for $i' \in I'$ (all in one message)
  end
  for all $J = 1 : p$ and $I = 1 : p$ in parallel
    set $I' = \{\ i' > i, I = i' \bmod p\}$
    set $J' = \{\ j' > i, J = j' \bmod p\}$
    processor (I, J) updates $A_{i',j'} = A_{i',j'} - L_{i',i}U_{i,j'}$ for $i' \in I'$, $j' \in J'$
  end
end

Each processor stores $(n/(p\mathcal{B})) \times (n/(p\mathcal{B}))$ blocks. That is, the size of the sets I' and J' for $i = 1$ is $n/(p\mathcal{B})$. In each of the $n/\mathcal{B}$ phases of the algorithm a processor sends and receives a constant number of messages whose total size is $O(\mathcal{B} \times (n/p))$. The total time spent on communication in the algorithm is $O(\alpha n \lg p/\mathcal{B} + \beta n^2/p)$.

The amount of work per processor in a phase is

$$O\left(\mathcal{B}^3 \times \left(\frac{n}{p\mathcal{B}}\right)^2\right) = O\left(\mathcal{B}\left(\frac{n}{p}\right)^2\right) .$$

Hence, the computation to communication ratio in the algorithm is n/p, the same as in the 2D matrix-multiplication and in the triangular solver.

This algorithm is asymptotically work-efficient even when there is only one block per processor ($\mathcal{B} = n/p$), but its load balance and work efficiency improves with smaller block-size $\mathcal{B}$.

The new 3D LU algorithm presented in this work does no perform any numerical pivoting, a technique for enhancing the accuracy and stability of the LU factorization algorithm. We assume that the input matrix can be factored without pivoting, which is true for some classes of matrices, such as diagonally dominant ones.

CHAPTER 3

Communication-Efficient Triangular Solvers

This chapter 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 3-dimensional assignment of processors to the 3-dimensional computation structure of the substitution algorithm. We start with a description of the algorithm in 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 later.

We write a system of n linear equations with unknown $x_1, x_2, ..., x_n$

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n &= b_2 \\
&\cdot \\
&\cdot \\
&\cdot \\
a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n &= b_n
\end{aligned}$$

as a matrix equation $Ax = b$, where A is an n by n matrix and x and b are n-vectors. When we have multiple systems all with the same coefficient matrix A but with different right hand sides $b_1, b_2, ..., b_m$, we write all the systems as a matrix equation $AX = B$ where X and B are n by m matrices.

We refer to processors using their position in the 3-dimensional processor grid. For example, processor ijk is the processor in position (i, j, k) in the grid. We also need to refer to entire 2-dimensional 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 3-Dimensional grid of processors G, of l-by-m-by-n processors, is written as G_{ijk}, where $1 \leq i \leq l$, $1 \leq j \leq m$, $1 \leq k \leq n$.

3.1. A Simple Case

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 3-dimensional 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 (noncontiguous) 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 serially performed, 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 2-dimensional 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 3.1.1. *In a syncronous 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 3.1.2. *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 3.1.3. *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 $C_n = O(n^2 p \lg(n/p))$. □

3.2. 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 $O(n^2p)$. It is asymptotically the same as the amount that the parallel 3D matrix multiplication performs. The key to the improvment 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 delaid 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 3.2.1. *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 3.2.2. *The number of computational steps that the algorithm requires is $O(n^3/p^3 + n^2/p)$.*

PROOF. The proof of Theorem 3.1.1 applies here as well. □

COROLLARY 3.2.3. *The algorithm is asymptotically work efficient when $p \leq \sqrt{n}$.*

THEOREM 3.2.4. *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. $\square$

3.3. A More Detailed Analysis of Communication

The previous sections discussed the asymptotic behavior of the algorithm. Since the reduction in communiction 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.

3.4. The General Case

This section shows how to relax all the assumptions that we have made earlier about the number of processors and the size of the problem. The more general cases require a moderately more complex algorithm. We perform some analysis on the generalized algorithm, but we do not generalize all the results that we have obtained so far. Our objectives in the analyses that follow is to show that the behavior of the generalized algorithm is similar to that of the simple algorithm, and not to derive exact expressions for time and communication for each case.

3.4.1. Processor Grids with Arbitrary Shapes. In most cases the number of processors P is not a perfect integer cube ($P = p^3$). Suppose that $P = p_1 \times p_2 \times p_3$ for some p_1, p_2, and p_3. To handle this case we stop the recursion when the problem size reaches $p = \sin(p_1, p_2, p_3)$. This ensures that all processors store exactly the same number of cube indices in a size-p problem. In fact, the index triplets that a processor is responsible for form a $\frac{p}{p_1} \times \frac{p}{p_2} \times \frac{p}{p_3}$ cube. In the simple case, $p_1 = p_2 = p_3 = p$ and each processor is responsible for a single cube index.

To generalize the algorithm for this case, we only need to explain how to solve a p-by-p triangular system with p right-hand sides on a p_1-by-p_2-by-p_3 processor grid. 3D matrix multiply works on any 3D processor grid, and the overall recursive algorithm remains unchanged.

The base case takes $O((p/p_3)(p + p^2/p_1p_2))$ steps. Each ik layer in the processor grid stores a copy of A on a p_1-by-p_2 processor subgrid and solves p/p_3 linear systems. Consider the solution of one system (one right-hand side). The processors in the layer can solve for the first p_2 unknowns and eliminate them from the p equations is $O(p_2 + p/p_1)$, since it takes a processor p/p_1 steps to eliminate one unknown from all the equations that it stores. The number of steps is $O(p_2 + p/p_1)$ and not $O(p_2 \times p/p_1)$ since we use pipelining. There are p/p_2 groups of p_2 unknowns, so solving for a single right-hand side takes $O(p + p^2/p_1p_2)$. Multiple linear systems are independent so the total number of steps is $O((p/p_3)(p + p^2/p_1p_2))$.

The amount of communication in this generalized base case is $\Theta(p^2(p_1 + p_2 + p_3))$. There are p^2 unknowns, and each one, after it is solved for, is broadcast down a column of p_1 processors. There are also p^2 elements in the right-hand-side matrix, and each one is updated once by a row of p_2 processors. Finally, we need to replicate A on each of the p_3 layers.

From here on to the end of the chapter we take p to be $p = \sin(p_1, p_2, p_3)$ where the number of processors is $P = p_1 \times p_2 \times p_3$.

3.4.2. Unbalanced Divide and Conquer. Suppose that $n = p \times L$ where L is not a power of 2. We handle this case by splitting the matrices such that A_{11} (and also B_{11} and X_{11}) is $p\lfloor L/2\rfloor$-by-$p\lfloor L/2\rfloor$ and A_{22} is $p\lceil L/2\rceil$-by-$p\lceil L/2\rceil$. This implies that the recursive triangular solver must be prepared to solve n-by-n linear systems with m right-hand sides, where m is also a multiple of p. At the base of the recursion, we have to solve a p-by-p linear system with m right-hand sides. We simply solve groups of p right-hand sides one after the other. The 3D matrix multiplication algorithm does not require square inputs, so there is no problem with this step in the recursion.

The difference between n and m is always 0, p, or $-p$. Therefore, the shapes of the matrices that are multiplied are never far from square, and the number of right-hand sides in the base case is either 0, p, or $2p$. This behavior ensures that the algorithm does not lose its "3D-ness".

3.4.3. Arbitrary Matrix Orders. When n is not divisible by p, we conceptually pad the matrices with zeros to the next multiple of p. This is done by maintaining the dimensions of the blocks that we operate on, not by storing explicit zeros. The partial blocks at the end of the matrix may cause some load imbalance, but they do not consume more memory.

3.4.4. Block-Cyclic Data Layouts. Stopping the recursion at p-by-p leads to parallel triangular solves where each processor performs only $O(1)$ work and in which single-word messages must be sent and received. We can reduce the number of messages by laying out the matrices using a block-cyclic distribution with block size r and stopping the recursion at rp-by-rp blocks. This reduces the number of messages; the same amount of data is communicated using fewer, larger messages.

This technique also reduces the amount of parallelism. We have seen that the basic algorithm is work efficient as long as $p = O(\sqrt{n})$. With a block-cyclic data layout and schedule, we must have $rp = O(\sqrt{n})$ or $p = O(\sqrt{n}/r)$. When r is large, more work is done inside base cases, which are inefficient, and less work is done in the context of efficient matrix multiplications.

CHAPTER 4

Communication-Efficient Triangular Factorization

4.1. The 3-Dimensional Base Case

Like the 3D triangular solver, the new LU-factorization algorithm is recursive. This section describes the base case of the recursion. The recursion stops when a 3D grid of p-by-p-by-p processors factors a p-by-p matrix. The most important property of the algorithm that we present here is that it lays out its output, L and U, in a distribution that is consistent with the higher levels of the recursion. This allows us to obtain an elegant recursive 3D factorization algorithm. The algorithm uses processors inefficiently, but it turns out that the inefficiency in the leaves of the recursion does not have a significant impact on the overall efficiency of the algorithm.

The algorithm that we use in the base case is a particular parallel schedule of the following sequential code:

for $k = 1 : n$
    factor $a_{kk} = l_{kk}u_{kk}$ (set $l_{kk} \leftarrow 1$, $u_{kk} \leftarrow a_{kk}$)
    for $i = k + 1 : n$
        $l_{ik} = a_{ik}/u_{kk}$
    end
    for $j = k + 1 : n$
        $u_{kj} = a_{kj}/l_{kk}$
    end
    for $i = k + 1 : n$
        for $j = k + 1 : n$
            $a_{ij} = a_{ij} - l_{ik}u_{kj}$
        end
    end
end

In the parallel algorithm, we perform the iteration of the outer loop in a sequence, one after the other. But the body of the outer loop is almost entirely parallelized. The iteration begins with one processor factoring a_{kk}. Next, $n-k$ processors perform the iteration of the first i loop concurrently, and another $n-k$ processors perform the iterations of the first j loop. Finally, another group of $(n-k)^2$ processors perform concurrently all the iterations of doubly-nested i-j loop.

We assign processors to operations in the algorithm in a simple way. We label the processors according to their position in the 3D processor grid. Processor ijk in the grid for some i, j, and k performs at most one operation in the kth iteration of the outer loop and nothing in other iterations of the outer loop. If $i = j = k$, then the processor performs the factorization of a_{kk}. If $i > j = k$, then the processor computes l_{ik}. If $j > i = k$, then the processor computes u_{kj}. If $i > k$ and $j > k$, then the processor updates a_{ij}. Otherwise, the processor does nothing.

Information flows in this algorithm such that each processor receives the data that it needs exactly when it needs it. The results of factorization of a_{kk} is broadcasted to the processors that compute l_{ik} for $i > k$ and to the processors that compute u_{kj} for $j > k$. After computed, l_{ik} is broadcasted to processors in the ith row, and u_{kj} is broadcasted to processors in the jth column. The processors for which $i > k$ and $j > k$, after updating a_{ij}, send their results to their neighbors in the $(k + 1)$st ij layer. In the next iteration, the algorithm factors $a_{k+1,k+1}$, and so on.

THEOREM 4.1.1. *At the end of p iterations, the L factor is distributed on each ik layer, and the U factor is distributed on each kj layer.*

PROOF. At the first iteration, the left column of L and the upper row of U are computed and replicated on all the ik layers and all the kj layers respectively. At the end of this iteration, the updated a_{ij}s for $i, j > 1$ are distributed on processors of the second ij layer for which $i, j > 1$.

Now, assume that during the $l - 1$ first iterations, the $l - 1$ left columns of L and the $l-1$ upper rows of U are computed and replicated on all the ik layers and all the kj layers respectively. Also assume that at the end of the $l - 1$ iteration, the updated a_{ij}s for $i, j > l$ are distributed on processors of the lth ij layer for which $i, j > l$.

Then, during the lth iteration, a_{ll} is first factorized, and the results are replicated on all the ik layers (L) and all the kj layers (U). After that, the lth row of the lth layer computes the lth row of U, while the lth column computes the lth column of L. According to the algorithm, at the end of this computation, the lth row of U, and the lth column of L are replicated to all the ik layers and all the kj layers respectively. This concludes the proof. □

4.2. A Recursive Algorithm for LU Factorization

The algorithm that we just described is used as the base case in a recursive algorithm. The complete algorithm is a straightforward recursive LU factorization algorithm.

1. If A is p-by-p, factor it using the algorithm in Section 4.1 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 3.2.
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 $\widetilde{A_{22}} = L_{22}U_{22}$.

The amount of communication that this algorithm performs is asymptotically the same as the amount of communication in....

THEOREM 4.2.1. *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 $2n^3/p^3$. The time to perform a triangular solve with n right hand sides is $O(n^3/p^3+n^2/p)$, as shown in Theorem 3.1.1. We therefore have the following recurrence for the number of steps,
$$T_n \le \begin{cases} 2T_{n/2} + c(n^3/p^3 + n^2/p) & \text{when } n > p \\ p & \text{when } n = p \end{cases}$$
for some constant c.

It's not hard to verify by induction that the solution of this recurrence is $T_n \le d(n^3/p^3 + n^2/p) = O(n^3/p^3 + n^2/p)$ for some constant d. The base of the induction holds with any $d \ge 1$ since $T_p = p \le 1(1+p)$. Assume that the claim holds for $n/2$. We have
$$\begin{aligned} T_n &\le 2T_{n/2} + c(n^3/p^3 + n^2/p) \\ &\le 2d(n^3/8p^3 + n^2/4p) + c(n^3/p^3 + n^2/p) \\ &= (d/4 + c)(n^3/p^3) + (d/2 + c)(n^2/p) \\ &\le (d/2 + c)(n^3/p^3 + n^2/p) . \end{aligned}$$
The inductive claim holds for any d such that $d/2 + c \le d$, or $d \ge \max(1, 2c)$, because for such d we have $d/2 + c \le d$. ☐

COROLLARY 4.2.2. *The algorithm is asymptotically work efficient when $p \le \sqrt{n}$.*

PROOF. When $p \le \sqrt{n}$ the product of execution steps by processors is bounded by the amount of work in the sequential algorithm,
$$p^3 \times O(n^3/p^3 + n^2/p) = O(n^3 + n^2p^2) \le O(n^3) .$$

□

THEOREM 4.2.3. *The amount of communication in the algorithm is $O(n^2p)$.*

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 order-n matrices, as well as the amount to solve an order-n triangular linear system with n right hand sides, is $O(pn^2)$. We therefore have the following recurrence for the amount of communication,

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

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

Although we can improve the algorithm by removing redundant communication, as we have done for the triangular solver, we cannot improve the asymptotic behavior of the algorithm. This algorithm does not benefit much from this technique because it only makes 2 half-size recursive calls, as opposed to 4 in the triangular solver. Because the amount of communication that the algorithm performs on problems of size n is proportional to n^2, the total amount of communication drops in all the recursive children of a routine is only half of the amount that the parent performs. Hence, most of the communication is performed at the root of the recursion. In the triangular solver, on the other hand, which makes 4 recursive calls, the children perform about the same amount of communication as the parent, so each "level" in the recursion performs the same amount of communication, leading to a $\lg(n/p)$ factor more communication than the root performs. Hence, applying the same technique here would improve the constants but not the asymptotic behavior.

4.3. The General Case

This algorithm can be applied to arbitrary processor grids, matrix orders, and block-cyclic data layouts. The techniques are identical to the techniques that we used to generalize the 3D triangular solver, so the details are omitted.

CHAPTER 5

Implementation and Experimental Results

We implemented the algorithms that the previous chapters describe and conducted experiments to evaluate their performance. This chapter describes the implementation, as well as experimental results. We focus on the performance of the *LU* factorization algorithm, which calls the triangular solve, and not on the performance of the triangular solver alone, since we expect that it will rarely called directly.

5.1. Implementation

We implemented the algorithms in C using the Message Passing Interface (MPI [**10, 20**]). The implementation assume that the input matrices are laid out in a block-cyclic distribution on faces of the processor grid. The user is responsible for creating the 3D processor grid (using standard MPI functions) and for laying out the input matrices. The user, therefore, decides on the dimensions of the grid and on the block size of the distribution. By creating the 3D processor grid with an MPI call that creates such a grid, the user allows MPI to embed the processor grid onto the physical parallel computer in a way that minimizes communication, if the MPI implementation is capable of such "smart" embeddings.

This approach is similar to the approach that ScaLAPACK [**7, 4**] uses. ScaLAPACK is a library of parallel distributed-memory algorithms for linear algebra computations. The user of ScaLAPACK must set up a 2D processor grid and lay out the input matrices on the grid.

We tested the algorithm for correctness by writing the input matrices and output factors into files that Matlab can read and checking the accuracy of the factorization in Matlab (Matlab is an interactive environment for matrix computations).

5.2. Computational Environment for the Experiments

We conducted experiments on two parallel computers, a cluster of workstations running Linux and an SGI Origin parallel computer.

The cluster of workstations contain dual-CPU computers with Intel Pentium III processors running at 500 or 550MHz (the cluster also contains 600MHz dual-CPU computers but we didn't use these in our experiments). The computers are interconnected using a 3Com fast Ethernet switch (100Mbits/s per computer). The workstations that we used in our experiments were idle but other workstations in the cluster

Matrix Order	Our 3D LU	ScaLAPACK's 2D Cholesky
1000	3.03	0.81 (1.82)
2000	10.80	3.18 (7.16)
4000	49.05	16.32 (36.72)

TABLE 1. The running time, in seconds, of our 3D LU factorization algorithm on a 2-by-2-by-2 processor grid with the running of ScaLAPACK's LL^T algorithm (e.i. Cholesky) on a grid of 3-by-3 processors. Note that Cholesky requires half the amount of work and communication that LU requires, and that ScaLAPACK used one processor more than our algorithm. The numbers in parenthesis are the ScaLAPACK running times scaled by 2.25, the extra amount of work per processor in our algorithm. A fair comparison is between the numbers in parenthesis and the 3D numbers. The matrices are laid out in block-cyclic distributions with block size 50.

may have been used, which might have slightly degrades communication performance. We only used one processor in each node in our experiment, in order to create a parallel machine with a uniform communication fabric. 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. The cluster has 34 computers but not all were running and we never used more than 8.

The SGI Origin system has 112 400MHz R12000 processors. The entire system runs under a single operating system, but its memory is physically distributed among dual-CPU nodes. The operating system was IRIX version 6.5 and we used SGI's C compiler, their implementation of MPI, and sequential subroutines from SGI's scientific subroutine library. During our experiments only 108 processors were available, and between 8 and 11 processors were in use by other programs. We note that this machine was still in testing after installation, so its performance may have been less that optimal due to configuration or hardware problems.

5.3. Experimental Results

Table 1 shows the results of our experiments on the cluster. The table compares the running time of our LU factorization algorithm on a 2-by-2-by-2 grid with the running of ScaLAPACK's LL^T algorithm (e.i. Cholesky) on a grid of 3-by-3 processors. We compare our LU algorithm with ScaLAPACK's Cholesky, rather than ScaLAPACK's LU, since ScaLAPACK's LU performs partial pivoting, which affect performance. Our algorithm is more similar, in terms of both communication and computation, to a Cholesky factorization algorithm, except that it performs almost exactly twice the amount of work and communication. Furthermore, our algorithm only used 8 processor while ScaLAPACK's used 9, in order to have a "best-case" processor grid for each algorithm. Thus, one should multiply ScaLAPACK's running times by $2 \times (9/8) = 2.25$ in order to fairly compare them to the running times of our algorithm.

The matrices are laid out in block-cyclic distributions with block size 50, which we have determined in preliminary experiments to be near-optimal for both algorithms.

The main conclusion that one should draw from the table is that our algorithm is competitive with ScaLAPACK's 2D algorithm, even when P is small. Our algorithm is slower, but not by much—for large n, it is slower by about 33%. Given that ScaLAPACK is production-quality software, we expect it to be more highly tuned. We do not claim that our algorithm can be made to run faster or as fast as ScaLAPACK in all cases. Some of the performance difference in favor of ScaLAPACK is probably due to different constants hidden in the asymptotic notation, and it is quite likely that the constants in our algorithm are slightly larger. Our main point is that even if the constants are larger in our algorithm and, therefore, our algorithm performs more communication for small P, the difference is not large.

Our analysis shows that the amount of communication in our algorithm scales proportionally to $P^{1/3}$, whereas the amount in ScaLAPACK's 2D algorithm scales proportionally to $P^{1/2}$, so we expect our algorithm to become faster as P grows. We did not have enough processors to test this claim.

A secondary but important conclusion to be drawn from Table 1 is that both algorithm parallelize well, even on a machine with slow communication. A node running at 500MHz is capable of sustaining about 330 million floating-point operations per second (330Mflops) in a matrix-multiplication subroutine (measured using ATLAS). The TCP/IP throughput of one node is about 6MBytes/sec, or less than 1 million words per second. Clearly, this parallel computer has slow communication. The computational rate of ScaLAPACK on matrices of order 4000 is about 145Mflops per node, which represents a speedup of about 4 over the peak performance of one node. Our algorithm is, as we have seen, somewhat slower, but still presents a nice speedup.

Obviously, on smaller matrices a larger fraction of the running time is spent on communication and therefore little or no effective speedup is achieved. When we scale down the order of the matrix by a factor

# processors→ Matrix Order ↓	1	8	27	64
500		0.26	0.26	0.36
1000	4.80	0.97	0.90	1.26
2000	34.00	4.39	3.77	4.69
4000		35.46	22.17	20.65

Table 2. The performance of our 3D *LU* factorization algorithm on an SGI Origin. The table shows the number, in seconds, to factor a matrix. The block size here too is 50.

of 2, the number of messages remains the same, the total volume of communication goes does by a factor of 4, and the number of floating point-point operations goes down by almost a factor of 8. Therefore, on small matrices, communication dominates the running time, in both algorithms.

Table 2 shows the results of our experiments on an SGI Origin parallel computer. The table shows the running time, in seconds, of our 3D *LU* factorization algorithm running on 1, 8, 27, and 64 processors, always arranged in a perfect 3D cube.

The table shows that the algorithm achieves excellent speedups as long as the matrix is large enough compared to the number of processors. For example, on matrices of size 2000 we achieve a speedup of 7.75 with 8 processors.

When the number of processors grow, for a fixed matrix size, the speedup improves more slowly and finally decreases. This is due to more communication and to more work being done in the inefficient base case of the recursion. For example, on matrices of sizes 1000 and 2000 performance improves only slightly when we increase the number of processors from 8 to 27. The running times in both cases actually increase when we add more processors.

The speedups improve, however, when we solve larger problems. The running times on matrices of size 4000 improve monotonically when we increase the number of processors to 27 and even 64. The efficiency, of course, drops.

CHAPTER 6

Communication Lower Bounds for Matrix Multiplication

In this chapter we analyze the amount of interprocessor communication that matrix multiplication algorithms must perform on a distributed-memory parallel computer. We assume again a conventional matrix multiplication algorithm in which each c_{ij} is computed as a sum of products $a_{ik}b_{kj}$.

We begin by showing a lower bound on the amount of communication that any parallel load-balanced algorithm must perform. The so-called 3-dimensional algorithm achieves this bound.

THEOREM 6.0.1. *Consider the conventional multiplication of two n-by-n matrices on a P-processor distributed-memory parallel computer. A processor that performs at least $\epsilon n^3/P$ scalar multiplications for some $\epsilon > 0$ must send or receive at least*

$$\left(\frac{\epsilon}{2}\right)^{2/3} \frac{n^2}{P^{2/3}} - N_B - N_E$$

words, where N_B is the number of elements of A and B that the processor stores before the computation begins, and where N_E is the number of elements of C that the processor computes without communication and stores at the end of the computation.

PROOF. We assume for now that the processor stores no elements of A and B initially and that it must send out all the partial sums that it computes.

We now analyze the number of scalar multiplications possible with N words. The N words can be words of A, of B, or words containing partial summations of elements in the product C. The analysis is similar to the analysis in the proof in [**21**]. We denote the set of rows of A from which the processor receives $\sqrt{N}$ elements or more by S^1, and we denote by S^2 the rest of the rows. There are at most $\sqrt{N}$ rows in S^1. The number of scalar multiplications involving elements of A from rows in S^1 is at most $N\sqrt{N}$. The number of multiplications involving elements of A from rows in S^2 is also bounded by $N\sqrt{N}$, which is the maximal number of elements in a row of S^2 times the number of elements in C, that partial sums of products of them are computed by the processor. Therefore, the number of scalar multiplications that the processor can perform is at most $2N\sqrt{N}$.

Therefore, for a processor that performs at least $\epsilon n^3/P$ multiplications we must have

$$2N\sqrt{N} > \epsilon n^3/P$$

or

$$N \geq \left(\frac{\epsilon}{2}\right)^{2/3} \frac{n^2}{P^{2/3}} \; .$$

□

The simplest way to apply this theorem is to assume that $N_E = N_B = 0$. That is, that no processor stores any element of the input when the algorithm starts and that the processors must send out all the elements of the output. Under this assumption, the amount of communication that the 3D algorithm performs is asymptotically optimal.

We now turn our attention to matrix-multiplication computations with a limited amount of memory. The analysis models algorithms that cannot replicate the inputs arbitrarily, and in particular, 2D algorithms that do not replicate at all.

THEOREM 6.0.2. *Consider the conventional multiplication of two n-by-n matrices on a P-processor distributed-memory parallel computer, where each processor has M words of memory. If each processor performs $\Theta(n^3/P)$ scalar multiplications and M is at most $O(n^2/P^{2/3})$, then each processor must perform at least $\Omega(n^3/P\sqrt{M})$ words with the other processors.*

PROOF. Let us look at processor P_k for any $1 \leq k \leq P$. We decompose a given schedule for P_k of a conventional data-flow matrix multiplication into phases, such that during each phase, exactly M words are sent or received by P_k, except perhaps the last phase.

We say that c_{ij} is *alive* during phase q if during phase q, P_k computes products $a_{il}b_{lj}$ for some l, and if some partial sum of these products is either in memory space of P_k at the end of the phase, or is sent from P_k during the phase. By the definition of a live c_{ij}, there can be at most $2M$ live c_{ij}'s during a phase. Note that for each triplet (i, j, l), the multiplication $a_{il}b_{lj}$ must be performed in some phase in which c_{ij} is alive. A multiplication $a_{il}b_{lj}$ that is performed in a phase in which c_{ij} is not alive does not represent progress toward the scheduling of the data-flow graph, so without loss of generality we assume that there are no such multiplications in the schedule.

There can also be at most $2M$ distinct elements of A and at most $2M$ distinct elements of B in the memory space of P_k during phase q, because every such element must be either in memory space of P_k at the beginning of the phase, or it must be received from another processor during the phase. We denote the set of elements of A, that are in the memory space of P_k during phase q, by A_q.

How many multiplications can processor P_k perform in phase q?

We denote by S_q^1 the set of rows of A with $\sqrt{M}$ or more elements of A_q, and by S_q^2 the set of rows with fewer elements in A_q. Clearly, the size of S_q^1 is at most $2\sqrt{M}$.

We begin by bounding the number of scalar multiplications during phase q, that involve elements of A from rows in S_q^1. Each element of B is used exactly once when a row of A is multiplied by B. Therefore, the number of scalar multiplications in phase q that involve elements of A from rows in S_q^1 is bounded by the size of S_q^1 times the number of elements of B in memory space of P_k during the phase, or $2\sqrt{M}\times 2M = 4M^{3/2}$.

We now bound the number of scalar multiplications during phase q that involve elements of A from rows in S_q^2. Each c_{ij} is a product of a row of A and a column of B. The number of multiplications that involve elements of A from rows in S_q^2 is bounded by the number of live c_{ij}'s times the maximal number of elements in a row of S_q^2 that are in A_q, or $2M\sqrt{M} = 2M^{3/2}$.

We conclude that the number of multiplications per phase is at most $6M^{3/2}$. Since the total number of multiplications, that are performed by P_k, is $\Theta(n^3/P)$, the number of phases is at least $\Theta(\lceil (n^3/P)/M^{3/2}\rceil)$. So, if $M = O(n^2/P^{2/3})$, the number of words that are sent or received by P_k is at least $\Theta(M(n^3/P)/M^{3/2})$. Therefore, for $M = O(n^2/P^{2/3})$, the number of words transferred is at least $\Theta(n^3/P\sqrt{M})$. $\square$

Conclusion 6.0.3. Assume the conventional data-flow matrix multiplication of two n-by-n matrices should be performed by a P-processor distributed-memory computer, where the size of memory of each processor is $M = \Theta(n^2/P)$, and that each processor should perform $\Theta(n^3/P)$ scalar multiplications. Then the 2D matrix-multiplication algorithm (e.g., Cannon's) is the optimal in terms of amount of communication.

Proof. Each processor must communicate at least

$$\Omega(n^3/P\sqrt{M}) = \Omega(n^3/P\sqrt{n^2/P}) = \Omega(n^2/\sqrt{P})$$

words with other processors. Summing over all processors we find that the algorithm must transfer $\Omega(\sqrt{P}n^2)$ between processors. $\square$

CHAPTER 7

Conclusions, Future Work, and Open Problems

Our main conclusion is that our new linear solver is more communication-efficient than any existing dense linear solver. It also uses more storage than existing solvers. Like the 3D matrix multiplication algorithm, the new algorithm reduces communication by replication.

More specifically, our algorithm perform asymptotically a factor of $P^{1/6}$ less communication than any existing algorithm, where P is the number of processors. The algorithm is general and works for matrices of any order and processor grids $p_1 \times p_2 \times p_3$ for any p_1, p_2, and p_3. The algorithm is more efficient when the processor grid is a perfect cube or nearly so. Our experimental results show that when P is small our algorithm spends more time on communication than existing algorithms, however. The difference is not large and we expect our algorithm to be faster with large P.

It would be fairly straightforward to extend our work by designing a 3D Cholesky factorization algorithm. Our algorithm does not perform any form of pivoting, so it is numerically unstable for general matrices (it is stable for symmetric positive-definite and for diagonally-dominant matrices). The most widely used class of matrices whose factorization does not require pivoting is the class of symmetric positive-definite matrices. It is possible to factor such matrices using half the amount of work and storage that the LU algorithm performs using a Cholesky factorization $A = LL^T$. The main challenge in designing a 3D Cholesky algorithm is to use only half the number of processors, most likely in a prism-like half cube grid. Using half as many processors would ensure that the Cholesky algorithm exploits processors about as efficiently as the algorithm that this thesis presents. If we use a full processor cube, the efficiency of the Cholesky algorithm would be about one half of the efficiency of LU, since it would perform half the amount of work in the same amount of time (and storage).

A more challenging problem is to design a 3D partial pivoting LU factorization algorithm. Is there a pivoting LU algorithm that performs only $O(pn^2) = O(P^{1/3}n^2)$ communication and is asymptotically work efficient when $p = O(\sqrt{n})$? This is an open question.

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. Communicatio 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] L. E. Cannon. *A Cellular Computer to Implement the Kalman filter algorithm*. PhD thesis, Montana State University, 1969.

[6] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. A proposal for a set of parallel basic linear algebra subprograms. Technical Report CS-95-292, University of Tennessee, May 1995.

[7] 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.

[8] 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.

[9] Eliezer Dekel, David Nassimi, and Sartaj Sahni. Parallel matrix and graph algorithms. *SIAM Journal on Computing*, 10:657–675, 1981.

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

[11] G. C. Fox, S. W. Otto, and A. J. G. Hey. Matrix algorithms on a hypercube. i: Matrix multiplication. *Parallel Computing*, 4:17–31, 1987.

[12] W. Morven Gentleman. Some complexity results for matrix computatiosn on parallel processors. *Journal of the ACM*, 25(1):112–115, 1978.

[13] 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.

[14] Ching-Tien Ho, S. Lennart Johnsson, and Alan Edelman. Matrix multiplication on hypercubes using full bandwidth and constant storage. In *Proceedings of The 6th Distributed Memory Computing Conference*, pages 447–451. IEEE Computer Society Press, 1991. Also published as TR-19-91, Center for

Research in Computing Technology, Harvard University, and available online from `http://rodin.cs.uh.edu/ johnsson/rpt.html`.

[15] S. Lennart Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *Journal of Parallel and Distributed Computing*, 2:133–172, 1987.

[16] S. Lennart Johnsson. Data parallel programming and basic linear algebra subroutines. In J. R. Rice, editor, *Mathematical Aspects of Scientific Software*, volume 14 of *IMA Volumes in Mathematics and Its Applications*, pages 183–196. Springer-Verlag, 1988.

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

[18] Woody Lichtenstein and S. Lennart Johnsson. Block cyclic dense linear algebra. *SIAM Journal on Scientific Computing*, 14:1257–1288, 1993.

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

[20] 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.

[21] Sivan Toledo. A survey of out-of-core algorithms in numerical linear algebra. In James M. Abello and Jeffrey Scott Vitter, editors, *External Memory Algorithms*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 161–179. American Mathematical Society, 1999.

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

[23] Robert A. van de Geijn. *Using PLAPACK*. MIT Press, 1997.