

ornl

**OAK RIDGE
NATIONAL
LABORATORY**

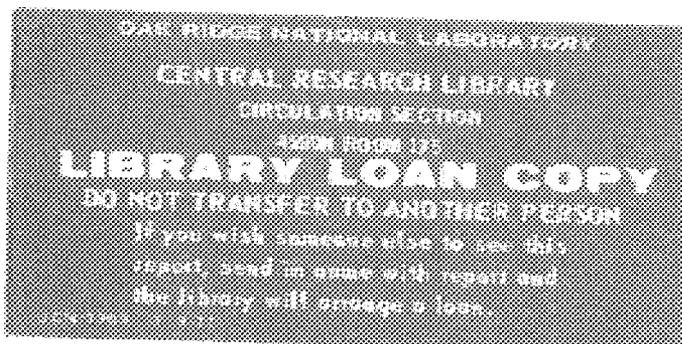
MARTIN MARIETTA



ORNL/TM-10384

Parallel Solution of Triangular Systems on Distributed-Memory Multiprocessors

M. T. Heath
C. H. Romine



OPERATED BY
MARTIN MARIETTA ENERGY SYSTEMS, INC.
FOR THE UNITED STATES
DEPARTMENT OF ENERGY

Printed in the United States of America. Available from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road, Springfield, Virginia 22161
NTIS price codes—Printed Copy: A04; Microfiche A01

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Engineering Physics and Mathematics Division

Mathematical Sciences Section

**PARALLEL SOLUTION OF TRIANGULAR SYSTEMS
ON DISTRIBUTED-MEMORY MULTIPROCESSORS**

M. T. Heath

C. H. Romine

Date Published - March 1987

Research was supported by the
Applied Mathematical Sciences Research Program
of the Office of Energy Research,
U. S. Department of Energy

Prepared by the
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
operated by
Martin Marietta Energy Systems, Inc.
for the
U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC05-84OR21400



3 4456 0153516 8

Table of Contents

Abstract.	1
1. Introduction.	1
1.1. Distributed-memory multiprocessors.	2
1.2. Parallel matrix computations.	2
2. Algorithms.	3
2.1. Serial algorithms.	3
2.2. Parallel algorithms.	4
2.3. Fan-out and fan-in algorithms.	5
2.4. Wavefront algorithms.	7
2.5. Cyclic algorithms.	9
3. Analysis	12
3.1. Fan-out and fan-in algorithms.	12
3.2. Cyclic algorithms.	13
3.3. Wavefront algorithms.	13
3.4. Comparison of models.	16
4. Numerical experiments.	17
4.1. Preliminary tests.	17
4.1.1. Connectivity.	17
4.1.2. Acceleration.	18
4.1.3. Mapping.	19
4.1.4. Segment size.	19
4.2. Comprehensive tests.	19
4.2.1. Performance as a function of p	19
4.2.2. Performance as a function of n	20
5. Conclusion.	20
References.	20
Appendix.	22
Illustrations.	34

Parallel Solution of Triangular Systems on Distributed-Memory Multiprocessors

Michael T. Heath* and Charles H. Romine*

Abstract. Several parallel algorithms are presented for solving triangular systems of linear equations on distributed-memory multiprocessors. New wavefront algorithms are developed for both row-oriented and column-oriented matrix storage. Performance of the new algorithms and several previously proposed algorithms is analyzed theoretically and illustrated empirically using implementations on commercially available hypercube multiprocessors.

1. Introduction. On conventional serial computers, the solution of triangular systems of linear equations is often thought of as an essentially trivial computation. The $O(n^2)$ arithmetic operations required for the triangular solution are usually a mere postscript to the dominant cost of the $O(n^3)$ arithmetic operations required to factor a general matrix into a product of triangular matrices. On the other hand, if repeated triangular solutions are required, for example, in computing the inverse of a matrix or in using a triangular preconditioner to accelerate an iterative method, then the cost of triangular solutions takes on greater significance.

On many types of multiprocessor architectures, particularly those with distributed memory, necessary communication among processors causes the relative costs of matrix factorization and triangular solution to be quite different from the serial case, with good efficiency being much more difficult to attain for triangular solution than for matrix factorization. Thus, on some multiprocessor architectures, the triangular solution phase can require a significant portion of the total time for solving a general system of linear equations and is therefore worthy of closer study than it has traditionally received. The surveys of Heller [10] and Ortega and Voigt [15] cite numerous parallel algorithms for solving triangular systems, but most of them require a very large number of processors (as many as $O(n^3)$), and are therefore of mainly theoretical interest.

In this paper we will examine and compare, both theoretically and empirically, several practical algorithms for solving triangular systems on various types of distributed-memory multiprocessors. Although progress in parallel triangular solution algorithms has lagged somewhat behind that for parallel factorization algorithms (see, *e.g.*, [7] and references therein), several recently proposed algorithms, as well as some new ones proposed here, can attain satisfactory efficiency on distributed-memory multiprocessors. In the remainder of this section we outline some background material on distributed-memory multiprocessors and on parallel matrix computations. Section 2 presents the motivation for and statement of several algorithms, including two new wavefront algorithms, for solving triangular systems of equations on such architectures. Section 3 contains theoretical analyses of the algorithms and Section 4 gives results of extensive numerical experiments comparing the algorithms on hypercubes.

* Mathematical Sciences Section, Oak Ridge National Laboratory, P.O. Box Y, Oak Ridge, Tennessee 37831. Research supported by the Applied Mathematical Sciences Research Program, Office of Energy Research, U.S. Department of Energy under contract DE-AC05-84OR21400 with Martin Marietta Energy Systems Inc.

1.1. Distributed-memory multiprocessors. In solving a problem on a multiprocessor system, the computational work is divided among the processors. Ideally, the workload should be evenly balanced and concurrency maximized so that all of the processors are kept busy doing useful work as much as possible. In a distributed-memory system, the problem data (*e.g.*, the matrix and right-hand-side vector) must also be partitioned and spread across the memories of individual processors. If one processor needs data stored in the memory of another, the necessary information is sent between the processors through a network interconnecting the processors. Possible interconnection schemes include bus, ring, grid, tree, and hypercube networks. Most of the triangular solution algorithms we will discuss can be implemented on many different types of multiprocessor networks. Hypercubes were used in all of our experiments, however, because of their ready availability and because many other networks are included as subsets (*see, e.g.*, [18]).

The cost of communication between adjacent processors in a message-passing system can usually be modeled with reasonable accuracy by the linear form

$$t = \alpha + \beta M, \quad (1.1)$$

where α is a start-up cost for any message independent of its length, β is the incremental cost per unit length, and M is the length of the message in bytes or words. There may also be a packetizing effect in some systems, in which large messages are broken up into smaller pieces for transmission. The relative magnitudes of α and β determine the most efficient message size for a given system. Thus, for example, if α is relatively large, then for a given communication volume, small messages will be much less efficient than large messages. Another important characteristic determining the overall efficiency of parallel algorithms is the relative cost of communication and computation. Thus, for example, if communication is relatively slow, then coarse-grain algorithms in which a relatively large amount of computation is done between communications will be more efficient than fine-grain algorithms. Typical measured communication parameters for some commercially available hypercubes are shown in Table 1, which is taken from [4]. The values shown for α and β were obtained by a least squares fit to data taken over a wide range of message sizes, while the communication/computation cost ratio is based on the cost of transmitting an eight-byte message vs. the cost of a floating-point multiplication. Note that the start-up cost is relatively high and communication is relatively slow compared to floating-point computation for all of these machines.

Table 1. *Approximate communication parameters for some commercially available hypercubes.*

Parameter	Ametek	Intel	Ncube
α (start-up cost in microseconds)	560	860	450
β (transfer cost in microseconds/byte)	9.5	1.8	2.4
communication/computation cost ratio	19	26	32

1.2. Parallel matrix computations. The above considerations have a direct and important bearing on the design of parallel algorithms for matrix factorization and triangular solution on distributed-memory multiprocessors. In particular, they explain why good efficiency is much more difficult to attain for triangular solution than for matrix factorization on multiprocessors having characteristics such as those shown in Table 1. Simply stated, triangular solution is inherently a finer-grain process that has an order of magnitude less computation over which to

amortize its significant communication cost. Moreover, the communication required by triangular solution algorithms tends to be composed of small messages, so that start-up overhead is a relatively large proportion of the total communication cost. The result is that on commercial hypercubes, communication cost in solving triangular systems tends to dominate. Thus, efficiency as high as 50% for triangular solution is difficult to attain, whereas efficiencies of 80-90% are easily attained for matrix factorization. These issues will be discussed in greater detail and illustrated in subsequent sections.

The characteristics of the target architecture critically affect the partitioning of matrices for factorization on a distributed-memory multiprocessor, and hence the distributed data structures with which subsequent triangular solutions will have to work. Several possible partitionings are plausible, including those by rows, columns, submatrices, or diagonals. Partitioning by diagonals can be effective for matrices having various special structures (*e.g.*, banded, circulant, Toeplitz), but is usually not optimal for general matrices. Partitioning by submatrices tends to minimize the resulting communication volume, since it minimizes the perimeter of the individual pieces [6]. However, it also tends to spread this volume over a relatively large number of messages, yielding poor performance on machines having high start-up costs for communication. *We will therefore restrict our attention to partitioning by rows and by columns.*

Given such a partitioning, the rows or columns can be mapped onto the processors in numerous ways, including wrap, block, and reflection mappings (see, *e.g.*, [7] for definitions and comparisons). Due to its good load balancing properties, an effective mapping in many contexts is given by wrapping (or interleaving) the rows or columns onto the processors, much as one would deal cards. One row or column is assigned to each processor (in some given order) until all processors are exhausted, whereupon the assignment returns to the first processor and continues through the processors again (in the same order), and so on until all rows or columns have been assigned. In a hypercube, taking the processors in the order given by a binary reflected Gray code ensures that adjacent rows or columns will be assigned to processors that are physically connected. The correctness of most of the triangular solution algorithms we will discuss is independent of the mapping employed, although performance is often best with wrap mapping; in a few cases noted below, wrap mapping is essential to the correctness of the algorithm. Insensitivity to the mapping is a desirable trait because in some cases, for example, when pivoting is required during the factorization to maintain numerical stability, it may not be convenient to preserve a preassigned mapping.

2. Algorithms. We will present two basic types of parallel algorithms for solving triangular systems on distributed-memory multiprocessors; in addition, there are many variations on each algorithm, some major and some minor. One fundamental distinction is whether the matrix is stored by rows or by columns: row-oriented algorithms and column-oriented algorithms come in "dual" pairs. The triangular matrix can be either lower or upper triangular, so that either forward or back substitution is required, respectively. The interconnection network among processors can be used in various ways to meet the communication needs of a given basic algorithm, especially with a flexible network such as a hypercube. Finally, there are various techniques that can potentially enhance performance, at the expense of complicating the basic algorithms somewhat.

2.1. Serial algorithms. In order to establish notation and help motivate the parallel algorithms, we first state two serial algorithms that have the same dual relationship we will see later in pairs of parallel algorithms. For definiteness, we consider the solution of the lower triangular linear system

$$Lx = b,$$

where L is a lower triangular matrix of order n , b is a known right-hand-side vector of dimension n , and x is the unknown solution vector of dimension n . On a serial computer, this system is solved by forward substitution using a doubly nested loop that can be ordered in either of two ways:

Vector-sum algorithm

```

for  $j = 1$  to  $n$ 
   $x_j = b_j / L_{jj}$ 
  for  $i = j + 1$  to  $n$ 
     $b_i = b_i - x_j L_{ij}$ 

```

Scalar-product algorithm

```

for  $i = 1$  to  $n$ 
  for  $j = 1$  to  $i - 1$ 
     $b_i = b_i - x_j L_{ij}$ 
   $x_i = b_i / L_{ii}$ 

```

The names we use for these algorithms were chosen to emphasize the duality between them and to characterize the computations in their respective inner loops. In the vector-sum algorithm, the inner loop updates the vector b by a multiple of a column of L . In the scalar-product algorithm, the inner loop updates a single component of b by an inner-product of a row of L and the portion of x already computed. (It is tempting to use the names outer-product and inner-product for the two algorithms, but the inner-loop computation in the vector-sum algorithm is not really an outer product.)

2.2. Parallel algorithms. In partitioning the data and computational work for solving triangular systems on distributed-memory multiprocessors, we will consider four possibilities: partitioning the matrix by rows or by columns, and partitioning the work according to the inner loop or the outer loop. Due to data dependencies in the problem, we do not have complete freedom to exploit parallelism in solving a triangular system of equations. In particular, each component of the solution vector must be *completed* in the correct sequential order dictated by forward or back substitution. Thus, our only opportunities for exploiting parallelism are in the computations that modify the right-hand-side vector prior to the division that determines the next component of the solution. We may exploit parallelism, while still satisfying this precedence constraint, in two distinct ways: in the inner loop or in the outer loop. The work within the inner loop can be partitioned and distributed across the processors while maintaining strictly serial execution of the outer loop. Alternatively, successive executions of the outer loop can be partially overlapped so that different processors are working on different components of the solution simultaneously, yet the sequential order of completion is still honored. We present algorithms of both types below.

The choice of inner loop or outer loop parallelism determines the appropriate partitioning of the matrix for a given algorithm. To partition the work in the inner loop of the vector-sum algorithm, the matrix and right-hand side must be partitioned and distributed by rows, so that each processor can update the portion of b it owns by the appropriate column of L in those rows it owns, and these updates can occur simultaneously. Similarly, to partition the work in the inner loop of the scalar-product algorithm, the matrix must be partitioned and distributed by columns, so that each processor can contribute to the inner-product those terms corresponding to the columns of L it owns, and these "subtotals" can be computed simultaneously.

Exploiting parallelism in the outer loop suggests the opposite data organization. Thus, in the vector-sum algorithm, if the matrix is partitioned and distributed by columns, then the updating due to each column must be computed solely by the one processor that owns it. Similarly, in the scalar-product algorithm, if the matrix is partitioned and distributed by rows, then each inner product will be computed entirely by the one processor that owns the corresponding row. Despite this serial execution of the inner loops, parallelism is attainable in both cases by overlapping the work on successive components of the solution, as we will demonstrate below.

2.3. Fan-out and fan-in algorithms. Partitioning the work in the inner loop of the vector-sum and scalar-product algorithms leads to what we will call the fan-out and fan-in parallel algorithms, respectively, for solving triangular systems. The names refer to the communication pattern of these algorithms, in which information is either broadcast from one processor to all others or gathered into one processor from all others. The duality between fan-out and fan-in communication matches that between the vector-sum and scalar-product algorithms, as will soon be made clear.

In implementing the vector-sum algorithm using fan-out communication, we assume that the matrix L and vector b are distributed among the processors by rows; for each processor, the row numbers it is assigned are contained in the set *myrows*. (All of the sets in the algorithms to follow have a natural order associated with them, and when we indicate a loop over the elements of a set, the elements are to be taken in that natural order.) We adopt the notation that processor *map*(j) is assigned row j of L (or column j if the assignment is by columns), and *map* is known by all processors. The result of the operation *fan-out*(t , *root*) is that processor *root* sends, and all other processors receive, t . The details of the fan-out broadcast depend on the particular interconnection network. The fan-out operation provides all necessary synchronization in the algorithm below, since we assume that each processor waits until an expected message arrives before proceeding. The fan-out vector-sum algorithm was first stated by Kuck [11], who called it the "column-sweep" algorithm. The basic algorithm for each processor is as follows:

Fan-out vector-sum algorithm

```

for  $j=1$  to  $n$  do
  if  $j \in \text{myrows}$  then  $x_j = b_j / L_{jj}$ 
  fan-out ( $x_j$ , map( $j$ ))
  for  $i \in \text{myrows}$ ,  $i > j$ ,
     $b_i = b_i - x_j L_{ij}$ 

```

In implementing the scalar-product algorithm using fan-in communication, we assume that the matrix L is distributed among the processors by columns; for each processor, the column numbers it is assigned are contained in the set *mycols*. The components of b are distributed among the processors correspondingly. The result of the operation *fan-in*(t , *root*) is that processor *root* receives the *sum* of the t 's over all processors. The details of the fan-in operation depend on the particular interconnection network. The fan-in algorithm is due to Romine and Ortega [17], [16]. The basic algorithm for each processor is as follows:

Fan-in scalar-product algorithm

```

for  $i = 1$  to  $n$  do
   $t = 0$ 
  for  $j \in \text{mycols}, j < i,$ 
     $t = t + x_j L_{ij}$ 
   $s = \text{fan-in}(t, \text{map}(i))$ 
  if  $i \in \text{mycols}$  then  $x_i = (b_i - s) / L_{ii}$ 

```

With most multiprocessor networks, the fan-out or fan-in of information is implemented by means of a spanning tree, with information either originating or culminating, respectively, at the root. The necessary flow of information is implemented by a “bucket brigade” from processor to processor. In a richly connected network, such as a hypercube, the terms “fan-out” and “fan-in” can be interpreted quite literally, and global communication is quite efficient. In more sparsely connected topologies, such as a ring, the degree of fan-out or fan-in in the spanning tree is necessarily limited, so that the height of the spanning tree (equivalently, the diameter of the network) is much greater. In a fully connected network (*e.g.*, a cross-bar), or one which efficiently emulates full connections, the global dissemination or collection of information can be accomplished directly between any node and all other nodes. With a bus architecture, the duality between fan-out and fan-in breaks down somewhat, since global broadcasting can be accomplished with a single message, while collecting global information into a single processor requires multiple messages. For our experiments on hypercubes, we have implemented both fan-out and fan-in algorithms using four different types of communication:

- minimal spanning tree (logarithmic fan-out/fan-in)
- unidirectional ring
- bidirectional ring
- complete connectivity (simulated by automatic routing)

The relative performance of these options will be examined in Sections 3 and 4.

The fan-out and fan-in algorithms were stated above in their simplest forms. Both are subject to modifications that can potentially enhance performance. Consider the j -th step of the fan-out algorithm, in which the computation of x_j is completed. Following the fan-out of x_j , each processor updates *all* of its components of b ; not until the next iteration of the outer loop does each processor check whether it is responsible for computing x_{j+1} . Processor $\text{map}(j+1)$ could in fact have computed x_{j+1} as soon as b_{j+1} was updated, *before* updating its remaining components of b , and therefore could have started the next fan-out somewhat earlier (of course, it must eventually complete the postponed updates). Such a strategy of sending out results needed by other processors at the earliest possible moment is intended to minimize possible idle waiting times in the receiving processors. The extent to which the potential gain in performance is realized in practice depends on a number of factors, including the particular connectivity used for the fan-out and the particular mapping of the rows onto the processors. Since messages will arrive at unpredictable times, the implementation of this strategy requires a communication system capable of queuing incoming messages in system buffers until the user program is ready to receive them, and this capability may not be available in some message-passing multiprocessors. By contrast, the basic fan-out algorithm stated above is easily implemented on systems that support only synchronous communication.

By analogy with the “send-ahead” strategy for accelerating the fan-out algorithm, Cleary has suggested a “compute-ahead” strategy for accelerating the fan-in algorithm [3]. His idea is based on the observation that processor $\text{map}(i)$, because it must compute x_i , will be the last

processor to compute its contribution to the inner product for the next component of the solution, x_{i+1} . In order to avoid being the bottleneck at the $i+1$ -st stage, processor $map(i)$ could go ahead and compute most of its contribution to the $i+1$ -st inner product (all but the one term corresponding to x_i) while it would otherwise be idle waiting for the contributions from the other processors to the i -th inner product. After computing x_i , processor $map(i)$ will then require only a trivial amount of computing to complete its contribution to the $i+1$ -st inner product and therefore will not unnecessarily delay the computation of x_{i+1} . Again, the effectiveness of this potential performance enhancement depends on the communication pattern used and the mapping of the columns onto the processors, among other factors. Note that both of these modifications introduce some degree of pipelining into the basic fan-out and fan-in algorithms by partially overlapping successive executions of the outer loops, although their dominant source of parallelism is still found in the inner loops.

2.4. Wavefront algorithms. We turn now to algorithms whose source of parallelism is the overlapping, or pipelining, of work on many components of the solution at once. First, consider the vector-sum algorithm with the matrix partitioned and distributed by columns. After processor $map(j)$ computes x_j , the work of updating b cannot be shared with the other processors, since only processor $map(j)$ has direct access to column j of L . Serial execution of the inner loop by a single processor, however, would limit us to an entirely serial algorithm unless we can somehow overlap successive iterations of the outer loop so that all processors are working simultaneously on different components of the solution. This can be accomplished by breaking the update vector into segments and pipelining the segments through the processors in a wavefront fashion. In the discussion and algorithm to follow, we introduce the n -vector z in which to accumulate the updates to b , so that b itself need not be collected into any one processor, but instead its components remain distributed among the processors in the same manner as the columns of L .

Processor $map(1)$ first computes x_1 , then proceeds to compute the components $z_i = x_1 L_{i1}$ of the update vector z . After computing the first σ such components, $1 \leq \sigma \leq n$, processor $map(1)$ sends them on to processor $map(2)$ so that the latter can compute x_2 and begin further updates of z . Meanwhile, processor $map(1)$ has resumed work on the next σ components of z , and so on through successive segments of σ components each (if σ does not divide n , then the last segment of z may have fewer than σ components). As soon as it completes updating each segment of z , each processor forwards the segment to the next processor. Depending on a number of factors (the segment size, the mapping of the columns onto the processors, the relative speeds of communication and computation, etc.), it may be possible for all of the processors to become busy simultaneously, each working on a different segment. In effect, the segment size σ is an adjustable parameter that controls the granularity of the algorithm. A smaller segment size tends to increase the potential amount of concurrency, but also increases the number of messages required, which may be prohibitively expensive on a message-passing system with high start-up cost. A larger segment size, on the other hand, reduces the number of messages, but may severely limit the amount of concurrency attainable (in the extreme, $\sigma = n$ gives a purely serial algorithm). Thus, there is a trade-off in performance as segment size varies, and the optimal value depends on the characteristics of the particular architecture on which the algorithm is implemented. We will address this issue analytically in Section 3 and empirically in Section 4.

We now state the wavefront vector-sum algorithm somewhat more formally, although in the interest of brevity and simplicity we omit the initialization phase of the algorithm, checks for termination conditions, and the details of computing how many segments each processor should expect at each stage. These details can be found in the program listings in the appendix below. In the following pseudo-code, *segment* is a set containing at most σ components of the update vector z . The algorithm for each processor is as follows:

Wavefront vector-sum algorithm

```

for  $j \in \text{mycols}$  do
  for  $k = 1$  to #segments
    receive segment
    if  $k = 1$  then do
       $x_j = (b_j - z_j) / L_{jj}$ 
       $\text{segment} = \text{segment} - \{z_j\}$ 
    for  $z_i \in \text{segment}$  do
       $z_i = z_i + x_j L_{ij}$ 
    if  $|\text{segment}| > 0$  then send segment to processor  $\text{map}(j+1)$ 

```

Several additional points about this algorithm bear mentioning. We no longer have a simple serial outer loop that goes through all values from 1 to n , since each processor no longer contributes directly to the computation of every component of the solution. Another interesting feature is that the “first” segment shrinks by one element after each component of the solution is computed, disappearing entirely after σ steps, at which time the next segment becomes the “first” segment. By the time the algorithm reaches the final component of the solution, only one segment remains and it contains only one element. This declining communication volume as the algorithm progresses offsets to some extent the undesirable property that segments may pass through the same processor multiple times (albeit in altered form).

A fundamental assumption behind the wavefront algorithm is that it is always possible to send messages from processor $\text{map}(j)$ to processor $\text{map}(j+1)$. Whether processors holding contiguous columns of the matrix are physically adjacent in the multiprocessor network depends on the interconnection scheme and the mapping of columns to processors. In a ring, for example, such direct communication would be possible only if the mapping were compatible with the ordering of the processors in the ring. Even if processors are not directly connected, however, some systems provide automatic routing through intermediate nodes between arbitrary sender and destination, although there is usually a performance penalty associated with longer paths. In most hypercubes, for example, the wavefront vector-sum algorithm is valid for any mapping, but performs best using the wrap mapping with Gray code ordering.

The wavefront idea can also be applied to the scalar-product algorithm, producing a parallel algorithm that is the dual of the wavefront vector-sum algorithm. Consider the scalar-product algorithm with the matrix partitioned and distributed by rows. Computation of the i -th inner product cannot be shared, since only processor $\text{map}(i)$ has direct access to row i of L . Thus, once again, if we are to attain any concurrency it must be through the overlapping of work on different components of the solution by multiple processors. The dual concept here is to break the solution vector x into segments that are pipelined through the processors in a wavefront fashion.

Processor $\text{map}(1)$ first computes x_1 and sends it to processor $\text{map}(2)$. Processor $\text{map}(2)$ can then compute the second inner product and x_2 . Processor $\text{map}(2)$ now sends both x_1 and x_2 (i.e., a segment of x of size 2) to processor $\text{map}(3)$, which uses them in computing the third inner product. This process continues (serially in this early stage) until σ components of x have been computed, at which point the receiving processors begin forwarding any full-sized segments *before* they are used in computing inner products, so that subsequent processors can get started on their inner products. Forwarding of the one segment that is currently incomplete is delayed until the next component of x can be appended to it. Depending on the segment size and other factors, it may be possible for all processors to become busy simultaneously, each working on a different segment. As before, the segment size σ is an adjustable parameter that determines the granularity of the algorithm, with a similar

performance trade-off between communication and concurrency as σ varies. The optimal value of σ depends on the characteristics of the particular architecture. We now give a somewhat more formal statement of the algorithm, again omitting such details as initialization and termination. The algorithm for each processor is as follows:

Wavefront scalar-product algorithm

```

for  $i \in \text{myrows}$  do
  for  $k = 1$  to  $\# \text{segments} - 1$ 
    receive segment
    send segment to processor  $\text{map}(i+1)$ 
    for  $x_j \in \text{segment}$  do
       $b_i = b_i - x_j L_{ij}$ 
    receive segment /* last segment may be empty */
    for  $x_j \in \text{segment}$  do
       $b_i = b_i - x_j L_{ij}$ 
     $x_i = b_i / L_{ii}$ 
     $\text{segment} = \text{segment} \cup \{x_i\}$ 
    send segment to processor  $\text{map}(i+1)$ 

```

Similar remarks apply to this algorithm as were made for the wavefront vector-sum algorithm, except that many of its features are the “duals” of those for the earlier algorithm. Thus, for example, instead of starting with a full set of segments that shrink and eventually disappear until no segments remain, the scalar-product version starts with no segments, but segments appear and grow until there is a full set of them. Again, the algorithm depends on the ability to send messages between processors holding contiguous rows of the matrix, which may or may not be possible and may or may not be direct, depending on the interconnection network and the mapping of rows to processors. On most hypercubes, the wavefront scalar-product algorithm is valid for any mapping, but performs best using the wrap mapping with Gray code ordering. We will see in Section 4 that the two wavefront algorithms performed similarly, but by no means identically, in our experiments.

To the best of our knowledge, both of the wavefront algorithms presented here are new, although many other wavefront algorithms (and closely related dataflow and systolic algorithms) have been proposed for various matrix computations (see, *e.g.*, [12], [14]), and the idea of an adjustable segment size has also been used in other contexts (see, *e.g.*, [9]). Parallel wavefront algorithms for solving triangular systems were proposed by Evans and Dunbar [5], but their algorithms have a very different flavor from those presented here. In particular, their algorithms require at least $(n-1)/2$ processors and they involve a rather complicated reassignment of processors to rows as the algorithms proceed that would be difficult to implement efficiently in a distributed-memory environment.

2.5. Cyclic algorithms. The motivation for the final pair of dual algorithms we present is to attempt to gain performance by taking specific advantage of the wrap mapping to attain minimal communication volume. The resulting algorithms are therefore less widely applicable than the other algorithms presented above, but they can have very attractive performance characteristics under some conditions. We use the term “cyclic” to describe these algorithms for reasons that will become clear shortly. Cyclic algorithms bear a superficial resemblance to the wavefront algorithms in that they are based on partitioning the outer loop of the vector-sum and scalar-product algorithms, and they send a segment of z or x between processors. Their motivating philosophy and performance characteristics, however, are entirely different from the wavefront algorithms. In particular, rather than having a variable number of segments of an adjustable length, cyclic algorithms circulate a single segment of a fixed and specially chosen size, namely $p-1$, where p is the number of processors.

We first describe the cyclic vector-sum algorithm, with the matrix partitioned and distributed by columns according to the wrap mapping. This algorithm is due to Li and Coleman [13]. A segment of size $p-1$ passes from processor to processor, one step for each column of the matrix, cycling through all of the other $p-1$ processors before returning to a given processor (hence the name cyclic). The function of the segment is to contain partially accumulated components of the update vector z . At stage j of the algorithm, processor $map(j)$ receives the segment from processor $map(j-1)$ and uses its first element (which has accumulated all necessary prior updates except that computed by processor $map(j)$ itself) to determine x_j . Processor $map(j)$ then modifies the segment by deleting the first element, updating the remaining $p-1$ elements, and appending a new element to begin accumulation of the next component of z . Processor $map(j)$ then sends the segment to processor $map(j+1)$, where a similar process is repeated. After forwarding the modified segment, processor $map(j)$ then computes the remaining update components due to x_j , which will be needed when the segment returns to processor $map(j)$ again. These latter update computations, which take place in each processor while the segment passes through the other processors, provide the source of concurrency in the algorithm, since the computations on the segment itself are strictly serial.

We now state the cyclic vector-sum algorithm somewhat more formally. We need to make a distinction between the updates that accumulate in the elements of the segment and the updates computed by each processor while the segment is circulating elsewhere. We denote the former by the vector z and the latter by the vector t . For convenience, we use the corresponding row numbers in the matrix as subscripts for the elements z_i of the segment; in a real program one would use relative positions in a smaller array (see program listing in appendix). Again, for simplicity and brevity, we omit details such as initialization and termination conditions. In particular, toward the end of the algorithm (for the last $n-p+1$ columns), the segment is less than $p-1$ in size, shrinking by one element each time until termination. The algorithm for each processor is as follows:

Cyclic vector-sum algorithm

```

for  $j \in mycols$  do
  receive segment
   $x_j = (b_j - z_j - t_j) / L_{jj}$ 
  segment = segment -  $\{z_j\}$ 
  for  $z_i \in segment$  do
     $z_i = z_i + t_i + x_j L_{ij}$ 
   $z_{j+p-1} = t_{j+p-1} + x_j L_{j+p-1,j}$ 
  segment = segment  $\cup \{z_{j+p-1}\}$ 
  send segment to processor  $map(j+1)$ 
  for  $i = j+p$  to  $n$ 
     $t_i = t_i + x_j L_{ij}$ 

```

We turn now to the cyclic scalar-product algorithm, with the matrix partitioned and distributed by rows according to the wrap mapping. This algorithm is due to Chamberlain [1]. In this algorithm, the segment that passes from processor to processor contains components of the solution vector x . At stage i of the algorithm, processor $map(i)$ receives the segment from processor $map(i-1)$ and uses the elements of x it contains to complete the i -th inner product (processor $map(i)$ will have previously accumulated all other terms in the inner product while the segment was circulating through the other processors), so that x_i can then be computed. Processor $map(i)$ now modifies the segment by deleting the first element and appending the new element x_i just computed. Processor $map(i)$ then sends the segment to processor $map(i+1)$, where a similar process is repeated. After forwarding the modified segment, processor $map(i)$ then computes the partial inner products that use the components

of the segment, which will be further accumulated when the segment returns to processor $map(i)$ again. These latter computations, which take place in each processor while the segment passes through the other processors, provide the source of concurrency in the algorithm, since the computations on the segment itself are strictly serial.

We now state the cyclic scalar-product algorithm somewhat more formally. For convenience, we use the actual elements of the solution vector x to denote the elements of the segment; in a real program one would use relative positions in a smaller array (see program listing in appendix). Again, we omit details such as initialization and termination conditions. In particular, in the beginning of the algorithm (for the first $p-1$ rows), the segment contains fewer than $p-1$ elements; it grows by one element each time until it contains $p-1$ elements, where it remains fixed in size thereafter. The algorithm for each processor is as follows:

Cyclic scalar-product algorithm

```

for  $i \in myrows$  do
  receive segment
  for  $x_j \in segment$  do
     $b_i = b_i - x_j L_{ij}$ 
     $x_i = b_i / L_{ii}$ 
     $segment = segment - \{x_{i-p}\} \cup \{x_i\}$ 
  send segment to processor  $map(i+1)$ 
  for  $m \in myrows, m > i,$ 
    for  $x_j \in segment$  do
       $b_m = b_m - x_j L_{mj}$ 

```

In both cyclic algorithms, the moving segment must pass through *all* other processors before returning to any given processor. Thus, the correctness of these algorithms depends on the use of the wrap mapping. The cyclic algorithms, like the wavefront algorithms, also depend on the ability to send messages between pairs of processors containing consecutive rows or columns. For this reason, the cyclic algorithms were originally proposed as algorithms for rings or hypercubes using the wrap mapping with Gray code ordering, but they might also be suitable for other message-passing multiprocessors as well. For a bus-based distributed-memory system, in particular, the cyclic algorithms have the highly desirable property that only one pair of processors is communicating at any one time. In contrast, most of the other algorithms we have considered would encounter contention for the bus due to simultaneous message traffic. On the other hand, this lack of concurrency in communication implies that the cyclic algorithms use very little of the available bandwidth in more richly connected multiprocessor networks.

The efficiency of the cyclic algorithms depends on whether the time interval between successive appearances of the segment at any given processor is balanced by the time required for the intervening computations in that processor. Since the latency of the segment cycle tends to grow with the number of processors, there may be a point for a given problem size at which the use of more processors will increase rather than decrease the execution time. The proposers of the cyclic algorithms have suggested modifications to both algorithms aimed at improving their performance for large numbers of processors [1], [13]. These modifications, which include reorganizing the computations for better load balancing and breaking a hypercube into several subrings, have not been worked out in detail, however, and we have not implemented them. We have also considered a hybrid combination of the cyclic and wavefront algorithms that would attempt to combine the best features of both, but this also remains unimplemented. Our experiments reported in Section 4 used only the basic wavefront and cyclic algorithms.

3. Analysis. In this section we present theoretical analyses of the algorithms described in Section 2. For those algorithms whose analyses have appeared in the literature, we merely cite known results; readers should consult the original references for further details. For the new wavefront algorithms, we present a more detailed analysis. As originally presented, the analyses of the various algorithms have been based on a variety of models of communication and computation. To make the theoretical results for different algorithms more directly comparable, we couch all performance estimates in terms of the model of communication given by (1.1). This approach provides a convenient means of accounting for the different message sizes employed by the various algorithms. In addition, we use *flops* (linked scalar multiply-add pairs of the form $ax+y$) as the basic units of arithmetic computation. For a given machine, we assume that one *flop* (including any necessary indexing and addressing overhead) requires f microseconds. For compatibility of units, we take β in (1.1) to be the message transfer cost per *word* rather than per byte.

In modeling performance for a specific machine, we use the machine constants shown in Table 2. These differ slightly from those in Table 1 because in our experiments we used a faster version of the Ncube hypercube than was used in [4] (8 MHz clock speed instead of 7 MHz) and because we restricted the least squares fit of (1.1) to the range of message lengths actually used in our experiments.

Table 2. *Machine parameters used in theoretical models of performance on hypercubes.*

Parameter	Ncube	Intel
α (microseconds)	384	1108
β (microseconds/word)	10.4	2.9
f (microseconds/flop)	35	70

3.1. Fan-out and fan-in algorithms. The performance of the fan-out and fan-in algorithms is analyzed in detail in [17] and [16] for the case in which logarithmic hypercube communication is used. Upper bounds on execution time are obtained by assuming that there is no overlap between successive stages of the algorithms, nor between communication and computation. Under these assumptions, an upper bound on execution time for the row-oriented fan-out algorithm is

$$T = \frac{1}{2p}(n^2 + 2np - 2n)f + (n-1)(\log_2 p)(\alpha + \beta). \quad (3.1)$$

Similarly, an upper bound on execution time for the column-oriented fan-in algorithm is

$$T = \frac{1}{2p}(n^2 + (2 + \log_2 p)np - 2n)f + (n-1)(\log_2 p)(\alpha + \beta), \quad (3.2)$$

which is the same as for the fan-out algorithm except for an additional $\frac{1}{2}n(\log_2 p)f$ to account for the sums done during the fan-in.

These upper bounds can be adapted to other communication schemes simply by replacing the $\log_2 p$ latency of the cube fan-out/fan-in by another appropriate maximum path length. However, the validity of the underlying assumptions of nonoverlapping stages and nonoverlapping communication and computation becomes more untenable for some of these communication schemes. With logarithmic fan-out or fan-in, these assumptions are not grossly in error, due to the fact that many of the processors are occupied with forwarding messages

throughout the $\log_2 p$ stages of the fan-out or fan-in. But with a ring, for example, each processor forwards only one message and can then resume computation. Thus, if the matrix is mapped onto the ring in a compatible order so that successive stages can be pipelined, the high communication latency of ring fan-out or fan-in can be almost completely masked (as will be shown experimentally below). Moreover, many stages may be in progress simultaneously, thereby significantly reducing execution time below the value that a pessimistic upper bound such as (3.1) or (3.2) would suggest. Use of the "compute-ahead" or "send-ahead" acceleration techniques discussed in Section 2.3 would further invalidate the nonoverlapping assumptions.

In a somewhat more subtle form, these effects also have a bearing on the relative performance of the fan-out and fan-in algorithms using logarithmic hypercube communication. The slightly larger upper bound for the cube fan-in algorithm compared to the cube fan-out algorithm is offset by an effect discussed in detail in [16]. Specifically, the fan-in, which originates at the leaves and terminates at the root of the spanning tree, frees more processors earlier to resume computing than the fan-out, which originates at the root and terminates at the leaves. The fan-in algorithm should therefore permit a greater degree of overlapping than the fan-out algorithm, thereby enhancing performance. Given these offsetting factors, either algorithm could be superior, depending on the specific machine and implementation.

3.2. Cyclic algorithms. The performance of the column-oriented cyclic algorithm is analyzed in [13]. It is shown that for a given number of processors p , the execution time of the algorithm increases linearly with n until n reaches a threshold, and increases quadratically thereafter. More specifically,

$$T = [n(t_p + p) - \frac{1}{2} p(p-1) - t_p] f \quad \text{whenever } n \leq p(t_p + p), \quad (3.3a)$$

and

$$T = \left[\frac{1}{2p} (n^2 + np) + \frac{p}{2} ((t_p + p)^2 - t_p^2 - p + 1) - t_p \right] f \quad \text{otherwise,} \quad (3.3b)$$

where t_p is the cost, measured in *flops*, of sending a message of length $p-1$ to a neighboring processor. Thus, in our model of communication performance, $t_p = (\alpha + \beta(p-1))/f$. For the hypercubes listed in Table 1, the minimum value for t_p is approximately 12.

A rough calculation is given in [13] of the value of n required to achieve 50 percent efficiency with the column-oriented cyclic algorithm using a given number of processors. This value is approximately $n = p(t_p + p)$, which suggests that the algorithm will be highly efficient for small p , but much less efficient for large p . For example, if $p = 8$ and $t_p = 12$, then this model predicts that 50 percent efficiency will be achieved for $n = 160$. However, for $p = 64$, n must be greater than 4800 for the algorithm to achieve an efficiency of 50 percent. There appears to be something of a paradox in the two-phase performance behavior given by (3.3): one would expect linear behavior to be superior to quadratic behavior, yet it is when the linear portion is brief that the algorithm performs well, while a lengthier linear portion indicates relatively poor performance. The explanation is that since the underlying computational task is $O(n^2)$, linear behavior can hold only when communication dominates computation, so that an early transition to quadratic behavior is necessary for high efficiency.

Chamberlain [1] does not provide a detailed performance analysis of the row-oriented cyclic algorithm. However, the characteristics of that algorithm are sufficiently similar to the column-oriented version that similar behavior should be expected.

3.3. Wavefront algorithms. Since the wavefront algorithms are new, we will give a more detailed analysis of their performance. Unlike the other algorithms, the performance of the

wavefront algorithms depends upon the choice of an adjustable parameter, the segment size σ . In addition to describing the performance of the algorithms, the theoretical performance models should also enable us to predict the optimal choice of σ to minimize execution time.

We begin by analyzing the column-oriented wavefront algorithm. We assume that the columns are distributed to the processors using a wrap mapping in Gray code order. Several important observations will make the behavior of the algorithm clear. First, each segment can be worked on by only one processor at a time, and the ideal situation is to have all processors working simultaneously on different segments. Hence, the segment size σ should be chosen so that there are at least p segments (*i.e.*, $\sigma \leq n/p$). Second, once the start-up phase of the algorithm is over so that all processors have become busy, then as long as there are at least p segments remaining (recall that segments disappear throughout the algorithm), all processors will remain busy. (This does not imply that one should take $\sigma = 1$ to maximize the number of segments. While this would keep all processors busy throughout most of the algorithm, almost all of their time would be spent performing communication unless the latter were extraordinarily inexpensive). Third, when fewer than p segments remain, the time required for the last segment to make one circuit through all the processors is independent of the number of segments remaining, since each processor will be ready to receive this segment when it is sent. Fourth, at any stage of the algorithm, all but (perhaps) the first of the segments will be of length σ . Thus, to simplify the model, we assume that the cost of each communication is $\alpha + \beta\sigma$, and the cost of the operations performed on each segment is σf .

To simplify notation in the following argument, we will number both the processors and the columns of the matrix starting from 0 rather than 1. For convenience, we model the total execution time as the total execution time of the first processor (*i.e.*, processor $map(0)$, which under the wrap mapping is processor 0) and assume that p divides n . The execution time can be split into two parts, depending upon whether there are at least p segments remaining. We must determine which of the columns contained in processor 0 is the first to have only p segments remaining. Under the wrap mapping, the first processor contains all columns kp ($0 \leq k < n/p$). Column kp will have $\lceil (n - kp)/\sigma \rceil$ segments. Setting this equal to p we see that when $k = (n/p) - \sigma$, column kp will have p segments.

To compute the first part of the execution time (when the number of segments is greater than p), we now let k range from 0 to $(n/p) - \sigma$. For each column kp in processor 0, there are $(n - kp)/\sigma$ segments to be processed. Hence, since no idle time occurs during this phase of the computation, the execution time is

$$\sum_{k=0}^{\frac{n}{p}-\sigma} \left(\frac{n - kp}{\sigma} \right) (\alpha + \beta\sigma + f\sigma). \quad (3.4)$$

Once fewer than p segments remain, we must account for the induced idle time in processor 0. Since we have noted that the time required for the final segment to make one circuit through all the processors is now independent of the number of segments, we can model the execution time by assuming that p segments remain until the computation is complete. That is, regardless of the number of remaining segments, the time required for the last segment to return to processor 0, including the idle time, is the same as if p segments remained. This yields

$$\sum_{k=\frac{n}{p}-\sigma+1}^{\frac{n}{p}-1} p(\alpha + \beta\sigma + f\sigma) \quad (3.5)$$

as the expression for the second part of the execution time.

Adding the expressions in (3.4) and (3.5) together to obtain the total execution time yields

$$T = \frac{1}{2p}(n^2 + np + \sigma(\sigma - 1)p^2) \left(\frac{\alpha}{\sigma} + \beta + f \right). \quad (3.6)$$

The performance model of the column-oriented wavefront algorithm given by (3.6) exhibits the expected trade-off between communication and computation as the segment length varies. Specifically, a large value of σ reduces the communication start-up cost (given by the α term) at the expense of greater computation cost, while a small value of σ does the opposite. Fig. 1a shows the execution time as a function of σ predicted by (3.6) for a typical problem ($n = 500$) using various numbers of processors and the Ncube machine parameters. (In producing Fig. 1a, whenever $\sigma > n/p$, the term given by (3.4) drops out and the model reduces to linear behavior given by $T = n(\alpha + \beta\sigma + f\sigma)$). Fig. 1a should be compared to Fig. 4a in Section 4, which shows the same curves resulting from actual runs on the Ncube hypercube.

To determine the optimal segment size to minimize execution time, we differentiate (3.6) with respect to σ and equate to 0, producing (after some algebraic manipulation)

$$2(\beta + f)p\sigma^3 + (\alpha - (\beta + f))p\sigma^2 - \alpha n \left(\frac{n}{p} + 1 \right) = 0. \quad (3.7)$$

(These calculations have been verified by the Maple symbolic algebra package [2].) For convenience, we define the machine constant $K = \alpha/(\beta + f)$, so that (3.7) is simplified to

$$2\sigma^3 + (K - 1)\sigma^2 - \frac{Kn}{p} \left(\frac{n}{p} + 1 \right) = 0. \quad (3.8)$$

For the Ncube and Intel hypercubes, the constant K has the values 8.5 and 15.2, respectively. Using these values for K , Newton's method was applied to find the roots of equation (3.8). In each case, only one real root was found, which was taken to be the optimal segment size σ . Table 3 gives the computed values of σ for various n and p using the values of K corresponding to the two hypercubes. These theoretical results based on the model are compared with numerical experiments in Section 4.

Table 3. Estimated optimal values of segment size σ for the wavefront algorithms.

p	column-oriented algorithm				row-oriented algorithm			
	$n = 500$		$n = 1000$		$n = 500$		$n = 1000$	
	Ncube	Intel	Ncube	Intel	Ncube	Intel	Ncube	Intel
4	39	47	63	76	58	73	90	120
8	24	29	39	47	37	58	58	88
16	15	18	24	29	23	37	42	69
32	9	11	15	18	13	20	23	48
64	6	6	9	11	8	10	13	20

We turn now to an analysis of the row-oriented wavefront algorithm. The most important difference between the row- and column-oriented wavefront algorithms is that at a given stage of the row-oriented algorithm, all but the last (currently incomplete) segment can be forwarded immediately after receipt by each processor, *before* any floating-point processing of the segment is done. This permits more than one processor to be working on the same segment at the same time (unlike the column-oriented algorithm), which means that it may not be necessary to have a full set of p segments in order to keep all processors busy. We would therefore expect that the optimal segment size for the row-oriented algorithm might be somewhat larger than for the column-oriented algorithm, since a larger segment size would tend to reduce communication cost without necessarily sacrificing computational parallelism.

To simplify the analysis, we make similar assumptions (wrap mapping, p divides n , etc.) to those made for the analysis of the column-oriented wavefront algorithm. We focus our attention on the first segment (which contains the first σ components of x). Until the first segment is complete, there is an initial serial phase whose duration is about

$$\sum_{k=1}^{\sigma} (\alpha + \beta \sigma + f \sigma). \quad (3.9)$$

Beyond this point, the first segment and all other complete segments are forwarded immediately upon receipt by each processor. Thus, the first segment will be propagated without delay, provided the receiving processors are not busy with computation when it arrives. Initially, the processors are waiting in an idle state for the first segment to arrive, but as the algorithm progresses each processor has more and more work to do between successive appearances of the first segment, so that eventually a processor may not be able to finish work on other segments before the first segment reappears. We wish to determine this point, prior to which the algorithm is communication bound and after which the algorithm is computation bound.

As long as the first segment propagates without delay, each circuit through all of the processors takes time

$$p(\alpha + \beta \sigma). \quad (3.10)$$

After forwarding the first segment, a given processor must process it and all remaining segments before receiving the first segment back again. If there are currently j segments, the time required for this processing will be about

$$j(\alpha + \beta \sigma + f \sigma). \quad (3.11)$$

To determine the trade-off point we equate (3.10) and (3.11), so that when there are more than $\lceil K_{\sigma} p \rceil$ segments, where

$$K_{\sigma} = \frac{\alpha + \beta \sigma}{\alpha + \beta \sigma + f \sigma}, \quad (3.12)$$

the algorithm will be computation bound. Note that K_{σ} is always less than 1, in agreement with our expectation that fewer than p segments are required for full processor utilization.

We can now determine the total execution time by summing up the initial serial phase given by (3.9), the communication bound phase given by (3.10), and the computation bound phase given by (3.11). If a given processor contains all rows kp ($0 \leq k < n/p$), then the trade-off point between the latter two phases will occur at row number $\lceil K_{\sigma} \sigma \rceil$, so the total execution time is given by

$$T = \sum_{k=1}^{\sigma} (\alpha + \beta \sigma + f \sigma) + \sum_{k=0}^{\lceil K_{\sigma} \sigma \rceil} \left(\frac{kp}{\sigma} \right) (\alpha + \beta \sigma) + \sum_{k=\lceil K_{\sigma} \sigma \rceil + 1}^{\frac{n}{p} - 1} p(\alpha + \beta \sigma + f \sigma). \quad (3.13)$$

Unfortunately, due to the complicated limits of summation, (3.13) cannot be reduced to a simple form comparable to (3.6), but it can be evaluated numerically for any given values of the parameters. This approach was used to generate the curves in Fig. 1b. A one-dimensional minimization procedure was used to determine the optimal segment sizes for the row-oriented wavefront algorithm shown in Table 3. We observe that the predicted optimal segment sizes are indeed somewhat larger than the corresponding ones for the column-oriented algorithm.

3.4. Comparison of models. The three basic column-oriented algorithms can be compared theoretically using the models given by (3.2), (3.3), and (3.6). Fig. 2 shows the performance of the three algorithms for fixed n as p varies. The hypercube fan-in algorithm is fairly

insensitive to the number of processors, with increasing communication overhead offsetting computational gains as p increases. The cyclic algorithm performs best for small p , reaching a minimum at $p=16$, but its execution time increases sharply for large p as the algorithm becomes strongly communication bound. The wavefront algorithm displays the opposite behavior, performing worst for small p but improving markedly as p grows.

Fig. 3 shows the performance of the three algorithms for fixed p as n varies. For this relatively large number of processors, the cyclic algorithm never gets out of the communication-bound range given by (3.3a) in which it displays linear behavior, and is consequently the poorest performer. The hypercube fan-in algorithm performs slightly better and does show quadratic behavior. The wavefront algorithm shows superior performance. Figs. 2 and 3 should be compared to Figs. 5a and 6a in Section 4, which show the same curves for actual runs on the Ncube hypercube.

4. Numerical experiments. We have conducted a series of numerical experiments on commercially available hypercubes to compare the basic triangular solution algorithms and to determine the effectiveness of the variations discussed in Section 2. Our programs are written in C and use single-precision floating-point data. While we believe the programs to be reasonably efficient, no attempt was made to optimize the source code through special techniques such as loop unrolling, nor were any assembler language modules used. The programs are written in a generic message-passing style, so that the source code can be ported without change across hypercubes from different manufacturers. The various algorithms were implemented in the same coding style and with equal care, so we believe that the tests represent their relative performance fairly. For sample listings of the programs, see the appendix below.

The hypercubes used in our tests were those shown in Table 1. At the time these tests were run, the Ametek hypercube available to us had only 16 processors, which was inadequate for running our full test suite. We therefore give results only for the Ncube and Intel machines, each of which had 64 processors. The results from the Ncube machine turned out to match the predictions of our theoretical models much more closely than did the Intel results. The principal reason for this is that the communication behavior of the Ncube machine is described almost exactly by the linear model (1.1), upon which our performance models are based, whereas the Intel machine deviates significantly from linear behavior as the message size varies. Thus, the parameter estimates for the Intel machine are less well determined and less meaningful. We also observed a substantially greater degree of nondeterminism in the Intel communication system, leading to somewhat erratic results that are inherently more difficult to model analytically. For these reasons we will focus primarily on the Ncube results, but will give enough Intel results to show that they were qualitatively similar.

4.1. Preliminary tests. We first give preliminary results on such issues as connectivity, mapping, and acceleration techniques. These preliminary findings will enable us to reduce the large number of algorithms and variations down to a more manageable number for the more comprehensive comparisons to follow, in which we study performance as a function of the number of processors and the size of problem.

4.1.1. Connectivity. As noted in Section 2.3, the fan-out and fan-in algorithms can be implemented using a number of different communication patterns, depending on the connectivity of the underlying network. Each of these types of communication has its advantages and disadvantages. The minimal spanning tree in a hypercube with p processors has a maximum path length of $\log_2 p$. Thus, logarithmic fan-out and fan-in have the smallest communication latency of any method of global communication, but at least some of the

processors must participate in up to $\log_2 p$ stages of the fan-out or fan-in procedure before resuming computational work. For brevity, we will refer to this type of communication as *cube* fan-out or fan-in. A unidirectional ring, in which messages flow in only one direction, has a maximum path length of $p-1$, but each processor forwards only one message, so that all processors can return to computational work quickly. This permits a pipelining effect, substantially overlapping communication with computation, which can result in a considerable gain in efficiency. This gain will be realized, however, only if the problem is mapped onto the processors in a compatible fashion. In a hypercube, the wrap mapping in Gray code order is one such mapping. A bidirectional ring, in which messages flow in both directions, has a smaller maximum path length of $p/2$, but is somewhat less amenable to pipelining. Finally, using complete connectivity requires no user-level forwarding at all, thus permitting pipelining at the user level, but requires that a large number of messages be sent sequentially and entails a great deal of system-level forwarding by intermediate processors.

Typical test results are shown in the first column of data in Tables 4a-b (for comparison, results for the wavefront and cyclic algorithms are also shown). Here the matrix has been mapped onto the processors using a wrap mapping in Gray code order. The unidirectional and bidirectional ring algorithms have similar performance and both substantially outperform the cube and completely-connected algorithms.

Table 4a. Execution time in seconds on the *N*cube hypercube for column-oriented algorithms on a matrix of order 1000 using 64 processors.

algorithm	wrap mapping		random mapping	
	basic	accel.	basic	accel.
cube fan-in	2.98	2.86	2.58	2.41
unidirectional ring fan-in	1.56	1.53	15.69	15.42
bidirectional ring fan-in	1.61	1.61	8.47	8.22
completely connected fan-in	1.80	1.78	5.33	5.34
wavefront	1.34	N/A	9.34	N/A
cyclic	4.08	N/A	N/A	N/A

Table 4b. Execution time in seconds on the *N*cube hypercube for row-oriented algorithms on a matrix of order 1000 using 64 processors.

algorithm	wrap mapping		random mapping	
	basic	accel.	basic	accel.
cube fan-out	2.55	2.38	2.91	2.80
unidirectional ring fan-out	1.48	1.63	15.59	16.22
bidirectional ring fan-out	1.72	1.84	10.02	10.02
completely connected fan-out	2.23	2.22	13.67	13.67
wavefront	1.03	N/A	6.53	N/A
cyclic	3.47	N/A	N/A	N/A

4.1.2. Acceleration. The second column of data in Tables 4a-b gives results for the “send-ahead” and “compute-ahead” acceleration techniques discussed in Section 2.3. The results are disappointing in that the hoped-for acceleration effects are not evident. Indeed, performance is worsened as often as it is helped by these modifications to the basic algorithms. This behavior is at odds with our prior experience using similar acceleration techniques in matrix factorization, where they provide a significant reduction in execution time. Matrix factorization is largely computation bound, however, so that any technique that enables processors to get an earlier start on computational tasks is likely to be beneficial. By

contrast, triangular solution algorithms are much more communication bound, so getting a head start on computational tasks is less significant. In fact, the acceleration techniques may even disrupt an otherwise smooth (systolic-like) flow of data.

4.1.3. Mapping. The third and fourth columns of data in Tables 4a-b show the performance of the algorithms with a random mapping of rows or columns to processors, as might result, for example, from pivoting for numerical stability during the factorization phase. We see that the algorithms based on logarithmic hypercube fan-out and fan-in are almost unaffected by the change in mapping, while performance of the other algorithms is substantially degraded. The orderly pipelining in the ring algorithms, which accounts for their superior performance using the Gray wrap mapping, is destroyed by a random mapping, resulting in performance that reflects the full latency caused by their longer communication paths. We conclude that the cube fan-in and fan-out algorithms are preferable over all other algorithms unless the mapping is known to be compatible with the ring algorithms.

4.1.4. Segment size. Our next series of tests is an empirical study of the effect of the segment size σ on the performance of the wavefront algorithms. Figs. 4a-b show the execution time of the wavefront algorithms as a function of segment size using various numbers of processors. The curves have the roughly convex shape one would expect from the trade-off between concurrency and communication overhead. These curves should be compared with Figs. 1a-b, which show the same curves predicted by the theoretical performance models of the wavefront algorithms. The optimal segment size for each curve predicted by the model and shown in Table 3 is indicated by the corresponding symbols along the horizontal axis in Figs. 4a-b. The agreement between model and experiment is remarkably good for the column-oriented algorithm. The model accurately predicts the magnitude and variation of the execution time as a function of σ . The model of the row-oriented algorithm is less accurate in predicting the actual execution time, but still predicts the optimal σ quite well.

4.2. Comprehensive tests. To keep the number of algorithms and variations manageable for our more comprehensive tests, we will discard the acceleration techniques, which proved largely ineffective anyway. We will restrict our attention to the Gray wrap mapping, since this is the only mapping for which every algorithm is valid and since the performance models are based on it. Since the two types of ring communication perform similarly for the wrap mapping, we will use only the simpler unidirectional ring communication, hereafter referred to simply as ring communication. Similarly, we discard completely-connected communication, since its performance at best resembles that of cube fan-out and fan-in. This leaves us with four types of basic algorithms (cube and ring fan-out or fan-in, wavefront, and cyclic), and two types of storage organization (rows and columns) on which to make more extensive comparisons. In particular, we are interested in the performance of the algorithms as the number of processors and the size of the problem vary.

4.2.1. Performance as a function of p . Figs. 5a-b give results for a matrix of fixed size using a varying number of processors. The cyclic algorithms perform extremely well for 16 or fewer processors, but their performance degrades rapidly as p grows. The wavefront algorithms behave in just the opposite manner: they fare badly for small p , but their performance improves markedly as p increases. Performance of the fan-in and fan-out algorithms is less sensitive to the number of processors; ring communication performs better than cube communication, with the gap widening significantly as p grows. Fig. 5a should be compared with Fig. 2, which shows the same curves for three of the four algorithms using the theoretical performance models. There are slight differences between predicted and observed behavior, but the models capture remarkably well the order of magnitude and general shape of the performance curves and the crossover points between them.

4.2.2. Performance as a function of n . Figs. 6a-d give results for matrices of varying order using a fixed number of processors, this time including Intel as well as Ncube results. We used the maximum number of processors available (64 for both hypercubes) for this test because our primary interest is in the effectiveness of parallel algorithms for very large numbers of processors. Generally speaking, the wavefront algorithms perform best, with ring fan-in or fan-out second, cube fan-in or fan-out third, and the cyclic algorithms worst. There are some exceptions to this order, however: on both machines there is a crossover point beyond which the ring fan-in algorithm outperforms the column-oriented wavefront algorithm for very large problems, and on the Intel machine the cube fan-out algorithm performs uniformly more poorly than the row-oriented cyclic algorithm. Fig. 6a should be compared with Fig. 3, which shows the same curves for three of the four algorithms using the theoretical performance models. Again, agreement between predicted and observed behavior is quite good.

5. Conclusion. We have presented two new wavefront algorithms for solving triangular systems of equations on distributed-memory multiprocessors, with matrices partitioned and distributed among the processors by rows or by columns. Both algorithms feature an adjustable parameter, the segment size, that controls the granularity of the algorithm and can be tuned to a specific architecture. Performance models were developed for both algorithms that accurately predict the optimal segment size for given machine parameters.

The new wavefront algorithms were placed in the context of other message-passing algorithms for solving triangular systems. These basic algorithms and several variations were implemented and tested on commercially available hypercube multiprocessors. No one type of algorithm proved to be superior in all cases. If the matrix is not compatibly mapped onto the processors, then logarithmic hypercube fan-in and fan-out are the best performing algorithms. The best absolute performance occurs, however, when the matrix is mapped compatibly onto the processors, such as in the Gray wrap mapping. In this case, for small numbers of processors cyclic algorithms are best, while for large numbers of processors wavefront algorithms are best. Ring fan-in and fan-out perform well across a wide range of numbers of processors, but provide the overall winner only in the case of a very large matrix mapped by columns onto a large number of processors.

Parallel algorithms for solving triangular systems tend to be inherently less efficient than similar algorithms for matrix factorization on the same architectures. The difference is due to the relatively high proportion of communication required for triangular solution compared to that for factorization. On the other hand, only one of the triangular solution algorithms we have discussed (column-sweep) was known prior to 1986. This substantial recent progress has produced algorithms with acceptable efficiency, so that the triangular solution phase no longer requires a severely disproportionate amount of the total time in solving a general system of equations, regardless of whether the matrix is partitioned by rows or by columns. Moreover, further investigation may lead to still more efficient algorithms. A combination of the best features of the cyclic and wavefront algorithms appears especially promising, since their regions of greatest effectiveness are complementary.

References

- [1] R. M. Chamberlain, *An algorithm for LU factorization with partial pivoting on the hypercube*, Tech. Rept. CCS 86/11, Dept. of Science and Technology, Chr. Michelsen Institute, Bergen, Norway, June 1986.
- [2] B. W. Char, G. J. Fee, K. O. Geddes, G. H. Gonnet and M. B. Monagan, *A tutorial introduction to Maple*, J. Symbolic Comput., 2 (1986), pp. 179-200. (Maple is an interactive system for algebraic computation developed and distributed by the Symbolic

Computation Group of the Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1.)

- [3] A. Cleary, private communication, Dept. of Applied Mathematics, University of Virginia, Charlottesville, Virginia, October 1986.
- [4] T. H. Dunigan, *Hypercube performance*, to appear in *Hypercube Multiprocessors 1987*, ed. by M. T. Heath, SIAM, Philadelphia, 1987.
- [5] D. J. Evans and R. C. Dunbar, *The parallel solution of triangular systems of equations*, IEEE Trans. Computers, C-32 (1983), pp. 201-204.
- [6] G. C. Fox, *Square matrix decompositions - symmetric, local, scattered*, Tech. Rept. HM-97, California Institute of Technology, Pasadena, California, 1984.
- [7] G. A. Geist and M. T. Heath, *Matrix factorization on a hypercube multiprocessor*, in *Hypercube Multiprocessors 1986*, ed. by M. T. Heath, SIAM, Philadelphia, 1986, pp. 161-180.
- [8] M. T. Heath and C. H. Romine, *Parallel solution of triangular systems on distributed-memory multiprocessors*, manuscript submitted for publication in *SIAM J. Sci. Stat. Comput.*, 1987.
- [9] M. T. Heath and D. C. Sorensen, *A pipelined Givens method for computing the QR factorization of a sparse matrix*, *Linear Algebra Appl.*, 77 (1986), pp. 189-203.
- [10] D. E. Heller, *A survey of parallel algorithms in numerical linear algebra*, *SIAM Rev.*, 20 (1978), pp. 740-777.
- [11] D. J. Kuck, *Parallel processing of ordinary programs*, *Advances in Computers*, 15 (1976), pp. 119-179.
- [12] S. Y. Kung, *On supercomputing with systolic/wavefront array processors*, *Proc. IEEE*, 72 (1984), pp. 867-884.
- [13] G. Li and T. F. Coleman, *A parallel triangular solver for a hypercube multiprocessor*, Tech. Rept. TR 86-787, Dept. of Computer Science, Cornell University, Ithaca, New York, October 1986.
- [14] D. P. O'Leary and G. W. Stewart, *Data-flow algorithms for parallel matrix computations*, *Comm. ACM*, 28 (1985), pp. 840-853.
- [15] J. M. Ortega and R. G. Voigt, *Solution of partial differential equations on vector and parallel computers*, *SIAM Rev.*, 27 (1985), pp. 149-240.
- [16] C. H. Romine, *Parallel solution of triangular systems on a hypercube*, to appear in *Hypercube Multiprocessors 1987*, ed. by M. T. Heath, SIAM, Philadelphia, 1987.
- [17] C. H. Romine and J. M. Ortega, *Parallel solution of triangular systems of equations*, Tech. Rept. RM-86-05, Dept. of Applied Mathematics, University of Virginia, Charlottesville, Virginia, August 1986.
- [18] C. L. Scitz, *The cosmic cube*, *Comm. ACM*, 28 (1985), pp. 22-33.

Appendix

In this appendix we give selected program listings for our implementations of the algorithms presented in this paper. Our purpose is to provide details that were omitted from the high-level statements of the algorithms given earlier. These listings may also help readers in making comparisons with their own programs and timings.

The programs that follow are straightforward implementations of the algorithms. We believe that the coding is clean enough to make our performance results meaningful, but no special attempt was made to optimize the source code for highest possible performance. All of the algorithms were implemented in the same coding style and with the same degree of care, so comparisons based on these programs should be fair. The close correspondence between our empirical results and the results from our theoretical performance models bears this out.

The programs are written in a generic message-passing style so that the source code can run on hypercubes from different manufacturers without change. The generic communication procedures *nsend* and *nrecv* in turn invoke the native communication procedures of a given machine. A receive call blocks further execution by the receiving processor until the expected message arrives. The parameters of the communication procedures include a channel identifier *ci*, a message buffer, the length of the message in bytes, the source or destination node, and a message *type*. The latter permits discrimination among messages in the input queue of a processor and can be helpful in synchronization.

The following program listings include the eight basic algorithms comprehensively compared in Section 4.2. To illustrate the other two topologies and the two acceleration techniques, we also give program listings for the column-oriented fully connected fan-in algorithm with compute-ahead and the row-oriented bidirectional ring fan-out algorithm with send-ahead. All of the programs solve a lower triangular system by forward substitution. We have also implemented all of the algorithms in this paper to solve upper triangular systems by backward substitution. In most cases the programs are similar to their forward-substitution counterparts, with the loops simply running backward. The wavefront algorithm is a bit trickier to adapt, and the unidirectional ring algorithms communicate in the opposite direction around the ring.

The parameters for the triangular solution procedures are as follows:

<i>n</i>	size of matrix
<i>ncols</i> or <i>nrows</i>	number of columns or rows owned by this processor
<i>map</i>	vector giving mapping of columns or rows to processors
<i>mycols</i> or <i>myrows</i>	the set of columns or rows owned by this processor
<i>col</i> or <i>row</i>	vector of pointers to the columns or rows of matrix
<i>b</i>	components of right-hand-side vector owned by this processor (on exit contains components of solution computed by this processor)
<i>p</i>	number of processors
<i>me</i>	identification number of this processor
<i>ci</i>	communication channel identifier

In addition, for the wavefront algorithms the parameter *segment* gives the segment size. The ring algorithms use a few utility routines for determining neighbors in a Gray code ordering, and these are given in a separate listing at the end.

```

col_cf ( n, ncols, map, mycols, col, b, p, me, ci )
    int n, ncols, *map, *mycols, p, me, ci ;
    float **col, *b ;

/*
 * lower triangular forward solve
 * column-oriented cube fan-in algorithm
 */
{
    int i, j, k ;
    float t ;

    j = 0 ;
    if ( map[0] == me )
    {
        b[j] /= *col[j] ;
        j++ ;
    }
    for ( k = 1 ; k < n ; k++ )
    {
        t = 0 ;
        for ( i = 0 ; i < j ; i++ )
            t += b[i] * *(col[i]+k-mycols[i]) ;
        fan_in ( ci, me, k, &t, map[k], p ) ;
        if ( map[k] == me )
        {
            b[j] -= t ;
            b[j] /= *col[j] ;
            j++ ;
        }
    }
}

fan_in ( ci, mynode, type, sum, root, p )
    int mynode, type, root, p ;
    float *sum ;

/*
 * global sum over all processors
 * using minimal spanning tree with given root
 */
{
    int me, bytes, cnt, node ;
    float t ;

    bytes = sizeof(float) ;
    me = mynode^root ;
    p /= 2 ;
    if ( me < p )
    {
        nrecv ( ci, &t, bytes, &cnt, &node, &type ) ;
        *sum += t ;
        if ( p != 1 ) fan_in ( ci, mynode, type, sum, root, p ) ;
    }
    else nsend ( ci, sum, bytes, (me-p)^root, type ) ;
}

```

```

row_cf ( n, nrows, map, myrows, row, b, p, me, ci )
    int n, nrows, *map, *myrows, p, me, ci ;
    float **row, *b ;
/*
 * lower triangular forward solve
 * row-oriented cube fan-out algorithm
 */
{
    int i, j, k ;
    float x ;

    j = 0 ;
    for ( k = 0 ; k < n-1 ; k++ )
    {
        if ( map[k] == me )
        {
            b[j] /= *(row[j]+k) ;
            x = b[j] ;
            j++ ;
        }
        fan_out ( ci, me, k, &x, sizeof(float), map[k], p ) ;
        for ( i = j ; i < nrows ; i++ )
            b[i] -= x * *(row[i]+k) ;
    }
    if ( map[n-1] == me ) b[j] /= *(row[j]+n-1) ;
}

fan_out ( ci, mynode, type, vec, bytes, root, p )
    int ci, mynode, type, bytes, root, p ;
    char *vec ;
/*
 * broadcast vector vec of length bytes to all processors
 * using minimal spanning tree with given root
 */
{
    int me, cnt, node ;

    me = mynode^root ;
    p /= 2 ;
    if ( me < p )
    {
        if ( p != 1 ) fan_out( ci, mynode, type, vec, bytes, root, p ) ;
        nsend ( ci, vec, bytes, (me+p)^root, type ) ;
    }
    else
    {
        nrecv ( ci, vec, bytes, &cnt, &node, &type ) ;
    }
}

```

```

col_rf ( n, ncols, map, mycols, col, b, p, me, ci )
    int n, ncols, *map, *mycols, p, me, ci ;
    float **col, *b ;
/*
 * lower triangular forward solve
 * column-oriented ring fan-in algorithm
 */
(
    int i, j, k, cnt, node, type, forward, back ;
    float sum, t ;

    forward = right(me,p) ;
    back = left(me,p) ;
    j = 0 ;
    if ( map[0] == me )
    (
        b[j] /= *col[j] ;
        j++ ;
    )
    for ( k = 1 ; k < n ; k++ )
    (
        t = 0 ;
        for ( i = 0 ; i < j ; i++ )
            t += b[i] * *(col[i]+k-mycols[i]) ;
        sum = 0 ;
        if ( back != map[k] )
        (
            type = k ;
            nrecv ( ci, &sum, sizeof(float), &cnt, &node, &type ) ;
        )
        sum += t ;
        if ( map[k] == me )
        (
            b[j] -= sum ;
            b[j] /= *col[j] ;
            j++ ;
        )
        else nsend ( ci, &sum, sizeof(float), forward, k ) ;
    )
)

```

```

row_rf ( n, nrows, map, myrows, row, b, p, me, ci )
    int n, nrows, *map, *myrows, p, me, ci ;
    float **row, *b ;
/*
 * lower triangular forward solve
 * row-oriented ring fan-out algorithm
 */
{
    int i, j, k, cnt, node, type, forward ;
    float x ;

    forward = right(me,p) ;
    j = 0 ;
    for ( k = 0 ; k < n-1 ; k++ )
    {
        if ( map[k] == me )
        {
            b[j] /= *(row[j]+k) ;
            x = b[j] ;
            nsend ( ci, &x, sizeof(float), forward, k ) ;
            j++ ;
        }
        else
        {
            type = k ;
            nrecv ( ci, &x, sizeof(float), &cnt, &node, &type ) ;
            if ( forward != map[k] ) nsend ( ci, &x, cnt, forward, type ) ;
        }
        for ( i = j ; i < nrows ; i++ )
            b[i] -= x * *(row[i]+k) ;
    }
    if ( map[n-1] == me ) b[j] /= *(row[j]+n-1) ;
}

```

```

col_wv ( segment, n, ncols, map, mycols, col, b, p, me, ci )
    int segment, n, ncols, *map, *mycols, p, me, ci ;
    float **col, *b ;
/*
 * lower triangular forward solve
 * column-oriented wavefront algorithm
 */
{
    int i, j, k, m, lim, tag, bytes, cnt, node, type ;
    float *z ;

    bytes = segment*sizeof(float) ;
    z = (float *)malloc(bytes) ;

    j = 0 ;
    for ( k = 0 ; k < n ; k++ )
    {
        if ( map[k] == me )
        {
            tag = 1 ;
            m = 0 ;
            while ( m < n-k )
            {
                if ( k > 0 )
                {
                    type = k ;
                    nrecv ( ci, z, bytes, &cnt, &node, &type ) ;
                }
                else
                {
                    cnt = segment*sizeof(float) ;
                    for ( i = 0 ; i < segment ; i++ ) z[i] = 0 ;
                }
                lim = cnt/sizeof(float) ;
                if ( lim > n-m ) lim = n-m ;
                if ( tag )
                {
                    b[j] -= z[0] ;
                    b[j] /= *col[j] ;
                    m++ ;
                }
                for ( i = tag ; i < lim ; i++ )
                {
                    z[i] += b[j] * *(col[j]+m) ;
                    m++ ;
                }
                if ( k < n-1 && lim-tag > 0 )
                    nsend ( ci, &z[tag], (lim-tag)*sizeof(float), map[k+1], k+1 ) ;
                tag = 0 ;
            }
            j++ ;
        }
    }
    free(z) ;
}

```

```

row_wv ( segment, n, nrows, map, myrows, row, b, p, me, ci )
    int segment, n, nrows, *map, *myrows, p, me, ci ;
    float **row, *b ;
/*
 * lower triangular forward solve
 * row-oriented wavefront algorithm
 */
{
    int i, j, k, m, lim, nseg, iseg, bytes, cnt, node, type ;
    float *z ;

    bytes = segment*sizeof(float) ;
    z = (float *)malloc(bytes) ;

    j = 0 ;
    for ( k = 0 ; k < n ; k++ )
    {
        if ( map[k] == me )
        {
            m = 0 ;
            nseg = k/segment ;
            for ( iseg = 0 ; iseg < nseg ; iseg++ )
            {
                type = k ;
                nrecv ( ci, z, bytes, &cnt, &node, &type ) ;
                if ( k < n-1 ) nsend ( ci, z, cnt, map[k+1], k+1 ) ;
                for ( i = 0 ; i < segment ; i++ )
                {
                    b[j] -= z[i] * *(row[j]+m) ;
                    m++ ;
                }
            }
            if ( k > 0 && k%segment != 0 )
            {
                type = k ;
                nrecv ( ci, z, bytes, &cnt, &node, &type ) ;
            }
            else cnt = 0 ;
            lim = cnt/sizeof(float) ;
            for ( i = 0 ; i < lim ; i++ )
            {
                b[j] -= z[i] * *(row[j]+m) ;
                m++ ;
            }
            b[j] /= *(row[j]+k) ;
            if ( k < n-1 )
            {
                z[lim] = b[j] ;
                nsend ( ci, z, (lim+1)*sizeof(float), map[k+1], k+1 ) ;
            }
            j++ ;
        }
    }
    free(z) ;
}

```

```

col_cy ( n, ncols, map, mycols, col, b, p, me, ci )
    int n, ncols, *map, *mycols, p, me, ci ;
    float **col, *b ;
/*
 * lower triangular forward solve
 * column-oriented cyclic algorithm
 */
{
    int i, j, k, lim, bytes, cnt, node, type ;
    float *z, *t ;

    bytes = (p-1)*sizeof(float) ;
    z = (float *)malloc(p*sizeof(float)) ;
    for ( i = 0 ; i < p ; i++ ) z[i] = 0 ;
    t = (float *)malloc(n*sizeof(float)) ;
    for ( i = 0 ; i < n ; i++ ) t[i] = 0 ;

    j = 0 ;
    for ( k = 0 ; k < n ; k++ )
    {
        if ( map[k] == me )
        {
            if ( k > 0 )
            {
                type = k ;
                nrecv ( ci, z, bytes, &cnt, &node, &type ) ;
            }
            b[j] -= z[0] + t[k] ;
            b[j] /= *col[j] ;
            if ( k < n-1 )
            {
                if ( p <= n-k )
                {
                    lim = p-1 ;
                    for ( i = 1 ; i < lim ; i++ )
                        z[i] += t[k+i] + b[j] * *(col[j]+i) ;
                    z[lim] = t[k+lim] + b[j] * *(col[j]+lim) ;
                }
                else
                {
                    lim = n-k ;
                    for ( i = 1 ; i < lim ; i++ )
                        z[i] += t[k+i] + b[j] * *(col[j]+i) ;
                }
                nsend ( ci, z+1, lim*sizeof(float), map[k+1], k+1 ) ;
                for ( i = k+p ; i < n ; i++ )
                    t[i] += b[j] * *(col[j]+i-k) ;
            }
            j++ ;
        }
    }
    free ( t ) ;
    free ( z ) ;
}

```

```

row_cy ( n, nrows, map, myrows, row, b, p, me, ci )
    int n, nrows, *map, *myrows, p, me, ci ;
    float **row, *b ;
/*
 * lower triangular forward solve
 * row-oriented cyclic algorithm
 */
{
    int i, j, k, m, lim, tag, bytes, cnt, node, type ;
    float *z ;

    bytes = (p-1)*sizeof(float) ;
    z = (float *)malloc(p*sizeof(float)) ;

    j = 0 ;
    for ( k = 0 ; k < n ; k++ )
    {
        if ( map[k] == me )
        {
            if ( k > 0 )
            {
                type = k ;
                nrecv ( ci, z, bytes, &cnt, &node, &type ) ;
            }
            else cnt = 0 ;
            lim = cnt/sizeof(float) ;
            for ( i = 0 ; i < lim ; i++ )
                b[j] -= z[i] * *(row[j]+i) ;
            b[j] /= *(row[j]+k) ;
            if ( k < n-1 )
            {
                z[lim] = b[j] ;
                tag = lim < p-1 ? 0 : 1 ;
                nsend ( ci, z+tag, (lim+1-tag)*sizeof(float), map[k+1], k+1 ) ;
                for ( m = j+1 ; m < nrows ; m++ )
                    for ( i = 0 ; i <= lim ; i++ )
                        b[m] -= z[i] * *(row[m]+k+i) ;
            }
            j++ ;
        }
    }
    free (z) ;
}

```

```

col_ff ( n, ncols, map, mycols, col, b, p, me, ci )
    int n, ncols, *map, *mycols, p, me, ci ;
    float **col, *b ;
/*
 * lower triangular forward solve
 * column-oriented fully connected fan-in algorithm with compute-ahead
 */
{
    int i, j, k, cnt, node, type ;
    float sum, t, next ;

    j = 0 ;
    if ( map[0] == me )
    {
        b[j] /= *col[j] ;
        t = b[j] * *(col[j]+1) ;
        j++ ;
    }
    for ( k = 1 ; k < n ; k++ )
    {
        if ( map[k-1] != me )
        {
            t = 0 ;
            for ( i = 0 ; i < j ; i++ )
                t += b[i] * *(col[i]+k-mycols[i]) ;
        }
        if ( map[k] == me )
        {
            next = 0 ;
            if ( k < n-1 ) for ( i = 0 ; i < j ; i++ )
                next += b[i] * *(col[i]+k+1-mycols[i]) ;
            sum = t ;
            for ( i = 1 ; i < p ; i++ )
            {
                type = k ;
                nrecv ( ci, &t, sizeof(float), &cnt, &node, &type ) ;
                sum += t ;
            }
            b[j] -= sum ;
            b[j] /= *col[j] ;
            if ( k < n-1 ) t = next + b[j] * *(col[j]+1) ;
            j++ ;
        }
        else nsend ( ci, &t, sizeof(float), map[k], k ) ;
    }
}

```

```

row_bf ( n, nrows, map, myrows, row, b, p, me, ci )
    int n, nrows, *map, *myrows, p, me, ci ;
    float **row, *b ;
/*
 * lower triangular forward solve
 * row-oriented bidirectional ring fan-out algorithm with send-ahead
 */
{
    int i, j, k, m, cnt, node, type, forward, back, antipode ;
    float x ;

    forward = right(me,p) ;
    back = left(me,p) ;
    j = 0 ;
    if ( map[0] == me )
    {
        b[j] /= *row[j] ;
        nsend ( ci, &b[j], sizeof(float), forward, 0 ) ;
        nsend ( ci, &b[j], sizeof(float), back, 0 ) ;
        nsend ( ci, &b[j], sizeof(float), me, 0 ) ;
        j++ ;
    }
    for ( k = 0 ; k < n-1 ; k++ )
    {
        type = k ;
        nrecv ( ci, &x, sizeof(float), &cnt, &node, &type ) ;
        antipode = gray((invgray(map[k])+p/2)%p) ;
        if ( me != antipode && me != map[k] )
        {
            if ( node == back ) nsend ( ci, &x, cnt, forward, type ) ;
            else if ( back != antipode ) nsend ( ci, &x, cnt, back, type ) ;
        }
        for ( i = j ; i < nrows ; i++ )
        {
            b[i] -= x * *(row[i]+k) ;
            if ( k == myrows[j]-1 )
            {
                m = myrows[j] ;
                b[j] /= *(row[j]+m) ;
                if ( myrows[j] < n-1 )
                {
                    nsend ( ci, &b[j], sizeof(float), forward, myrows[j] ) ;
                    nsend ( ci, &b[j], sizeof(float), back, myrows[j] ) ;
                    nsend ( ci, &b[j], sizeof(float), me, myrows[j] ) ;
                }
                j++ ;
            }
        }
    }
}
}

```

```
int right ( node, p )
    int node, p ;
{
    int gray(), invgray() ;
    return( gray((invgray(node)+1)%p) ) ;
}

int left ( node, p )
    int node, p ;
{
    int gray(), invgray() ;
    return( gray((invgray(node)+p-1)%p) ) ;
}

int gray (i)
    int i ;
{
    return( (i>>1)^i ) ;
}

int invgray (i)
    int i ;
{
    int k ;
    k = i ;
    while ( k > 0 )
        {
            k >>= 1 ;
            i ^= k ;
        }
    return (i) ;
}
```

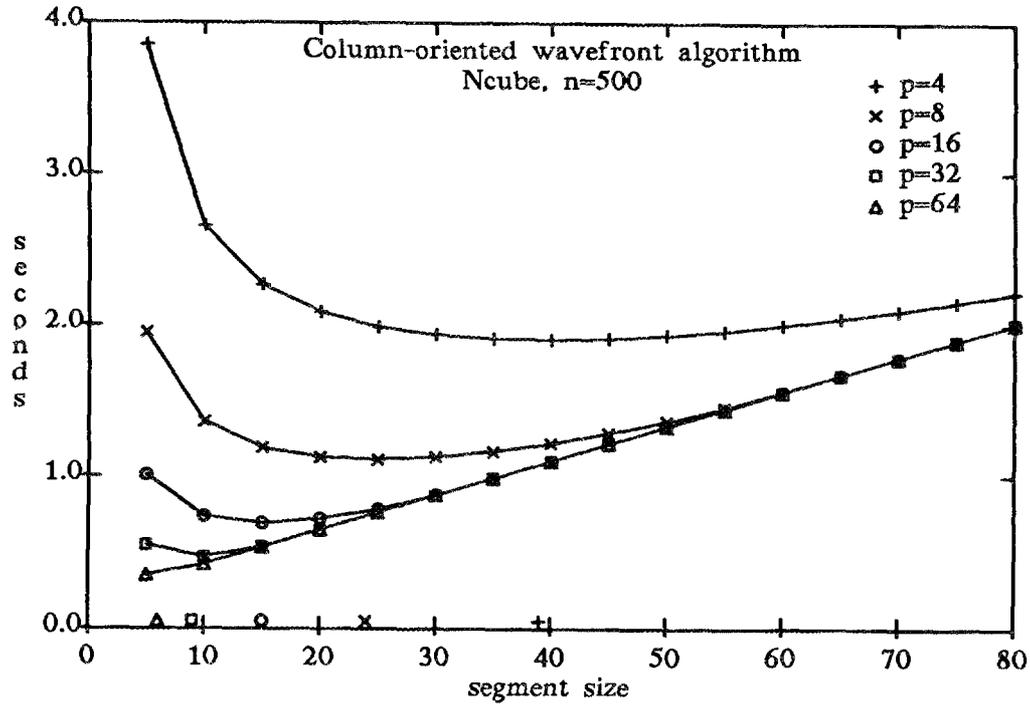


Fig. 1a. Predicted execution time as a function of segment size for the column-oriented wavefront algorithm on the Ncube hypercube with a matrix of order 500.

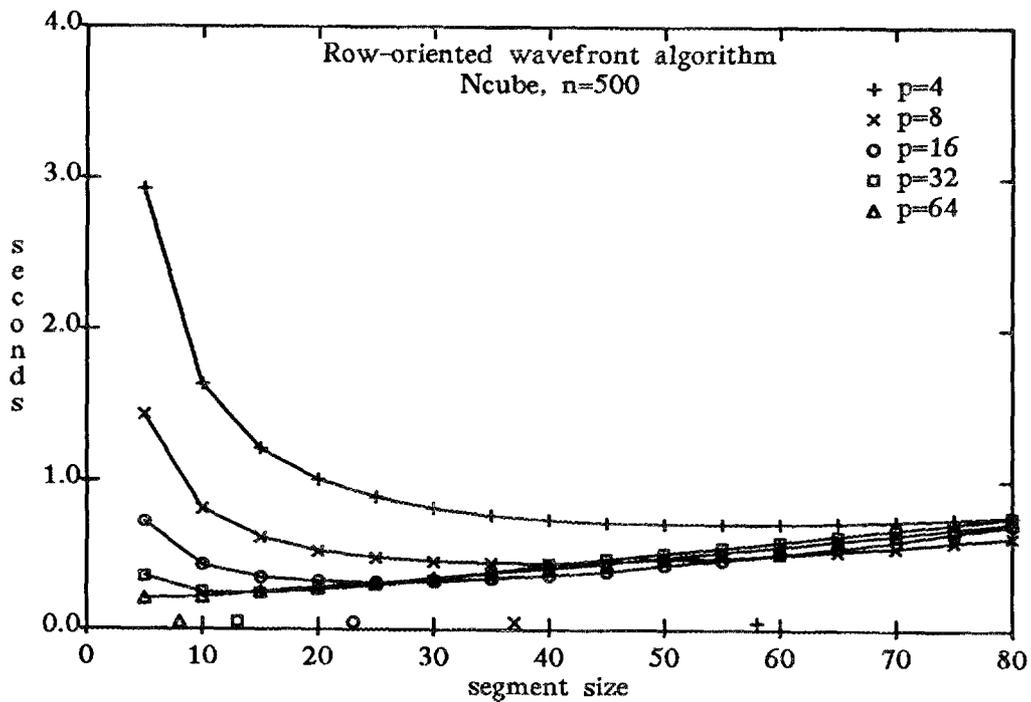


Fig. 1b. Predicted execution time as a function of segment size for the row-oriented wavefront algorithm on the Ncube hypercube with a matrix of order 500.

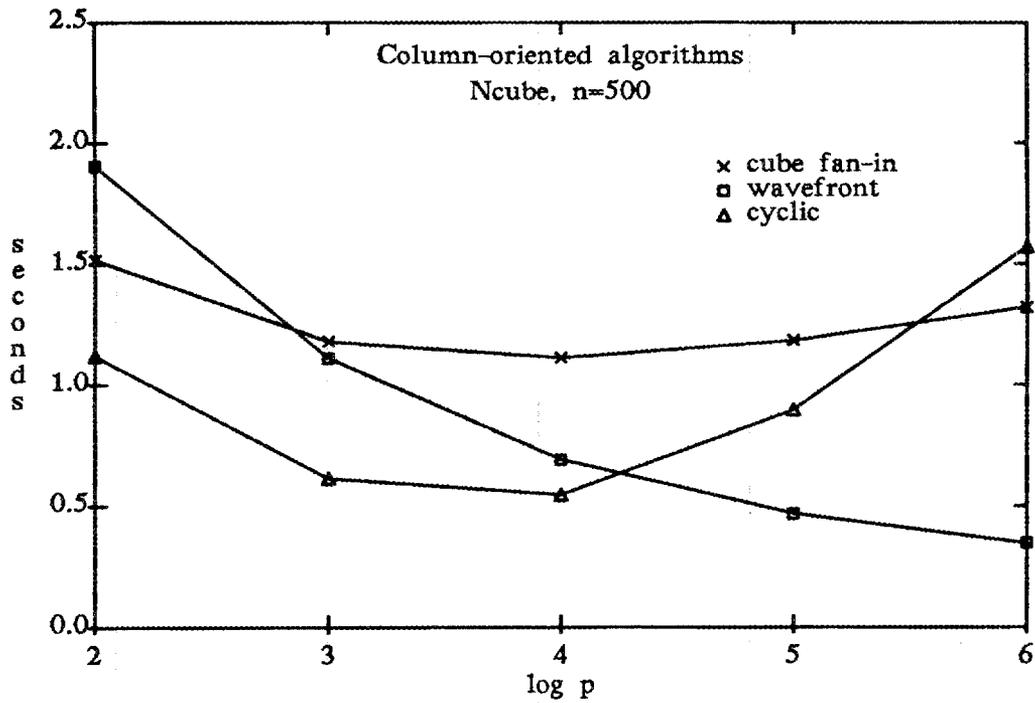


Fig. 2. Predicted execution time as a function of number of processors for column-oriented algorithms on the Ncube hypercube with a matrix of order 500.

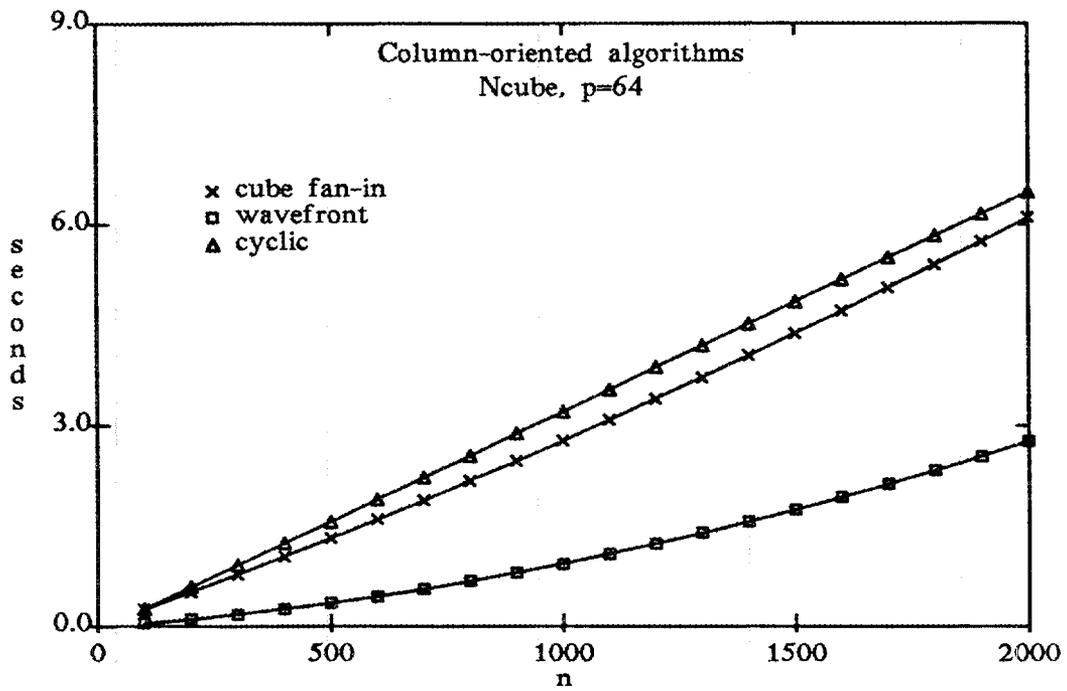


Fig. 3. Predicted execution time as a function of matrix size for column-oriented algorithms on the Ncube hypercube using 64 processors.

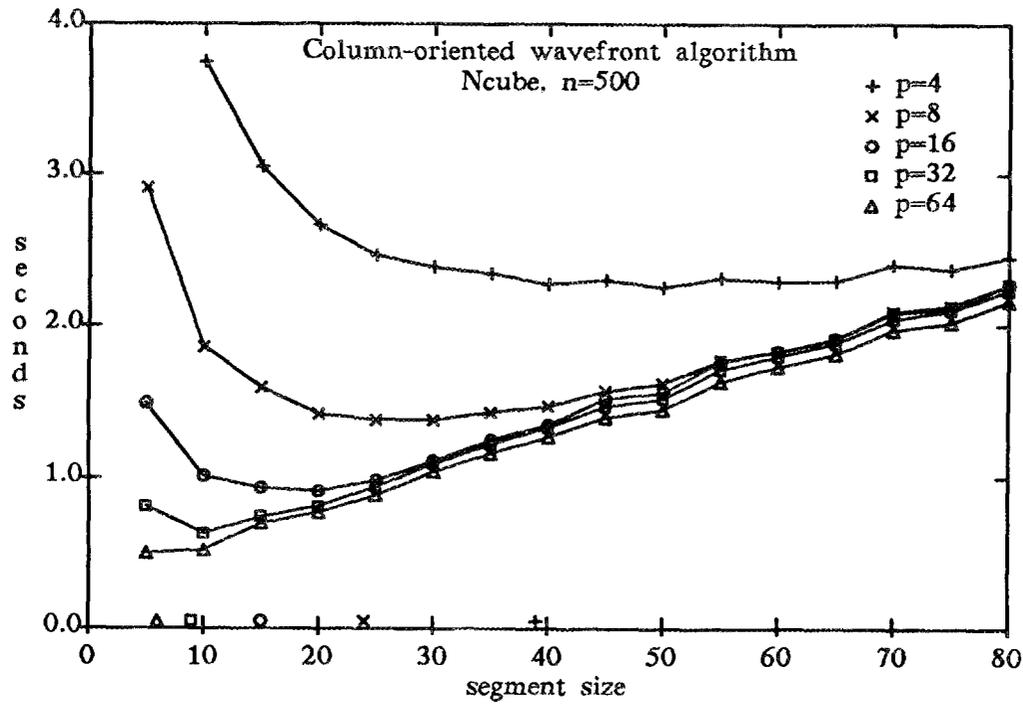


Fig. 4a. Execution time as a function of segment size for the column-oriented wavefront algorithm on the Ncube hypercube with a matrix of order 500.

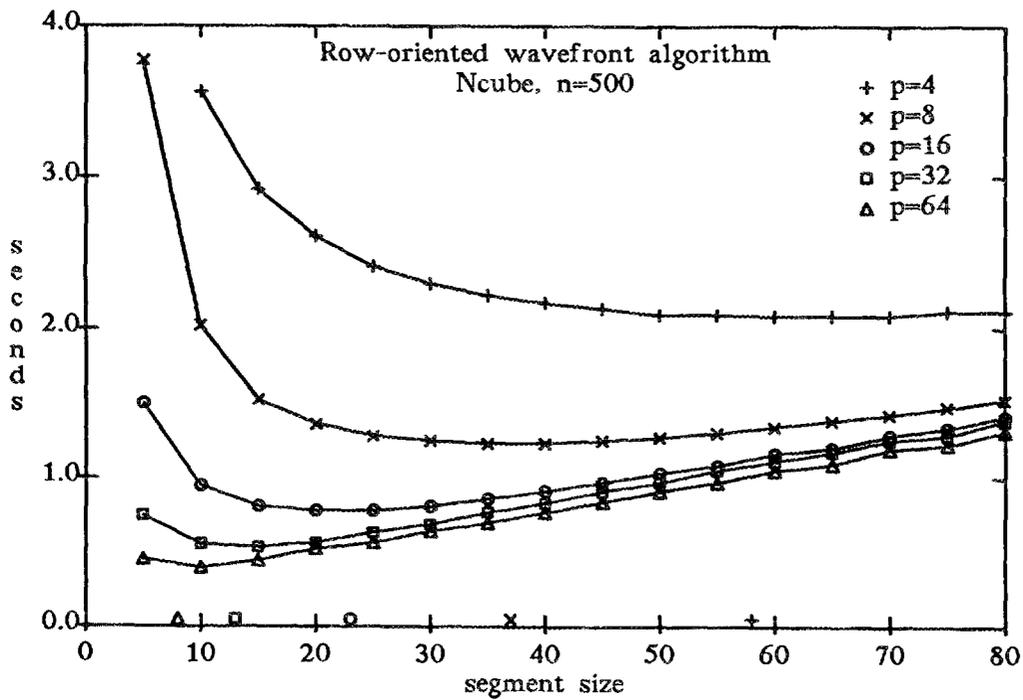


Fig. 4b. Execution time as a function of segment size for the row-oriented wavefront algorithm on the Ncube hypercube with a matrix of order 500.

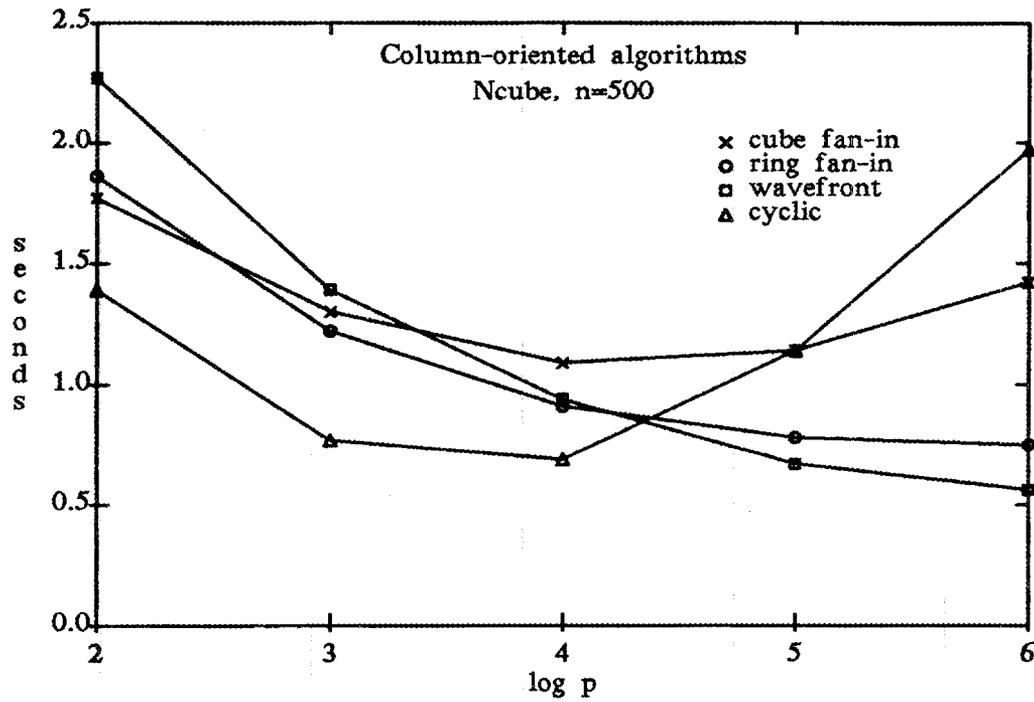


Fig. 5a. Execution time as a function of number of processors for column-oriented algorithms on the Ncube hypercube with a matrix of order 500.

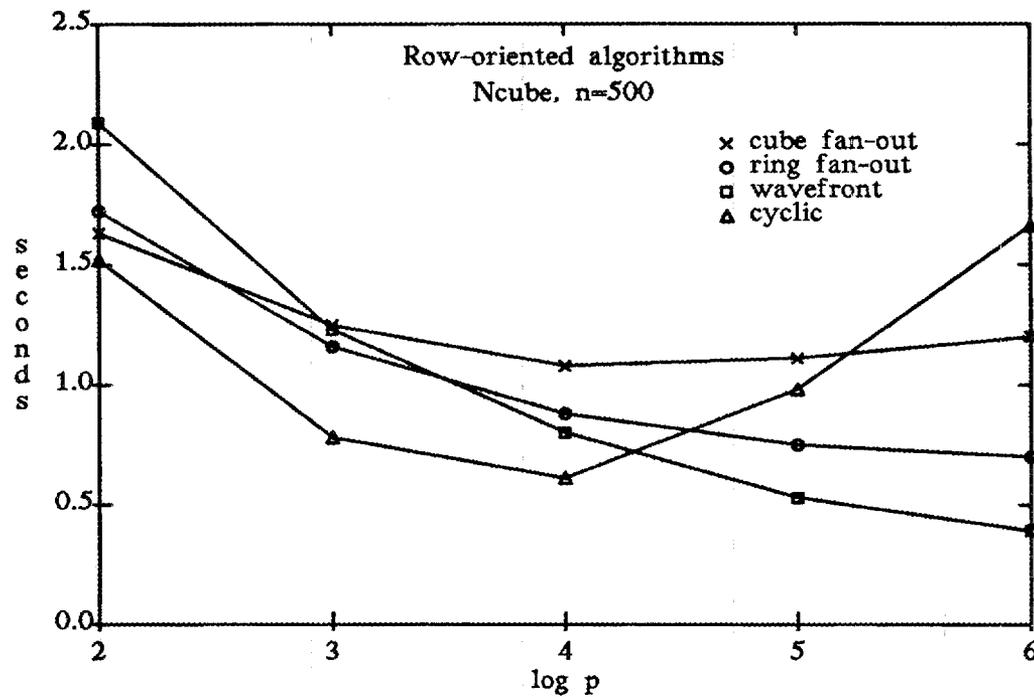


Fig. 5b. Execution time as a function of number of processors for row-oriented algorithms on the Ncube hypercube with a matrix of order 500.

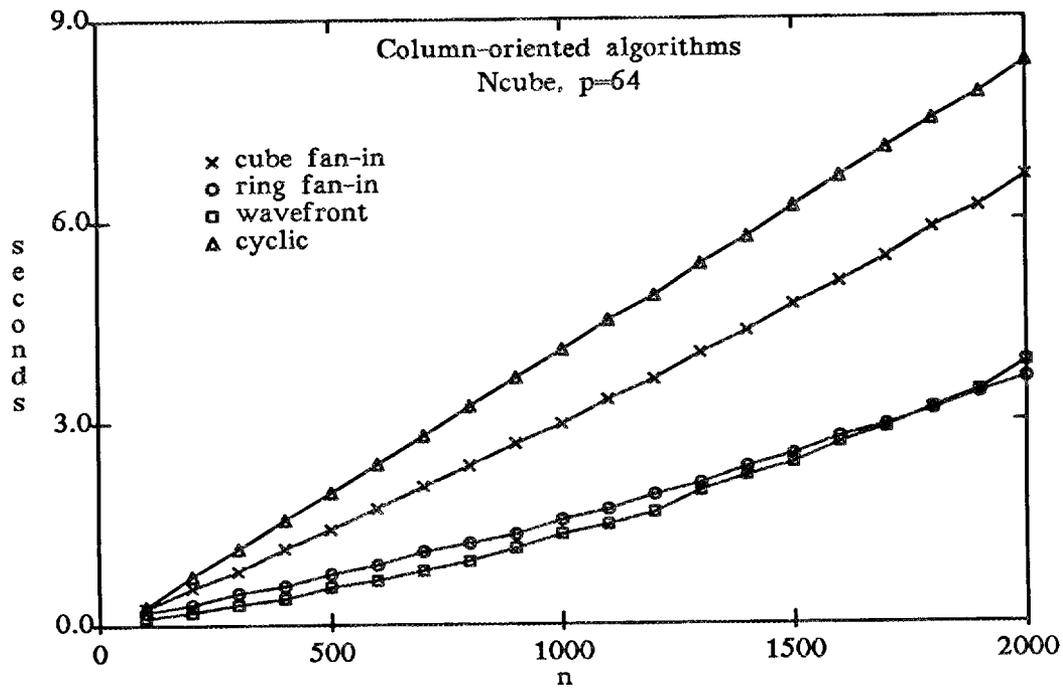


Fig. 6a. Execution time as a function of matrix size for column-oriented algorithms on the Ncube hypercube using 64 processors.

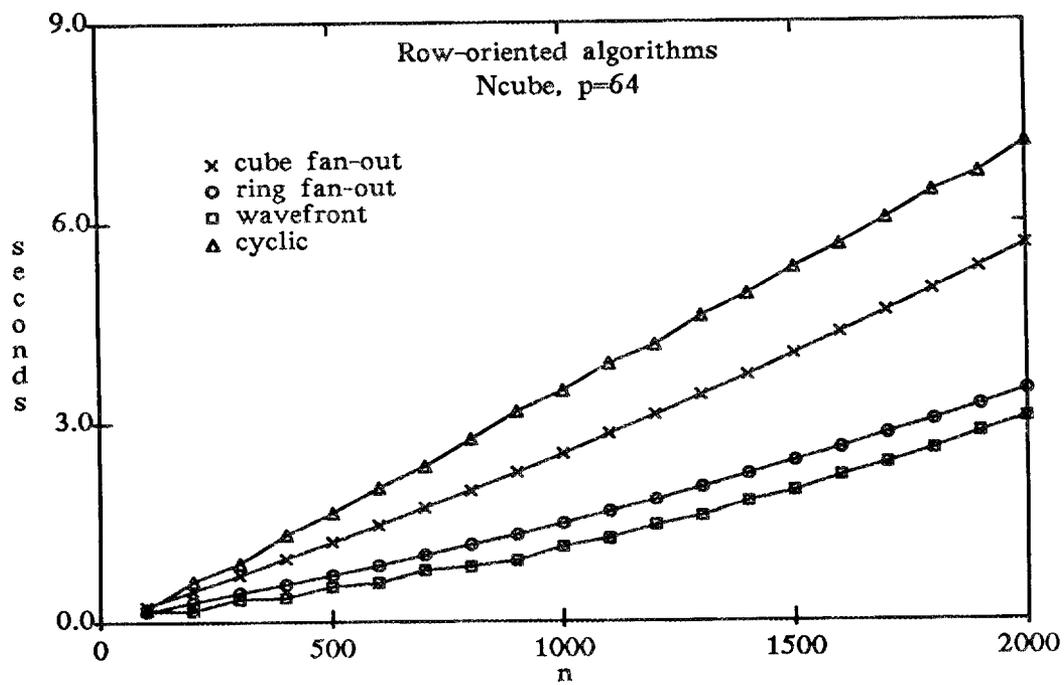


Fig. 6b. Execution time as a function of matrix size for row-oriented algorithms on the Ncube hypercube using 64 processors.

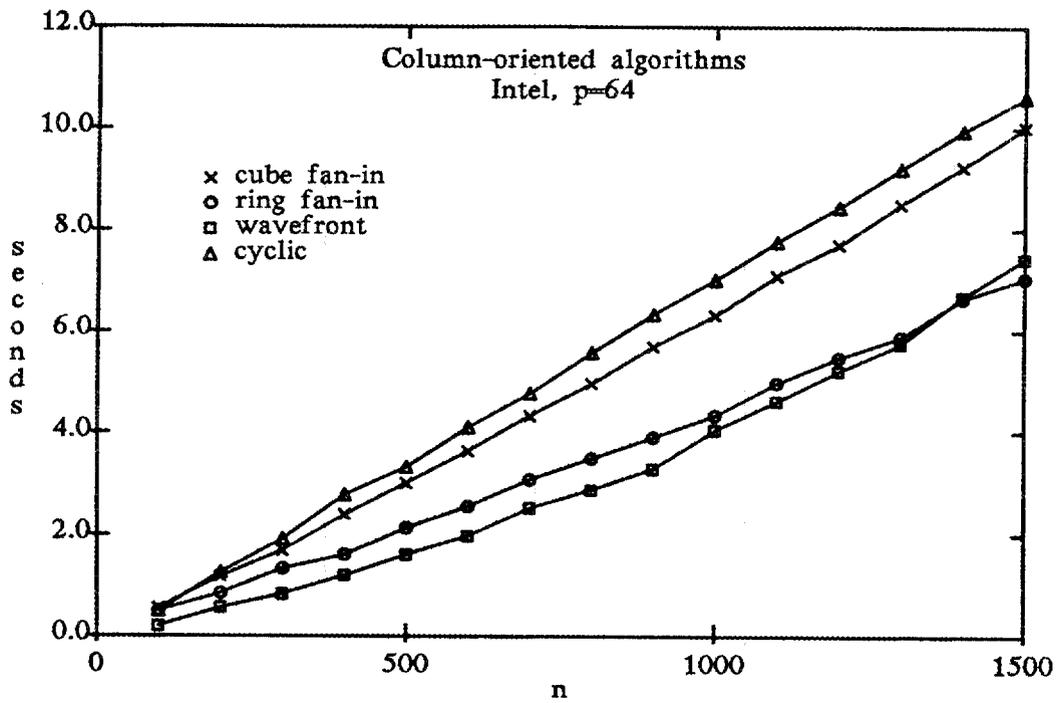


Fig. 6c. Execution time as a function of matrix size for column-oriented algorithms on the Intel hypercube using 64 processors.

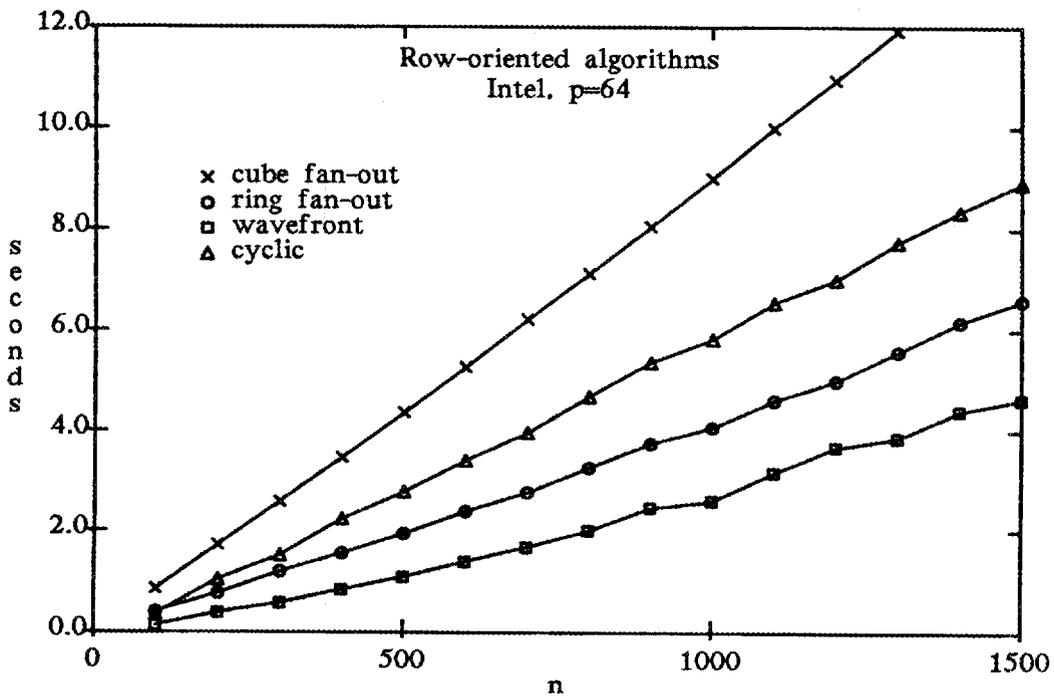


Fig. 6d. Execution time as a function of matrix size for row-oriented algorithms on the Intel hypercube using 64 processors.

INTERNAL DISTRIBUTION

- | | | | |
|--------|----------------------------|--------|---|
| 1-2. | R. F. Harbison | 31. | R. M. Haralick (Consultant) |
| 3-7. | M. T. Heath | 32. | D. Steiner (Consultant) |
| 8-12. | J. K. Ingersoll | 33. | Central Research Library |
| 13-17. | F. C. Maienschein | 34. | K-25 Plant Library |
| 18-22. | C. H. Romine | 35. | ORNL Patent Office |
| 23-27. | R. C. Ward | 36. | Y-12 Technical Library
/Document Reference Station |
| 28. | A. Zucker | 37. | Laboratory Records - RC |
| 29. | P. W. Dickson (Consultant) | 38-39. | Laboratory Records Department |
| 30. | G. H. Golub (Consultant) | | |

EXTERNAL DISTRIBUTION

40. Dr. Donald M. Austin, Office of Scientific Computing, Office of Energy Research, ER-7, Germantown Building, U.S. Department of Energy, Washington, DC 20545
41. Lawrence J. Baker, Exxon Production Research Company, P.O.Box 2189, Houston, TX 77252-2189
42. Dr. Jesse L. Barlow, Department of Computer Science, Pennsylvania State University, University Park, PA 16802
43. Dr. Adam Beguelin, Department of Computer Science, University of Colorado, Boulder, CO 80309
44. Prof. Ake Bjorck, Department of Mathematics, Linkoping University, Linkoping 58183, Sweden
45. Dr. Bill L. Buzbee, C-3, Applications Support & Research, Los Alamos National Laboratory, P.O. Box 1663, Los Alamos, NM 87545
46. Dr. Donald A. Calahan, Department of Electrical and Computer Engineering, University of Michigan, Ann Arbor, MI 48109
47. Dr. Richard Chamberlain, Intel Scientific Computers, Intel International LTD., Pipers Way, Swindon, England SN31RJ
48. Dr. Tony Chan, Department of Computer Science, Yale University, P.O. Box 2158 Yale Station, New Haven, CT 06520
49. Dr. Jagdish Chandra, Army Research Office, P.O. Box 12211, Research Triangle Park, North Carolina 27709
50. Andrew Cleary, Department of Applied Mathematics, University of Virginia, Charlottesville, VA 22903
51. Dr. Tom Coleman, Computer Science Department, Cornell University, Ithaca, NY 14853
52. Dr. Paul Concus, Mathematics and Computing, Lawrence Berkeley Laboratory, Berkeley, CA 94720
53. Dr. Jane K. Cullum, IBM T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598

54. Dr. George Cybenko, Department of Computer Science, Tufts University, Medford, MA 02155
55. Dr. George J. Davis, Department of Mathematics, Georgia State University, Atlanta, GA 30303
56. Dr. Jack J. Dongarra, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439
57. Dr. Stanley Eisenstat, Department of Computer Science, Yale University, P.O. Box 2158 Yale Station, New Haven, CT 06520
58. Dr. Howard C. Elman, Computer Science Department, University of Maryland, College Park, MD 20742
59. Dr. Albert M. Erisman, Boeing Computer Services, 565 Andover Park West, Tukwila, WA 98188
60. Dr. Jeff Fier, Computer Research Division, Ametek Corporation, 610 North Santa Anita Avenue, Arcadia, CA 91006
61. Dr. Geoffrey C. Fox, Booth Computing Center 158-79, California Institute of Technology, Pasadena, CA 91125
62. Dr. Paul O. Frederickson, Computing Division, Los Alamos National Laboratory, Los Alamos, NM 87545
63. Dr. Robert E. Funderlic, Department of Computer Science, North Carolina State University, Raleigh, NC 27650
64. Dr. Dennis B. Gannon, Computer Science Department, Indiana University, Bloomington, IN 47405
65. Dr. David M. Gay, Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974
66. Dr. C. William Gear, Computer Science Department, University of Illinois, Urbana, Illinois 61801
67. Dr. Don E. Heller, Physics and Computer Science Department, Shell Development Co., P.O. Box 481, Houston, TX 77001
68. Dr. Robert E. Huddleston, Computation Department, Lawrence Livermore National Laboratory, P.O. Box 808, Livermore, CA 94550
69. Dr. Ilse Ipsen, Department of Computer Science, Yale University, P.O. Box 2158 Yale Station, New Haven, CT 06520
70. Dr. Harry Jordan, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO 80309
71. Dr. Linda Kaufman, Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974
72. Dr. Robert J. Kee, Applied Mathematics Division 8331, Sandia National Laboratories, Livermore, CA 94550
73. Dr. David Kuck, Computer Science Department, University of Illinois, Urbana, IL 61801
74. Dr. Richard Lau, Office of Naval Research, 1030 E. Green Street, Pasadena, CA 91101
75. Dr. Alan J. Laub, Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106
76. Dr. Robert L. Launer, Army Research Office, P.O. Box 12211, Research Triangle Park, North Carolina 27709
77. Prof. Peter D. Lax, Director, Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012

78. Dr. Michael R. Leuze, Computer Science Department, Box 1679 Station B, Vanderbilt University, Nashville, TN 37235
79. Dr. Joseph Liu, Department of Computer Science, York University, 4700 Keele Street, Downsview, Ontario, Canada M3J 1P3
80. Dr. Franklin Luk, Electrical Engineering Department, Cornell University, Ithaca, NY 14853
81. James G. Malone General Motors Research Laboratories, Warren, Michigan 48090-9055
82. Dr. Thomas A. Manteuffel, Computing Division, Los Alamos National Laboratory, Los Alamos, NM 87545
83. Dr. Paul C. Messina, Applied Mathematics Division, Argonne National Laboratory, Argonne, IL 60439
84. Dr. Cleve Moler, Intel Scientific Computers, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006
85. Dr. Dianne P. O'Leary, Computer Science Department, University of Maryland, College Park, MD 20742
86. Maj. C. E. Oliver, Office of the Chief Scientist, Air Force Weapons Laboratory, Kirtland Air Force Base, Albuquerque, NM 87115
87. Dr. James M. Ortega, Department of Applied Mathematics, University of Virginia, Charlottesville, VA 22903
88. Prof. Chris Paige, Basser Department of Computer Science, Madsen Building F09, University of Sydney, N.S.W., Sydney, Australia 2006
89. Dr. John F. Palmer, NCUBE Corporation, 915 E. LaVie Lane, Tempe, AZ 85284
90. Prof. Beresford N. Parlett, Department of Mathematics, University of California, Berkeley, CA 94720
91. Prof. Merrell Patrick, Department of Computer Science, Duke University, Durham, NC 27706
92. Dr. Robert J. Plemmons, Departments of Mathematics and Computer Science, North Carolina State University, Raleigh, NC 27650
93. Dr. John K. Reid, CSS Division, Building 8.9, AERE Harwell, Didcot, Oxon, England OX11 0RA
94. Dr. John R. Rice, Computer Science Department, Purdue University, West Lafayette, IN 47907
95. Dr. Garry Rodrigue, Numerical Mathematics Group, Lawrence Livermore Laboratory, Livermore, CA 94550
96. Dr. Donald J. Rose, Department of Computer Science, Duke University, Durham, NC 27706
97. Dr. Ahmed H. Sameh, Computer Science Department, University of Illinois, Urbana, IL 61801
98. Dr. Michael Saunders, Systems Optimization Laboratory, Operations Research Department, Stanford University, Stanford, CA 94305
99. Dr. Robert Schreiber, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180
100. Dr. Martin H. Schultz, Department of Computer Science, Yale University, P.O. Box 2158 Yale Station, New Haven, CT 06520

101. Dr. David S. Scott, Intel Scientific Computers, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006
102. Dr. Lawrence F. Shampine, Mathematics Department Southern Methodist University Dallas, Texas 75275
103. Dr. Danny C. Sorensen, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439
104. Prof. G. W. Stewart, Computer Science Department, University of Maryland, College Park, MD 20742
105. Capt. John P. Thomas, Air Force Office of Scientific Research, Building 410, Bolling Air Force Base, Washington, DC 20332
106. Prof. Charles Van Loan, Department of Computer Science, Cornell University, Ithaca, NY 14853
107. Dr. Robert G. Voigt, ICASE, MS 132-C, NASA Langley Research Center, Hampton, VA 23665
108. Dr. Andrew B. White, Computing Division, Los Alamos National Laboratory, Los Alamos, NM 87545
109. Mr. Patrick H. Worley, Computer Science Department, Stanford University, Stanford, CA 94305
110. Dr. Arthur Wouk, Army Research Office, P.O. Box 12211, Research Triangle Park, North Carolina 27709
111. Dr. Margaret Wright, Systems Optimization Laboratory, Operations Research Department, Stanford University, Stanford, CA 94305
112. Office of Assistant Manager for Energy Research and Development, Department of Energy, Oak Ridge Operations Office, Oak Ridge, TN 37830
- 113-142.
Technical Information Center