NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

A SURVEY OF PARALLEL ALGORITHMS IN NUMERICAL LINEAR ALGEBRA

Don Heller

February 1976

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213

This work has been supported by the National Science Foundation under Grant MCS75-222-55 and the Office of Naval Research under Contract N00014-76-C-0370, NR 044-422.

TABLE OF CONTENTS

Page

1. Introduction 1
2. Parallel and Pipeline Computers 7
    a. Parallel Computers 7
    b. Pipeline Computers 12
    c. Consequences of the Models 16
3. Basic Algorithms 22
    a. General Expressions and Recurrences 22
    b. Inner Products and Related Computations 25
    c. The Fast Fourier Transform 28
4. Linear Systems 30
    a. General Dense Matrices 30
    b. Triangular Systems 38
    c. Tridiagonal Systems 43
    d. Block Tridiagonal and Band Systems 49
    e. Systems Arising from Differential Equations 52
    f. General Sparse Matrices 60
5. Eigenvalues 63

Acknowledgments 69
References 69

ABSTRACT

The existence of parallel and pipeline computers has inspired a new approach to algorithmic analysis. Classical numerical methods are generally unable to exploit multiple processors and powerful vector-oriented hardware. Efficient parallel algorithms can be created by reformulating familiar algorithms or by discovering new ones, and the results are often surprising. A comprehensive survey of parallel techniques for problems in linear algebra is given. Specific topics include: relevant computer models and their consequences, evaluation of ubiquitous arithmetic expressions, solution of linear systems of equations, and computation of eigenvalues.

Watson: "You have formed a theory, then?"

Holmes: "At least I have got a grip of the essential facts of the case. I shall enumerate them to you, for nothing clears up a case so much as stating it to another person, and I can hardly expect your cooperation if I do not show you the position from which we start."

-- Sir Arthur Conan Doyle, Silver Blaze

1. INTRODUCTION

Numerical algorithms can be generally classified in various ways, such as algebraic vs. analytic, finite vs. infinite, exact vs. approximate. Within recent years a new classification has become important: sequential vs. parallel, brought about by the development of parallel and pipeline computers. These devices allow concurrent arithmetic processing, can easily handle large volumes of information, and often provide hardware facilities for many inherently parallel operations found in numerical linear algebra.

In previous surveys of numerical parallel algorithms, Miranker [71] described early work in several areas, and Poole and Voigt [74] prepared a general annotated bibliography. Our present intention is to provide a more complete and up-to-date discussion of parallel methods for linear systems of equations and eigenvalue problems, along with background information concerning the computer models and fundamental techniques. Original material has been included to create a unified treatment. We consider algorithms using both exact and finite-precision arithmetic, and the inherent complexity of computations. As will be seen, good algorithms for parallel and pipeline computers may be strikingly different from good algorithms for ordinary computers.

The idea behind parallel computers is that programs using P "central" processors should run P times faster than otherwise identical programs using only one processor, although experience and theory show that the actual speedup is often much smaller. Examples of large parallel computers are the Carnegie-Mellon University C.mmp, constructed from up to 16 asynchronous minicomputer processors (Wulf and Bell [72]), and the University of Illinois Illiac IV, with 64 synchronous processing elements under the direction of a single control unit (Barnes et al. [68], Bouknight et al. [72]). The Illiac IV was built by Burroughs Corporation and is now located at NASA Ames Research Center. The Goodyear Aerospace STARAN, with an associative memory and up to 32 processors that can perform serial-by-bit operations on 256 words simultaneously, is a somewhat different approach to high-speed computation (Rudolph [72]), and will not be considered here.

The idea behind pipeline computers is essentially that of an assembly line: if the same arithmetic operation is going to be repeated many times, throughput can be greatly increased by dividing the operation into a sequence of sub-tasks and maintaining a flow of operand pairs in various states of completion. This process is especially suited to vector computations, and vectors are taken as fundamental data types. Thus pipeline computers are also known as vector computers. Available pipeline computers are the Control Data Corporation STAR-100, with two pipes (Hintz and Tate [72]), the Texas Instruments Advanced Scientific Computer, with up to four pipes (Watson [72]), and the recently announced CRAY-1 (Cray [75]). The IBM 2938 Array Processor is also pipelined (Ruggiero and Coryell [69]).

Section 2 of the survey is devoted to a general discussion of parallel and pipeline computers, and how certain architectural features can affect the behavior of algorithms. We consider two realistic models and a theoretical model in which an unlimited number of parallel processors are available. These are not mathematical models of computation, but minimal descriptions intended to allow brief and hopefully meaningful analyses. Of prime importance are the observations that decreases in computation time depend on the ability to move data quickly, and that data must be arranged to conform with the architectural restrictions of the computer.

Section 3 concerns the evaluation of general arithmetic expressions and some important special cases, such as recurrences, inner products and matrix multiplication. The solution of a linear system of equations $Ax = v$ is discussed in Section 4. Csanky [75] has recently answered a major theoretical question concerning bounds on the number of parallel computation steps required to compute x, but the method displayed is unstable. Stable algorithms, such as Gaussian elimination with pivoting, use considerably more time and considerably fewer resources. If A has a regular pattern of zero and nonzero elements then much faster stable algorithms can be derived, and these approach the known theoretical lower bounds on computation time. Finally, parallel eigenvalue calculations are considered in Section 5; the brevity of this section reflects the relatively small number of results.

For simplicity of language we will refer to algorithms for parallel or pipeline computers as parallel algorithms, and hope that no confusion results. Actually, there are many common characteristics, especially the ability to process vectors efficiently.

Parallel algorithms depend on one simple yet crucial observation: independent computations may be executed simultaneously. A set of computations is said to be independent if each result variable appears in only one computation. For example, in vector addition the set of component sums is independent. Thus two N-vectors may be added in a single step using N parallel processors; the vector addition is said to exhibit inherent parallelism, as is any algorithm with large sets of independent computations.

A parallel algorithm may be created by recognizing the inherent parallelism of a sequential algorithm (i.e., an algorithm for a single-processor computer). While many familiar algorithms really are sequential, standard operations of linear algebra often have considerable inherent parallelism. It is sometimes necessary to restructure an algorithm to increase this property. This can involve reordering a linear system or reorganizing a computation to spread operations across several processors. The latter technique leads to the important theoretical result that the inner product of two N-vectors can be computed in $\lceil \log N \rceil + 1$* steps using N processors, and it can be shown that there is no faster algorithm. It is also possible to use an automatic search for parallelism in sequential programs (for example, see Kuck [73]), but we will not consider this aspect. In fact, good sequential programs sometimes obscure their parallelism precisely because they are written for sequential computers.

We use a parenthesized list of indices to emphasize the inherent parallelism of a set of computations in which the operations are identical and only the operands differ. For example, if $C = AB$, $A \in R^{m \times n}$, $B \in R^{n \times p}$, a sequential

*All logarithms are base 2. $\lceil x \rceil$ is the unique integer satisfying $x \le \lceil x \rceil < x+1$.

program to compute C is

```
for i = 1 step 1 until m do
  for j = 1 step 1 until p do
    begin s ← 0; for k = 1 step 1 until n do
                    s ← s + a_ik b_kj;
        c_ij ← s
    end.
```

In the parenthesis notation this becomes

$$s_{ij} \leftarrow 0,\ (1 \le i \le m;\ 1 \le j \le p);$$

for $k = 1$ **step** 1 **until** n **do**

$$s_{ij} \leftarrow s_{ij} + a_{ik}b_{kj},\ (1 \le i \le m;\ 1 \le j \le p);$$

$$c_{ij} \leftarrow s_{ij},\ (1 \le i \le m;\ 1 \le j \le p)$$

or even

$$c_{ij} \leftarrow \sum_{k=1}^{n} a_{ik}b_{kj},\ (1 \le i \le m;\ 1 \le j \le p)$$

when we do not want to be restricted to any particular method for computing the summation.

Once an algorithm has been given, we would like to know how good it is: how much time and how many resources are needed, are these requirements minimal or nearly so, can the algorithm be reconstructed to use fewer resources and still have a respectable running time, is it numerically stable? There are two viewpoints for this analysis. On the one hand, we want to know how difficult a problem can be, so we ask how much work must be done by any algorithm

that solves the problem (complexity). On the other hand, we want to discover algorithms that will lead to reliable, efficient programs (simplicity), though it is always difficult to predict the effect of a specific idea on the execution time of a program. The critical problem for any computation model is to identify which parts of an algorithm are most important, and put the most effort into optimizing those parts (see, e.g., Moler [72], Parlett and Wang [75]).

For the practitioner, the hope is that parallelism will allow the cost-effective and fast solution of larger and more complicated problems. For the mathematician and computer scientist, there are many interesting theoretical questions about the complexity and simplicity of computations.

2. PARALLEL AND PIPELINE COMPUTERS

In this section we describe certain features of parallel and pipeline computers, and some of the ways in which these features affect parallel algorithms. In many cases the choice of an algorithm will be dictated by non-arithmetic considerations, such as storage and communication requirements, and relative performances will be affected by widely varying operating characteristics. Section 2.c is concerned with consequences of the models for numerical algorithms, especially lower bounds on computation time.

For more information about programming and implementation, see Lawrie et al. [75], Newell and Robertson [75], Sameh and Layman [74], Stevenson [75], the technical reports listed by Poole and Voigt [74], and the March 1975 issue of SIGPLAN Notices, which contains the Proceedings of a Conference on Programming Languages and Compilers for Parallel and Vector Machines. Other relevant surveys are Baer [73], Miller [73], Owens [73], and Stone [73b]; architectural issues are discussed by Stone [75b] and T. C. Chen [75].

2.a. Parallel Computers

Our model of parallel computation depends on the use of P identical processors, although real computers like C.mmp allow some differences in the processors. We regard the number of processors as the important parameter. It is assumed that the standard rounding error hypotheses hold for finite precision arithmetic, and that double precision accumulation of inner products is possible. The basic operations (e.g., +, -, ×, /, max, min, and order relations) take two inputs and produce one result in time t_{op}, which depends only on the operation and the data types, and not on the number of processors. It is possible to generalize to r inputs, but as long as r is fixed its value is of minor importance.

We assume a large finite primary memory accessible by each processor, and make the simple but very strong assumption that any processor can obtain any piece of information in unit time. Register load/store costs and I/O for secondary storage will be ignored.

In reality, all parallel algorithms must deal with the complex problems of data manipulation, storage allocation, memory interference and interprocessor communication presented by existing parallel computers. For example, the chaotic relaxation methods (Section 4.e) are designed to handle delays caused by conflicting requests to shared memory. To reduce these conflicts, parallel computers provide each processor with a local memory, along with a communication network to permit data transfers between processors. Since it is far too costly to link each processor directly to all others, a restricted network must be implemented, causing an accessing delay due to increased path lengths between processors. <u>It is possible for this delay to seriously affect program execution times</u>. The Illiac IV network arranges the 64 processors as an 8x8 grid and connects each processor to its four neighbors. This leads to skewed storage of matrices if it is desirable to access rows and columns with equal ease (Kuck [68]).

Instructions for each processor come from the individual processor or from a central control unit. In the first case, the instructions may differ across the processors, yielding the Multiple Instruction Stream - Multiple Data Stream model (MIMD); the processors need not operate synchronously. In the second case, each processor executes the same instruction at the same time, though using different data and depending on a local on/off switch (a mask in the Illiac IV terminology). This is the Single Instruction Stream -

Multiple Data Stream model (SIMD). The terms MIMD, SIMD were first used by Flynn [66]; this is not the only possible taxonomy, but it fits our purposes. The greater part of the survey concerns algorithms for SIMD computers, reflecting both present knowledge and the inherent simplicity of the model.

For definiteness we assume that the processors are synchronous* and count one step for each set of operations performed simultaneously. Thus the important consideration is the number of steps, and not the total number of operations performed by all the processors. Of course, any operations performed in parallel must be independent.

Various models of parallel computation depend on the number of processors. The practical model has a <u>fixed</u> number of processors, independent of the application. The theoretical model allows <u>unlimited</u> parallelism ("sufficiently many processors"), so that the number may vary with the application. We distinguish two cases of unlimited parallelism. If a parallel algorithm solves a problem of size N using P(N) processors, the number of processors is (polynomially) <u>bounded</u> if $P(N) \le p(N)$ for some polynomial p and all N, and <u>essentially infinite</u> otherwise. The number of memory locations may be similarly treated.

Most of the algorithms considered in this survey involve bounded parallelism and memory. Unlimited parallelism is considered for theoretical purposes, but it can provide useful information for fixed parallelism if algorithms can be transformed to reduce the resource requirements. It should be noticed from the following discussion that bounded parallelism is preferred for this purpose, since algorithms using an essentially infinite number of processors

*This has two interpretations: if several different instructions are executed in parallel, then each must be allowed to finish before the next round of instructions is started, or, all instructions executed in parallel are identical.

perform an exponential number of operations per step.

Suppose we are given an algorithm using $P_1(N)$ processors and $T_1(N)$ steps, and we want to construct a new algorithm using $P_2(N) < P_1(N)$ processors and $T_2(N)$ steps, where T_2 is not much larger than T_1. Two principles, algorithm decomposition and problem decomposition (the names are due to Hyafil and Kung [74b]) underlie the transformations to a smaller number of processors. The first depends on a simulation argument: if q_i operations are performed in step i of algorithm 1, then this one step becomes $\lceil q_i/P_2 \rceil$ steps in algorithm 2. Each new step consists of at most P_2 operations performed in parallel, and (Brent [74])

$$T_2 = \sum_{i=1}^{T_1} \lceil q_i/P_2 \rceil \leq \sum_{i=1}^{T_1} (q_i+P_2-1)/P_2$$

$$= T_1 + (q-T_1)/P_2, \quad q = \sum_{i=1}^{T_1} q_i .$$

Problem decomposition depends on the observation that there is often an M such that $P_1(M) \leq P_2(N)$. By partitioning the original problem (size N) into smaller problems (size M) and applying algorithm 1 to the small problems, a new "bootstrapped" algorithm may be obtained. The solution of a linear system by block elimination is a familiar example of this principle.

Recursive doubling, a powerful method of generating parallel algorithms, is related to problem decomposition. The idea is to repeatedly separate each computation into two independent parts of equal complexity, which are then computed in parallel. This is actually a special case of divide-and-conquer. For example,

$$\sum_{i=1}^{N} a_i = \left(\sum_{i=1}^{n-1} a_i\right) + \left(\sum_{i=n}^{N} a_i\right), \ n = \lceil N/2 \rceil ,$$

and by further applications of this splitting the summation can be computed in $\lceil \log N \rceil$ steps using $N/2$ processors. The subproblems need not be smaller versions of the original problem, but should exhibit associative properties so that the partitioning can be continued. Recursive doubling has been applied to several problems, notably in the work of Stone and Kogge on recurrence relations. This is discussed briefly in Section 3.

It is of considerable practical interest to be able to measure the effectiveness of parallelism. For a given problem, parameterized by N, let $T_1(N)$ be the running time of the best known sequential algorithm, and let $T_P(N)$ be the running time of a parallel algorithm using P processors. Speed-up, defined as $S_P(N) = T_1(N)/T_P(N)$, measures the improvement in solution time using parallelism, while efficiency, defined as $E_P(N) = S_P(N)/P$, attempts to measure how well the processing power is being used. A simple argument shows that $S_P \leq P$ and $E_P \leq 1$. Note that $E_1 = 1$, so we don't necessarily want to choose the number of processors in order to maximize this function.

For a class of problems it is necessary to consider some different measures. Let f(x) be the probability that we want to solve problem x taken from class X. Kung and Traub [74] define speedup on the average as

$$SA_P(X,f) = \left(\sum_{x \in X} f(x) T_1(x)\right) \Big/ \left(\sum_{x \in X} f(x) T_P(x)\right)$$

and the average speedup as

$$AS_P(X,f) = \sum_{x \in X} f(x)T_1(x)/T_P(x).$$

Each of these functions has different characteristics and, depending on f, can expose different features of an algorithm.

The goal is to construct algorithms exhibiting linear (in P) speedup and hence utilizing the processors efficiently. That is, for problems of size N we want an asymptotic speedup of the form $S_P(N) = cP - g(P,N)$, with $0 < c \leq 1$, $0 \leq g(P,N) = o(1)$ as $N \to \infty$; c should be independent of P and close to 1. It is suggestive to think of g as a penalty for the use of parallelism on small problems, so for large problems the rewards should outweigh the penalties.

However, linear speedup is not always possible. There are certain computations for which the maximal speedup is $S_P(N) < k$ for a constant k, and such a computation clearly makes poor use of parallelism. For many important problems in linear algebra the best speedup is $S_P(N) = cP/\log P - g(P,N)$, which is acceptable though less than linear.

2.b. Pipeline Computers

We now consider a different approach to high-speed computation, the class of pipeline or vector computers. These are SIMD machines, but their speed is achieved primarily by a form of instruction lookahead in a single processor rather than by use of multiple processors.

By partitioning floating point operations (+, ×, etc.) into more basic sub-operations (exponent adjustment, mantissa arithmetic, etc.) an assembly line structure or pipeline can be set up for repetitive calculations such as componentwise vector operations and inner products. Successive completed results leave the pipeline at a rate determined by the memory transfer rate

and the internal stage delay, and not by the total time required for each arithmetic operation. To simplify memory transfers, vector operands on the STAR must be contiguous blocks of memory locations. Vector operands on the ASC and Cray-1 can be any sequence of locations in arithmetic progression. Interleaving is used to increase transfer rates from relatively slow core memories.

The execution time for a vector operation consists of two parts, an initial delay (called the vector startup time) and the sequential appearance of completed results. We denote this time as $\tau_{op}N + \sigma_{op}$, where N is the length of the vectors involved. For scalar operations we continue to denote the time as t_{op}. Typical values of τ, σ and t for the STAR are given in Table 1; the (core-to-core) vector times include load/store costs while the (register-to-register) scalar times do not.

TABLE 1
Selected CDC STAR Instructions, 64 Bit Floating Point Operands (CDC [74])

operation	t	τ	σ	$\lceil \sigma/(t-\tau) \rceil$	N such that $\tau N+\sigma = 1.5\tau N$
add, subtract	13	.5	96	8	384
multiply	17	1	156	10	312
divide	47	2	156	4	156
square root	73	2	152	3	152
floor, ceiling	11	.5	90	5	360
$a_{n+1} - a_n$	(13)	.5	94	(8)	376
{ a = b (branch)	15-46	-	-	-	-
{ A = B (set condition code)	-	.5	95	(3-7)	380
{ a < b (branch)	15-46	-	-	-	-
{ maximum	-	6	85	(3-10)	29
summation	(13)	4	98	(11)	49
inner product	(30)	4	100	(4)	50

Times in the preceding table are given as multiples of the 40 nano-second cycle time, under certain assumptions. Deviations from these assumptions can cause σ to increase.

Two immediate consequences of the timing formula $\tau N + \sigma$ are the linear dependence of execution time on the number of operations performed, and the encouragement to use vector operations when τ is much smaller than t. For $N > \sigma/(t-\tau)$ it is better to use one vector operation than N scalar operations. Moreover, the use of long vectors is encouraged in order to minimize the effects of σ, which is generally much larger than τ. The last column of Table 1 shows how long the vectors must be so that the actual result rate is 50% greater than the asymptotic result rate.

The STAR also provides another set of instructions for use with sparse vectors, each of which consists of a packed vector of nonzero components and a bit vector describing the true position of these components. The general timing formula for a sparse vector operation is $\tau n + \rho N + \sigma$, where n is the number of nonzeros in the result vector and N is the number of bits in the operand vectors. For a sparse addition, $\tau = 1$, $\rho = 1/8$, $\sigma = 183$.

Many of the problems in constructing programs for pipeline computers involve the isolation of vector operations in a particular setting, and data manipulation to change conceptual data structures into the vector operand format. Consider, for example, the addition of two columns of an $N \times N$ matrix that is stored by rows. Since a vector on the STAR is defined to be a contiguous sequence of memory locations, a matrix column is not a vector, contrary to the usual mathematical interpretation. A vector addition ($O(N)$ time) must therefore be preceded by a special extraction operation ($O(N^2)$ time)

to copy the required column elements into vector operand form. A better data structure or N scalar operations will preserve the linear running time, but these solutions are not always possible or pleasing. The more general definition of a vector used by the ASC avoids this difficulty and the column addition may be performed directly with a vector operation in O(N) time.

A completely different data manipulation problem can occur when secondary storage (i.e., discs) must be used (Lynch [74], Knight, Poole and Voigt [75]). With current technology, the pipeline computation rate is so much greater than the disc transfer rate that, for certain large linear systems, the anticipated I/O time overwhelms the anticipated computation time. The arithmetic unit will be idle for a significant period, simply waiting for its operands. This situation is by no means unique to the pipeline architecture, and will occur in any computer system with a mismatch between computation and transfer rates. It is suggested that intermediate quantities be recomputed rather than stored on disc, with the expectation that a large increase in computation time will be offset by a greater decrease in the I/O time.

We note one side effect of studying vector computers, which occurred during the transition from the CDC 7600 to the STAR at Lawrence Livermore Laboratory (Owens [73], Zwackenberg [75]). LRLTRAN, the local dialect of Fortran, was extended to include vector operations reflecting the STAR instructions, and existing LRLTRAN programs were rewritten using the extensions. Prior to the arrival of the STAR, the new programs were run on the 7600, and it was found that they ran 1.2 to 2.8 times faster than the original codes, since the software vector instructions took greater advantage of the 7600's

lookahead and segmented functional units. This has become known as the "vector 7600" effect.

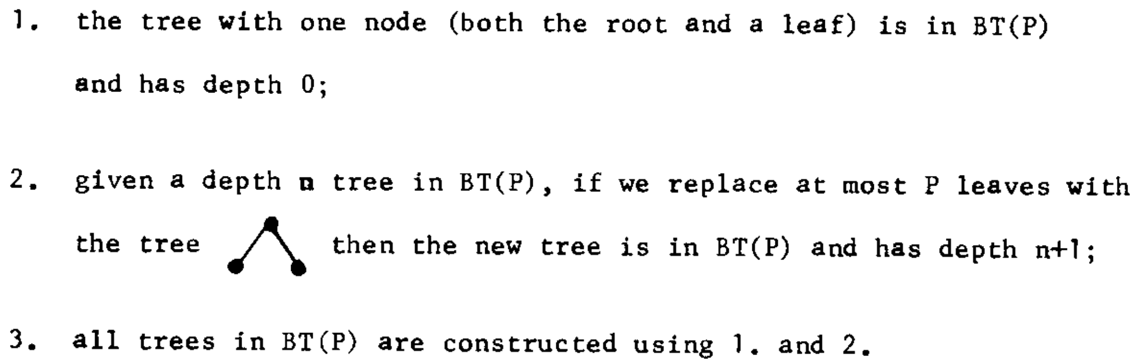
To simplify further discussion, as we did with the model of parallel computation, we will ignore the time required for data manipulation and concentrate on the arithmetic time.

For a problem of size N, where the best scalar algorithm uses $R_s(N)$ operations, suppose we are considering an algorithm using the vector instructions and a total of $R_v(N)$ operations. If N is large we certainly want to avoid the situation where the use of vector operations is harmful rather than beneficial. Following Lambiotte and Voigt [75], we say that the vector algorithm is asymptotically consistent if $R_v(N) = O(R_s(N))$ as $N \to \infty$. An inconsistent vector algorithm, though not universally useful, may still be applicable for some values of N, and it is possible that a consistent algorithm may not be applicable at all because of a large asymptotic constant, but vector algorithms of interest will generally be consistent, take advantage of the increased result rate, and minimize the effects of the startup time σ.

2.c. Consequences of the Models

The fact that the parallel processing element uses only unary and binary operations allows the immediate conclusion of a simple lower bound on the computation time for a problem with N inputs and a single output. Clearly N-1 binary operations are necessary and may be sufficient to compute the result sequentially, but we are interested in knowing how many of these operations may be executed in parallel. For this purpose, consider the set BT(P) of rooted binary trees defined as follows:

1. the tree with one node (both the root and a leaf) is in BT(P) and has depth 0;

2. given a depth n tree in BT(P), if we replace at most P leaves with the tree  then the new tree is in BT(P) and has depth n+1;

3. all trees in BT(P) are constructed using 1. and 2.

A depth n tree in BT(P) corresponds to n steps of parallel computation with P processors. It is important to note that depth is different from height, which is defined as the maximum number of branches between the root and a leaf. While not equal in general, we always have height $\le$ depth. Now, let L(P,n) be the maximum number of leaves on a depth n tree in BT(P), and let m(P,N) be the minimum depth of any tree in BT(P) with N leaves. Clearly

$L(P,0) = 1$,

$L(P,n+1) = \min(L(P,n)+P,\ 2L(P,n))$, and

$L(P,n-1) < N \le L(P,n)$ implies $m(P,N) = n$.

Letting $k = \lceil \log P \rceil$, we have $L(P,n) = 2^{\min(k,n)} + P\max(0,n-k)$, so

$$m(P,N) = \min(k, \lceil \log N \rceil) + \max(0, \lceil (N-2^k)/P \rceil).$$

In summary, at least m(P,N) steps are required to compute one result from N inputs using P processors.

Munro and Paterson [73] have derived a similar but more general lower bound on computation time. This result is important because it allows the translation of complexity theorems from sequential computation to parallel computation.

<u>Theorem</u>. If at least q operations are required to compute a single number Q, then any algorithm using P processors to compute Q must take at least $m(P,q+1)$ steps.

<u>Proof</u>. The maximum number of operations that can be done in n steps with P processors to compute one result is $L(P,n)-1$, which is just the maximum number of non-leaf nodes on a depth n tree in BT(P). Let t be the unique positive integer such that $L(P,t-1)-1 < q \leq L(P,t)-1$, so that $t = m(P,q+1)$. Fewer than t steps cannot compute Q, although we cannot conclude that t steps are sufficient. ■

Actually, we have described the class of optimal algorithms to compute $A_N = a_1 \circ a_2 \circ \ldots \circ a_N$, where o is any associative operation. Each tree in BT(P) with N leaves and minimal depth represents an algorithm for P processors, which we will call an <u>associative fan-in algorithm</u>, and which uses $m(P,N)$ steps and N-1 operations. Figure 1 shows one such tree for $N = 8$, $P = 3$. These methods are more familiarly known as log-sum or log-product algorithms for the original applications in which $\circ = +$ or $\times$, $P = \lfloor N/2 \rfloor$ and $m(P,N) = \lceil \log N \rceil$; we have already used the summation example to illustrate the recursive doubling technique.

In spite of their simplicity, the fundamental importance of the associative fan-in algorithms should not be underestimated. We note that $m(P,N)$ is approximately $N/P + \log(P/2)$ for $P < N$, and $m(P,N) = \lceil \log N \rceil$ for $P \geq N$. Thus the obtainable speedups in computing A_N are $S_P(N) = P - O(1/N)$ and $S_N(N) = N/\log N$. It is seen that although linear speedup is impossible for certain values of P, the fan-in algorithms are optimal in the sense of achieving minimal computation time for all values of P.

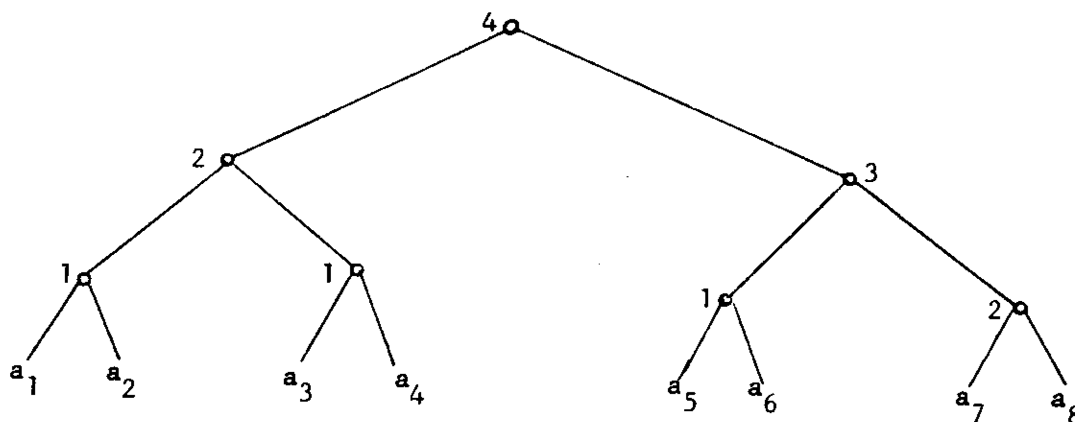


Figure 1
Computation of a_1 o ... o a_8 with 3 processors.
Numbers next to a node denote the step in which the operation is performed.

As an allied question, we can ask how few processors are actually needed to compute A_N in minimal time. Muraoka [71] showed that if $n = \lceil \log N \rceil$ and

$$P(N) = \begin{cases} \lceil (N-2^{n-2})/2 \rceil & \text{if } 2^{n-1} < N \le \frac{3}{2}\, 2^{n-1} \\ N - 2^{n-1} & \text{if } \frac{3}{2}\, 2^{n-1} \le N \le 2^n \end{cases}$$

then $m(P(N),N) = n$ and $m(P(N)-1,N) = n+1$. Thus $P(N)$ processors are necessary and sufficient to evaluate A_N in n steps. Kogge [72c] examined the broader problem of evaluating A_j, $1 \le j \le N$, in n steps, and also considered constraints on the communication networks between processors. Clearly $P(N)$ processors are still necessary, but the constraints and additional computations make the sufficiency question more difficult, and only partial results are available. One simple result that we will need later is that the powers z^j, $1 \le j \le N$, may be computed in n steps using $\max(2^{n-2}, N-2^{n-1}) \le N/2$ processors.

Another useful observation is that if $P(N) = \lceil N/\log N \rceil$ then $m(P(N),N) < 2\log N$ and $S_P(N) > P/2$. Thus we can reduce the number of processors somewhat and obtain both a logarithmic running time and linear speedup.

The error analysis for summations and products using the associative fan-in algorithm is not difficult. If h is the height of the computation tree then it may be shown that

$$fl\left(\sum_{i=1}^{N} a_i\right) = \sum_{i=1}^{N} a_i(1+\xi_i), \quad |\xi_i| \le h\mu,$$

$$fl\left(\prod_{i=1}^{N} a_i\right) = a_1 \prod_{i=2}^{N} a_i(1+\epsilon_i), \quad |\epsilon_i| \le \mu,$$

where μ is a small constant depending on the floating point number system (Babuska [68], Kogge [72b]). Babuska (also Viten'ko [68]) has observed that the fully branched log-sum algorithm ($h = \lceil \log N \rceil$) yields the best stability bounds of any algorithm using N-1 additions. This does not preclude use of pseudo-double precision summation methods comparable to the ones proposed for sequential computation (e.g., Kahan [71]), since these use more than N-1 additions.

Finally, lower bounds for pipeline computers may be compared to lower bounds for parallel computers with a fixed number of processors, for in each case the computation time is bounded below by a <u>linear</u> function of the number of operations actually performed. Indeed, we can estimate the time for a parallel vector operation as $tN/P + s$, where s includes overhead and the "wrap-up" time $t \log P/2$ for operations with a single result. This is in sharp contrast to the "sufficiently many processors" model, where vector additions

can be done in a single step and summations in a logarithmic number of steps. It is important to be aware of the differences between a fixed and unlimited number of processors, and that algorithms executed under the two models will have quite different operating characteristics.

We observe that, for the limited model of arithmetic time and as an intuitive guide only, a good algorithm for a pipeline computer should also be a good algorithm for a SIMD computer with fixed parallelism, and vice versa. One of the flaws of this comparison is the relative importance of lower order terms, for σ will typically be much larger than s, and we have to be careful in discussing algorithms that are expected to be fast "in the limit", the definition of which can change considerably.

3. BASIC ALGORITHMS

Thus far we have established that certain simple but useful computations can be performed optimally in parallel. In fact, arbitrary arithmetic expressions with N terms can be evaluated in O(m(P,N)) steps under the P processor MIMD model. Much effort has gone into improving the asymptotic constants and in consideration of special expressions such as polynomials and recurrences. Important cases for linear algebra are the vector inner product, matrix multiplication, linear combinations of vectors, vector norms, linear recurrences, and the fast Fourier transform. The results of this section mostly apply to parallel computers, for no general theory presently exists concerning the optimal evaluation of arbitrary expressions on a pipeline computer.

3.a. General Expressions and Recurrences

In deriving the associative fan-in algorithms we were able to use the associativity of o in order to reduce the depth of the computation tree representing the sequential algorithm $A_1 = a_1$, $A_{i+1} = A_i \circ a_{i+1}$. The idea of restructuring a sequential computation has been successfully extended to general arithmetic expressions; Brent [73] summarizes early work and basic techniques. Since an upper bound is desired, it may be assumed that the expression's directed computation graph is actually a tree. Constants are treated as indeterminates and all indeterminates must be distinct. Tree manipulations with the flavor of recursive doubling are then applied, reducing the tree depth and hence the computation time. Under the MIMD model, Table 2 shows recent results for an expression with N indeterminates. The P processor cases are derived using the algorithm decomposition principle.

TABLE 2
Parallel Computation Times

Expressions with Division

# of processors	# of steps	source
$O(N^{1.44})$	$2.88\log N + O(1)$	Muller and Preparata [75]
$3N$	$4\log N + O(1)$	Brent [73]
P	$5N/2P + O(\log^2 N)$	Winograd [75]

Expressions without Division

# of processors	# of steps	source
$O(N^{1.817})$	$2.08\log N + O(1)$	Preparata and Muller [75]
N	$4\log N + O(1)$	Brent [73]
P	$3N/2P + O(\log^2 N)$	Winograd [75]

Brent's algorithm for expressions without division is numerically stable, and an example due to W. Miller (Brent [74]) shows that the corresponding algorithm for expressions with division can be unstable. The stability of the other evaluation techniques has not yet been investigated. Hyafil and Kung [74a] show that in Winograd's scheme (no divisions) the constant term 3/2 cannot be decreased by much. This is done by considering the first order linear recurrence $x_0 = a_0$, $x_i = b_i x_{i-1} + a_i$, and showing that any algorithm to evaluate x_n in t steps must perform at least $3n-t/2$ operations. This result quantifies part of the folk wisdom of parallel computation, that there is a tradeoff between speed and the amount of work actually performed. Using the fact that we must have $Pt \geq 3n-t/2$ (otherwise t steps could not be sufficient) we conclude that $t \geq 3n/(P+1/2)$. A slightly better result can be obtained from the Munro-Paterson theorem, as $t \geq m(P,3n-t/2+1)$ implies $t \geq (3n + P(\lceil \log P \rceil - 2) + 1)/(P + 1/2)$. The substitution $N = 2n+1$ gives the desired result.

Investigation of linear recurrences began with two different applications, polynomial evaluation and the solution of tridiagonal linear systems.

Horner's rule, which uniquely minimizes the sequential computation time (Borodin [71]), is a familiar example of a first order linear recurrence, and was one of the first methods to be reconstructed for parallel computation. The situation is now quite well understood, and Munro and Paterson [73] have described some asymptotically optimal algorithms. With N processors an Nth degree polynomial can be evaluated in $\log N + (2\log N)^{1/2} + O(1)$ steps, and with P processors in $m(P,2N+1) + O(1)$ steps.

Stone [73a] observed that an N×N tridiagonal linear system can be solved in O(logN) steps if the first N terms of first and second order linear recurrences can be evaluated in O(logN) steps using N processors. This is discussed in more detail in Section 4.c, and an algorithm to compute m^{th} order linear recurrences ($x_{-m+1}, \ldots, x_0$ given, $x_i = \sum_{j=1}^{m} a_{ij}x_{i-j} + b_i$, $i \geq 1$) is given in Section 4.b, although this is slightly different than the recursive doubling techniques used by Stone.

A series of papers by Kogge and Stone (Kogge [72a,b,c], [74], Kogge and Stone [73], cf. Trout [72] and Heller [74b]) dealt with various generalizations and improvements of Stone's original recurrence methods, although linear-like properties are still required. For example, if x_0 is given and $x_{i+1} = \frac{a_i x_i + b_i}{c_i x_i + d_i}$, then x_j, $1 \leq j \leq N$, may be computed in O(logN) steps with N processors. Work by Kung [74] and Hyafil and Kung [75] shows that the linear recurrence is really very special, and that only a constant speedup is possible for general rational and nonlinear recurrences. That is, there can be no fast algorithms, regardless of how many processors are available.

We note that, although the computation of N terms of a low order linear recurrence on a pipeline computer generally requires use of the scalar mode

because of short vector lengths, Kogge [73] shows how to adapt results from the parallel computation of recurrences to the design of special purpose pipelines. By careful use of parallel computation networks and feedback loops, the maximal output rate of one result per cycle may be achieved.

3.b. Inner Products and Related Computations

One of the most important applications of the associative fan-in algorithm is the computation of inner products. Winograd [70] has shown that N multiplications and N-1 additions are required to compute $a^Tb = \sum_{i=1}^{N} a_ib_i$, so with P processors it is necessary to use at least m(P,2N) steps. This lower bound is achievable on an MIMD machine by a slight alteration of the fan-in method. On a SIMD machine $\lceil N/P \rceil + m(P,N)$ steps are needed, since we cannot perform different operations in the same step. In either case, a^Tb requires $\lceil \log N \rceil + 1$ steps given N processors. As before, the error analysis is not difficult, for

$$fl\left(\sum_{i=1}^{N} a_ib_i\right) = \sum_{i=1}^{N} a_ib_i(1+\xi_i), \ |\xi_i| \le h\mu,$$

where $h \le m(P,2N)+1$ is the height of the computation tree used. Double precision accumulation is possible as in sequential computation.

In considering the N processor evaluation of a^Tb it is important to observe that while half of the operations are multiplications, almost all of the computational steps involve performing the additions. In sequential computation it is often possible to estimate an algorithm's performance solely by counting multiplications, but this is definitely not the case for parallel computation. All operations must be counted.

On a vector computer the inner product is generally included as a hardware instruction taking time $\tau_{dot}N + \sigma_{dot}$, where $\tau_{dot} < \tau_x + \tau_{sum}$. Even if $\tau_{dot} = \tau_x + \tau_{sum}$ it is better to use the hardware inner product since it will save one vector startup time, which is not insignificant.

The vector p-norms can also be computed in parallel using the associative fan-in algorithm. Here $\|v\|_p = \left(\sum_{i=1}^{N} |v_i|^p\right)^{\frac{1}{p}}$, so for $1 \le p < \infty$ we require $\lceil N/P \rceil$ unary steps to compute $|v_i|^p$, $m(P,N)$ addition steps for the summation, and one step for $\|v\|_p$. $\|v\|_\infty = \max|v_i|$ is directly computable in $\lceil N/P \rceil + m(P,N)$ steps since the maximum operation is associative. The most frequent cases are $p = 1$, 2, and ∞, and on a pipeline machine these may be computed using the absolute value, summation, inner product and maximum vector instructions. Similarly, the 1 and ∞ matrix norms are easily computable. In actual practice the choice of norm and computational method will be greatly affected by the particular application and computer, and the issues involved in this choice are far from being resolved.

An inner product is, of course, a special case of matrix multiplication, being the product of $1 \times N$ and $N \times 1$ matrices. More generally, the product of $m \times n$ and $n \times p$ matrices may be optimally computed in $\lceil \log n \rceil + 1$ steps with mnp processors, since each component of the $m \times p$ result is an inner product of n-vectors. If fewer than mnp processors are available then asymptotically fast algorithms may be developed using tradeoffs between the fast sequential algorithms (Strassen [69]) and the usual sequential algorithm for partitioned matrices. For example, if A and B are $N \times N$ and we have a fast sequential method using cN^α steps, $\alpha = \log 7 = 2.81$, then with 8 processors $C = AB$ may be computed in $cN^\alpha/7 + N^2/8$ steps. If A_{11} and B_{11} are $\lceil N/2 \rceil \times \lceil N/2 \rceil$,

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \; B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \text{ then}$$

$$C = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}.$$

Each $A_{ik}B_{kj}$ matrix product is computed in one processor in $c(N/2)^{\alpha}$ steps, and the four matrix additions are each done with two processors in $(N/2)^2/2$ steps. Thus the speedup is $S_8(N) = 7 - O(N^{-.81})$, which is satisfactory.

A matrix-vector multiplication $c = Ab$ is equivalent to the formation of a linear combination of the columns of A, and here we use the fact that the computations for the components of c are independent. Thus

$$c_i \leftarrow \sum_{j=1}^{n} a_{ij}b_j, \; (1 \le i \le N)$$

if A is $N \times n$. Special cases of importance are $n = 2$, 3 or N, and it is seen that good speedups are obtainable for any values of N, n and P. The choice of a particular method for a pipeline computer would depend on the storage scheme used for A and the relative costs of an inner product and a vector multiply-and-add.

Muraoka and Kuck [73] have considered the evaluation of a conformable sequence of matrix products $A_1A_2 \ldots A_n$, where A_i is either $1 \times N$, $N \times N$, or $N \times 1$, using unlimited parallelism. It is necessary to associate the products correctly in order to minimize the computation time, and a minimal weight parsing algorithm is given.

The parallel evaluation of arbitrary matrix expressions is discussed by Maruyama [73] and Kuck and Maruyama [75], who generalize earlier results for scalar expressions. If, with unlimited parallelism, $N \times N$ matrices may be added, multiplied and inverted in t_A, t_M and t_I steps respectively, then any matrix expression involving n $N \times N$ matrices and no inversions may be evaluated in $2\lceil \log n \rceil (t_A + t_M)$ steps. If inversions are necessary then $\lceil \log n \rceil (2t_A + 3t_M + t_I) - t_M + t_I$ steps are sufficient.

3.c. The Fast Fourier Transform

The discrete Fourier transform of an N-vector $a = (a_0, \ldots, a_{N-1})$ is another N-vector b, where

$$b_i = \sum_{j=0}^{N-1} \omega^{ij} a_j, \quad 0 \le i \le N-1,$$

and ω is the principal N^{th} root of unity. We assume for this section only that all arithmetic is done with complex numbers. The transform is just a matrix-vector multiplication $b = Fa$, so with N^2 processors we only need to generate F ($\lceil \log N \rceil$ steps given ω) and perform the multiplication ($\lceil \log N \rceil + 1$ steps). The fast Fourier transform (Cooley and Tukey [65]) allows us to compute b just as quickly but only using N processors, or in $O(N\log N/P)$ steps with P processors (Pease [68]).

For simplicity assume that $N = 2^{n+1}$, and for $0 \le r \le N-1$, $0 \le k \le n$, let

$$r = [r_0 r_1 \ldots r_n] = \sum_{j=0}^{n} r_{n-j} 2^j, \quad r_i = 0 \text{ or } 1,$$

$$f(r,k) = [r_0 \ldots r_{k-1} 0 r_{k+1} \ldots r_n],$$
$$g(r,k) = [r_k r_{k-1} \ldots r_0 0 \ldots 0],$$
$$h(r,k) = [r_0 \ldots r_{k-1} 1 r_{k+1} \ldots r_n],$$
$$rev(r) = [r_n \ldots r_0] = g(r,n).$$

The parallel FFT is performed as follows:

$z_i \leftarrow \omega^i$, $(0 \le i \le N-1)$;

$c_i \leftarrow a_i$, $(0 \le i \le N-1)$;

<u>for</u> k = 0 <u>step</u> 1 <u>until</u> n <u>do</u>

$\quad c_i \leftarrow c_{f(i,k)} + z_{g(i,k)} c_{h(i,k)}$, $(0 \le i \le N-1)$;

$b_i \leftarrow c_{rev(i)}$, $(0 \le i \le N-1)$.

Except for the initial computation of z_i, which may be done in a variety of ways, the algorithm runs in $2\lceil N/P \rceil \lceil \log N \rceil$ complex arithmetic steps with P processors. In essence, F has been factored into $\lceil \log N \rceil$ very simple matrices, and the cumulative product is computed. However, the interprocessor data movements are just as important as the arithmetic costs, and it may be observed that either $f(r,k) = r$ and $h(r,k) = r + 2^{n-k}$ or $f(r,k) = r - 2^{n-k}$ and $h(r,k) = r$, so the movements for c within the loop are well structured (Pease [68], Stone [71]).

4. LINEAR SYSTEMS

In this section we consider the parallel solution of arbitrary linear systems and some special systems of practical interest; most of the methods are direct rather than iterative. Lambiotte [75] covers many of these topics with respect to the CDC STAR, and there are a large number of technical reports dealing with implementation on Illiac IV (see Poole and Voigt [74]). Throughout this section A will denote a real nonsingular matrix, and we will attempt to solve $Ax = v$, $v \in R^N$, and $AX = B$, $B \in R^{N \times M}$.

4.a. General Dense Matrices

If $x = A^{-1}v$ then each component of x depends on the N^2 components of A and the N components of v, so at least $m(P,N^2+N)$ steps are required to compute x with P processors. This number is about $2\log N$ for large P. Supposing that the parallel processor uses w-bit floating point arithmetic, $w < \infty$, let S be the set of all N-vectors such that $z = fl(z)$; there are at most $2^w N$ vectors in S. Now, it may be shown that if $y \in S$ satisfies

$$\|Ay-v\|_\infty = \min_{z \in S} \|Az-v\|_\infty.$$

then $(A+E)y = v$, where $\|E\|_\infty$ is small. Using $N^2|S|$ processors we can compute $\|Az-v\|_\infty$, for each $z \in S$, in $1 + \lceil\log(N+1)\rceil + \lceil\log N\rceil$ steps. The minimization over S may be done in $\lceil\log|S|\rceil$ steps, for a total of at most $3\log N + w + 4$ steps. Nevertheless, we must reject this approach, for it requires an excessive number of processors and operations when we regard w as a problem parameter, hence giving no useful information for an actual parallel program, and also fails to give a clue for the treatment of real numbers ($w = \infty$).

Let $T(P,N,M)$ be the minimum number of steps required to compute $A^{-1}B$ using P processors; the trivial lower bound $T(P,N,1) \geq m(P,N^2+N)$ has just been mentioned. It is clear that it is no more difficult to compute $A^{-1}B$ than $A^{-1}v$; simply compute, in parallel, $A^{-1}B_j$ where B_j is the jth column of B. That is, $T(MP,N,M) = T(P,N,1)$. It is also clear that T is an increasing function of N for any value of P. No one has yet been able to advance much beyond this lower bound for general parallel algorithms.

For some time it was believed that the Gauss-Jordan elimination algorithm would prove to be the fastest way to compute $A^{-1}v$ given any number of processors. The running time of this method is linear in N when pivoting is not used, and a major open problem was to close the gap between the logarithmic lower bounds and the linear upper bounds for $T(P,N,1)$. In fact it is not hard to give a plausibility argument (but not a proof!) that at least N steps are needed for any recursive algorithm such as elimination. We now know that A^{-1} may be computed in $O(\log^2 N)$ steps using a bounded number of processors (Csanky [75]). All the mathematical tools involved in this result are classical, but its importance is drawn from the modern notions of sequential and parallel computation.

Suppose that the eigenvalues of A are α_j, $1 \leq j \leq N$, and its characteristic polynomial is

$$f(z) = \prod_{j=1}^{N} (z-\alpha_j) = \det(zI-A)$$

$$= z^N + c_1 z^{N-1} + \ldots + c_{N-1}z + c_N.$$

Let $s_i = \mathrm{trace}(A^i) = \sum_{j=1}^{N} \alpha_j^i$; by the Newton identities relating the zeros and

coefficients of a polynomial we have the triangular system

$$\begin{pmatrix} 1 & & & & & \\ s_1 & 2 & & & & \\ s_2 & s_1 & 3 & & & \\ s_3 & s_2 & s_1 & 4 & & \\ \vdots & & & & \ddots & \\ s_{N-1} & \cdots & & & s_1 & N \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} -s_1 \\ -s_2 \\ -s_3 \\ -s_4 \\ \vdots \\ -s_N \end{pmatrix} .$$

By the Cayley-Hamilton Theorem $f(A) = 0$, so we can write

$$A^{-1} = -(A^{N-1} + c_1 A^{N-2} + \ldots + c_{N-2}A + c_{N-1}I)/c_N.$$

Note that $c_N = (-1)^N \det(A) \neq 0$ since A is nonsingular. The method is to compute all the powers of A and the traces s_i, solve for the coefficients of the characteristic polynomial, and finally use the powers and coefficients to evaluate A^{-1}.

To determine the running time, recall that z^i, $1 \le i \le N$, can be computed in $n = \lceil \log N \rceil$ steps with $N/2$ processors. This is easily extended to show that A^i, $1 \le i \le N$, can be computed in $n(n+1)$ steps with $N(N^3)/2$ processors. Since $\mathrm{trace}(A) = \sum_{j=1}^{N} a_{jj}$, we can compute s_i, $1 \le i \le N$, in n steps using N^2 processors. Later, in Section 4.b, we will show that a triangular linear system can be solved in $(n+1)(n+2)/2$ steps using $N^3/68 + O(N^2)$ processors. (This particular result is due to Chen and Kuck [75].) Finally, A^{-1} may be computed in n+2 steps using N^3 processors. Since $x = A^{-1}v$ requires only n+1 more steps with N^2 processors, we have

$$2\log N < T(N^4/2,N,1) \le (3/2)\log^2 N + O(\log N).$$

The number of processors required can be reduced somewhat while preserving the $O(\log^2 N)$ time by computing inner products in $O(\log N)$ steps using $O(N/\log N)$ processors. Thus if $P = O(N^4/\log N)$, $T(P,N,1) = O(\log^2 N)$. Note also that if A is $M \times N$, $M \geq N$, and A has full column rank, then we can compute $A^\dagger$, the generalized inverse of A, since $A^T A$ is positive definite and $A^\dagger = (A^T A)^{-1} A$.

As for stability, the evaluation of c_N is extremely sensitive to rounding errors committed in the evaluation of the traces s_i. In many cases severe cancellation will occur (see the discussion of Leverrier's method in Wilkinson [65]), and a very large number of figures must be carried in order to obtain a reasonable computed value of A^{-1}.

In summary, we have an excellent theoretical result, but it will be of little help in creating programs for real parallel computers. For this we must return to the standard elimination methods, which are known to be stable and which have enough inherent parallelism to allow efficient execution on parallel and pipeline computers. We suspect that there is a conservation law for linear systems, which states that if a stability criterion is to be met then a certain number of arithmetic steps must be performed. We believe that techniques developed by W. Miller [75] are worth pursuing for parallel computation.

We discuss four elimination methods: Gauss-Jordan, the LU and QR decompositions, and a method due to Pease [74]. Each has different merits according to the computing model and characteristics of A. In each case we identify B with columns N+1 through N+M of A, and compute $X = A^{-1}B$ by performing operations on whole rows of the augmented matrix. For notational convenience "row i" will refer to the ith row of (A,B).

The Gauss-Jordan algorithm is the simplest to describe. Assuming pivoting is not necessary for stability, we have

<u>Algorithm G-J</u>:

<u>for</u> j = 1 <u>step</u> 1 <u>until</u> N <u>do</u>

row i ← row i - (a_{ij}/a_{jj})row j, $(1 \le i \le N,\ i \ne j)$;

$x_{ij} \leftarrow a_{ij}/a_{ii}$, $(1 \le i \le N;\ N+1 \le j \le N+M)$.

Using (N-1)(N+M) processors this requires 3N+1 steps. Note that A^{-1} can be computed in parallel with $A^{-1}B$, so that $Y = A^{-1}C$ is later obtainable by a matrix multiplication.

If $a_{jj} = 0$ at some point in Algorithm G-J, then it is sufficient (mathematically at least) to search column j below the diagonal to discover a non-zero pivot before performing the elimination step. This adds an additional number of comparison steps equal to

$$\sum_{j=1}^{N-1} \lceil \log(N-j) \rceil = Nn - 2^n - n + 1,\ n = \lceil \log(N-1) \rceil .$$

Thus the effort expended in partial pivoting can overwhelm the arithmetic effort, in contrast to the single processor case where pivoting does not radically change the running time.

Sameh and Kuck [75b] describe one way to overcome the pivoting problem. Using the square-root-free Givens transformations (Gentleman [73]), a diagonal matrix D, an upper triangular matrix R and an orthogonal matrix Q, such that $QA = D^{1/2}R$, can be obtained in 8N-7 steps (only two of them involving square roots) with N^2 processors. Q is computed implicitly as a

product of simpler matrices. For notational purposes we use a procedure Rotate(i,j) which applies a root-free Givens transformation to rows i-1 and i in order to eliminate a_{ij}, where $1 \le j < i \le N$. Rotate takes care of managing the scale factors in D, and the rows are interchanged if necessary for stability. The triangularization algorithm is thus

```
for k = 1 step 1 until N-1 do
   begin Rotate(N-2p,k-p), (0 ≤ p ≤ min(k-1,N-k-1));
         Rotate(N-2p-1,k-p), (0 ≤ p ≤ min(k-1,N-k-2))
   end.
```

Once A has been reduced to triangular form, any of the methods of Section 4.b can be used to solve the resulting triangular system.

If only N processors are available (cf. Pease [67]) the Gauss-Jordan algorithm without pivoting uses $N^2 + 2NM + M$ arithmetic steps. Partial pivoting is not costly since O(NlogN) comparison steps still suffice.

<u>Algorithm G-J (N processors)</u>:

```
for j = 1 step 1 until N do
   begin t_i ← a_ij/a_jj, (1 ≤ i ≤ N, i ≠ j);
      for k = j+1 step 1 until N+M do
         a_ik ← a_ik - t_i a_jk, (1 ≤ i ≤ N, i ≠ j)
   end;
for j = N+1 step 1 until N+M do
   x_ij ← a_ij/a_ii, (1 ≤ i ≤ N).
```

This is, of course, only one of many possibilities. It has the advantage that the only data movements needed are to spread a column across the processors and to broadcast one number to all processors. The algorithm decomposition modification to P processors is easily done.

For a pipeline computer it is better to use Gaussian elimination, as this minimizes the number of arithmetics when we are limited to operations on whole rows and columns (Kluyuev and Kokovkin-Shcherbak [65]). The method due to Strassen [69], which uses asymptotically fewer operations by performing block elimination, requires some complicated addressing schemes that may be quite difficult to implement. Because of the restrictive definition of vector operands on the STAR, it is necessary to provide two sets of Gauss elimination routines, depending on whether A is stored by rows or columns (Lambiotte [75]). In addition, the symmetric factorization $A = LDL^T$ when A is positive definite requires both routines, for if L is stored by rows (columns) then L^T is stored by columns (rows).

The preceding elimination algorithms are familiar ones from sequential computation, and we have really only given parallel implementations of them. Pease [74] recently presented a new algorithm for the solution of a general system (cf. Pease [69]). Although the method would require $O(N^2 \log N)$ steps using N processors (as opposed to Gauss-Jordan's $O(N^2)$ steps), it is interesting in that any parallel computer with an interprocessor communication network designed for the FFT can implement it reasonably well. In particular, special purpose FFT devices could be modified to support the algorithm, and it is expected that this will be the primary application; it is not recommended for other situations.

We give a simple recursive definition of Pease's method, though this does not illuminate relations with the FFT. Like other elimination algorithms it works by premultiplying the augmented matrix. Suppose for simplicity that $N = 2^n$.

Algorithm P(n):

<u>if</u> $n = 0$ <u>then</u> $x \leftarrow A^{-1}v$ <u>else</u>

<u>begin</u>

$$\text{let } A = \begin{pmatrix} A_1 & E_1 \\ E_2 & A_2 \end{pmatrix},\ x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix},\ v = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix},$$

where A_1 and A_2 are $2^{n-1} \times 2^{n-1}$;

simultaneously solve $A_1(F_1, g_1) = (E_1, v_1)$

and $A_2(F_2, g_2) = (E_2, v_2)$

by Algorithm P(n-1);

simultaneously solve $(I - F_1F_2)\ x_1 = (g_1 - F_1g_2)$,

and $(I - F_2F_1)x_2 = (g_2 - F_2g_1)$

by Algorithm P(n-1)

<u>end</u>.

In the first application of Algorithm P(n-1), $Ax = v$ is transformed into

$$\begin{pmatrix} I & F_1 \\ F_2 & I \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} g_1 \\ g_2 \end{pmatrix}.$$

This system is premultiplied by

$$\begin{pmatrix} I & -F_1 \\ -F_2 & I \end{pmatrix}$$

to form

$$\begin{pmatrix} I - F_1F_2 & 0 \\ 0 & I - F_2F_1 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} g_1 - F_1g_2 \\ g_2 - F_2g_1 \end{pmatrix}.$$

The last step is simply the cyclic reduction method originally suggested for implicit use with large scale iterative methods (Varga [62]). The numerical stability of Algorithm P has not yet been discussed thoroughly; a manageable pivoting scheme must be given with a corresponding error analysis.

4.b. Triangular Systems

We have seen that the solution of dense systems requires the ability to solve more specialized systems quickly. Suppose now that A is a nonsingular triangular matrix; without loss of generality we can assume A is lower triangular. It is easily verified that the straightforward sequential solution of $Ax = v$ requires N^2 arithmetic operations. The parallel solution of $Ax = v$ was first considered by Heller [74a], who showed that x can be computed in $O(\log^2 N)$ steps with $O(N^4)$ processors. The original method was rather complicated, involving the evaluation of lower Hessenberg determinants by recursive doubling, and the result has since been improved by reducing the processor requirement to $O(N^3)$. It is of some interest to consider three distinct but similar algorithms for this problem. These algorithms succeed by use of recursion and doubling, in that the number of completed computations (i.e., components of x) doubles at each stage.

The first method (Chen and Kuck [75]), is a variation of Gauss-Jordan elimination, and might be called elimination by diagonals. "Row i" again refers to a row of the augmented matrix, where for notational simplicity we set $a_{ij} = 0$ if $i \le 0$ or $j \le 0$.

<u>Algorithm C-K</u>:

row i $\leftarrow$ (row i)/a_{ii}, $(1 \le i \le N)$;

<u>for</u> j = 1 <u>step</u> j <u>until</u> N-1 <u>do</u>

$$\text{row } i \leftarrow \text{row } i - \sum_{k=j}^{2j-1} a_{i,i-k}\text{row } i\text{-}k, \quad (j+1 \le i \le N);$$

$x_i \leftarrow a_{i,N+1}$, $(1 \le i \le N)$.

Using less than $N^2(N+1)/2$ processors, for each j we can do all the multiplications in the loop in parallel, followed by the log-sum addition of j+1 rows. The time is thus, for $n = \lceil \log N \rceil$,

$$1 + \sum_{k=0}^{n-1} 1 + \lceil \log(2^k+1) \rceil = (n^2+3n+2)/2 = O(\log^2 N).$$

By a closer analysis of the number of processors needed when only essential operations are performed, $N^3/68 + O(N^2)$ processors suffice. Chen [75] shows that if A is a Toeplitz matrix $(a_{ij} = \beta_{i-j})$ then $N^2/4$ processors are sufficient. It is also seen that AX = B can be solved in $O(\log^2 N)$ steps with no more than $N^2(N+M)$ processors.

The second method (Borodin and Munro [75]) uses partitioning to invert A. Let

$$A = \begin{pmatrix} A_1 & 0 \\ A_2 & A_3 \end{pmatrix}$$

where A_1 and A_3 are lower triangular and A_1 is m $\times$ m, so that

$$A^{-1} = \begin{pmatrix} A_1^{-1} & 0 \\ -A_3^{-1}A_2A_1^{-1} & A_3^{-1} \end{pmatrix}.$$

The algorithm proceeds in two stages, first simultaneously inverting A_1 and A_3 and solving $A_3Y = A_2$, and then multiplying Y and A_1^{-1}. The total time is

$$t(1) = 1,$$
$$t(N) = \min_{1 \le m < N} (\max(t(m), t(N-m)) + 1 + \lceil \log m \rceil)$$

using $O(N^3)$ processors. The proper choice for m is $\lceil N/2 \rceil$; thus $t(2^n) = (n^2+n)/2$ and in general $t(N) = O(\log^2 N)$.

A third method was obtained independently by Heller [74b] and Orcutt [74]. Supposing that A has a unit diagonal, let $A = I - L$, where L is strictly lower triangular. Since $L^N = 0$

$$x = A^{-1}v = (I + L + L^2 + \dots + L^{N-1})v$$
$$= (I + L^{2^{n-1}})(I + L^{2^{n-2}}) \dots (I + L)v$$

where again $n = \lceil \log N \rceil$. x is computed by repeatedly squaring L and accumulating matrix-vector products according to the above formula. The time required is at most $n^2 + n$ steps using at most $N^2(N+1)$ processors.

This technique also provides a simple algorithm for linear recurrences. Suppose that $a_{ij} = 0$ for $i-j > m$, so the system $Ax = v$ represents an mth order linear recurrence with initial values. We assume that $m \ll N$. Partition A into $m \times m$ blocks A_{ij}, $1 \le i, j \le n = \lceil N/m \rceil$ (A_{nn} may be smaller than $m \times m$); there are only two block diagonals that are nonzero, namely A_{ii} and $A_{i,i-1}$. Now consider

$$A^* = \mathrm{diag}(A_{11}^{-1},\ldots,A_{nn}^{-1})A,$$
$$v^* = \mathrm{diag}(A_{11}^{-1},\ldots,A_{nn}^{-1})v,$$

and note that $A_{ii}^* = I$, $A_{i,i-1}^* = A_{ii}^{-1}A_{i,i-1}$. Using $2m^2N$ processors, A^* and v^* may be computed in $O(\log^2 m)$ steps since A_{ii} is triangular. Moreover, powers of $L = I - A^*$ all have a single block diagonal. To compute $L^{2^{i+1}}$ from L^{2^i} requires the parallel computation of $n\text{-}2^{i+1}$ $m \times m$ matrix products, which may be done in $1 + \lceil \log m \rceil$ steps. Collecting these results, x (the first N terms of the recurrence) may be computed in $O(\log m \log N)$ steps. It is not hard to show that $O(m^2 \log N)$ steps suffice with N processors.

We next apply algorithm and problem decomposition in order to reduce the processor requirements for triangular systems below N^3 and still obtain fast algorithms. These results are due to Hyafil and Kung [74b]; similar ideas appear in Chen [75].

First consider the following scheme to solve $Ax = v$.

$x_i \leftarrow v_i$, $(1 \le i \le N)$;

<u>for</u> j = 1 <u>step</u> 1 <u>until</u> N <u>do</u>

<u>begin</u> $x_j \leftarrow x_j/a_{jj}$;

$x_i \leftarrow x_i - a_{ij}x_j$, $(j+1 \le i \le N)$

<u>end</u>.

Using $P \le N$ processors this requires

$$\sum_{j=1}^{N} 1 + \lceil (N-j)/P \rceil < N^2/P + 2N$$

steps. Using vector instructions on a pipeline machine, the arithmetic cost would be

$$(\tau_+ + \tau_x)N^2/2 + (t_\div - \tau_+/2 - \tau_x/2 + \sigma_+ + \sigma_x)N.$$

These are both acceptable speedups, since the scalar time would be

$$(t_+ + t_x)N^2/2 + (t_\div - t_+/2 - t_x/2)N.$$

Now suppose that $P = N^r$ processors are available, $1 < r < 3$. Let $m = \lfloor P^{1/3} \rfloor$ and partition A into $m \times m$ blocks A_{ij}, $1 \le i, j \le n = \lceil N/m \rceil$ (A_{nn} may be smaller than $m \times m$). Similarly partition x and v, and apply the following algorithm:

$x_i \leftarrow v_i$, $(1 \le i \le n)$;

<u>for</u> $j = 1$ <u>step</u> 1 <u>until</u> n <u>do</u>

<u>begin</u> $x_j \leftarrow A_{jj}^{-1} x_j$;

$x_i \leftarrow x_i - A_{ij} x_j$, $(j+1 \le i \le n)$

<u>end</u>.

The computation of $A_{jj}^{-1} x_j$ is done using the fast algorithm, since m^3 processors are available. The total time required is, for $k = \lceil \log m \rceil$,

$$\sum_{j=1}^{n} (k^2+3k+2)/2 + \lceil (n-j)/m \rceil (k+2)$$

$$= O(nk^2 + n^2k/m) = O(N^{1-r/3}\log^2 N + N^{2-r}\log N).$$

If $3/2 \le r < 3$ the first term dominates, and if $1 < r < 3/2$ the second term dominates. Thus there is a tradeoff between the use of the fast algorithm and the cost of combining the results of its application.

4.c. Tridiagonal Systems

Stone [73a] first discussed the solution of a tridiagonal system on a parallel computer, relating the LDU decomposition of A to first and second order linear recurrences. The fact that these recurrences are relevant was not new, but Stone developed recursive doubling algorithms to compute the necessary terms in O(logN) steps with N processors. Thus it is possible to solve tridiagonal systems in nearly minimal time, since at least m(N,4N-2) (about logN) steps are necessary.

For notational convenience we write a tridiagonal matrix as a triple of vectors, so

$$A = \begin{pmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & & \\ & a_3 & b_3 & c_3 & & \\ & & \cdot & \cdot & \cdot & \\ & & & \cdot & \cdot & \cdot \\ & & & \cdot & \cdot & \cdot \\ & & & a_{N-1} & b_{N-1} & c_{N-1} \\ & & & & a_N & b_N \end{pmatrix} = (a_j, b_j, c_j)$$

where we assume $a_1 = c_N = 0$. The LDU factorization expresses

$$A = LDU = (l_j,1,0)(0,d_j,0)(0,1,u_j)$$

and it is easily verified that

$$\begin{aligned} d_1 &= b_1, \\ d_j &= b_j - a_j c_{j-1}/d_{j-1}, \quad 2 \le j \le N, \\ l_j &= a_j/d_{j-1}, \quad 2 \le j \le N, \\ u_j &= c_j/d_j, \quad 1 \le j \le N. \end{aligned}$$

$Ax = v$ is then solved by solving $Lw = v$ and $Ux = D^{-1}w$. The bidiagonal systems represent first order linear recurrences, and $D^{-1}w$ is computable in a single parallel division step using N processors. Since L and U are completely determined by D, all that remains is a fast method of computing D. This is found by defining $p_0 = 1$, $p_1 = b_1$, $p_j = b_j p_{j-1} - a_j c_{j-1} p_{j-2}$ and observing that $d_j = p_j/p_{j-1}$. Thus the parallel computation of a second order recurrence completes the algorithm.

Unfortunately this method will fail if pivoting is necessary in the sequential factorization. One way to avoid this problem is to consider the QR factorization of A, where Q is formed from the product of N-1 Givens rotations and R is upper triangular with $r_{ij} = 0$ if $j-i > 2$. Sameh and Kuck [75b] show that by use of linear recurrences Q and R may be computed in O(logN) steps with N processors, so A may be solved in O(logN) steps. We delay the presentation of the details of this method until Section 5, where the QR algorithm for the eigenvalues of a symmetric matrix is discussed.

The methods of odd-even elimination and reduction are another class of parallel algorithms with some quite different characteristics. We first discuss odd-even elimination. In keeping with previous algorithms, we describe the method in terms of row operations on the augmented matrix, although a computer implementation would be in terms of the vectors describing A and v. It is essential that pivoting not be used, so there are some restrictions on the application of the algorithm. For notational convenience, let $a_{ij} = \delta_{ij}$ for indices outside the ranges $1 \le i \le N$, $1 \le j \le N+1$.

<u>Algorithm E</u>:

<u>for</u> $k = 1$ <u>step</u> k <u>until</u> $N-1$ <u>do</u>

$$\text{row } i \leftarrow \text{row } i - a_{i,i-k}(\text{row } i-k)/a_{i-k,i-k} - a_{i,i+k}(\text{row } i+k)/a_{i+k,i+k}, \quad (1 \le i \le N);$$

$$x_i \leftarrow a_{i,N+1}/a_{ii}, \quad (1 \le i \le N).$$

This is not the only possible row operation, but others differ only by scaling. The row elimination preserves the fact that A has only three diagonals, but as the algorithm progresses the diagonals move further and further apart, until only a diagonal matrix remains. Thus each execution of the loop body requires 13 steps with N processors, for a total of $O(\log N)$ steps to solve $Ax = v$. In addition, it may be shown that if A is strictly diagonally dominant then the two off-diagonals decrease in magnitude relative to the main diagonal, and the rate of decrease is quadratic (Stone [75a], Jordan [74], Heller [74c]). If an approximate solution is desired it is therefore possible to leave the loop before it is completed and use the computed x as the approximation.

Algorithm E has an important variation, which was actually the original formulation (Hockney [65]), namely odd-even (or cyclic) reduction. It is called reduction because it implicitly generates a sequence of tridiagonal systems $A^{(i)}x^{(i)} = v^{(i)}$, each half the size of the previous system and formed by eliminating the odd-indexed variables and saving the even-indexed variables. $x^{(i)}$ is then obtained by back substituting $x^{(i+1)}$, finally arriving at $x^{(0)}$, which is the solution to the original problem. For simplicity, assume that $N = 2^n-1$, $n \ge 1$, and let $x_0 = x_{N+1} = 0$.

Algorithm R:

for k = 1 step k until N do

$$\text{row } i \leftarrow \text{row } i - a_{i,i-k}(\text{row } i-k)/a_{i-k,i-k} - a_{i,i+k}(\text{row } i+k)/a_{i+k,i+k},$$
$$(i = 2k, 4k, \ldots, 2^n-2k);$$

for $k = 2^{n-1}$ step $-k/2$ until 1 do

$$x_i \leftarrow (a_{i,N+1} - a_{i,i-k}x_{i-k} - a_{i,i+k}x_{i+k})/a_{ii},$$
$$(i = k, 3k, 5k, \ldots, 2^n-k).$$

The reduction algorithm has several advantages over the elimination algorithm, despite the fact that it is slower when N processors are available. The first observation is that it is equivalent to Gaussian elimination (in the usual sequential sense, without pivoting) applied to PAP^T, P a particular permutation matrix. In fact, this is just the nested dissection ordering (Widlund [72], Birkhoff and George [73]). The concept of reordering a system to increase the inherent parallelism of a sequential algorithm is a very important one, as will be seen in later examples. The second observation is that O(N) arithmetic operations are performed, as opposed to O(NlogN) operations in odd-even elimination and in the recurrence modifications of the LU and QR factorizations. For large values of N, the reduction algorithm is therefore preferred for implementation on pipeline and parallel computers with fixed parallelism, where the operation count is as important as the step count. In the terminology of Section 2.b, odd-even reduction is asymptotically consistent, while the other methods are not.

Program timings for the CDC STAR are given by Lambiotte and Voigt [75], comparing odd-even reduction, sequential Gaussian elimination, and a

consistent variation of Stone's method; Stone [75a] gives a different consistent variation. For N smaller than about 100 the sequential algorithm is best, as the effect of vector startups is felt by the other two methods. For N larger than about 100, odd-even reduction is indeed the best method; for very large N (about 2^{16}) it requires about 14% of the computation time for sequential Gaussian elimination, and about 20% of the time for the consistent recursive doubling algorithm. It is also possible to create a polyalgorithm by using odd-even reduction until a reduced system is obtained where the sequential method is faster.

If an approximate solution is satisfactory and previous computations yield good initial approximations an iterative method may be indicated. One obvious technique is to reorder the equations according to the red-black scheme, so

$$PAP^T = \begin{pmatrix} D_1 & A_2 \\ A_2 & D_1 \end{pmatrix}$$

where A_1 and A_2 are bidiagonal and D_1 and D_2 are diagonal. Because of this decoupling, the SOR-type methods have considerable inherent parallelism, and are easily adapted to parallel computation (Lambiotte and Voigt [75]).

A second technique, proposed by Traub [73] and further developed by Heller, Stevenson and Traub [74], changes the sequential LDU factorization into a three-stage iteration. If

$$A = A_L + A_D + A_U,$$
$$A_L = (a_j, 0, 0),$$
$$A_D = (0, b_j, 0),$$
$$A_U = (0, 0, c_j),$$

then $D = A_D - A_L D^{-1} A_U$, so a natural iteration is $D^{(i)} = A_D - A_L (D^{(i-1)})^{-1} A_U$. Once this iteration has converged, L and U are computed and $Lw = v$ and $Ux = D^{-1}w$ are solved using the Jacobi iteration, which is inherently parallel. It may be shown that all three iterations converge linearly under weak conditions on A. Because three stages are used and because inaccuracies in one stage limit the attainable accuracy in later stages it is important to carefully choose the number of iterations used. However, unlike SOR no optimal parameters need to be calculated, so termination is the only difficult issue.

It is possible to alter the three iterations as given in order to improve the anticipated performance on real computers. Suppose first that division is much more expensive than addition or multiplication. On the Illiac IV, $t_+ = 7$, $t_\times = 9$, $t_\div = 56$ cycles (Burroughs [72]). To avoid division, compute $N = D^{-1}$ instead of D and use one step of the Newton iteration for z^{-1}. The new iteration is thus

$$N^{(i)} = N^{(i-1)}(2I - (A_D - A_L N^{(i-1)} A_U) N^{(i-1)})$$

Here $N^{(i-1)}$ is taken to be an approximation to $(A_D - A_L N^{(i-1)} A_U)^{-1}$. Convergence of the N iteration is slightly slower than the D iteration initially, but improves as more accuracy is obtained. This variation may be useful if $t_\div > 3t_\times + t_+$, as is the case for Illiac IV.

For pipeline computers a different approach is suggested. If M iterations are used in the first stage, which is

$d_j^{(0)} \leftarrow$ "initial value", $(1 \le j \le N)$;

<u>for</u> i = 1 <u>step</u> 1 <u>until</u> "termination" <u>do</u>

$d_j^{(i)} \leftarrow b_j - a_j c_{j-1}/d_{j-1}^{(i-1)}$, $(2 \le j \le N)$,

the arithmetic cost is $M((\tau_{\div} + \tau_{+})N + (\sigma_{\div} + \sigma_{+}))$, not counting the cost of precomputing $-a_j c_{j-1}$. Now consider splitting the loop:

$d_j^{(0)} \leftarrow$ "initial value", $(1 \le j \le N)$;

<u>for</u> i = 1 <u>step</u> 1 <u>until</u> "termination" <u>do</u>

<u>begin</u> $d_j^{(i)} \leftarrow b_j - a_j c_{j-1}/d_{j-1}^{(i-1)}$, $(1 \le j \le N,\ j \text{ even})$;

$d_j^{(i)} \leftarrow b_j - a_j c_{j-1}/d_{j-1}^{(i)}$, $(1 \le j \le N,\ j \text{ odd})$

<u>end</u>.

The cost per iteration is now $(\tau_{\div} + \tau_{+})N + 2(\sigma_{\div} + \sigma_{+})$, but only $M/2$ iterations are required because of the use of more recent information. Thus the total arithmetic cost is $M((\tau_{\div} + \tau_{+})N/2 + (\sigma_{\div} + \sigma_{+}))$ and the iteration has been sped up by nearly a factor of two. The same splitting technique may be applied to the second and third stages, and this corresponds exactly to the Gauss-Seidel iteration with the red-black ordering. Numerical experiments using the STAR timing information show that the "accelerated" three stage iteration will be faster than the red-black optimal SOR methods. It is also possible to define more general splittings which could give further improvements.

4.d. Block Tridiagonal and Band Systems

We now discuss some extensions of the parallel algorithms for tridiagonal matrices. A is a band matrix if $a_{ij} = 0$ when $|i-j| > m$ for some m; if m = 1 then A is tridiagonal. A is a block tridiagonal matrix if it can be partitioned as (A_{ij}) where A_{ii} is square and $A_{ij} = 0$ if $|i-j| > 1$. We

assume for simplicity that all the blocks are the same size. Any band matrix is also a block tridiagonal matrix, as is seen by taking $m \times m$ blocks, and any block tridiagonal matrix is clearly also a band matrix. Thus for theoretical purposes we can consider only one of the two cases, but for practical purposes it is more efficient to distinguish between them.

Suppose that A is block tridiagonal with $n \times n$ blocks. As with the tridiagonal case we write

$$A = \begin{pmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{N-1} & b_{N-1} & c_{N-1} \\ & & & a_N & b_N \end{pmatrix} = (a_j, b_j, c_j),$$

where $b_j \in R^{n \times n}$. x and v may be partitioned similarly. The study of block tridiagonal systems is made easier by the fact that elimination methods used for tridiagonal systems may be generalized to block elimination methods. As a rule, if the blocks are not mutually commutative then the diagonal blocks must be inverted, and this can create very serious storage problems when the blocks are originally large and sparse.

Block elimination adds another kind of inherent parallelism by allowing the parallel solution of block systems. For example, the block LU factorization

$d_1 \leftarrow b_1$; $g_1 \leftarrow v_1$;

<u>for</u> $j = 2$ <u>step</u> 1 <u>until</u> N <u>do</u>

<u>begin</u> $u_{j-1} \leftarrow d_{j-1}^{-1} c_{j-1}$; $f_{j-1} \leftarrow d_{j-1}^{-1} g_{j-1}$;

$d_j \leftarrow b_j - a_j u_{j-1}$; $g_j \leftarrow v_j - a_j f_{j-1}$

```
    end;
x_N ← d_N^{-1} g_N;
for j = N-1 step -1 until 1 do
    x_j ← f_j - u_j x_{j+1}
```

can use Gauss-Jordan elimination to compute u_{j-1} and f_{j-1}, as well as fast matrix multiplication in other computations. O(nN) steps are then used if n^2 processors are available. It may be observed, however, that the computation of the d sequence cannot be transferred into a second order linear matrix recurrence without assuming that certain blocks commute. If, for example, a_j is nonsingular, $2 \le j \le N$, then we can work with the new system $A^* x = v^*$, where we take $a_1 = I$ and $a^* = (I, a_j^{-1} b_j, a_j^{-1} c_j)$, $v^* = (a_j^{-1} v_j)$. The transformation to a recurrence will now succeed, and an O(n) + O(logn logN) algorithm can be formally defined.

The odd-even algorithms are also easily extended to block tridiagonal systems. Heller [74c] discusses this in some detail, and shows that if A is strictly diagonally dominant then the norms of the off-diagonal blocks relative to the diagonal blocks decrease quadratically, just as in the tridiagonal case. An important observation for the block case is that the additional storage required by odd-even reduction is about 2N blocks, while A itself requires 3N blocks.

As an extension of the algorithms for tridiagonal matrices and m^{th} order linear recurrences, Hyafil and Kung [75] describe an algorithm for $O(m^2N)$ processors requiring O(m) + O(logm logN) steps for a system of bandwidth m. Tewarson [68] describes another method using a matrix recurrence, and this also shows that banded systems can be solved in this time. Vector

algorithms for the root-free band Cholesky, symmetric band Gaussian elimination and profile elimination methods are discussed by Lambiotte [75].

4.e. Systems Arising from Differential Equations

The area of differential equations has exerted a great influence on parallel computation, for it provides a range of difficult and important problems. Because of these special applications, parallel computers have been designed to support in hardware some of the operations naturally occuring in the solution of differential equations. As examples, the interprocessor connections on the Illiac IV are precisely those of the five point finite-difference molecule used for two dimensional elliptic equations, and the CDC STAR provides vector instructions for differencing and averaging.

Although our present emphasis is on the treatment of discrete systems derived from continuous systems, the parallel solution of differential equations must be taken as a total package. It is essential that the discretization be chosen with parallelism in mind, so that the linear system may be generated and solved efficiently. While many standard discretization techniques do yield inherent parallelism in sequential algorithms, this can be greatly enhanced by making some basic alterations, mostly with regard to boundary conditions. One of the important and restrictive characteristics of SIMD and pipeline computers is the necessity of processing data in a homogeneous manner. Boundary conditions form an exceptional case in a discretization, and if not handled correctly will also form an exceptional case in a program. The probable effect of this would be to break up the flow of parallel operations with lengthy segments of sequential code. It is not our purpose to discuss this topic in much more detail, so we assume that

the discretization is given and will look for parallel algorithms based on properties of the linear system $Ax = v$.

Equations in one space dimension generally give rise to banded matrices with small bandwidth, and these problems have already been discussed. Two-dimensional elliptic equations and systems of one dimensional equations often yield block tridiagonal linear systems; the blocks may be large and sparse or small and dense. If the underlying geometry is, for instance, a rectangle, the system will have a very regular pattern of nonzero elements, and it is the existence of this pattern that makes the parallel solution feasible. General sparse matrices are much more difficult to handle in parallel, although hardware facilities such as the STAR's sparse vector instructions are designed to aid in this effort. We will return to this later.

It is important to be able to solve some systems very well; Poisson's equation on a rectangle is a prime example (Buzbee [73]). Assume a regular $N \times M$ grid with the usual five point discretization, Dirichlet boundary conditions, the nodes numbered by rows and from right to left within a row. The system $Ax = v$ is then block tridiagonal with constant block diagonals; that is, $A = (-I, B, -I)$, $B = (-1, 4, -1)$. The eigenvalues and eigenvectors of the diagonal blocks are known analytically, and in fact the matrix of eigenvectors represents a discrete sine transformation. Using parallel FFT techniques, the transformation may be applied simultaneously to each row of the grid in time $O(\log N)$. The original system is now decoupled into N independent $M \times M$ tridiagonal systems, one for each column of the grid. These may be solved in $O(\log M)$ steps using any of the methods already described. An inverse sine transformation is then applied to recover the solution. The

net result is that using O(NM) processors the (NM) $\times$ (NM) system may be solved in O(logNM) steps. The same decoupling technique may be applied to separable equations of the form

$$\frac{\partial}{\partial x} (a(x) \frac{\partial}{\partial x} u) + \frac{\partial}{\partial y} (b(y) \frac{\partial}{\partial y} u) + cu = f$$

on a rectangle, and to the biharmonic equation (Sameh, Chen and Kuck [74]). The biharmonic is a bit more difficult to handle, but with an N $\times$ N grid it may be solved in O(N) steps using $O(N^3)$ processors, or in O(NlogN) steps with $O(N^2)$ processors.

The odd-even reduction algorithm is another attractive method for Poisson's equation (Buzbee, Golub and Nielson [70], Ericksen [72]), although because of excessive storage requirements it is not feasible to use the block version of Algorithm R. Instead, assuming there are 2^n-1 diagonal blocks in A, the sequence of reduced block tridiagonal systems is

$$(-I,\ B^{(i)},\ -I)x^{(i)} = v^{(i)},$$

each with 2^{n-i}-1 diagonal blocks, and

$$B^{(i)} = (B^{(i-1)})^2 - 2I,\ B^{(0)} = B.$$

For numerical stability $v^{(i)}$ is represented by vectors $p^{(i)}$ and $q^{(i)}$, where

$$v^{(i)} = (0,\ B^{(i)},\ 0)p^{(i)} + q^{(i)}.$$

The inherent parallelism earlier demonstrated for odd-even reduction is still present. The computation of $p^{(i)}$ and $q^{(i)}$ requires the solution of systems involving $B^{(i)}$, but $B^{(i)}$ is a polynomial in B of degree 2^i, and this polynomial

can be factored analytically into its linear terms. Thus $B^{(i)}$ can be represented by the polynomial itself, does not need to be stored directly, and $B^{(i)-1}$ can be computed by solving 2^i tridiagonal systems. Sweet [74] has extended the method to cover cases other than 2^n-1 diagonal blocks, and Swarztrauber [74] gives another extension for separable equations using the storage-efficient polynomial representation.

For the Poisson equation we have used finite differences and a particular ordering of the nodes of the grid. A more general approach for square grids ($N \times N$) using a different ordering is given by Liu [74]. The only assumption necessary is that A is symmetric positive definite and that a_{ij} can be nonzero only if nodes x_i and x_j are corners of the same elementary square. This admits a large class of elliptic equations and finite difference or finite element methods; in the rowwise ordering A would be block tridiagonal. Liu shows that the nested dissection ordering (George [73], Birkhoff and George [73]) coupled with the LDL^T decomposition allows the solution of $Ax = v$ in $O(N)$ steps using $O(N^2)$ processors. We note that $O(N^3)$ multiplications are required to form the decomposition sequentially (Hoffman, Martin and Rose [73]). Nested dissection partitions the square grid into five disjoint subsets of grid points such that when A is reordered to group these subsets together we have

$$PAP^T = \begin{pmatrix} A_1 & & & & C_1^T \\ & A_2 & & & C_2^T \\ & & A_3 & & C_3^T \\ & & & A_4 & C_4^T \\ C_1 & C_2 & C_3 & C_4 & A_5 \end{pmatrix} .$$

The first four subsets are square, so the dissection may be repeated recursively; the fifth is "+" shaped and separates the first four from each other. It is clear that the inherent parallelism of the sequential elimination has been greatly increased by use of this ordering. To reduce the probability of memory interference without increasing the solution time, Liu suggests a new ordering, called doubly nested dissection. Like nested dissection it recursively uses "+" shaped separating sets, but the crosses are now double width, providing more complete independence of the square subsets.

Lambiotte [75] reports on the implementation of nested dissection on the CDC STAR. By taking advantage of the ability to predict the number and location of nonzeros per row of L^T it is possible to obtain an algorithm consistent in both storage ($O(N^2 \log N)$ words) and time ($O(N^3)$). However, the vectorization is complicated enough so that the standard band methods are expected to be more efficient (both in terms of runtime and programming costs) for moderate values of N.

From a theoretical standpoint direct methods are sufficient, and it would appear that there is no need for iterative methods. However, from a practical standpoint there will still be some cases, such as irregular domains, nonseparable elliptic equations and three-dimensional problems, where an iterative solution will be attractive. In addition, many iterative methods for linear systems can be used to define iterative methods for nonlinear systems (Ortega and Rheinboldt [70]), so we can also learn something about the parallel solution of these more difficult problems.

A large class of iterations is defined by a splitting of the matrix A (Varga [62]). Two matrices A_1 and A_2 are chosen such that $A = A_1 - A_2$ and systems involving A_1 may be solved "easily". Given an estimate $x^{(0)}$, improved

estimates are generated by the rule $A_1x^{(i+1)} = A_2x^{(i)} + v$. A number of techniques may be used to accelerate convergence of the basic method, but the success of the iteration still depends crucially on the proper choice of the structure of A_1.

One approach is to let A_1 be any of the special forms already discussed. In the case of nonseparable elliptic equations a good choice for A_1 is a close relative of the Poisson matrix (Concus and Golub [73]). Of course, the simplest choice is $A_1 = \text{diag}(A)$, yielding the Jacobi iteration, but convergence is generally so slow that the method is not competitive. Gilmore [71] suggests use of Jacobi overrelaxation with an associative processor, but slow convergence is again a detracting factor, as well as the fact that for a consistently ordered matrix the optimal relaxation factor is $\omega = 1$.

A second approach looks for efficient implementations of standard sequential iterations. To illustrate, suppose we have a two-dimensional elliptic equation and an $N \times N$ grid with a five point finite difference approximation. It can be observed that a point SOR method with the rowwise ordering has essentially no inherent parallelism. By choosing a different ordering of the nodes this situation can be greatly improved without sacrificing the rate of convergence. The basic schemes are the red-black ordering, the diagonal ordering and subdomain partitioning. These ideas have appeared in many independent publications, with several minor variations, so it is impossible to give all credits, but Karp, Miller and Winograd [67] and Morice [72] are good sources.

The red-black ordering (or any other 2-cyclic ordering (Varga [62])) enables us to write

$$PAP^T = \begin{pmatrix} D_1 & F \\ G & D_2 \end{pmatrix},$$

D_1 and D_2 diagonal, so the SOR methods may be described wholely in terms of vector operations. In fact, these vectors have average length $N^2/2$, which is an attractive feature for pipeline computers. The diagonal ordering decomposes the grid as 2N-1 diagonals, creating an independence of computations that allows all the nodes of a diagonal to be updated simultaneously. The subdomain approach is suitable for a parallel computer with a small number of processors: each processor is assigned to a group of points, which it updates sequentially.

These methods all use an ordering and partitioning of A such that the smaller systems arising in block SOR are actually diagonal. Of course, the natural ordering could be used with line SOR, where only tridiagonal systems need to be solved. The alternating direction implicit methods similarly require only the solution of tridiagonal systems. The first half step of the iteration simultaneously solves a tridiagonal system for each row of the grid, and the second half step does the same for each column. Lambiotte [75] presents a STAR implementation of ADI for Poisson's equation, in which the grid is stored by columns, the row systems are solved by simultaneous execution of the usual sequential algorithm, and the column systems are solved as one $N^2 \times N^2$ tridiagonal system using odd-even reduction.

Hayes [74] reports on some tests conducted on the ASC to compare the standard iterative schemes (not including ADI) for the solution of Laplace's equation on a unit square. For mesh size $h = 1/80$ (N=79) the best method is the cyclic Chebyshev semi-iterative scheme, which is essentially SOR in the red-

black ordering but with better parameters. It is interesting to note, however, that the symmetric SOR semi-iterative method (Young [72]) is the closest competitor. This method uses the natural rowwise ordering and the bulk of the computation has essentially no inherent parallelism, but it is still a good method because it converges so quickly ($O(h^{-1/2})$ iterations vs. $O(h^{-1})$ iterations for SOR and CCSI). Lambiotte [75] shows that it is advantageous to consider SSOR-SI based on the diagonal ordering. The good rate of convergence is preserved, and the diagonal ordering allows the use of vector instructions. Although the vectors are only of length N, as opposed to $N^2/2$ with the red-black ordering, for large values of N this becomes less important than the convergence rate.

To close this subsection, we consider some iterative methods for parallel computers with a small number of asynchronous parallel processors. The chaotic relaxation methods, originally developed for the iterative solution of linear systems (Chazan and Miranker [69], Donelly [71]) and now extended to nonlinear systems (Robert, Charnay and Musy [75]), take advantage of asynchronous computation by randomly relaxing components in parallel. The current values of the solution are stored in a common memory, and the iteration is terminated when some condition is met; e.g., one processor is given the task of checking for convergence. There must be some way of guaranteeing that, if the parallel program were allowed to run forever, each component would be updated infinitely often, for otherwise convergence could not occur. Because synchronization is not part of the algorithm, it is expected that programming will be considerably easier, especially since memory access conflicts can be resolved by the computer system itself, and not by the chaotic

relaxation programs. Chazan and Miranker [69] show that convergence occurs if $\rho(|B|) < 1$, $B = I - \text{diag}(A)^{-1}A$, independent of the schedule of computations; if $\rho(|B|) \geq 1$ then there exists a schedule for which convergence does not occur, even when all components are updated infinitely often.

A related class of iterations for the P processor SIMD model was given by Robert [70]. The system is partitioned into $P \times P$ blocks, and groups of components are updated simultaneously and explicitly as in the Jacobi iteration, rather than simultaneously and implicitly as in the usual block iterations. With one processor the method reduces to Gauss-Seidel iteration, and with N processors it is the Jacobi iteration. If A is an M-matrix it is possible to compare the rates of convergence for different values of P: if P_1 divides P_2 exactly, and M_1 and M_2 are the iteration matrices for Robert's methods, then $\rho(M_1) \leq \rho(M_2) < 1$. It is possible to show that for a given value of P the block Gauss-Seidel iteration is asymptotically faster, so Robert's method will be less time consuming if the block systems are difficult to solve.

4.f. General Sparse Matrices

At the 1968 IBM Sparse Matrix Conference the hope was expressed that parallel machines would meet the special needs of this area (Wolfe [68]). The sparse vector instructions on the CDC STAR, for example, were designed for packed storage and faster execution times. However, as Lambiotte [75] shows for nested dissection, the storage costs using sparse vectors can actually increase over scalar methods. If we choose to represent the $N \times N$ matrix A as a sparse vector of length N^2, then N^2 bits ($\lceil N^2/64 \rceil$ words) are needed to define the structure of A, regardless of its sparsity. Moreover,

since the time for a sparse vector operation depends on the number of bits in each order vector, the total running time may be adversely affected.

While most of the matrix elements are zero, the nonzeros need not be distributed in a regular pattern, and this can complicate parallel solutions using the LU decomposition. It may be possible to automatically reorder the system to create a regular pattern. One criterion is to find a permutation P with the following property (Calahan [73]): if $A_0 = PAP^T$,

$$A_{i-1} = \begin{pmatrix} D_i & F_i \\ G_i & H_i \end{pmatrix}, \quad A_i = H_i - G_i D_i^{-1} F_i,$$

then the D_i's are diagonal, nonsingular, and their average size is as large as possible. The permutation used by odd-even reduction has this property. The automatic generation of P can be quite difficult; if the sparse system is at the inner loop of Newton's method for a nonlinear system then the preprocessing cost may be spread across a large number of outer iterations.

Another basic approach is to consider A only as an operator to be applied to a vector. It is assumed that there is some highly parallel procedure to evaluate Az and z^TA given z; it is not even necessary to generate A explicitly. Any sequence of operations on whole vectors (e.g., linear combinations and inner products) is naturally parallel so the method of conjugate gradients may be used as either a direct or iterative method if A is positive definite (Palmer [74], Reid [72]). If A is symmetric but indefinite the operator approach can also be used to derive direct methods related to the Lanczos tridiagonalization (Paige and Saunders [75]). If A is unsymmetric then an orthogonal bidiagonalization of A can be found with

reasonable economy (Cline [74]). As this decomposition proceeds, minimal residual approximations can be generated using current information. At present these methods appear to be the most favorable for synchronous parallelism; for asynchronous parallelism the chaotic relaxation schemes are the natural approach.

5. EIGENVALUES

In this section we consider a rather different problem of linear algebra. Unlike the solution of a linear system, where the exact answer can be produced in a finite number of steps, the exact calculation of eigenvalues is an infinite process. In actual computation the process is terminated at some point. One fundamental technique is to construct a sequence of matrices A_i, all similar to A and converging to a matrix T from which the eigenvalues are more easily obtained. We address only the finite problem of generating the next matrix in the sequence. A second technique is to convert A into a form where the characteristic polynomial $f(\lambda) = \det(A-\lambda I)$ is easily evaluated, and to approximate the zeros of f by iteration or bisection. If only the largest eigenvalues and corresponding eigenvectors are desired, variations of the power method are directly applicable, as we only need to compute Az, inner products and linear combinations of vectors. However, we will not consider this method, or the general problem of computing eigenvectors.

First, suppose that A is a dense, real symmetric matrix, and let $A_0 = A$. In Jacobi's method $A_{i+1} = R_i A_i R_i^T$, where R_i is a plane rotation matrix designed to annihilate (for some $p < q$ depending on i) the off-diagonal elements $a_{pq}^{(i)}$ and $a_{qp}^{(i)}$; that is, $a_{pq}^{(i+1)} = a_{qp}^{(i+1)} = 0$. Each rotation reduces the sum of squares of the off-diagonal elements, so A_i converges to a diagonal matrix, and the iteration is halted when the sum of squares is sufficiently small. The premultiplication $R_i A_i$ replaces rows p and q with linear combinations of themselves, and the postmultiplication is analagous. Thus we can exploit parallelism by using the inherent parallelism of row and column

operations, but Sameh [71] (also Kuck and Sameh [71]) shows that further improvements can be made. In this parallel variation we take $Q_i = M_i A_i M_i^T$, $M_i = T_i P_i$, P_i a permutation matrix and, supposing $N = 2n$,

$$T_i = \mathrm{diag}(T_1^{(i)}, T_2^{(i)}, \ldots, T_n^{(i)}),$$

$$T_j^{(i)} = \begin{pmatrix} \cos \varphi_j^{(i)} & \sin \varphi_j^{(i)} \\ -\sin \varphi_j^{(i)} & \cos \varphi_j^{(i)} \end{pmatrix}.$$

Thus n rotations have been overlapped, and N off-diagonal elements can be annihilated simultaneously. The permutations ensure that all the off-diagonal positions will be subjected to annihilation.

If A is not symmetric, then the same overlapping technique can be applied to the Jacobi-like algorithm proposed by Eberlein [62]. A parallel version is given by Sameh [71] . Here $A_{i+1} = U_i A_i U_i^{-1}$, $U_i = S_i T_i P_i$, P_i and T_i as above, and

$$S_i = \mathrm{diag}(S^{(i)}, S^{(i)}, \ldots, S^{(i)}),$$

$$S^{(i)} = \begin{pmatrix} \cosh \psi_i & \sinh \psi_i \\ \sinh \psi_i & \cosh \psi_i \end{pmatrix}.$$

The biorthogonalization algorithm (Hestenes [58]) can also benefit from overlapping column operations (Hall [73]). In this method the columns of A are repeatedly orthogonalized in pairs so that, in the limit, $AW = Q$, where W is orthogonal and $Q^T Q = D^2$ is diagonal. A Jacobi sequence for $A^T A$ is implicitly produced, and the singular value decomposition $A = USV^T$ is found by setting $U = QD^{-1}$, $S = D$, $V = W$.

Many methods can be applied more efficiently if the dense matrix A is transformed into a simpler form $A^* = QAQ^T$ where Q is orthogonal. If A is symmetric then A^* is taken to be symmetric tridiagonal; otherwise A^* is upper Hessenberg. The outer product formulations given by Wilkinson [65] (pp. 290-2, 347-9) show the inherent parallelism of the Householder transformations. It must be observed, however, that the effective vector lengths decrease at each step, and this may be a disadvantage in some parallel implementations, despite the fact that it is always an advantage in sequential computation.

Now, suppose that A is symmetric and reduced to tridiagonal form. It is well known that the characteristic polynomial may be evaluated by a second order linear recurrence. Kuck and Sameh [71] suggest using a multiple Sturm sequence and bisection algorithm for N processors, evaluating f at N points simultaneously and operating with N intervals. Because derivatives of f are also easy to evaluate, parallel root finding methods can be used. Of course, the algorithms to evaluate second order recurrences in O(logN) steps given N processors are applicable. One place where these occur is in the LR algorithm: A_{i+1} is obtained by factoring $A_i - k_i I = L_i R_i$, where L_i is unit lower bidiagonal and R_i is upper bidiagonal, and setting $A_{i+1} = R_i L_i + k_i I$. The shift is necessary to accelerate convergence. Explicitly, suppose that $A_i - k_i I = (a_j, b_j, c_j)$. If $g_1 = b_1$, $g_j = b_j - a_j c_{j-1}/g_{j-1}$, $j = 2, \ldots, N$, then

$$A_{i+1} = (a_j g_j/g_{j-1},\ g_j + a_{j+1} c_j g_j + k_i,\ c_j)$$

and convergence is to an upper bidiagonal matrix. As in Stone's method for the solution of a tridiagonal system, the g sequence can be computed in

O(logN) steps with N processors, and the rest of the computations are entirely parallel. Similar results hold for the LL^T algorithm.

For stability reasons it is often preferable to use the QR algorithm. $A_i - k_iI$ is reduced to upper triangular form by N-1 plane rotations, obtaining $A_i - k_iI = Q_iR_i$, Q_i orthogonal and R_i upper triangular. A_{i+1} is then computed from $R_iQ_i + k_i$ as in the LR algorithm, although symmetry is preserved and convergence is to a diagonal matrix. Sameh and Kuck [75a] have recently shown that recurrence techniques are also applicable to this method. The basic sequential algorithm is, for $A_i - k_iI = (a_{j-1}, b_j, a_j)$, $A_{i+1} = (\alpha_{j-1}, \beta_j, \alpha_j)$,

$p_1 \leftarrow b_1$; $c_0 \leftarrow 1$;

<u>for</u> j = 1 <u>step</u> 1 <u>until</u> N-1 <u>do</u>

<u>begin</u> $r_j \leftarrow (p_j^2 + a_j^2)^{1/2}$;

$c_j \leftarrow p_j/r_j$; $s_j \leftarrow a_j/r_j$;

$p_{j+1} \leftarrow c_j b_{j+1} - s_j c_{j-1} a_j$

<u>end</u>;

$r_N \leftarrow p_N$;

$q_j \leftarrow s_j b_{j+1} + c_j c_{j-1} a_j$, $(1 \le j \le N-1)$;

$\beta_j \leftarrow p_j c_{j-1} + q_j s_j + k_i$, $(1 \le j \le N-1)$;

$\beta_N \leftarrow p_N c_{N-1} + k_i$;

$\alpha_j \leftarrow r_{j+1} s_j$, $(1 \le j \le N-1)$;

$\alpha_N \leftarrow 0$.

As can be seen the major bottleneck is the sequential computation of the r, s, c and p sequences. Sameh and Kuck introduce the following sequences:

$$w_0 = 1,\ w_1 = b_1,\ w_{j+1} = b_{j+1}w_j - a_j^2 w_{j-1},$$
$$j = 1, \ldots, N-1,$$

$$z_0 = 1,\ z_j = a_j^2 z_{j-1} + w_j^2,\ j = 1, \ldots, N.$$

By induction it can be shown that $w_j/w_{j-1} = p_j/c_{j-1}$ and $z_j/z_{j-1} = r_j^2$, so the parallel QR step may be expressed as

compute w_j, $0 \le j \le N$;

compute z_j, $0 \le j \le N$;

$r_j^2 \leftarrow z_j/z_{j-1}$, $(1 \le j \le N)$;

$s_j^2 \leftarrow b_j^2/r_j^2$, $(1 \le j \le N-1)$;

$\beta_j \leftarrow s_j^2 b_{j+1} + w_j w_{j-1}(a_j^2 + r_j^2)/z_j + k_i$,
$(1 \le j \le N-1)$;

$\beta_N \leftarrow w_N w_{N-1}/z_{N-1} + k_i$;

$\alpha_j^2 \leftarrow r_{j+1}^2 s_j^2$, $(1 \le j \le N-1)$;

$\alpha_N^2 \leftarrow 0$.

Thus the QR iteration requires O(logN) steps with N processors. Note that the squares of the off-diagonal elements are used, and that only rational operations are necessary (cf. Reinsch [71]). Despite possible ill-conditioning of the linear systems implicit in the computation of w_j and z_j, Sameh and Kuck report that they have been able to obtain good results in simulated execution on a sequential computer.

We now consider the nonsymmetric case, and assume that A has been reduced to upper Hessenberg form by similarity transformation. The standard sequential method for real matrices is the QR algorithm with double origin

shift, which avoids the need for complex arithmetic. The double shift is actually performed by using a similarity transform $C_i = P_i A_i P_i^T$, where P_i is a Householder transformation and C_i is upper Hessenberg except for $c_{31}^{(i)}$, $c_{41}^{(i)}$ and $c_{42}^{(i)}$, which are nonzero. A_{i+2} is then obtained by reducing C_i to upper Hessenberg form, and we have

$$A_{i+2} = Q_i^T A_i Q_i,$$

$$(A_i - k_i I)(A_i - k_{i+1} I) = Q_i R_i.$$

Either plane rotations or Householder transformations may be used in the reduction; the latter is preferred in sequential computation since fewer multiplications are used.

Unfortunately, the parallel implementation of the double QR step may be adversely affected by some of the properties that make it so attractive for sequential computers. Investigations by Kuck and Sameh [71] for the Illiac IV and by Ward [76] for the CDC STAR show that the necessary use of short vectors in either the pre- or postmultiplications and in deflation can lead to inefficiencies. On the Illiac this is manifested by low processor efficiency, and on the STAR by the fact that the time for the double step is $\alpha N^2 + O(N)$, where α depends on the vector startup time σ. This is clearly not a good situation, and additional work is needed to find better parallel implementations.

As an alternative to the QR algorithm, Ward [76] suggests the use of Laguerre iteration with Hyman's method of evaluating $f(\lambda)$ and its derivatives (Wilkinson [65]). In fact, Hyman's method corresponds to the solution of an upper triangular linear system, so this method is attractive when parallel evaluation of f is used.

Acknowledgments. Valuable comments on the manuscript were received from H. T. Kung, D. K. Stevenson, J. F. Traub and R. G. Voigt. This work would not have been possible without the continued support and encouragement of J. F. Traub.

References

Babuska [68]. I. Babuska, "Numerical stability in mathematical analysis," IFIP Congress 1968, North-Holland, Amsterdam, 1969, vol. 1, pp. 11-23.

Baer [73]. J. L. Baer, "A survey of some theoretical aspects of multiprocessing," Computing Surveys, vol. 5, 1973, pp. 31-80.

Barnes et al. [68]. G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, R. A. Stoker, "The Illiac IV computer," IEEE Trans. on Comp., vol. C-17, 1968, pp. 746-757.

Birkhoff and George [73]. G. Birkhoff and A. George, "Elimination by nested dissection," in Complexity of Sequential and Parallel Numerical Algorithms, J. F. Traub, ed., Academic Press, N. Y., 1973, pp. 221-269.

Borodin [71]. A. Borodin, "Horner's rule is uniquely optimal," in Theory of Machines and Computations, Z. Kohavi and A. Paz, eds., Academic Press, N. Y., 1971, pp. 45-58.

Borodin and Munro [75]. A. Borodin and I. Munro, The Computational Complexity of Algebraic and Numeric Problems, American Elsevier, N. Y., 1975.

Bouknight et al. [72]. W. J. Bouknight, S. A. Denenberg, D. E. McIntyre, J. M. Randall, A. H. Sameh, D. L. Slotnick, "The Illiac IV system," Proc. IEEE, vol. 60, 1972, pp. 369-388.

Brent [73]. R. P. Brent, "The parallel evaluation of arithmetic expressions in logarithmic time," in Complexity of Sequential and Parallel Numerical Algorithms, J. F. Traub, ed., Academic Press, N. Y., 1973, pp. 83-102.

Brent [74]. R. P. Brent, "The parallel evaluation of general arithmetic expressions," J. ACM, vol. 21, 1974, pp. 201-206.

Burroughs [72]. Burroughs Corp., Illiac IV Systems Characteristics and Programming Manual, Paoli, Pa., May, 1972.

Buzbee [73]. B. L. Buzbee, "A fast Poisson solver amenable to parallel computation," IEEE Trans. on Comp., vol. C-22, 1973, pp. 793-796.

Buzbee, Golub and Nielson [70]. B. L. Buzbee, G. H. Golub, and C. W. Nielson, "On direct methods for solving Poisson's equations," SIAM J. Num. Anal., vol. 7, 1970, pp. 627-656.

Calahan [73]. D. Calahan, "Parallel solution of sparse simultaneous linear equations," Dept. of Elec. Eng., Univ. of Michigan, Ann Arbor, 1973.

CDC [74]. Control Data Corporation, CDC STAR-100 Instruction Execution Times, preliminary version 2, Arden Hills, Minn., January, 1974.

Chazan and Miranker [69]. D. Chazan and W. L. Miranker, "Chaotic relaxation," Lin. Alg. Appl., vol. 2, 1969, pp. 199-222.

Chen [75]. S. C. Chen, "Speedup of iterative programs in multiprocessing systems," Dissertation, Dept. of Comp. Sci., Univ. of Illinois, Urbana, January, 1975.

Chen and Kuck [75]. S. C. Chen and D. J. Kuck, "Time and parallel processor bounds for linear recurrence systems," IEEE Trans. on Comp., vol. C-24, 1975, pp. 701-717.

T. C. Chen [75]. T. C. Chen, "Overlap and pipeline processing," in Introduction to Computer Architecture, H. S. Stone, ed., Science Research Associates, Palo Alto, Calif., 1975, pp. 375-431.

Cline [74]. A. K. Cline, "A Lanczos-type method for the solution of large sparse systems of linear equations," contributed paper, Second Langley Conf. on Sci. Comp., Virginia Beach, October, 1974.

Concus and Golub [73]. P. Concus and G. H. Golub, "Use of fast direct methods for the efficient numerical solution of nonseparable elliptic equations," SIAM J. Num. Anal., vol. 10, 1973, pp. 1103-1120.

Cooley and Tukey [65]. J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," Math. Comp., vol. 19, 1965, pp. 297-301.

Cray [75]. Cray Research, Inc., "Cray-1 Computer," Chippewa Falls, Wis., 1975.

Csanky [75]. L. Csanky, "Fast parallel matrix inversion algorithms," contributed paper, 16th Ann. Symp. on Foundations of Computer Science (SWAT), Berkeley, October, 1975.

Donelly [71]. J. D. P. Donelly, "Periodic chaotic relaxation," Lin. Alg. Appl., vol. 4, 1971, pp. 117-128.

Eberlein [62]. P. J. Eberlein, "A Jacobi-like method for the automatic computation of eigenvalues and eigenvectors of an arbitrary matrix," J. SIAM, vol. 10, 1962, pp. 74-88.

Ericksen [72]. J. H. Ericksen, "Iterative and direct methods for solving Poisson's equation and their adaptability to ILLIAC IV," Center for Advanced Computation, University of Illinois, Urbana, 1972.

Flynn [66]. M. J. Flynn, "Very high-speed computing systems," Proc. IEEE, vol. 54, 1966, pp. 1901-1909.

Gentleman [73]. W. M. Gentleman, "Least squares computations by Givens transformations without square roots," J. Inst. Math. Applics., vol. 12, 1973, pp. 329-336.

George [73]. J. A. George, "Nested dissection of a regular finite element mesh," SIAM J. Numer. Anal., vol. 10, 1973, pp. 345-363.

Gilmore [71]. P. A. Gilmore, "Parallel relaxation," Goodyear Aerospace Corp., Akron, Ohio, July, 1971.

Hall [73]. J. C. Hall, "Examination of numerical methods for singular value decompositions," M.S. thesis, University of Colorado, Boulder, 1973.

Hayes [74]. L. Hayes, "Comparative analysis of iterative techniques for solving Laplace's equation on the unit square on a parallel processor," M.S. thesis, Dept. of Math., University of Texas, Austin, 1974.

Heller [74a]. D. Heller, "A determinant theorem with applications to parallel algorithms," SIAM J. Num. Anal., vol. 11, 1974, pp. 559-568.

Heller [74b]. D. Heller, "On the efficient computation of recurrence relations," ICASE, Hampton, Va.; Dept. of Computer Science, Carnegie-Mellon University, June, 1974.

Heller [74c]. D. Heller, "Some aspects of the cyclic reduction algorithm for block tridiagonal linear systems," ICASE, Hampton, Va.; Dept. of Computer Science, Carnegie-Mellon University, December, 1974.

Heller, Stevenson and Traub [74]. D. Heller, D. K. Stevenson, J. F. Traub, "Accelerated iterative methods for the solution of tridiagonal linear systems on parallel computers," Department of Computer Science, Carnegie-Mellon University, December, 1974.

Hestenes [58]. M. R. Hestenes, "Inversion of matrices by biorthogonalization and related results," J. SIAM, vol. 6, 1958, pp. 51-90.

Hintz and Tate [72]. R. G. Hintz and D. P. Tate, "Control Data STAR-100 Processor design," COMPCON-72 Digest of Papers, IEEE Comp. Soc., 1972, pp. 1-4.

Hockney [65]. R. W. Hockney, "A fast direct solution of Poisson's equation using Fourier analysis," J. ACM, vol. 12, 1965, pp. 95-113.

Hoffman, Martin and Rose [73]. A. J. Hoffman, M. S. Martin and D. J. Rose, "Complexity bounds for regular finite difference and finite element grids," SIAM J. Numer. Anal., vol. 10, 1973, pp. 364-369.

Hyafil and Kung [74a]. L. Hyafil and H. T. Kung, "The complexity of parallel evaluation of linear recurrences," Proc. 7th Ann. ACM Symp. on Theory of Computing, 1975, pp. 12-22.

Hyafil and Kung [74b]. L. Hyafil and H. T. Kung, "Parallel algorithms for solving triangular linear systems with small parallelism," Department of Computer Science, Carnegie-Mellon University, December, 1974.

Hyafil and Kung [75]. L. Hyafil and H. T. Kung, "Bounds on the speedups of parallel evaluation of recurrences," Second USA-Japan Comp. Conf. Proc., August, 1975, pp. 178-182.

Jordan [74]. T. L. Jordan, "A new parallel algorithm for diagonally dominant tridiagonal matrices," Los Alamos Sci. Lab., Los Alamos, N. M., 1974.

Kahan [71]. W. Kahan, "A survey of error analysis," IFIP Congress 1971, North-Holland, Amsterdam, 1972, vol. 2, pp. 1214-1239.

Karp, Miller and Winograd [67]. R. M. Karp, R. E. Miller, S. Winograd, "The organization of computations for uniform recurrence relations," J.ACM, vol. 14, 1967, pp. 563-590.

Klyuyev and Kokovkin-Shcherbak [65]. V. V. Klyuyev and N. I. Kokovkin-Shcherbak, "On the minimization of the number of arithmetic operations for the solution of linear algebraic systems of equations," trans. by G. J. Tee, Department of Computer Science, Stanford University, 1965.

Knight, Poole and Voigt [75]. J. C. Knight, W. G. Poole, Jr., and R. G. Voigt, "System balance analysis for vector computers," ICASE, Hampton, Virginia, March, 1975.

Kogge [72a]. P. M. Kogge, "Parallel algorithms for the efficient solution of recurrence problems," Digital Systems Lab., Stanford University, September, 1972.

Kogge [72b]. P. M. Kogge, "The numerical stability of parallel algorithms for solving recurrence problems," Digital Systems Lab., Stanford University, September, 1972.

Kogge [72c]. P. M. Kogge, "Minimal parallelism in the solution of recurrence problems," Digital Systems Lab., Stanford University, September, 1972.

Kogge [73]. P. M. Kogge, "Maximal rate pipeline solutions to recurrence problems," Proc. First Ann. Symp. on Comp. Architecture, Gainesville, Florida, 1973, pp. 71-76.

Kogge [74]. P. M. Kogge, "Parallel solution of recurrence problems," IBM J. Res. Deve., vol. 18, 1974, pp. 138-148.

Kogge and Stone [73]. P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," IEEE Trans. on Comp., vol. C-22, 1973, pp. 786-793.

Kuck [68]. D. J. Kuck, "Illiac IV software and application programming," IEEE Trans. on Comp., vol. C-17, 1968, pp. 758-770.

Kuck [73]. D. J. Kuck, "Multioperation machine computational complexity," in Complexity of Sequential and Parallel Numerical Algorithms, J. F. Traub, ed., Academic Press, N. Y., 1973, pp. 17-47.

Kuck and Maruyama [75]. D. J. Kuck and K. Maruyama, "Time bounds on the parallel evaluation of arithmetic expressions," SIAM J. Comput., vol. 4, 1975, pp. 147-162.

Kuck and Sameh [71]. D. J. Kuck and A. H. Sameh, "Parallel computation of eigenvalues of real matrices," IFIP Congress 1971, North-Holland, Amsterdam, 1972, vol. 2, pp. 1266-1272.

Kung [74]. H. T. Kung, "New algorithms and lower bounds for the parallel evaluation of certain rational expressions," Proc. Sixth Ann. ACM Symp. on Theory of Comput., pp. 323-333.

Kung and Traub [74]. H. T. Kung and J. F. Traub, "Methodologies for studying the speedups gained from parallelism," contributed paper, Second Langley Conf. on Sci. Comp., Virginia Beach, October, 1974.

Lambiotte [75]. J. J. Lambiotte, Jr., "The solution of linear systems of equations on a vector computer," Dissertation, University of Virginia, 1975.

Lambiotte and Voigt [75]. J. J. Lambiotte, Jr. and R. G. Voigt, "The solution of tridiagonal linear systems on the CDC STAR-100 computer," ACM Trans. on Math. Software, Vol. 1, 1975, pp. 308-329.

Lawrie et al. [75]. D. H. Lawrie, T. Layman, D. Baer and J. M. Randal, "Glypnir, a programming language for Illiac IV," Comm. ACM, vol. 18, 1975, pp. 157-164.

Liu [74]. J. W. H. Liu, "The solution of mesh equations on a parallel computer," Department of Computer Science, University of Waterloo, October, 1974.

Lynch [74]. W. C. Lynch, "How to stuff an array processor," Third Texas Conf. on Comp. Systems, November, 1974.

Maruyama [73]. K. Maruyama, "The parallel evaluation of matrix expressions," IBM T. J. Watson Research Center, Yorktown Heights, N. Y., 1973.

Miller [73]. R. E. Miller, "A comparison of some theoretical models of parallel computation," IEEE Trans. on Comp., vol. C-22, 1973, pp. 710-717.

W. Miller [75]. W. Miller, "Computational complexity and numerical stability," SIAM J. Comput., vol. 4, 1975, pp. 97-107.

Miranker [71]. W. L. Miranker, "A survey of parallelism in numerical analysis," SIAM Review, vol. 13, 1971, pp. 524-547.

Moler [72]. C. B. Moler, "Matrix computations with Fortran and paging," Comm. ACM, vol. 15, 1972, pp. 268-270.

Morice [72]. Ph. Morice, "Calcul parallele et decomposition dans la resolution d'equations aux derivees partielles de type elliptique," IRIA, June, 1972.

Muller and Preparata [75]. D. E. Muller and F. P. Preparata, "Upper bound to the time for parallel evaluation of arithmetic expressions," contributed paper, Symposium on Analytic Computational Complexity, Carnegie-Mellon University, April, 1975.

Munro and Paterson [73]. I. Munro and M. Paterson, "Optimal algorithms for parallel polynomial evaluation," J. Comp. Syst. Sci., vol. 7, 1973, pp. 189-198.

Muraoka [71]. Y. Muraoka, "Parallelism exposure and exploitation," Dissertation, Department of Computer Science, University of Illinois, Urbana, 1971.

Muraoka and Kuck [73]. Y. Muraoka and D. J. Kuck, "On the time required for a sequence of matrix products," Comm. ACM, vol. 16, 1973, pp. 22-26.

Newell and Robertson [75]. A. Newell and G. Robertson, "Some issues in programming multi-mini-processors," Department of Computer Science, Carnegie-Mellon University, January, 1975.

Orcutt [74]. S. E. Orcutt, Jr., "Computer organization and algorithms for very high-speed computations," Dissertation, Department of Electrical Engineering, Stanford University, 1974.

Ortega and Rheinboldt [70]. J. M. Ortega and W. C. Rheinboldt, Iterative Solution of Nonlinear Equations in Several Variables, Academic Press, N. Y., 1970.

Owens [73]. J. L. Owens, "The influence of machine organization on algorithms," in Complexity of Sequential and Parallel Numerical Algorithms, J. F. Traub, ed., Academic Press, N. Y., 1973, pp. 111-130.

Paige and Saunders [75]. C. C. Paige and M. A. Saunders, "Solution of sparse indefinite systems of equations," SIAM J. Numer. Anal., vol. 12, 1975, pp. 617-629.

Palmer [74]. J. Palmer, "Conjugate direction methods and parallel computing," Dissertation, Department of Computer Science, Stanford University, 1974.

Parlett and Wang [75]. B. N. Parlett and Y. Wang, "The influence of the compiler on the cost of mathematical software - in particular on the cost of triangular factorization," ACM Trans. on Math. Software, vol. 1, 1975, pp. 35-46.

Pease [67]. M. C. Pease, "Matrix inversion using parallel processing," J.ACM, vol. 14, 1967, pp. 757-764.

Pease [68]. M. C. Pease, "An adaptation of the fast Fourier transform for parallel processing," J.ACM, vol. 15, 1968, pp. 252-264.

Pease [69]. M. C. Pease, "Inversion of matrices by partitioning," J.ACM, vol. 16, 1969, pp. 302-314.

Pease [74]. M. C. Pease, "The C(2,m) algorithm for matrix inversion," Stanford Research Institute, Menlo Park, California, 1974.

Poole and Voigt [74]. W. G. Poole, Jr. and R. G. Voigt, "Numerical algorithms for parallel and vector computers: An annotated bibliography," Computing Reviews, vol. 15, 1974, pp. 379-388.

Preparata and Muller [75]. F. P. Preparata and D. E. Muller, "Parallel evaluation of division-free expressions," contributed paper, Symposium on Analytic Computational Complexity, Carnegie-Mellon University, April, 1975.

Reid [72]. J. K. Reid, "The use of conjugate gradients for systems of linear equations possessing "Property A"," SIAM J. Numer. Anal., vol. 9, 1972, pp. 325-332.

Reinsch [71]. C. H. Reinsch, "A stable, rational QR algorithm for the computation of the eigenvalues of an Hermitian tridiagonal matrix," Math. Comp., vol. 25, 1971, pp. 591-597.

Robert [70]. F. Robert, "Methods iteratives 'serie-parallele'," C. R. Acad. Sci., Paris, vol. 271, 1970, pp. 847-850.

Robert, Charnay and Musy [75]. F. Robert, M. Charnay, and F. Musy, "Iterations chaotiques serie-parallele pour des equations non-lineaires de point fixe," Aplikace Matematiky, vol. 20, 1975, pp. 1-38.

Rudolph [72]. J. A. Rudolph, "A production implementation of an associative array processor - STARAN," AFIPS Fall 1972, AFIPS Press, Montvale, N. J., vol. 41, pt. 1, pp. 229-241.

Ruggiero and Coryell [69]. J. F. Ruggiero and D. A. Coryell, "An auxiliary processing system for array calculations," IBM Sys. J., vol. 8, 1969, pp. 118-135.

Sameh [71]. A. H. Sameh, "On Jacobi and Jacobi-like algorithms for a parallel computer," Math. Comp., vol. 25, 1971, pp. 579-590.

Sameh, Chen and Kuck [74]. A. H. Sameh, S. C. Chen and D. J. Kuck, "Parallel direct Poisson and biharmonic solvers," Department of Computer Science, University of Illinois, Urbana, July, 1974.

Sameh and Kuck [75a]. A. H. Sameh and D. J. Kuck, "A parallel QR algorithm for tridiagonal symmetric matrices," Department of Computer Science, University of Illinois, Urbana, February, 1975.

Sameh and Kuck [75b]. A. H. Sameh and D. J. Kuck, "Linear system solvers for parallel computers," Department of Computer Science, University of Illinois, Urbana, February, 1975.

Sameh and Layman [74]. A. H. Sameh and T. Layman, "Toward an Illiac IV library," contributed paper, Second Langley Conference on Sci. Comp., Virginia Beach, October, 1974.

Stevenson [75]. D. K. Stevenson, "Programming the Illiac," Department of Computer Science, Carnegie-Mellon University, to appear, 1975.

Stone [71]. H. S. Stone, "Parallel processing with the perfect shuffle," IEEE Trans. on Comp., vol. C-20, 1971, 153-161.

Stone [73a]. H. S. Stone, "An efficient parallel algorithm for the solution of a tridiagonal linear system of equations," J.ACM, vol. 20, 1973, pp. 27-38.

Stone [73b]. H. S. Stone, "Problems of parallel computation," in Complexity of Sequential and Parallel Numerical Algorithms, J. F. Traub, ed., Academic Press, N. Y., 1973, pp. 1-16.

Stone [75a]. H. S. Stone, "Parallel tridiagonal equation solvers," ACM Trans. on Math. Software, Vol. 1, 1975, pp. 289-307.

Stone [75b]. H. S. Stone, "Parallel computers," in Introduction to Computer Architecture, H. S. Stone, ed., Science Research Associates, Palo Alto, California, 1975, pp. 318-374.

Strassen [69]. V. Strassen, "Gaussian elimination is not optimal," Num. Math., vol. 13, 1969, pp. 354-356.

Swarztrauber [74]. P. N. Swarztrauber, "A direct method for the discrete solution of separable elliptic equations," SIAM J. Num. Anal., vol. 11, 1974, pp. 1136-1150.

Sweet [74]. R. A. Sweet, "A generalized cyclic reduction algorithm," SIAM J. Num. Anal., vol. 11, 1974, pp. 506-520.

Tewarson [68]. R. P. Tewarson, "Solution of linear equations with coefficient matrix in band form," BIT, vol. 8, 1968, pp. 53-58.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

Field	Entry
1. REPORT NUMBER	
2. GOVT ACCESSION NO.	
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle)	A SURVEY OF PARALLEL ALGORITHMS IN NUMERICAL LINEAR ALGEBRA
5. TYPE OF REPORT & PERIOD COVERED	Interim
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s)	Don Heller
8. CONTRACT OR GRANT NUMBER(s)	N00014-76-C-0370, NR044-422
9. PERFORMING ORGANIZATION NAME AND ADDRESS	Carnegie-Mellon University Computer Science Dept. Pittsburgh, PA 15213
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
11. CONTROLLING OFFICE NAME AND ADDRESS	Office of Naval Research Arlington, VA 22217
12. REPORT DATE	February 1976
13. NUMBER OF PAGES	81
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	
15. SECURITY CLASS (of this report)	UNCLASSIFIED
15a. DECLASSIFICATION DOWNGRADING SCHEDULE	

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release, distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The existence of parallel and pipeline computers has inspired a new approach to algorithmic analysis. Classical numerical methods are generally unable to exploit multiple processors and powerful vector-oriented hardware. Efficient parallel algorithms can be created by reformulating familiar algorithms or by discovering new ones, and the results are often surprising. A comprehensive survey of parallel techniques for problems in linear algebra is given. Specific topics include: relevant computer models and their comsequences, evaluation of ubiquitous arithmetic expressions, solution of linear systems of equations, and computation of eigenvalues.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Traub [73]. J. F. Traub, "Iterative solution of tridiagonal systems on parallel and vector computers," in Complexity of Sequential and Parallel Numerical Algorithms, J. F. Traub, ed., Academic Press, New York, 1973, pp. 49-82.

Trout [72]. H. R. G. Trout, "Parallel techniques," Department of Computer Science, University of Illinois, Urbana, October, 1972.

Varga [62]. R. S. Varga, Matrix Iterative Analysis, Prentice-Hall, Englewood Cliffs, New Jersey, 1962.

Viten'ko [68]. I. V. Viten'ko, "Optimum algorithms for adding and multiplying on computers with a floating point," USSR Comput. Math. and Math. Phys., vol. 8, 1968, pp. 183-195.

Ward [76]. R. C. Ward, "The QR algorithm and Hyman's method on vector computers," Math. Comp., Vol. 30, 1976, pp. 132-142.

Watson [72]. W. J. Watson, "The TI ASC, a highly modular and flexible supercomputer architecture," AFIPS Fall 1972, AFIPS Press, Montvale, N. J., vol. 41, pt. 1, pp. 221-229.

Widlund [72]. O. B. Widlund, "On the use of fast methods for separable finite difference equations for the solution of general elliptic problems," in Sparse Matrices and their Applications, D. J. Rose and R. A. Willoughby, eds., Plenum Press, N. Y., pp. 121-131.

Wilkinson [65]. J. H. Wilkinson, The Algebraic Eigenvalue Problem, Oxford University Press, London, 1965.

Winograd [70]. S. Winograd, "On the number of multiplications to compute certain functions," Comm. Pure and Appl. Math., vol. 23, 1970, pp. 165-179.

Winograd [75]. S. Winograd, "On the parallel evaluation of certain arithmetic expressions," J.ACM, vol. 22, 1975, pp. 477-492.

Wolfe [68]. P. Wolfe, chairman, "Panel discussion on new and needed work and open questions," in Sparse Matrix Proceedings, R. A. Willoughby, ed., IBM T. J. Watson Research Center, Yorktown Heights, N. Y., September, 1968.

Wulf and Bell [72]. W. A. Wulf and C. G. Bell, "C.mmp, a multi-mini-processor," AFIPS Fall 1972, AFIPS Press, Montvale, N. J., vol. 41, pt. 2, pp. 765-777.

Young [72]. D. M. Young, "Second-degree iterative methods for the solution of large linear systems," J. Approx. Th., vol. 5, 1972, pp. 137-148.

Zwackenberg [75]. R. G. Zwackenberg, "Vector extensions to LRLTRAN," SIGPLAN Notices, vol. 10, no. 3, March, 1975, pp. 77-86.