



## Parallel Algorithms for Dense Linear Algebra Computations

K. A. Gallivan; R. J. Plemmons; A. H. Sameh

*SIAM Review*, Vol. 32, No. 1 (Mar., 1990), 54-135.

Stable URL:

<http://links.jstor.org/sici?sici=0036-1445%28199003%2932%3A1%3C54%3APAFDLA%3E2.0.CO%3B2-6>

*SIAM Review* is currently published by Society for Industrial and Applied Mathematics.

---

Your use of the JSTOR archive indicates your acceptance of JSTOR's Terms and Conditions of Use, available at <http://www.jstor.org/about/terms.html>. JSTOR's Terms and Conditions of Use provides, in part, that unless you have obtained prior permission, you may not download an entire issue of a journal or multiple copies of articles, and you may use content in the JSTOR archive only for your personal, non-commercial use.

Please contact the publisher regarding any further use of this work. Publisher contact information may be obtained at <http://www.jstor.org/journals/siam.html>.

Each copy of any part of a JSTOR transmission must contain the same copyright notice that appears on the screen or printed page of such transmission.

---

JSTOR is an independent not-for-profit organization dedicated to creating and preserving a digital archive of scholarly journals. For more information regarding JSTOR, please contact [support@jstor.org](mailto:support@jstor.org).

## PARALLEL ALGORITHMS FOR DENSE LINEAR ALGEBRA COMPUTATIONS\*

K. A. GALLIVAN<sup>†</sup>, R. J. PLEMMONS<sup>‡</sup>, AND A. H. SAMEH<sup>†</sup>

**Abstract.** Scientific and engineering research is becoming increasingly dependent upon the development and implementation of efficient parallel algorithms on modern high-performance computers. Numerical linear algebra is an indispensable tool in such research and this paper attempts to collect and describe a selection of some of its more important parallel algorithms. The purpose is to review the current status and to provide an overall perspective of parallel algorithms for solving dense, banded, or block-structured problems arising in the major areas of direct solution of linear systems, least squares computations, eigenvalue and singular value computations, and rapid elliptic solvers. A major emphasis is given here to certain computational primitives whose efficient execution on parallel and vector computers is essential in order to obtain high performance algorithms.

**Key words.** numerical linear algebra, parallel computation

**AMS(MOS) subject classifications.** 65-02, 65F05, 65F15, 65F20, 65N20

**1. Introduction.** Numerical linear algebra algorithms form the most widely-used computational tools in science and engineering. Matrix computations, including the solution of systems of linear equations, least squares problems, and algebraic eigenvalue problems, govern the performance of many applications on vector and parallel computers. With this in mind we have attempted in this paper to collect and describe a selection of what we consider to be some of the more important parallel algorithms in dense matrix computations.

Since the early surveys on parallel numerical algorithms by Miranker [133], Sameh [153], and Heller [91] there has been an explosion of research activities on this topic. Some of this work was surveyed in the 1985 article by Ortega and Voigt [138]. Their main emphasis, however, was on the solution of partial differential equations on vector and parallel computers. We also point to the textbook by Hockney and Jesshope [100] which includes some material on programming linear algebra algorithms on parallel machines. More recently, Ortega, Voigt, and Romine produced an extensive bibliography of parallel and vector numerical algorithms [139]; and Ortega [137] published a textbook containing a discussion of direct and iterative methods for solving linear systems on vector and parallel computers.

Our purpose in the present paper is to provide an overall perspective of parallel algorithms for dense matrix computations in linear system solvers, least squares problems, eigenvalue and singular-value problems, as well as rapid elliptic solvers. In this paper, dense problems are taken to include block tridiagonal matrices in which each block is dense, as well as algorithms for banded matrices which are dense within the band. In particular, we concentrate on approaches to these problems that have been used on available, research and commercial, shared memory multivector architectures

---

\* Received by the editors March 6, 1989; accepted for publication (in revised form) October 31, 1989.

<sup>†</sup> Center for Supercomputing Research and Development, University of Illinois, Urbana, Illinois 61801. This research was supported by the Department of Energy under grant DE-FG02-85ER25001 and the National Science Foundation under grants NSF-MIP-8410110 and NSF-CCR-8717942.

<sup>‡</sup> Departments of Computer Science and Mathematics, North Carolina State University, Raleigh, North Carolina 27695-8206. The work of this author was supported by the Air Force Office of Scientific Research under grant AFOSR-88-0285 and by the National Science Foundation under grant DMS-85-21154.

with a modest number of processors and distributed memory architectures such as the hypercube.

Since the amount of literature in these areas is very large we have attempted to select representative work in each. As a result, the topics and the level of detail at which each is treated can not help but be biased by the authors' interest. For example, considerable attention is given here to the discussion and performance analysis of certain computational primitives and algorithms for high performance machines with hierarchical memory systems. Given recent developments in numerical software technology, we believe this is appropriate and timely.

Many important topics relevant to parallel algorithms in numerical linear algebra are not discussed in this survey. Iterative methods for linear systems are not mentioned since the recent text by Ortega [137] contains a fairly comprehensive review of that topic, especially as it relates to the numerical solution of partial differential equations. Parallel algorithms using special techniques for solving generally sparse problems in linear algebra will also not be considered in this particular survey. Although significant results have recently been obtained, the topic is of sufficient complexity and importance to require a separate survey for adequate treatment.

The organization of the rest of this paper is as follows. Section 2 briefly discusses some of the important aspects of the architecture and the way in which they influence algorithm design. Section 3 contains a discussion of the decomposition of algorithms into computational primitives of varying degrees of complexity. Matrix multiplication, blocksize analysis, and triangular system solvers are emphasized. Algorithms for *LU* and *LU*-like factorizations on both shared and distributed memory systems are considered in §4. Parallel factorization schemes for block-tridiagonal systems, which arise in numerous application areas, are discussed in detail. Section 5 concerns parallel orthogonal factorization methods on shared and distributed memory systems for solving least squares problems. Recursive least squares computations, on local memory hypercube architectures, are also discussed in terms of applications to computations in control and signal processing. Eigenvalue and singular value problems are considered in §6. Finally, §7 contains a review of parallel techniques for rapid elliptic solvers of importance in the solution of separable elliptic partial differential equations. In particular, recent domain decomposition, block cyclic reduction, and boundary integral domain decomposition schemes are examined.

**2. Architectures of interest.** To satisfy the steadily increasing demand for computational power by users in science and engineering, supercomputer architects have responded with systems that achieve the required level of performance via progressively complex synergistic effects of the interaction of hardware, system software (e.g., restructuring compilers and operating systems), and system architecture (e.g., multivector processors and multilevel hierarchical memories). Algorithm designers are faced with a large variety of system configurations even within a fairly generic architectural class such as shared memory multivector processors. Furthermore, for any particular system in the architectural class, a CRAY-2 or Cedar [117], the algorithm designer encounters a complex relationship between performance, architectural parameters (cache size, number of processors), and algorithmic parameters (method used, blocksizes). As a result, codes for scientific computing such as numerical linear algebra take the form of a parameterized family of algorithms that can respond to changes within a particular architecture, e.g., changing the size of cluster or global memory on Cedar, or when moving from one member of an architectural family to another, e.g., Cedar to CRAY-2. The latter adaptation may, of course, involve chang-

ing the method used completely, say from Gaussian elimination with partial pivoting to a decomposition based on pairwise pivoting.

There are several consequences of such a situation. First, algorithm designers must be sensitive to architecture/algorithm mapping issues and any discussion of parallel numerical algorithms is incomplete if these issues are not addressed. Second, one of the main thrusts of parallel computing research must be to change the situation. That is, if scientific computing is to reap the full benefits of parallel processing, cooperative research involving expertise in the areas of parallel software development (debugging, restructuring compilers, etc.), numerical algorithms, and parallel architectures is required to develop parallel languages and programming environments along with parallel computer systems that mitigate this architectural sensitivity. Such cooperative work is underway at several institutions.

The architecture that first caused a widespread and substantial algorithm redesign activity in numerical computing is the vector processor. Such processors exploit the concept of *pipelining* computations. This technique decomposes operations of interest, e.g., floating point multiplication, into multiple stages and implements a pipelined functional unit that allows multiple instances of the computation to proceed simultaneously — one in each stage of the pipe.<sup>1</sup> Such parallelism is typically very fine-grain and requires the identification in algorithms of large amounts of homogeneous work applied to vector objects. Fortunately, numerical linear algebra is rich in such operations and the vector processor can be used with reasonable success. From the point of view of the functional unit, the basic algorithmic parameter that influences performance is the *vector length*, i.e., the number of elements on which the basic computation is to be performed. Architectural parameters that determine the performance for a particular vector length include cycle time, the number of stages of the pipeline, as well as any other startup costs involved in preparing the functional unit for performing the computations. Various models have been proposed in the literature to characterize the relationship between algorithmic and architectural parameters that determine the performance of vector processors. Perhaps the best known is that of Hockney and Jesshope [100].

The Cyber 205 is a memory-to-memory vector processor that has been successfully used for scientific computation. On every cycle of a vector operation multiple operands are read from memory, each of the functional unit stages operate on a set of vector elements that are moving through the pipe, and an element of the result of the operation is written to memory. Obviously, the influence of the functional unit on algorithmic parameter choices is not the only consideration required. Heavy demands are placed on the memory system in that it must process two reads and a write (along with any other control I/O) in a single cycle. Typically, such demands are met by using a highly interleaved or parallel memory system with  $M > 1$  memory modules whose aggregate bandwidth matches or exceeds that demanded by the pipeline. Elements of vectors are then assigned across the memory modules in a simple interleaved form, e.g.,  $v(i)$  is in module  $i \bmod M$ , or using more complex skewing schemes [193]. As a result, the reference pattern to the memory modules generated by accessing elements of a vector is crucial in determining the rate at which the memory system can supply data to the processor. The algorithmic parameter that encapsulates this information is the *stride* of vector access. For example, accessing the column of an array stored in column-major order results in a stride of 1 while accessing a row of

---

<sup>1</sup> The details of the architectural tradeoffs involved in a vector processor are somewhat surprisingly subtle and complex. For an excellent discussion of some of them see [174].



the same array requires a stride of  $lda$  where  $lda$  is the leading dimension of the array data object.

Not all vector processors are implemented with the three computational memory ports (2 reads/1 write) required by a memory-to-memory processor. The CRAY-1, one CPU of a CRAY-2 and one computational element of an Alliant FX/8 are examples of register-based vector processors that have a single port to memory and, to compensate for the loss in data transfer bandwidth, provide a set of vector registers internal to the processor to store operands and results.<sup>2</sup> Each of the registers can hold a vector of sufficient length to effectively use the pipelined functional units available. The major consequence of this, considered in detail below, is that such processors require careful management of data transfer between memory and register in order to achieve reasonable performance. In particular, care must be taken to reuse a register operand several times before reloading the register or to accumulate as many partial results of successive computations in the same register before storing the values to memory, i.e., reducing the number of loads and stores, respectively.

Some register-based vector processors also use two other techniques to improve performance. The first is the use of parallelism across functional units and ports. Multiple instructions that have no internal resource conflict, e.g., adding two vector registers with the result placed in a third and loading of a fourth register from memory, are executed simultaneously, making as much use of the available resources as possible. This influences kernel design in that careful ordering of assembler level instructions can improve the exploitation of the processor.

The second technique is essentially functional unit parallelism with certain resource dependences managed by the hardware at runtime. The technique is called *chaining* and it allows the result of one operation to be routed into another operation as an operand while both operations are active. For example, on a machine without chaining, loading a vector from memory into a register and adding it to another register would require two distinct nonoverlapped vector operations and therefore two startup periods, etc. Chaining allows the elements of the vector loaded into the first register to be made available, after a small amount of time, for use by the adder before the load is completed. Essentially, it appears as if the vector addition was taking one of its operands directly from memory. For processors that handle chaining of instructions automatically at runtime, careful consideration of the order of instructions used in implementing an algorithm or kernel is required. Some other vector processors, however, make the chaining of functional units and the memory port an explicit part of the vector instruction set. For example, the Alliant FX/8 allows one argument of a vector instruction to be given as an address in memory, thereby chaining the memory port and the appropriate functional units. The best example of this is the workhorse of its instruction set, the triad, which computes  $v_1 \leftarrow v_2 + \alpha x$ , where  $v_1$  and  $v_2$  are vector registers,  $\alpha$  is a scalar, and  $x$  is a vector in memory. This instruction explicitly chains the floating point multiplier and adder and the memory port. Such instruction constructs greatly simplify the exploitation of the chaining capabilities of a vector processor at the cost of the loss of a certain amount of flexibility.

While vector processors have been used and can deliver substantial performance for many computations, the quest for even more speed led to the availability and continuing development of MIMD multiprocessors and multivector processors. The processors on such machines are capable of executing arbitrary code segments in

---

<sup>2</sup> Some register-based vector processors also have multiple ports to memory in an attempt to have the best of both worlds, e.g., one CPU of a CRAY X-MP.

parallel and therefore subsume, assuming appropriate overhead levels, the fine-grain parallelism of vector processors. Shared memory architectures have the generic structure shown in Fig. 1(a). They are characterized by the fact that the interconnection network links all of the processors to all of the memory modules, i.e., a *user-controlled* processor can access any element of memory *without* the aid of another *user-controlled* processor. There is no concept of a direct connection between a processor and some subset of the remaining processors, i.e., a connection that does not involve the shared memory modules. Of course, in practice, few shared memory machines strictly adhere to this simple characterization. Many have a small amount of local memory associated with, and only accessible by, each processor. The aggregate size of these local memories is usually relatively insignificant compared to the large shared memory available. As local memory sizes increase, the architecture moves toward the distributed end of the architectural spectrum. Not surprisingly, the ability of the network/memory system to supply data to the multiple processors at a sufficient rate is one of the key components of performance of shared memory architectures. As a result, the organization and proper exploitation of this system must be carefully considered when designing high-performance algorithms.

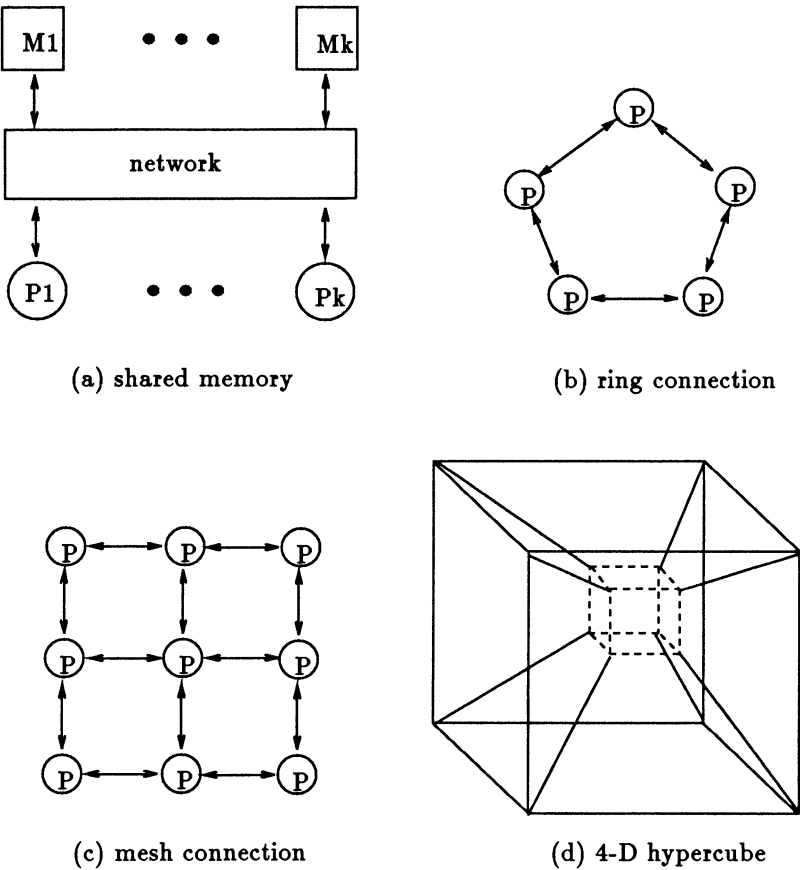


FIG. 1. Some memory/processor topologies.

The generic organization in Fig. 1 shows a highly interleaved or parallel memory system connected to the processors. This connection can take on several forms.

For a small number of processors and memory modules,  $p$ , a high-performance bus or crossbar switch can provide complete connectivity and reasonable performance. Unfortunately, such networks quickly become too costly as  $p$  increases. For larger systems, it is necessary to build scalable networks out of several smaller completely connected switches such as  $(s \times s)$ -crossbars. The  $\Omega$ -network of Lawrie [119] can connect  $p = s^k$  processors and memory modules with  $k$  network stages. Each stage comprises  $s^{k-1}$   $(s \times s)$ -crossbars, for a total of  $O(p \log_s p)$  switches. As with vector processors, data skewing schemes and access stride manipulation are important in balancing the memory bandwidth achieved with the aggregate computational rate of the processors. Ideally, the two should balance perfectly; in practice, keeping the two within a small multiple is achievable for numerical linear algebra computations via the skewing and stride manipulations or with the introduction of local memory (discussed below). As  $p$  increases, however, the latency for each memory access grows as  $O(k)$ . Fortunately, the addition of architectural features such as data prefetch mechanisms and local memory can provide some mitigation of this problem.

As mentioned above, one of the ways in which the performance of a large shared memory system can be improved is the introduction of local memories or caches with each processor. The idea is similar to the use of registers within vector processors in that data can be kept for reuse in small fast memory private to each processor. If sufficient data locality<sup>3</sup> is present in the computations the processor can proceed at a rate consistent with the data transfer bandwidth of the cache rather than the lower effective bandwidth of the large shared memory due to latency and conflicts. One difference between local memories/caches and vector registers, however, is that registers have a prescribed *shape* and must be used, for the most part, in vector operations; they must contain and be operated on as a vector  $v \in \Re^m$  where  $m$  is the vector length. On the other hand, local memory or caches can contain, up to a point, arbitrary data objects with no constraint on type or use. These differences can strongly affect the way that these architectural features influence algorithm parameter choices.

Another feature which can significantly influence the performance of an algorithm on a shared memory machine is the architectural support for synchronization of processors. These mechanisms are required for the assignment of parallel work to a processor and enforcing data dependences to ensure correct operation once the assignment is made. The support found on the various multiprocessors varies considerably. Some provide special purpose hardware for controlling small grain tasks on a moderate number of processors and simple TEST-AND-SET<sup>4</sup> synchronization in memory, e.g., the Alliant FX/8. Others provide more complex synchronization processors at the memory module or network level with capabilities such as FETCH-AND-OP or the Zhu-Yew primitives used on Cedar [196]. Finally, there are some which are oriented toward large-grain task parallelism which rely more on system-software-based synchronization mechanisms with relatively large cost to coordinate multiple tasks within a user's job, often at the same time with the tasks of other users.

The discussion above clearly shows that the optimization of algorithms for shared memory multivector architectures involve the consideration of the tradeoffs concern-

---

<sup>3</sup> A computation is said to have high data locality if the ratio of the data elements to the number of operations is small.

<sup>4</sup> The TEST-AND-SET operation allows for the indivisible action of accessing a memory location, testing its value, and setting the location if the test succeeds. It can be used as the basic building block of most synchronization primitives.

ing the influence of architectural features, such as parallelism, load balancing, vector computation, synchronization and parallel or hierarchical memory systems, on the choice of algorithm or kernel organization. Many of these are potentially contradictory. For example, increasing data locality by reorganizing the order of computations can directly conflict with the attempt to increase the vector length of other computations. The modeling and tradeoff analysis of these features will be discussed in detail below for selected topics.

Many shared memory parallel and multivector processors are commercially available over a wide range of price and performance. These include the Encore, Sequent, Alliant FX series, and supercomputers such as the CRAY X-MP and Y-MP, CRAY-2, and NEC. The Alliant FX/8 possesses most of the interesting architectural features that have influenced linear algebra algorithm design on shared memory processors recently; see the cluster blowup in Fig. 2. It consists of up to eight register-based vector processors or computational elements (CE's), each capable of delivering a peak rate of 11.75 Mflops for calculations using 64-bit data (two operations per cycle) implying a total peak rate of approximately 94 Mflops. The startup times for the vector instructions can reduce this rate significantly. For example, the vector triad instruction  $v \leftarrow v + \alpha x$  (the preferred instruction for achieving high performance in many codes) has a maximum performance of 68 Mflops. Each CE has eight 32-element vector registers and eight floating point scalar registers as well as other integer registers. The CE's are connected by a concurrency control bus (used as a synchronization facility). This mechanism allows an iteration of a parallel loop to be assigned to a processor within in time equivalent to a few floating point operations and provides synchronization support from lower iterations to higher iterations with a cost of a few cycles. As a result, the CE's can cooperate efficiently on parallel loops with iterations with a granularity of a small number of floating point operations.

There is only one memory port on each CE, like the CRAY-1 and a single CPU of the CRAY-2, therefore management of the vector registers is crucial. The CE's share the physical memory as well as a write-back cache that allows up to eight simultaneous accesses per cycle. The size of the cache can be configured from 64KB up to 512KB. The cache and the four-way interleaved main memory are connected through the main memory bus. Most of the detailed performance information for shared memory machines given below was obtained on this machine.

Distributed memory architectures can be roughly characterized in a fashion similar to that used above for shared memory. In particular, there are two major factors that distinguish them from shared memory architectures. These are the mode of memory access and the mode of synchronization.

On  $p$ -processor distributed memory machines with an aggregate memory size  $M$  each user-controlled processor has direct access to its local memory only, typically of size  $M/p$ . Accessing any other memory location requires the active participation of another user-controlled processor. As a result of this idea of direct interaction between processors to exchange data, distributed memory architectures are often identified by the topology of the connections between processors. Figure 1 illustrates three popular connection schemes. The ring topology (b) uses a linear nearest-neighbor bidirectional connection, essentially a linear array with a wrap-around connection between the first and last processor, while the mesh connection (c) provides two-dimensional nearest neighbor connections (wrap-around meshes are also used extensively). Both of these simple topologies work quite well for many numerical linear algebra algorithms. In particular, several algorithms are presented below for ring architectures. The hyper-

cube connection is perhaps the most discussed distributed memory topology recently. A four-dimensional cube is illustrated in (d). The connection patterns are, as the name implies, local connections in an arbitrarily dimensioned space. In general, a  $k$ -dimensional cube has  $2^k$  processors (vertices) each of which is connected to  $k$  other processors. It can be constructed from two  $(k - 1)$ -dimensional cubes by simply connecting corresponding vertices. As a result of this construction, the nodes have a very natural binary numbering scheme based on a Gray code. This construction also demonstrates one of the basic scalability problems of the hypercube in that the number of connections for a particular processor grows as the size of the cube increases as opposed to the constant local connection complexity of the simpler mesh and ring topologies. Many of the more common topologies, such as rings and meshes, can be embedded into a hypercube of appropriate dimension. In fact, many of the hypercube algorithms published use the cube as if it were one of the simpler topologies. Commercially available hypercubes include those by Ametek, Intel, and NCUBE.

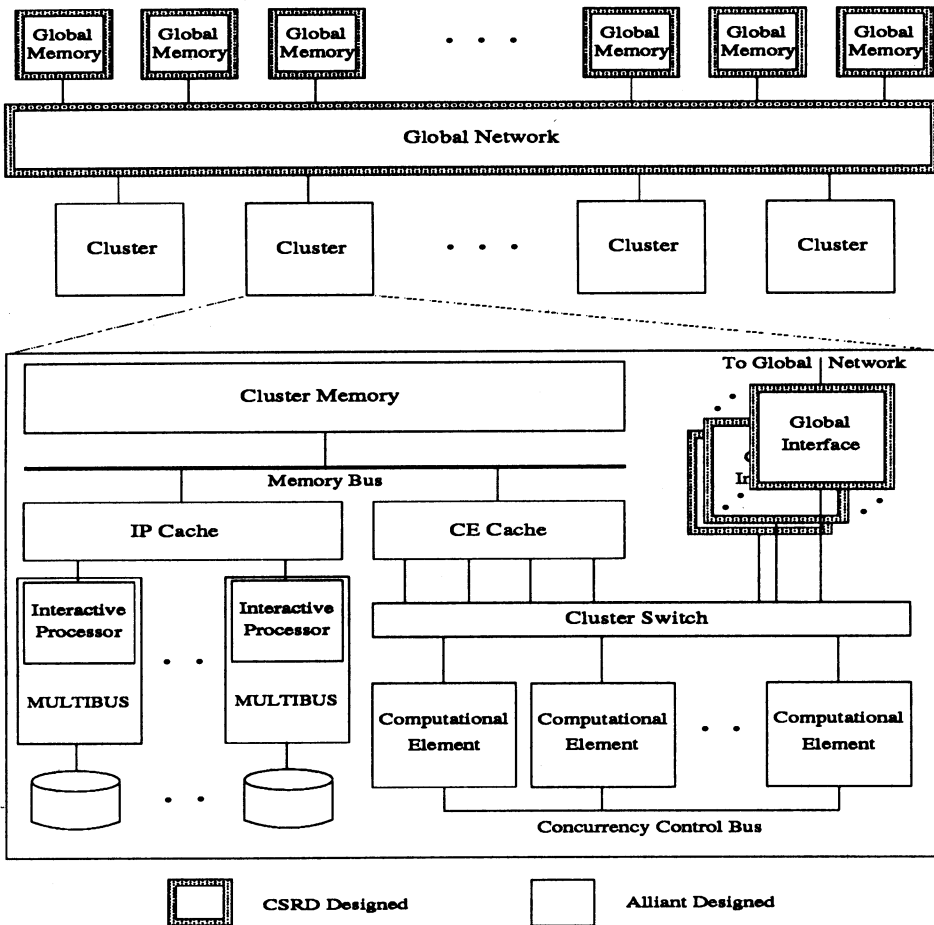


FIG. 2. The Cedar multiprocessor.

Synchronization on a distributed memory architecture, due to the memory accessing paradigm, is accomplished via a *data flow* mechanism rather than the indivisible update used in a large shared memory system. Computations can proceed on a pro-

cessor when due to its position in its local code the processor decides a computation is to be performed *and* all of the memory transactions involving operands for the computation in remote memory modules are complete. (These transactions are the interaction between the processors associated with the local memory and the remote memory modules mentioned above.) Clearly, since the synchronization is so enmeshed in the control and execution of interprocessor communication, the major algorithmic reorganization that can alter the efficiency of the synchronization on distributed memory machines is the partitioning of the computations (or similarly the data) so as to reduce the synchronization overhead required.

As we would expect, the algorithm/architecture mapping questions for a distributed memory machine change appreciably from those of shared memory. Since the machines tend to have more, but less powerful, processors, a key aspect of algorithm organization is the exposure of large amounts of parallelism. Once this is accomplished the major task is the partitioning of the data and the computations onto the processors. This partitioning must address several tradeoffs.

To reduce total execution time, a suitable balance must be achieved between the amount of communication required and efficient spreading of the parallel computations across the machine. One indicator of the efficient partitioning of the computations and data is the relationship between the load balance across processors and the amount of communication between processors. Typically, although not necessarily, a more balanced load produces a more parallel execution of the computations, ignoring for a moment delays due to communication. On the other hand, dispersing the computations over many processors may increase the amount of communication required and thereby negate the benefit of parallelism.

The property of data locality, which was very significant for shared memory machines in the management of registers and hierarchical memory systems, is also very important for some distributed memory machines in achieving the desired balance. Ideally, we would like to partition the data and computations across the processors and memory modules in such a way that a small amount of data is exchanged between processors at each stage of an algorithm, followed by the use of the received data in operations on many local data. As a result the cost of communication is completely amortized over the subsequent computations that make use of the data. If the partitioning of the computations and data also results in a balanced computational load the algorithm proceeds near the aggregate computational rate of the machine. This is, of course, identical to the hierarchical memory problem of amortizing a fetch of a data operand from the farthest level of memory by combining it with several operands in the nearest. Therefore many of the discussions to follow concerning the data-transfer-to-operations ratios that are motivated by shared hierarchical memory considerations are often directly applicable to the distributed memory case, although, as is shown below, there is often a tradeoff between data locality and the amount of exploitable parallelism.

Of course, there is a spectrum of architectures and a particular machine tends to have characteristics of both shared and distributed memory architectures. For these hybrid architectures efficient algorithms often involve a combination of techniques used to achieve high performance on the two extremes. An example of such an architecture that is used in this paper to facilitate the discussion of these algorithms is the Cedar system being built at the University of Illinois Center for Supercomputing Research and Development (see Fig. 2). It consists of clusters of vector processors connected to a large interleaved shared global memory — access to which can be accelerated

by data prefetching hardware. At this level it looks much like a conventional shared memory processor. However, each cluster is, in turn, a shared memory multivector processor, a slightly modified Alliant FX/8, whose cluster memory is accessible only by its CE's. The size of the cluster memory is fairly large and therefore the aggregate makes up a considerable distributed memory system. Consequently, the Cedar machine is characterized by its hierarchical organization in both memory and processing capabilities. The memory hierarchy consists of: vector registers private to each vector processor; cache and cluster memory shared by the processors within a cluster; and global memory shared by all processors in the system. Three levels of parallelism are also available: vectorization at the individual processor level, concurrency within each cluster, and global concurrency across clusters. Control and synchronization mechanisms between clusters are supported at two levels of granularity. The larger consists of large-grain tasks and multitasking synchronization primitives such as event waiting and posting similar to CRAY large-grain primitives. These primitives are relatively high cost in that they affect the state of the task from the point of view of the operating system, e.g., a task waiting for a task-level event is marked as blocked from execution and removed from the pool of tasks considered by the operating system when allocating computational resources. The second and lower-cost control mechanism is the SDOALL loop (for spread DOALL) which provides a self-scheduling loop structure whose iterations are grabbed and executed at the cluster level by *helper* tasks created at the initiation of the user's main task. Each iteration can then use the smaller grain parallelism and vectorization available within the cluster upon which it is executing. The medium grain SDOALL loop is ideal for moderately tight intercluster communication such as that required at the highest level of control in multicluster primitives with BLAS-like functionality that can be used in iterations such as the hybrid factorization routine presented in §4. Hardware support for synchronization between clusters on a much tighter level than the task events is supplied by synchronization processors, one per global memory module, which implements the Zhu-Yew synchronization primitives [196].

### 3. Computational primitives.

**3.1. Motivation.** The development of high-performance codes for a range of architectures is greatly simplified if the algorithms under consideration can be decomposed into computational primitives of varying degrees of complexity. As new architectures emerge, primitives with the appropriate functionality which exploit the novel architectural features are chosen and used to develop new forms of the algorithms. Over the years, such a strategy has been applied successfully to the development of dense linear algebra codes. These algorithms can be expressed in terms of computational primitives ranging from operations on matrix elements to those involving submatrices. As the pursuit of high performance has increased the complexity of computer architectures, the need to exploit this richness of decomposition has been reflected in the evolution of the Basic Linear Algebra Subroutines (BLAS).

The investigation of dense matrix algorithms in terms of decomposition into lower-level primitives such as the three levels of the BLAS has several advantages. First, for many presently available machines the computational granularity represented by single instances of the BLAS primitives from one of the levels or multiple instances executing simultaneously is sufficient for investigating the relative strengths and weaknesses of the architecture with respect to dense matrix computations. Consequently, since the primitive's computational complexity is manageable, it is possible to probe at an architecture/software level which is free of spurious software considerations

such as ways of tricking a restructuring compiler/code generator combination into producing the code we want. Thus, allowing meaningful conclusions to be reached about the most effective way to use a new machine.<sup>5</sup> Second, it aids in the identification of directions in language and restructuring technologies that would help in the implementation of high-performance scientific computing software. For example, matrix-manipulation constructs are already included in many proprietary extensions to Fortran due to the need for higher-level constructs to achieve high performance on some machines. Third, detailed knowledge of the efficient mapping of primitives to different architectures provides a way of thinking about algorithm design that facilitates the rapid generation of new versions of an algorithm by the direct manipulation of its algebraic formulation. (See the discussion of triangular system solvers below for a simple example.) Fourth, exposing the weaknesses of an architecture for the execution of basic primitives provides direction for architectural development. Finally, it simplifies the design of numerical software for nonexpert users. This typically occurs through the use of *total* primitives, i.e., primitives which hide all of the architectural details crucial to performance from the user. Code is designed in terms of a sequential series of calls to primitives which use all of the resources of the machine in the best way to achieve high performance. When such a strategy is possible a certain amount of performance portability is achieved as well. Unfortunately, many important architectures do not lend themselves to total primitives. Even in this case, however, the hiding of parts of the architecture via *partial* primitives is similarly beneficial. A user need only deal with managing the interaction of the partial primitives which may or may not execute simultaneously.

In this section, computational primitives from each level of the BLAS hierarchy are discussed and analyses of their efficiency on the architectures of interest in this paper are presented in various degrees of detail. Based on the discussion in §2 which indicates that the investigation of data locality is of great importance for both shared and distributed memory machines, special attention is given to identifying the strengths and weaknesses of each primitive in this regard and its relationship to the amount of exploitable parallelism.

**3.2. Architecture/algorithm analysis methodology.** The design of efficient computational primitives and algorithms that exploit them requires an understanding of the behavior of the algorithm/primitive performance as a function of certain system parameters (cache size, number of processors, etc.). It is particularly crucial that the analysis of this behavior identifies any contradictory trends that require tradeoff consideration, and the limits of performance improvement possible via a particular technique such as blocking. Additionally, preferences within a set of primitives can be identified by such an analysis, e.g., on certain architectures a rank-1 BLAS2 primitive does not perform as well as a matrix-vector multiplication. Ideally, the analysis should also yield insight into techniques a compiler could use to automatically restructure code to improve performance, e.g., on hierarchical memory systems [63], [75]. In this paper we are mostly concerned with analyses that concern the effects of hierarchical (registers, cache or local memory, global memory) or distributed memory systems.

As indicated earlier, the consideration of data locality and its relationship to the exploitable parallelism in an algorithm is a key activity in developing high-performance algorithms for both hierarchical shared memory and distributed memory

---

<sup>5</sup> Very loosely speaking this is usually the assembler level, i.e., the level at which the user has direct control over performance-critical algorithm/architecture tradeoffs.



architectures. In this section, we point out some performance modeling efforts concerning these tradeoffs that have appeared in the literature and present a summary of the techniques used on hierarchical shared memory architectures to produce some of the results discussed in later sections.

Several papers have appeared recently which discuss modeling the influence of a hierarchical memory on numerical algorithms, e.g., [3], [76], [99], [101]. Earlier work on virtual memory systems also discusses similar issues, e.g., the work of McKellar and Coffman [131], and Trivedi [185], [186]. In fact, the work of Trivedi performs many of the analyses for virtual memory systems that were later needed for both BLAS2 and BLAS3 such as the effect of blocking, loop orderings in the LU factorization, and prefetching. The details and assumptions for the hierarchical memory case, however, differ enough to require the further investigation that has taken place. Of particular interest here are studies by the groups at the University of Illinois on shared memory multivector processors (the Cedar Project) [9], [66], [67], [105] and at the California Institute of Technology on hypercubes (the Caltech Concurrent Computation Program) [59]–[61]. In these studies performance analyses were developed to express the influence of the block sizes, used in both the matrix multiplication primitives and the block algorithms built from them, on performance in terms of architectural parameters.

Gallivan, Jalby, Meier, and Sameh [67], [105] proposed the use of a decoupling methodology to analyze in terms of certain architectural parameters the trends in the relationship between the performance and the block sizes used when implementing BLAS3 primitives and block algorithms on a shared memory multivector processor. In particular, they considered an architecture comprising a moderate number ( $p$ ) of vector processors that share a small fast cache or local memory and a larger slower global memory. (The analysis is easily altered for the private cache or local memory case.) An example of such an architecture is the Alliant FX/8. In their methodology, two time components, whose sum is the total time for the algorithm, are analyzed separately. A region in the parameter space, i.e., the space of possible block size choices, that provides near-optimal behavior is produced for each time component. The intersection of these two regions yields a set of block sizes that should give near-optimal performance for the time function as a whole.

The first component considered is called the *arithmetic time* and is denoted  $T_a$ . This time represents the raw computational speed of the algorithm and is derived by ignoring the hierarchical nature of the memory system: it is the time required by the algorithm given that the cache is infinitely large. The second component of the time function considered is the degradation of the raw computational speed of the algorithm due to the use of a cache of size  $CS$  and a slower main memory. This component is called the *data loading overhead* and is denoted  $\Delta_l$ . The components  $T_a$  and  $\Delta_l$  are respectively proportional to the number of arithmetic operations and data transfers, from memory to cache, required by the algorithm; therefore, the total time for the algorithm is

$$(1) \quad T = T_a + \Delta_l = n_a \tau_a + n_l \tau_l,$$

where  $n_a$  and  $n_l$  are the number of operations and data transfers, and  $\tau_a$  and  $\tau_l$  are the associated proportionality constants or the “average” times for an operation and data load. *Note that no assumptions have been made concerning the overlap (or lack thereof) of computation and the loading of data in order to write  $T$  as a sum of these two terms.* The effect of such overlapping is seen through a reduction in

$\tau_l$ . This overlap effect can cause  $\tau_l$  to vary from zero, for machines which have a perfect prefetch capability from memory to cache, to  $t_l$ , where  $t_l$  is the amount of time it takes to transfer a single data element, for machines which must fetch data on demand sequentially from memory to cache.

The analysis of  $T_a$  considers the performance of the algorithm with respect to the architectural parameters of the multiple vector processors and the register-cache hierarchy under the assumption of an infinite cache. For some machines, the register-cache hierarchy is significant enough to require another application of the decoupling methodology with the added constraint of the *shape* of the registers. Typically, however, the analysis involves questions similar to those discussed concerning the BLAS2 below.

Rather than considering  $\Delta_l$  directly, the second portion of the analysis attempts a more modest goal. The data loading overhead can be analyzed so as to produce a region in the parameter space where the relative cost of the data loading  $\Delta_l/T_a$  is small. This analysis is accomplished by expressing  $\Delta_l/T_a$  in terms of two ratios: a *cache-miss* ratio and a *cost* ratio. Specifically,

$$(2) \quad \frac{\Delta_l}{T_a} = \lambda\mu$$

where  $\mu = n_l/n_a$  is the cache-miss ratio and  $\lambda = \tau_l/\tau_a$  is the cost ratio. For the purposes of qualitative analysis,  $\lambda$  can be bounded under various assumptions (average case, worst case, etc.) and trends in the behavior of the primitive or algorithm derived in terms of architectural parameters via the consideration of the behavior of the cache-miss ratio  $\mu$  as a function of the algorithm's blocksizes.

The utility of the results of the decoupling form of analysis depends upon the fact that the intersection of the near-optimal regions for each term is not empty or at least that the arithmetic time does not become unacceptably large when using parameter values in the region where small relative costs for data loading are achieved. For some algorithms this is not true; reducing the arithmetic time may directly conflict with reducing the relative cost of data loading. In some cases, a technique known as *multilevel blocking* can mitigate these conflicts [67]. In other cases, more machine-specific tradeoff studies must be performed. These studies typically involve probing the interaction of data motion to and from the various levels of memory and the underlying hardware to identify effective tradeoffs [64], [65].

On distributed memory machines, analyses in the spirit of the decoupling methodology can be performed. Fox, Otto, and Hey [59], [61] analyzed the efficiency of the *broadcast-multiply-roll* matrix multiplication algorithm and other numerical linear algebra algorithms on hypercubes in terms of similar parameters. In particular, they expressed efficiency in terms of the number of matrix elements per node (blocksize), the number of processors and a cost ratio  $t_{comm}/t_{flop}$  which gives the relative cost of communication to computation. Johnsson and Ho [110] presented a detailed analysis of matrix multiplication on a hypercube with special attention to the complexity of the *communication primitives* required and the associated data partitioning.

**3.3. First and second-level BLAS.** The first level of the BLAS comprises vector-vector operations such as dotproducts,  $\alpha \leftarrow x^T y$ , and vector triads (SAXPY),  $y \leftarrow y \pm \alpha x$  [121]. This level was used to implement the numerical linear algebra package LINPACK [38]. These primitives possess a simple one-dimensional parallelism especially suitable for vector processors with sufficient memory bandwidth to tolerate the high ratio of memory references to operations;  $\mu \approx \frac{3}{2}$  for the triad and  $\mu \approx 1$

for the dotproduct. The superiority of the dotproduct is due to the fact that it is a reduction operation that writes a scalar result after accumulating it in a register. The triad, on the other hand, produces a vector result and must therefore write  $n$  elements to memory in addition to reading the  $2n$  elements of the operands. For vector processors, performance tuning is limited to adjusting the vector length and stride of access. On multivector processors, both primitives are easily decomposed into several smaller versions of themselves for parallel execution. For the triad,  $\mu \approx \frac{3}{2} + \frac{p}{2n}$ , note that the fetch of  $\alpha$  becomes more significant, and  $\mu \approx 1 + \frac{p}{n}$  for the dotproduct, where  $p$  is the number of processors. As the number of processors increases to a maximum of  $n$ , the preference for the dotproduct over the triad is reversed. For  $p = n$  the triad requires  $O(1)$  time with  $\mu \approx 2$  while the dotproduct requires  $O(\log n)$  with  $\mu \approx 2$ . Such a reversal often occurs when considering large numbers of processors relative to the dimension of the primitive. The dependences graph of the reduction operation and its properties that produced a small  $\mu$  for a limited number of processors scale very poorly as  $p$  increases and translate directly into a relative increase in the amount of memory traffic required on a shared memory architecture and interprocessor communication on a distributed memory machine. (For a distributed memory machine, whether or not the reversal of preference occurs can depend strongly on the initial partitioning of the data.)

The advent of architectures with more than a few processors and high-performance register-based vector processors with limited processor-memory bandwidth such as the CRAY-1 exposed the limitations of the first level of the BLAS. New implementations of dense numerical linear algebra algorithms were developed which paid particular attention to vector register management and an emphasis on matrix-vector primitives resulted [24], [56]. This problem was later analyzed in a more systematic way in [42] and resulted in the definition of the extended BLAS or BLAS2 [40]. Architectures with a more substantial number of processors were also more efficiently used since matrix-vector operations consist essentially of multiple BLAS1 primitives that can be executed in parallel — roughly speaking they possess two-dimensional parallelism. The second level of the BLAS includes computations involving  $O(n^2)$  operations such as a matrix-vector multiplication,  $y \leftarrow y \pm Ax$ , and a rank-1 update,  $A \leftarrow A \pm xy^T$ . Note that these primitives subsume the triad and dotproduct BLAS1 primitives and become those primitives in the limit as one of the dimensions of  $A$  tends to 1. These primitives improve data locality in the sense that the number of memory references per operation can be reduced by accumulating the results of several vector operations in a vector register before writing to memory as in matrix-vector multiplication or by keeping in registers operands common to successive vector operations as in a rank-1 update. The two techniques, however, do not result in similar improvements in data locality. In general, it is preferable to write algorithms for register-based multivector processors in terms of matrix-vector multiplications rather than rank-1 updates.

To see this, consider first the efficiency of implementing the two BLAS2 primitives as a set of BLAS1 primitives each of the order of the matrix. (For the rank-1 it is only possible to use the triad; the matrix-vector multiplication allows a choice of primitives.) If the matrix dimensions  $n_1$  and  $n_2$  are larger than the register size<sup>6</sup> of any of the processors there is no possibility of efficient register reuse and the value of  $\mu$  remains at the disappointing BLAS1 level. For problems where either  $n_1$  or

---

<sup>6</sup> The term register size does not necessarily mean the vector length of a single vector register. It can also refer to the aggregate size of all of the vector registers used in a processor in a given implementation of the primitive.

$n_2$  is smaller than the register size, however, it is possible to reuse the registers in such a way that both primitives achieve their theoretical minimum values of  $\mu$ ;  $\mu = 1 + 1/2n_1 + 1/2n_2$  for the rank-1 update and  $\mu = 1/2 + 1/2n_1 + 1/n_2$  for the matrix-vector product. For the small rank-1, this local optimal is achieved by reading the small vector into vector register once and reusing it to form a triad with each row or column of the matrix in turn. As a result, each element of the matrix and the two vectors are loaded into the processor exactly once and the elements of the matrix are written exactly once — the optimal data transfer behavior for a rank-1 update. For the matrix-vector product, the technique depends upon whether  $n_1$  or  $n_2$  is the small dimension. If it is  $n_2$  then a technique similar to the rank-1 update is used. The vector  $x$  is loaded into a register once. Each row of  $A$  is read in turn and used in an inner product calculation with  $x$  in the register, and the result is then added to the appropriate element of  $y$  and written back to memory. Every data element is read and written the minimum number of times. If the small dimension is  $n_1$  then a slightly different technique is used. The result of the operation,  $y$ , is accumulated in a vector register, thereby suppressing the writes back to memory of partial sums.

As long as  $n_1$  or  $n_2$  do not get very small, which implies that the primitives are degenerating into a first level primitive, the values are an improvement compared to their limiting first level primitives. Of course, the rank-1 update still has a value of  $\mu$  similar to the dotproduct BLAS1 primitive, but it has the advantage of more exploitable parallelism. If these results could be maintained for arbitrary  $n_1$  and  $n_2$ , the superiority of the BLAS2 on register-based multivector processors would be established.

To show that this is indeed possible, we will exploit the richness of structure present in linear algebra computations and partition the primitives into smaller versions of themselves. This is accomplished by partitioning  $A$  into  $k_1k_2$  submatrices  $A_{ij} \in \mathbb{R}^{m_1 \times m_2}$ , where it is assumed for simplicity that  $n_i = k_i m_i$  with  $k_i$  and  $m_i$  integers, and partitioning  $x$  and  $y$  conformally. The blocksizes which determine the partitioning are chosen so that the smaller instances of the primitives are locally optimal with respect to their values of  $\mu$ .

The rank-1 update is thus reduced to  $k_1k_2$  independent small rank-1 updates. The resulting global  $\mu$  value for the entire rank-1 update is  $\mu = 1 + 1/2m_1 + 1/2m_2$ . Now consider its behavior as  $p$ , the number of register-based vector processors used, increases. For small and moderate  $p$ , one of the blocksizes, say  $m_1$ , could be taken equal to the corresponding dimension of the matrix,  $n_1$  (the choice of  $m_1$  or  $m_2$  simply depends upon the shape of the matrix and the exact number of processors). It follows that  $\mu = 1 + 1/2r + 1/2n_2$  where  $r$  is the register length. As  $p$  increases further, a true two-dimensional partitioning must be used. So we set  $p = k_1k_2$  which balances the computational load and the amount of data required by each processor. Since the register size determines the largest vector object we can work with and extra transfers to and from registers translate directly into additional time, we make  $m_1^{-1} + m_2^{-1}$  as small as possible under the constraint that either  $m_1 \leq r$  or  $m_2 \leq r$ , depending on the implementation chosen for the register-based smaller rank-1 update. Consequently,

$$\mu = 1 + \frac{p}{2n_1n_2}(m_1 + m_2)$$

and the algorithm requires  $O(m_1m_2)$  time. At the limit of available parallelism,  $p = n_1n_2$  and the rank-1 update requires  $O(1)$  time with  $\mu = 2$ . This is the same as the best BLAS1 primitive. This is not surprising since in the limit each processor is doing essentially the same scalar computation as the BLAS1 triad. The only difference

is that in the BLAS2 case there is much more exploitable parallelism. Note also that at some point while increasing the number of processors the vector length used by each processor will fall below the breakeven point for the use of the vector capability of the processor, and the switch should be made to scalar mode.

A similar decomposition technique can be used for the matrix-vector product primitive  $y \leftarrow y \pm Ax$ . The matrix is partitioned into submatrices  $A_{ij} \in \mathbb{R}^{m_1 \times m_2}$  and partitioning  $x$  and  $y$  conformally. The resulting algorithm is

```
do  $i = 1, k_1$ 
   $y_i \leftarrow y_i + A_{i1}x_1 + \dots + A_{ik_2}x_{k_2}$ 
end do
```

All of the basic computations  $z \leftarrow z + A_{ij}x_j$  can proceed in parallel with a fan-in dependence graph required on the update of the  $y_i$  if  $k_2 > 1$ . As before, for a small to moderate number of processors one of the  $m_i$  can be set to the register length and the other to the remaining dimension of  $A$ . If  $i = 1$  then no synchronization is required since  $k_2 = n_2$  and the loop can execute in parallel. The resulting global  $\mu$  is

$$\mu = \frac{1}{2} + \frac{1}{2r} + \frac{1}{n_2},$$

where  $r$  is the vector length. If  $i = 2$

$$\mu = \frac{1}{2} + \frac{1}{2n_1} + \frac{1}{r}.$$

In the latter case,  $k_1$  is equal to 1 and synchronization is required. However, since the number of processors is assumed small the partial sums from local matrix-vector products can be accumulated in a vector of length  $n_1$  private to each processor (not necessarily a register). After all processors are finished accumulating their partial sums, a simple fan-in of the results can be done. The time required is  $O(m_1 m_2)$ . Note that on a moderate number of processors the matrix-vector primitive is twice as efficient as the rank-1 primitive of the same size. Consequently, when implementing algorithms with BLAS2 primitives on a register-based multivector architecture with a moderate number of processors, a matrix-vector product-based algorithm will significantly outperform the same algorithm based on a rank-1 update.

As with the rank-1 update it is possible to derive an estimate of the time and the value of  $\mu$  for the case where a two-dimensional partitioning is used with  $p = k_1 k_2$ . In this case, not only must the transfers be computed for the small matrix-vector products performed by each of the processors, but also the transfers associated with the  $k_1$  independent fan-in trees which sum together the partial sums into the final values of  $y_i$  for  $1 \leq i \leq k_1$ . The time required is  $O(m_1 m_2) + O(m_1 \log_2 k_2)$  with

$$\mu = \frac{1}{2} + \frac{p}{n_1 n_2} \left[ 1 + m_2 \left( \frac{1}{2} + \frac{1}{n_2} \right) \right].$$

As with all of the other primitives, when  $p$  is as large as possible, in this case  $p = n_1 n_2$ , the value of  $\mu$  increases to approximately 2. Due to the reduction nature of the matrix-vector product, its time has a lower bound of  $O(\log n_2)$ .

The results above demonstrate several important points about first- and second-level BLAS primitives. The most important is that for register-based multivector

processors with a moderate number of processors, there can be a significant difference between the performance of a given algorithm when implemented in terms of the four primitives discussed above. This performance order is given from worse to best in terms of decreasing values of  $\mu$ . The triad with  $\mu \approx \frac{3}{2}$  does far too many spurious data transfers to be of use on a processor with a single port to memory. The dotproduct improves the ratio to  $\mu \approx 1$  but not all processors have high-performance capabilities. The BLAS2 rank-1 update primitive also has  $\mu \approx 1$  but it does not depend upon efficient reduction operations on vector registers being available on a processor and its extra dimension of parallelism makes it more flexible than the previous primitives. By far, however, the preferred primitive for such an architecture is the matrix-vector product due to its superior register management.

The second observation from the results above is how the preferences can reverse when the architecture used is radically altered. In this case we considered increasing the number of register-based vector processors available to the maximum needed. It was shown that in the limit all have similar register-memory transfer behavior and the nonreduction operations have a distinct advantage *if it is assumed that the data and computations have been partitioned ideally*. This last point is crucial. Our discussions implicitly assumed a shared memory architecture when increasing the number of processors. While the results do hold for certain distributed memory architectures, they can be very sensitive to the assumptions concerning initial data partitioning. If for some reason the data had been partitioned in a different way the trends need not be the same.

### 3.4. Third-level BLAS.

**3.4.1. Motivation.** The highest level of the BLAS is motivated by the use of memory hierarchies. On such systems, only the lowest level of the hierarchy (or in some cases the two lowest, e.g., registers and cache) are able to supply data at the computational bandwidth of the processors. Hence, data locality must be exploited to allow computations to involve mostly data located in the lowest levels. This allows the cost of the data transfer between levels to be amortized over several operations performed at the computational bandwidth of the processors. This problem of data reuse in the design of algorithms has been studied since the beginning of scientific computing. Early machines, which had small physical memories, required the use of secondary storage such as tape or disk to hold all of the data for a problem. Similar considerations were also needed on later machines with paged virtual memory systems. The *block* algorithms developed for such architectures relied on transferring large submatrices between different levels of storage, with prepagings in some cases, and localizing operations to achieve acceptable performance.

Of course, the resulting matrix-matrix primitives could have been used in algorithms for the machines which motivated the BLAS2. Indeed, as Calahan points out [23], the use of matrix-matrix modules was considered when developing algorithms for the CRAY-1. The hierarchy, however, was not distinct enough to achieve a significant advantage over BLAS2 primitives. The introduction of the CRAY X-MP and its additional memory ports delayed even further the move to the next level of the BLAS. It was finally caused by the availability of high-performance architectures which rely on the use of a hierarchical memory system and with more profound performance consequences when not used correctly. Agarwal and Gustavson designed matrix multiplication primitives and block algorithms for solving linear systems to exploit the cache memory on the IBM 3090 in the latter part of 1984. These evolved into the algorithms contained in ESSL, first released in the middle of 1985, for the IBM 3090

with vector processing capabilities [1], [84], [130], and more recently for the multi-processor version of the architecture [2]. A numerical linear algebra library based on block methods was developed and its performance analyzed in terms of architectural parameters in 1985 and early 1986 for a single cluster of the Cedar machine, the multivector processor Alliant FX/8 [9], [105], [156]. At approximately the same time, Calahan developed block LU factorization algorithms for one CPU of the CRAY-2 [23]. In 1985, Bischof and Van Loan developed the use of block Householder reflectors in computing the QR factorization and presented results on an FPS-164/MAX [16].

The development of these routines and numerical linear algebra libraries clearly demonstrated that a third level of primitives, or BLAS3, based on matrix-matrix computations was required to achieve high performance on the emerging architectures. Such primitives achieve a significant improvement in data locality, i.e., the data locality is no longer effectively independent of problem size as it is for the first two levels of the BLAS. Third-level primitives perform  $O(n^3)$  operations on  $O(n^2)$  data, and they increase the parallelism available by yet another dimension by essentially consisting of multiple independent BLAS2 primitives.

Since the reawakening of interest in block methods for linear algebra, many papers have appeared in the literature considering the topic on various machines, e.g., [5], [44], [149]. The techniques have become so accepted that some manufacturers now provide high-performance libraries which contain block methods and matrix-matrix primitives. Some, such as Alliant, provide matrix multiplication intrinsics i.e. their concurrent/vector processing extensions to Fortran. In 1987, an effort began to standardize for Fortran 77 the BLAS3 primitives and block methods for numerical linear algebra [35], [37], [39].

**3.4.2. Some algorithms.** The most basic BLAS3 primitive is a simple matrix operation of the form

$$(3) \quad C \leftarrow C + AB,$$

where  $C$ ,  $A$ , and  $B$  are  $n_1 \times n_3$ ,  $n_1 \times n_2$ , and  $n_2 \times n_3$  matrices, respectively. Clearly, this primitive subsumes the rank-1 update, ( $n_2 = 1$ ), and matrix-vector multiplication, ( $n_3 = 1$ ), BLAS2 primitives. In block algorithms, it is most often used as a rank- $\omega$  update ( $n_2 = \omega \ll n_1, n_3$ ) or a matrix multiplied by several vectors ( $n_3 = \omega \ll n_1, n_2$ ). The analysis of the parallel complexity of such a computation has been the subject of much study. In this section we give a brief summary of some generic algorithms and mention some implementations on various machines that have appeared recently in the literature.

The basic scalar computation can be expressed as

```

do r = 1, n3
  do s = 1, n1
    do t = 1, n2
      cs,r = cs,r + as,tbt,r
    end do
  end do
end do

```

where  $c_{s,r}$ ,  $a_{s,t}$ , and  $b_{t,r}$  denote the elements of  $C$ ,  $A$  respectively  $B$ .

There are three basic generic approaches to performing these computations which correspond to different choices of orderings of the loops. They are called the *inner*,

*middle*, and *outer* product methods due to the fundamental kernels used and correspond to the following code segments:

```
inner_product:
  do  $r = 1, n_3$ 
    do  $s = 1, n_1$ 
       $c_{s,r} = c_{s,r} + \text{inner\_prod}(a_{s,*}, b_{*,r})$ 
    end do
  end do
```

```
middle_product:
  do  $r = 1, n_3$ 
     $c_{*,r} = c_{*,r} + Ab_{*,r}$ 
  end do
```

and

```
outer_product:
  do  $t = 1, n_2$ 
     $C = C + a_{*,t}b_{t,*}^T$ 
  end do.
```

Each has its advantages and disadvantages for various problem shapes and architectures. All have immediate generalizations involving submatrices. These issues are discussed in the literature, e.g., [100], [137], in several places and will not be repeated here. We do note, however, that for register-based vector and multivector processors with one port to memory, the middle product algorithm facilitates the efficient use of the vector registers and data bandwidth to cache of each processor, and exploits the chaining of the multiplier, adder, and data fetch available on many systems. This is accomplished by performing, possibly in parallel, multiple matrix-vector products — the preferred BLAS2 primitive for vector register management. When the vector processors are such that register-register operations are significantly faster than chained operations from local memory or cache, a more sophisticated two-level generalization of the blocking strategy discussed below can be used to achieve high performance.

Madsen, Rodrigue, and Karush considered, for use on the CDC STAR-100 vector processor, a slightly more exotic matrix multiplication based on storing and manipulating the diagonals of matrices [127]. Their motivation was mitigating the performance degradation of the algorithms above for banded matrices and the difficulties in accessing the transpose of a matrix on some machines.

The BLAS3 primitive implemented for a single cluster of the Cedar machine [66], [67], [105] and applicable to machines with a moderate number of reasonably coupled multivector processors with a shared cache implements a block version of the basic matrix multiplication loops. It proceeds by partitioning the matrices  $C$ ,  $A$ , and  $B$  into submatrices  $C_{ij}$ ,  $A_{ik}$ , and  $B_{kj}$  whose dimensions are  $m_1 \times m_3$ ,  $m_1 \times m_2$ , and  $m_2 \times m_3$ , respectively. The basic loop is of the form

```
do  $i = 1, k_1$ 
  do  $k = 1, k_2$ 
    do  $j = 1, k_3$ 
```



$C_{ij} = C_{ij} + A_{ik} * B_{kj}$   
end do

end do

end do

where  $n_1 = k_1 m_1$ ,  $n_2 = k_2 m_2$ , and  $n_3 = k_3 m_3$ , and  $k_1$ ,  $k_2$ , and  $k_3$  are assumed to be positive integers for simplicity.

The block operations  $C_{ij} = C_{ij} + A_{ik} * B_{kj}$  possess a large amount of concurrent and vectorizable computations, so the algorithm proceeds by dedicating the full resources of the  $p$  vector processors to each of the block operations in turn. The kernel block multiplication can be computed by any of the basic concurrent/vector algorithms. As noted above the middle product algorithm which performs several multiplications of  $A_{ik}$  and columns of  $B_{kj}$  in parallel is well suited for register-based architectures like the Alliant FX/8, hence it is assumed in the analysis below.

There are, of course, several possible orderings of the block loops and several other kernels that can be used for the block operations.<sup>7</sup> If, for example, the processors are not tightly coupled enough parallelism can be moved to the block level. This can also be useful in the case of private caches or local memories for each processor. As is shown below this particular ordering (or one trivially related to it) is appropriate for use in the block algorithms discussed in later sections. However, when developing a robust BLAS3 library, kernels for the block operations which differ from those discussed below and alternate orderings must be analyzed so that selection of the appropriate form of the routine can be done at runtime based on the *shape* of the problem. This is especially important for cases with extreme shapes, e.g., guaranteeing smooth performance characteristics as the shapes become BLAS2-like.

Clearly, if the number of processors are increased to  $p = n_1 n_2 n_3$  the inner product form of the algorithm can generate the result in  $O(\log_2 n_2)$  time. For a shared memory machine, such an approach would place tremendous strain on a highly interleaved or parallel memory systems. As mentioned earlier, one way that such strain is mitigated is by assigning elements of structured variables to the memory banks in such a way as to minimize the chance of conflicts when accessing certain subsections of the data. For the inner product algorithm it is particularly important that the row and columns of matrices be accessible in a conflict free manner. One of the easiest memory module mapping strategies that achieves this goal dates back to the ILLIAC IV ([114], [115], also see [116]). The technique is called the *skewed storage scheme*. In it the elements of each row of a matrix are assigned in an interleaved fashion across the memory modules. However, when assigning the first element of a row it is placed in the memory module that is skewed by one from the module that contained the first element of the previous row. Any row or column of a matrix can now be accessed in a conflict free fashion. Matrix multiplication algorithms for the distributed memory ILLIAC IV were developed based on this scheme which can be easily adapted to the shared memory situation.

If we are willing to sacrifice some numerical stability, fast schemes which use less than  $O(n^3)$  operations can be used to multiply two matrices. In [95], Higham has analyzed this loss of stability for Strassen's method [175] and concluded that it does not preclude the effective use of the method as a BLAS3 kernel. Recently, Bailey has

<sup>7</sup> The  $i-j-k$  ordering of the block loops, for example, produces distinctly different block sizes and shapes [105]. Its use can be motivated by the desire to keep a block of  $C$  in cache while accumulating its final value. This implies that a block of  $A$  must reside in the cache simultaneously thereby altering the optimal shapes.

considered the use of Strassen's method to multiply matrices on the CRAY-2 [7]. The increased performance compared to CRAY's MXM library routine is achieved via the reduced operation count implicit in the method and the careful use of local memory via an algorithm due to Calahan. Speedups as high as 2.01 are reported compared to CRAY's library routine on a single CPU. Bailey also notes that the algorithm is very amenable to use on multiple CPU's of the CRAY-2 although no such results are presented.

The *broadcast-multiply-roll* algorithm for matrix multiplication described and analyzed by Fox et al. is representative of distributed memory algorithms [59]–[61]. (For other distributed memory algorithms see [78], [129], [135].) Consider the calculation of  $C \leftarrow C + AB$  where  $A, B, C \in \mathbb{R}^{n \times n}$ . Assume the processors are connected as a two-dimensional wrap-around mesh and the square subblock with index  $(i, j)$  of each matrix starts out in the memory of the processor correspondingly indexed. The algorithm consists of  $\sqrt{n}$  steps each of which consists of broadcast, multiply, and roll phases. In particular, on step  $i$  ( $i = 0, \dots, \sqrt{n} - 1$ ) the processor in each row owning  $A_{j, (j+i) \bmod \sqrt{n}}$  broadcasts it to the rest of the processors in the row which store it in a local work array  $T$ . Each processor then multiplies  $T$  by the subblock of  $B$  presently in its memory and adds it to the subblock of  $C$  that it owns. The final phase of each step consists of rolling the matrix  $B$  up one row in the mesh with appropriate wrap-around at the ends of the mesh. In other words, each processor transmits the subblock of  $B$  it has in its memory to the processor in the same column of the mesh but one row up. The repetition of this three-phase step  $\sqrt{n}$  times corresponds to the number of steps required to let each subblock of  $B$  return to its original processor.

Finally, Johnsson and Ho have considered the implementation of matrix multiplication on a hypercube [110]. In this work they consider the implementation of the computational primitive in terms of *communication* primitives some of which implicitly perform computations as the data move through the cube. As a result, users can write their algorithms as a sequence of calls to these data motion primitives in a fashion similar to the method advocated with respect to the computational primitives discussed above.

**3.4.3. Blocksize analysis.** In this section we summarize the application of the decoupling methodology to the matrix multiplication algorithm for the single cluster of the Cedar machine described above. Recall that the block level loops were

```

do  $i = 1, k_1$ 
  do  $k = 1, k_2$ 
    do  $j = 1, k_3$ 
       $C_{ij} = C_{ij} + A_{ik} * B_{kj}$ 
    end do
  end do
end do

```

where  $n_1 = k_1 m_1$ ,  $n_2 = k_2 m_2$ , and  $n_3 = k_3 m_3$ , and  $k_1$ ,  $k_2$ , and  $k_3$  are assumed to be positive integers. Each block operation  $C_{ij} = C_{ij} + A_{ik} * B_{kj}$  uses the resources of the  $p$  vector processors by performing matrix-vector products in parallel.

Values of  $m_1$ ,  $m_2$ , and  $m_3$  which yield near-optimal values of the arithmetic time for the kernel can be determined by an analysis similar to those presented above for the BLAS2. The essential tradeoffs require balancing the parallel and vector processing capabilities and the bandwidth restrictions due to the single port to memory on each processor. For the Alliant FX/8, the values of  $m_1$ ,  $m_2$ , and  $m_3$  chosen according to the preceding reasoning are:  $m_1 = 32k$  or is large;  $m_2 \geq 16$  to 32 depending on the

overhead surrounding the accumulation; and  $m_3 = 8k$  or is large.

The reduction of the data loading overhead reduces to a simple constrained minimization problem. Since the submatrices  $A_{ik}$  are associated with the inner loop, it is assumed that each  $A_{ik}$  is loaded once and kept in cache for the duration of the  $j$  loop. Similarly, it is assumed that each of the  $C_{ij}$  and  $B_{kj}$  are loaded into cache repeatedly. Note that the conservative approach is taken in that no distinction is made between reads and writes in that  $\lambda$  is set under the pessimistic assumption that anything loaded has to be written back whether or not it was updated. Some cases where this distinction becomes important are discussed below.

It is easily seen by considering the number of transfers required that the cache-miss ratio,  $\mu$ , is given by

$$(4) \quad \mu = \frac{1}{2m_1} + \frac{1}{2m_2} + \frac{1}{2n_3}.$$

The theoretical minimum, given an infinite cache, is

$$\mu = \frac{1}{2n_1} + \frac{1}{2n_2} + \frac{1}{2n_3}.$$

Constraints for the optimization of the terms involving  $m_1$  and  $m_2$  are generated by determining what amount of data must fit into cache at any given time and requiring that this quantity be bounded by the cache size  $CS$ . The final set of constraints come from the fact that the submatrices cannot be larger than the matrices being multiplied. Therefore, the minimization of the number of loads performed by the BLAS3 primitive is equivalent to the solution of the minimization problem

$$(5) \quad \min \rho(m_1, m_2) = m_1^{-1} + m_2^{-1}$$

$$\text{subject to } m_2(m_1 + p) \leq CS$$

$$1 \leq m_1 \leq n_1$$

$$1 \leq m_2 \leq n_2,$$

where  $CS$  is the cache size and  $p$  is the number of processors. The constraints trace a rectangle and an hyperbola in the  $(m_1, m_2)$ -plane.

The solution to the minimization problem separates the  $(n_1, n_2)$  plane into four distinct regions; two of which are of interest for the rank- $\omega$  update and matrix-times- $\omega$ -vectors primitives, and general large dense matrix multiplication (see [67] for details). These can be summarized as:

1. The value of  $m_3$  is arbitrary and taken to be  $n_3$ .
2. If  $n_2(n_1 + p) > CS$  and  $n_2 \leq CS(\sqrt{CS} + p)^{-1}$

$$m_1 = \frac{CS}{n_2} - p \quad \text{and} \quad m_2 = n_2.$$

3. If  $n_2(n_1 + p) > CS$ ,  $n_1 > \sqrt{CS}$ , and  $n_2 > CS(\sqrt{CS} + p)^{-1}$

$$m_1 = \sqrt{CS} \quad \text{and} \quad m_2 = \frac{CS}{\sqrt{CS} + p}.$$

Note that since the near-optimal region for the arithmetic time component was unbounded in the positive direction, there is a nontrivial intersection between it and the near-optimal region for the data loading component. This implies that, except for some boundary cases where  $n_1$ ,  $n_2$ , and/or  $n_3$  become small, the decoupling methodology does yield a strategy which can be used to choose near-optimal blocksizes for BLAS3 primitives. (The troublesome boundary cases can be handled by altering the block-loop ordering or choosing a different form of the block multiplication kernel.)

For the rank- $\omega$  primitive this results in a partitioning of the form

$$(6) \quad \begin{pmatrix} C_1 \\ \vdots \\ C_k \end{pmatrix} \leftarrow \begin{pmatrix} C_1 \\ \vdots \\ C_k \end{pmatrix} \pm \begin{pmatrix} A_1 \\ \vdots \\ A_k \end{pmatrix} B,$$

where the blocksizes are given by the case above with  $n_2 = \omega$  and small. Note that the block loops simplify to

```
do i = 1, k
    Ci = Ci + Ai * B
end do
```

and parallelism at the block-loop level becomes trivially exploitable when necessary. Also note that each block of the matrix  $C$  is read and written exactly once implying that this blocking maintains the minimum number of writes back to main memory.

For large dense matrix multiplication and for the matrix-times- $\omega$ -vectors primitive the partitioning is

$$(7) \quad \begin{pmatrix} C_1 \\ \vdots \\ C_k \end{pmatrix} \leftarrow \begin{pmatrix} C_1 \\ \vdots \\ C_k \end{pmatrix} \pm \begin{pmatrix} A_{11} & \cdots & A_{1m} \\ \vdots & \ddots & \vdots \\ A_{k1} & \cdots & A_{km} \end{pmatrix} \begin{pmatrix} B_1 \\ \vdots \\ B_m \end{pmatrix},$$

and the block loops reduce to

```
do i = 1, k
    do j = 1, m
        Ci = Ci + Aij * Bj
    end do
end do.
```

Once again block parallelism is obviously exploitable when needed. Note however that the blocks of  $C$  are written to several times. In general, these writes are not significant since the blocksizes have been chosen to reduce the significance of all transfers (including these writes) to a negligible level. The i-j-k block loop ordering can be used and analyzed in a similar fashion if it is desirable to accumulate a block of  $C$  in local memory. The blocksizes that result are, of course, different from the one shown above (see [105]).

The key observation with respect to the behavior of  $\mu$  for BLAS3 primitives is that it decreases hyperbolically as a function of  $m_1$  and  $m_2$ . (This assumes this particular block loop ordering but similar statements can be made about the others.)

It follows that the relative cost of transferring data decreases rapidly and reaches a global minimum of the form

$$(8) \quad \lambda\mu = \frac{\lambda}{\sqrt{CS}} + \frac{p\lambda}{2CS} + \frac{\lambda}{2n_3}.$$

Therefore, assuming that  $n_3$  is much larger than  $\sqrt{CS}$  (large dense matrix multiplication), data loading overhead can be reduced to  $O(1/\sqrt{CS})$ . This limit on the cache-miss ratio reduction due to blocking is consistent with the bound derived in Hong and Kung [101]. For BLAS3 primitives where one of the dimensions is smaller than the others, with value denoted  $\omega$ , the data loading overhead is a satisfactory  $O(1/\omega)$ .

The hyperbolic nature of the data loading overhead implies that reasonable performance can be achieved without increasing the block sizes to the near-optimal values given above. Of course, exactly how large  $m_1$  and  $m_2$  must be in order to reduce the data loading overhead to an acceptable amount depends on the cost ratio  $\lambda$  of the machine under consideration. The existence of a lower bound on the cache-miss ratio achievable by blocking does, however, have implications with respect to the block sizes used in block versions of linear algebra algorithms.

The expression for the data loading overhead based on (2) and (4) is also of the correct form for matrix multiplication primitives blocked for register usage in that hyperbolic behavior is also seen. The actual optimization process must be altered. The use of registers imposes shape constraints on block size choices and it is often more convenient not to decouple the two components of time. For the most part, however, the conclusions stated here still hold.

For hypercubes, the analysis of Fox, Otto, and Hey [61] derives a result in the same spirit as (8). They show that the efficiency (speedup divided by the number of processors) of the *broadcast-multiply-roll* matrix multiplication algorithms is

$$\epsilon \sim \frac{1}{1 - (c/\sqrt{n})t_{comm}/t_{flop}}$$

where  $t_{comm}$ ,  $t_{flop}$ , and  $n$  are the cost for communication of data, cost of a floating point operation, and the number of matrix elements stored locally in each processor (hence bounded by the local memory size). The constant  $c$  is 1 for the square subblock decomposition but is  $\sqrt{p}/2$  for the row decomposition, where  $p$  is the number of processors, indicating the superiority of square blocks for this type of matrix multiplication algorithm.

**3.4.4. Preferred BLAS3 primitives.** The preceding analysis also allows the issue of superiority of one BLAS3 primitive compared to the others to be addressed. Consider the comparison of the rank- $\omega$  primitive to the primitive which multiplies a matrix by  $\omega$  vectors. If  $\omega = 1$  this is the BLAS2 comparison discussed earlier and for the shared memory multivector processor analyzed above the matrix-vector multiplication primitive should be superior. On the other hand, if  $\omega = n$ , the two primitives are identical and no preference should be predicted by the analysis. Hence, the analysis should result in a preference which is parameterized by  $\omega$  with end conditions consistent with these two observations.

To make such a comparison we will restrict ourselves to the multivector shared hierarchical memory case considered above and to four partitionings of the primitives which exploit the knowledge that  $\omega$  is small compared to the other dimensions of the

matrices involved (denoted  $h$  and  $l$  below). Such a strategy was proposed in [105] and has been demonstrated effective on the Alliant FX/8. We will also distinguish between elements which are only read from memory into cache and those which require reading and writing. This allows us to be more precise than the conservative bounding of the cost of data transfer presented above. Also note that this affects the value of the cost ratio  $\lambda$  in that it need not be as large as required above.

The partitioning of the rank- $\omega$  update used is of the form given above in (6) but the values of the blocksizes are altered to reflect the more accurate analysis obtained by differentiating between reads and writes. (The qualitative conclusions of the previous analysis do not change.) Three different partitionings for the primitive which multiplies a matrix by  $\omega$  vectors are analyzed. Each is appropriate under various assumptions about the architecture and shape of the problem.

It is assumed that the primitives make use of code to perform the basic block operations which has been optimized for register-cache transfer and is able to maintain efficient use of the lowest levels of the hierarchy as the shape of the problem changes, i.e., the arithmetic time  $T_a$  has been parameterized according to  $\omega$  and the code adjusted accordingly. In this case, the source of differences in the performance of the two primitives is the amount of data transfer required between cache and main memory which is given by the ratio  $\mu$ . Below we derive and compare the value of  $\mu$  for each of the four implementations of the primitives.

The rank- $\omega$  update computes  $C \leftarrow C \pm AB$  where  $C \in \Re^{h \times l}$ ,  $A \in \Re^{h \times \omega}$ , and  $B \in \Re^{\omega \times l}$ . The partitioning used is shown in (6) where  $C_i \in \Re^{m \times l}$ ,  $A_i \in \Re^{m \times \omega}$ ,  $km = h$ , and  $m$  is the blocksize which must be determined. Note that we have used the knowledge of the analysis above to fix two blocksizes at  $\omega$  and  $l$ . The computations requires  $2hl\omega$  operations and the block loops are of the form

```
do  $i = 1, k$ 
     $C_i = C_i + A_i * B$ 
end do.
```

The primitive requires  $hl + h\omega + kl\omega$  loads from memory and  $hl$  writes back to memory. This partitioning/primitive combination is denoted *Form-1*.

The second primitive also computes  $C \leftarrow C \pm AB$ . In this case, however,  $C \in \Re^{h \times \omega}$ ,  $A \in \Re^{h \times l}$ , and  $B \in \Re^{l \times \omega}$ . As noted above, three partitionings are considered. The first two are of the form shown in (7). Both have the block loop form

```
do  $i = 1, k$ 
    do  $j = 1, m$ 
         $C_i = C_i + A_{ij} * B_j$ 
    end do
end do.
```

They differ in the constraints placed on the blocksizes.

The first version, denoted *Form-2*, results from applying the analysis of the previous section to the i-k-j loop ordering of the original triply nested loop form of the matrix multiplication primitive. One of the blocksizes is fixed at  $\omega$ . Specifically, the partitioning is such that  $A_i \in \Re^{m_1 \times m_2}$ ,  $k_1 m_1 = h$ ,  $k_2 m_2 = l$ , and  $C_i$  and  $B_i$  are dimensioned conformally. The blocksizes  $m_1$  and  $m_2$  are determined under the simplified constraint of  $m_1 m_2 \leq CS$ . *Form-2* requires  $hl + hl\omega(m_1^{-1} + m_2^{-1})$  loads and

$k_2 h \omega$  writes to memory.

The second version, denoted *Form-3*, results from analyzing the i-j-k loop ordering of the original triply nested loop form of the matrix multiplication primitive as in [105]. As before, one of the blocksizes is fixed at  $\omega$ . The partitioning is such that  $A_i \in \mathbb{R}^{m_1 \times m_2}$ ,  $k_1 m_1 = h$ ,  $k_2 m_2 = l$ , and  $C_i$  and  $B_i$  are dimensioned conformally. The blocksizes  $m_1$  and  $m_2$  are determined under the constraint of  $m_1(m_2 + \omega) \leq CS$ . This constraint is generated by requiring the accumulation in cache of a block  $C_i$  which implies that a  $C_i$  and the  $A_{ij}$  contributing to the product must fit in cache simultaneously. In [105] it is shown that this partitioning sets  $m_2$  to the value  $\tau$  where  $\tau$  is determined via the analysis of register-cache transfer cost. This simplifies the minimization problem and leaves only  $m_1$  to be determined. Form-3 requires  $hl + h\omega + hl\omega m_1^{-1}$  loads  $h\omega$  writes to memory. Additionally, it requires  $(k_2 - 1)h\omega$  writes to cache due to the local accumulation of  $C_i$ .

TABLE 1  
Comparison of the four forms of the BLAS3 primitives.

Form	$\mu$	$\mu_{opt}$	$\mu_{opt}(\sqrt{CS})$	Blocksizes
1	$\frac{1}{2m} + \frac{1}{\omega} + \frac{1}{2l}$	$\frac{1}{\omega} + \frac{\omega}{2CS} + \frac{1}{2l}$	$\frac{3}{2\sqrt{CS}} + \frac{1}{2l}$	$m = CS/\omega$
2	$\frac{1}{2\omega} + \frac{1}{2m_1} + \frac{1}{2m_2}$	$\frac{1}{2\omega} + \frac{\sqrt{2}}{\sqrt{CS}}$	$\frac{2\sqrt{2}+1}{2\sqrt{CS}}$	$m_1 = \sqrt{CS}/2$ $m_2 = \sqrt{2CS}$
3	$\frac{1}{2\omega} + \frac{1}{2m_1} + \frac{1}{l}$	$\frac{1}{2\omega} + \frac{\omega}{2CS} + \frac{1}{l} + \frac{\tau}{2CS}$	$\frac{1}{\sqrt{CS}} + \frac{1}{l} + \frac{\tau}{2CS}$	$m_1 = \frac{CS}{\omega + \tau}$
4	$\frac{1}{2\omega} + \frac{1}{2h} + \frac{1}{2m}$	$\frac{1}{2\omega} + \frac{\omega}{CS} + \frac{1}{2h}$	$\frac{3}{2\sqrt{CS}} + \frac{1}{2h}$	$m = CS/\omega$

The third version, denoted *Form-4*, applies the i-k-j ordering to the transpose of the matrix multiplication to determine blocksizes. This form is valuable for certain architecture/shape combinations. The resulting partitioning is of the form

$$(9) \quad C \leftarrow C \pm \begin{pmatrix} A_1 & \cdots & A_k \end{pmatrix} \begin{pmatrix} B_1 \\ \vdots \\ B_k \end{pmatrix},$$

where  $A_i \in \mathbb{R}^{h \times m}$ ,  $B_i \in \mathbb{R}^{m \times \omega}$  and  $km = l$ . The constraint  $m\omega \leq CS$  is applied. Form-4 requires  $hl + l\omega + 2kh\omega$  loads and  $kh\omega$  writes to memory. The block loops simplify to

```
do i = 1, k
  C = C + A_i * B_i
end do.
```

Note that if parallelism across the blocks is used this form requires synchronization (which is typically done on a subblock level).

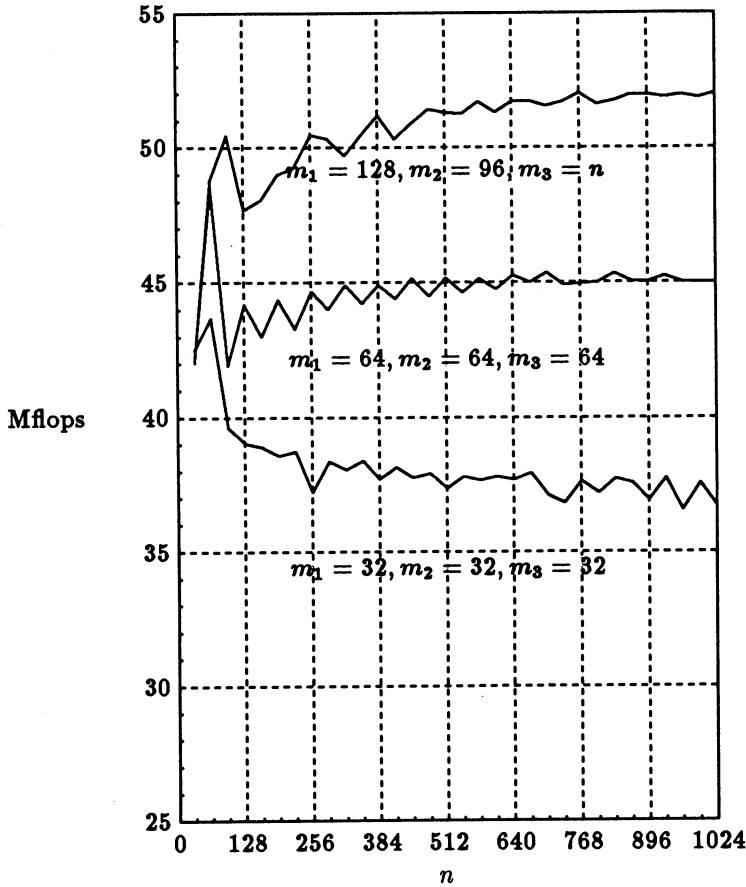


FIG. 3. Performance of square matrix multiplication on an Alliant FX/8.

Table 1 lists the results of analyzing each of the four forms presented above. The generic form of  $\mu$  is given in terms of the dimensions of the problem and the blocksizes used as well as its optimal value. Since the results of the analysis of the primitives given above and the analysis of the block methods which use them indicate that  $\omega = \sqrt{CS}$  represents a limit point on performance improvement the optimal  $\mu$  evaluated there is also given. Finally, the value of the blocksizes which give the optimal data loading cost are also listed. The values show clearly the well-known inferiority of the rank- $\omega$  by a factor of 2 when  $\omega$  is near 1, i.e., in the *near-BLAS2* regime. However, as  $\omega$  increases, the fact that one is up to a factor of two more than the other (though this multiple rapidly reduces as well) quickly becomes irrelevant since the relative cost of data transfer to computational work has become an insignificant performance consideration. As a result, given these partitionings and an architecture satisfying the assumptions of the analysis, we would not expect significant performance differences between the two primitives when  $\omega$  and the size of the matrices are large enough. Such observations have been verified on an Alliant FX/8. Consequently, one would not expect the performance of the block algorithms that use the two BLAS3 primitives,



e.g., a block  $LU$  algorithm, to be significantly different for sufficiently large problems<sup>8</sup>. It would also be expected that the trend in preference for non-reduction types of computations as the number of processors or the cost of processor synchronization increases seen with BLAS2 primitives carry over to the BLAS3.

**3.4.5. Experimental results.** The performance benefits of using BLAS3 primitives and carefully selecting blocksizes in their implementation has been demonstrated in the literature. In this section, we report briefly on experimental results on the Alliant FX/8. The experiments were performed executing the particular kernel many times and averaging to arrive at an estimate of the time spent in a single instance of the kernel. This technique was used to minimize the experimental error present on the Alliant when measuring a piece of code of short duration. As a consequence of this technique, the curves have two distinct parts. The first is characterized by a peak of high performance. This is the region where the kernel operates on a problem which fits in cache. The performance rate in this region gives some idea of the arithmetic component of the time function. It is interesting to compare this peak to the rest of the curve which corresponds to the kernel operating on a problem whose data is initially in main memory. When the asymptotic performance in the second region is close to the peak in cache the number of loads is being managed effectively.

Figure 3 illustrates the effect of blocksize on the performance of the BLAS3 primitive  $C \leftarrow C - AB$  where all three matrices are square and of order  $n$ . The blocksizes used for each curve are from low to high performance :  $m_1 = 32$ ,  $m_2 = 32$ , and  $m_3 = 32$ ;  $m_1 = 64$ ,  $m_2 = 64$ , and  $m_3 = 64$ ; and  $m_1 = 128$ ,  $m_2 = 96$ , and  $m_3 = n$ . It is clear from the asymptotic performance of the top curve that a significant portion of peak performance can be achieved by choosing the correct blocksizes. In this case an asymptotic rate of just below 52 Mflops is achieved on a machine with a peak rate, including vector startup, of 68 Mflops.

Figures 4 and 5 show the performance of various rank- $k$  updates. The parameters  $m_2$  and  $m_3$  are taken as  $k$  and  $n$  as recommended by the analysis of the BLAS3 primitive. The parameter  $m_1$  is taken to be 96 and 128 in the two figures, respectively. This parameter is kept constant for each figure to allow a fair comparison between the performances of the various kernels. Further, the BLAS3 analysis recommends  $m_1 = (CS/k) - p$ . In fact, for the values of  $k$  considered here, if  $m_1 \geq 96$  then the term in the expression for the number of loads for the rank- $k$  kernel which involves  $m_1$  is not significant compared to the term involving  $m_2$ .

These curves clearly show that increasing  $k$  yields increased performance and a significant portion of the effective peak computational rate is achievable. Also note that as  $k$  increases the difference in performance of two successive rank- $k$  kernels diminishes. Indeed, the  $k = 96$  curve was not included in Fig. 4 since it delivers performance virtually identical to the  $k = 64$  kernel.

It is instructive to compare the performance of the rank- $k$  kernel to typical BLAS and BLAS2 kernels. The BLAS kernels  $\alpha \leftarrow x^T y$  and  $y \leftarrow y \pm \alpha x$  achieve 11 Mflops and 7 Mflops, respectively, with their arguments in main memory. The BLAS2 matrix-vector product kernel achieves 18 to 20 Mflops.

**3.5. Triangular system solvers.** Solving triangular systems of linear equations, whether dense or sparse, is encountered in numerous applications. Even though the solution process consumes substantially less time than the associated factoriza-

<sup>8</sup> As is discussed later, when the ratio of the blocksize to the problem,  $\omega/n$ , is small other tradeoffs must be considered in the performance of block algorithms.

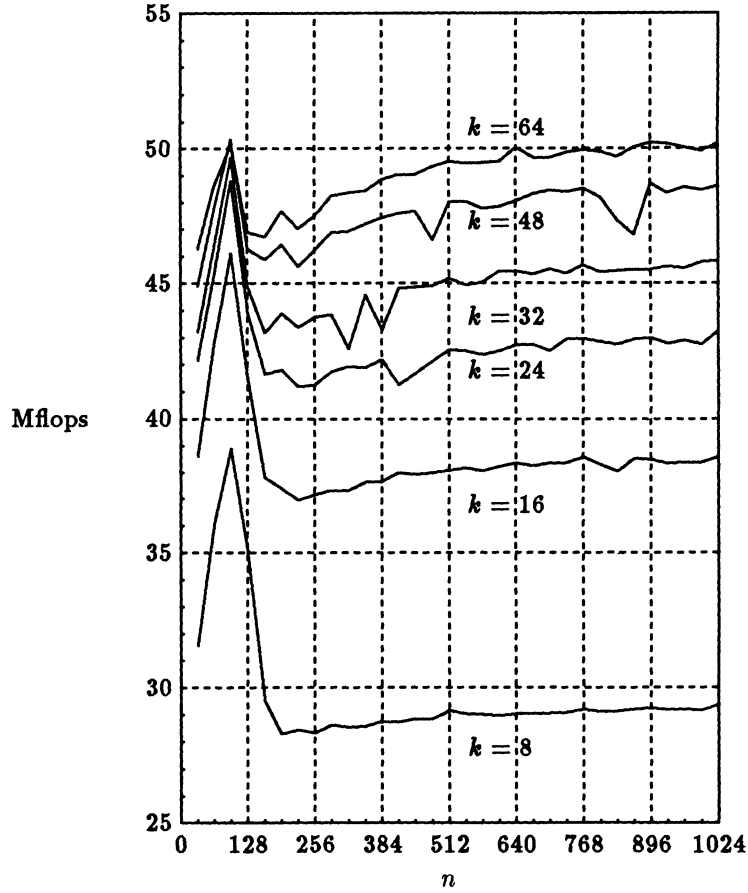


FIG. 4. Performance of rank- $k$  update with  $m_1 = 96$  on an Alliant FX/8.

tion stage, we often wish to solve these triangular systems repeatedly with different right-hand sides but with the same triangular matrix. Hence, it is vital to solve them as efficiently as possible on the architecture at hand.

There are two classical sequential algorithms for solving a lower triangular system  $Lx = f$ , where  $L = [\lambda_{ij}]$ ,  $f = [\phi_i]$ ,  $x = [\xi_i]$  and  $i, j = 1, 2, \dots, n$ . They differ in the fact that one is oriented towards rows, and the other columns. These algorithms are:

*Row-oriented :*

```

 $\xi_1 = \phi_1 / \lambda_{11}$ 
do  $i = 2, n$ 
  do  $j = 1, i - 1$ 
     $\phi_i = \phi_i - \lambda_{ij} \xi_j$ 
  enddo
   $\xi_i = \phi_i / \lambda_{ii}$ 
enddo
```

and

*Column-oriented :*

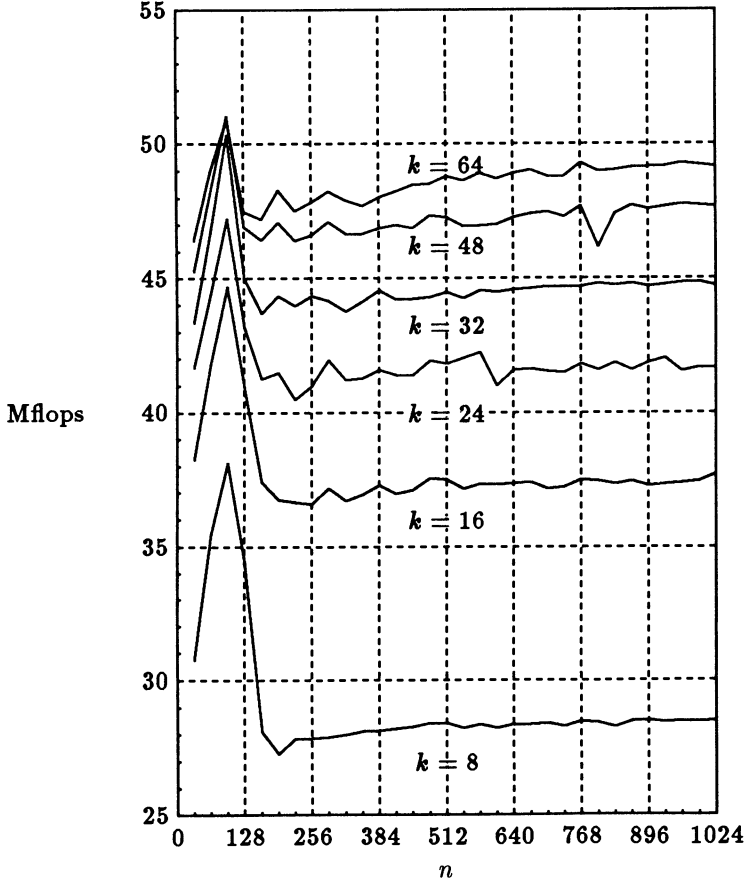


FIG. 5. Performance of rank- $k$  update with  $m_1 = 128$  on an Alliant FX/8.

```

do  $j = 1, n - 1$ 
   $\xi_j = \phi_j / \lambda_{jj}$ 
  do  $i = j + 1, n$ 
     $\phi_i = \phi_i - \lambda_{ij} \xi_j$ 
  end do
end do
 $\xi_n = \phi_n / \lambda_{nn}$ 

```

As is shown below, these two algorithms are the basis for many adaptations suitable for various vector and parallel architectures.

**3.5.1. Shared-memory triangular system solvers.** The inner loops of the row- and column-oriented versions vectorize trivially to yield algorithms based respectively on the BLAS operations of SAXPY and DOTPRODUCT. We refer to these algorithms as the row-sweep or forward-sweep, and the column-sweep [116].

Each step of the row-sweep algorithm requires less data motion than the corresponding step in the column-sweep algorithm; the DOTPRODUCT primitive reads two vectors and produces a scalar while the SAXPY reads two vectors and writes a third back to memory. If the vector processor has adequate bandwidth then, theoretically

at least, this should not be an important distinction. In practice, however, the reduced data traffic of the DOTPRODUCT may be preferable. (This assumes, of course, that the implementation of the DOTPRODUCT is not particularly expensive.<sup>9</sup>) The row-sweep algorithm can suffer from the fact that it accesses rows of the matrix. This can be remedied by storing the transpose of the lower triangular matrix, although in some cases this may not be an option, e.g., when the data placement has been determined by some other portion of the algorithm of which the triangular solve is a component.

For register-based vector processors with limited bandwidth to memory such as the CRAY-1 or a single processor of the Alliant FX/8 each of which has a single port to memory, the performance degradation due to excessive register transfers of the vector algorithms described above can be severe. Block forms of the algorithms must be considered. Let  $L^{(0)} = L$ ,  $f^{(0)} = f$ , and let each of  $L^{(j)}$ ,  $x^{(j)}$ , and  $f^{(j)}$  be of order  $(n - j\nu)$ ,  $j = 0, \dots, \frac{n}{\nu} - 1$  where

$$L^{(j)} = \begin{pmatrix} L_{11}^{(j)} & 0 \\ L_{21}^{(j)} & L_{22}^{(j)} \end{pmatrix}, x^{(j)} = \begin{pmatrix} x_1^{(j)} \\ x_2^{(j)} \end{pmatrix}, f^{(j)} = \begin{pmatrix} f_1^{(j)} \\ f_2^{(j)} \end{pmatrix}$$

with  $L_{11}^{(j)}$ ,  $x_1^{(j)}$ , and  $f_1^{(j)}$  being each of order  $\nu$  (we assume that  $\nu$  divides  $n$ ). The block column-sweep algorithm may then be described as:

*B.Col.Sweep* :

$$p = \frac{n}{\nu}$$

do  $j = 0, p - 2$

solve  $L_{11}^{(j)} x_1^{(j)} = f_1^{(j)}$  via Col.Sweep or Row.Sweep  
 $f^{(j+1)} = f_2^{(j)} - L_{21}^{(j)} x_1^{(j)}$

end do

solve  $L^{(p-1)} x^{(p-1)} = f^{(p-1)}$  via Col.Sweep or Row.Sweep.

Note that this blocking allows the registers to be used efficiently. The matrix-vector product which updates the right-hand side vector is blocked in the fashion described above to allow the accumulation in a vector register of the result of  $\nu$  vector operations before writing the register to memory rather than the one write per two reads of the triads in the nonblocked column-sweep. Similarly, the column-sweep algorithm can accumulate the solution to the triangular system  $L_{11}^{(j)} x_1^{(j)} = f_1^{(j)}$  in vector registers resulting in a data flow between registers and memory identical to that of a  $\nu \times \nu$  block of the matrix-vector product with the exception that the vector length reduces by one for each of the  $\nu$  operations.

A block row-sweep algorithm can also be derived which reduces the amount of register-memory traffic even further. Using the notation above, partition  $L$  so that each block row is of the form  $[C_i, L_i, 0]$  where  $C_i \in \mathbb{R}^{\nu \times (i-1)\nu}$  and  $L_i \in \mathbb{R}^{\nu \times \nu}$ . Let  $x = (x_1^T, \dots, x_p^T)^T$ ,  $x^{(j)} = (x_1^T, \dots, x_j^T)^T$ , and  $f = (f_1^T, \dots, f_p^T)^T$ , where  $x_i, f_i \in \mathbb{R}^\nu$ . The block algorithm is:

*B.Row.Sweep* :

---

<sup>9</sup> On some machines this is not necessarily a good assumption. The Alliant FX/8 has a considerable increase in the startup cost of the dotproduct compared to that of the triad instruction. Similarly, CRAY machines implement the dotproduct in a two-stage process. The first accumulates 64 partial sums in a vector register and the second reduces these sums to a scalar. The first phase has the memory reference pattern mentioned above but the second is memory intensive and its cost can be significant for smaller vectors.

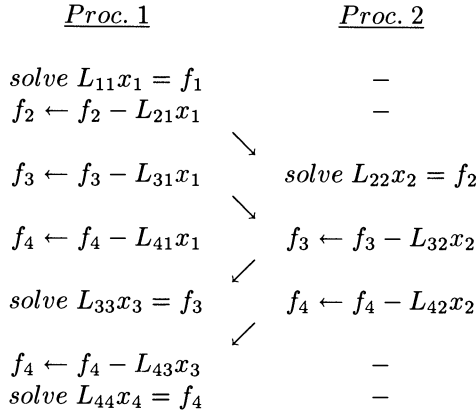


FIG. 6. Two processor DO-ACROSS synchronization pattern.

```

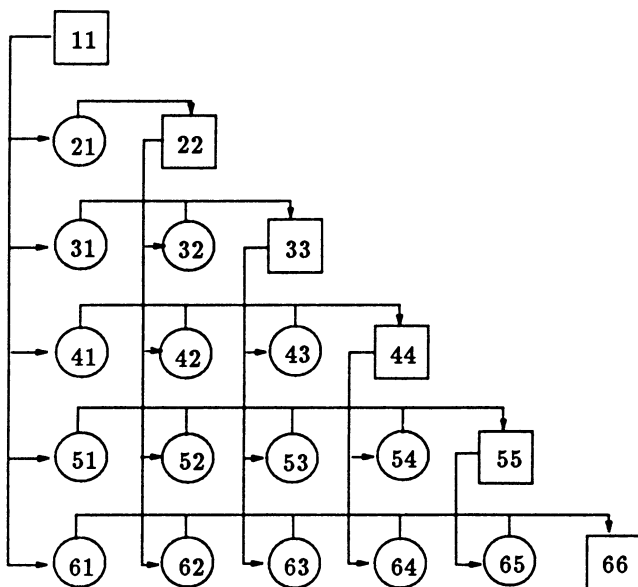
p = n / v
solve L1x1 = b1 via Col.Sweep or Row.Sweep
do j = 2, p
    fj = fj - Cjx(j-1)
    solve Ljxj = fj via Col.Sweep or Row.Sweep
end do.

```

This algorithm requires only one or two vector writes per block row computation depending upon whether or not the result of the matrix-vector product is left in registers for the triangular-solve primitive to use. This algorithm is characterized by the use of short and wide matrix-vector operations rather than the tall and narrow shapes of the block column-sweep. It is, of course, quite straightforward to combine the two approaches to use a more consistent shape throughout the algorithm.

Another triangular solver, which is also suited for both shared and distributed memory multiprocessors, is that based on the DO-ACROSS notion. For example, in the above sequential form of the column-oriented algorithm, the main point of a DO-ACROSS is that computing each  $\xi_j$  need not wait for the completion of the whole inner iteration  $i = j + 1, \dots, n$ . In fact, one processor may compute  $\xi_j$  soon after another processor has computed  $\phi_j := \phi_j - \lambda_{j,j-1}\xi_{j-1}$ . To minimize the synchronization overhead in a DO-ACROSS and efficiently use registers or local memory, the computation is performed by blocks. For example, if  $L = [L_{pq}]$ ,  $x = \{x_p\}$ ,  $f = \{f_p\}$ , and  $p, q = 1, \dots, 4$ , where each block is of order  $n/4$ , then the DO-ACROSS on two processors may be illustrated as shown in Fig. 6. Vectorization can be exploited in each of the calculations shown if each processor has vector capabilities. The particular parallel schedule used in the DO-ACROSS approach is, of course, highly dependent on the efficiency of the synchronization mechanisms provided on the multiprocessor of interest.

All of the methods presented thus far in this section can be viewed as reorganizations of the task graph in Fig. 7. The row-oriented algorithm executes each row in turn starting from the top and tasks within each row from left to right. The column-oriented, on the other hand, executes each column in turn starting from the left and tasks within a column from the top to bottom. The row and column sweeps

FIG. 7. *Triangular system solution dependence graph.*

on a vector machine merely vectorize the tasks within a row or column, respectively. Block versions of the algorithm interpret each node as corresponding to computations involving a submatrix rather than a single element. Careful consideration of the task graph, however, reveals certain limitations of all methods based upon it. Suppose that each node represents the operation on a submatrix of order  $m$  and  $n = km$ . The dependence graph implies that the maximum number of processors that can ever be active at the same time is  $k - 1$ . Further, the dependence graph has a critical path with  $O(k)$  length which establishes a fundamental limit to the speed at which these algorithms can solve a triangular system. To go faster we need a new dependence graph which relates the solution  $x$  to the data  $L$  and  $f$ .

The new dependence graph can be generated from recognizing the algebraic characterization of the column- and row-sweep algorithms. The algorithms can be easily described algebraically in terms of elementary unit lower triangular matrices. For example, assuming without loss of generality that  $\lambda_{ii} = 1$ , it follows that

$$L = \prod_{i=1}^{n-1} N_i^{-1} = \prod_{j=2}^n M_j^{-1},$$

where  $N_i = I - l_i e_i^T$ ,  $M_j = I - e_j v_j^T$ ,  $l_i$  is the vector corresponding to column  $i$  in  $L$  with the 1 on the diagonal removed and  $v_j$  is similarly constructed from row  $j$  of  $L$ . It is easy to see from the algebraic structure of  $N_i$  and  $M_j$  that multiplying them by a vector corresponds to the computational primitives of a triad and dotproduct, respectively. It follows immediately that the column-sweep and row-sweep algorithms are specified algebraically by (here with  $n = 8$ ):

$$(N_7(N_6(N_5(N_4(N_3(N_2(N_1 f)))))))$$

and

$$(M_8(M_7(M_6(M_5(M_4(M_3(M_2 f))))))).$$

The grouping of computations makes clear the source of the  $O(n)$  critical path in the dependence graph. Also a simple application of associativity can generate two algorithms that have a much shorter critical path. Specifically, the column-sweep expression can be transformed into

$$(((N_7 N_6)(N_5 N_4))((N_3 N_2)(N_1 f))).$$

Note the logarithmic nature of the critical path. The algorithm specified is called the *product form* and is due to Sameh and Brent [159]. Instead of performing the product  $(N_{n-1} \cdots N_2 N_1)f$  in  $(n-1)$  stages, we may form it in  $O(\log_2 n)$  stages. It can be shown by careful consideration of the structure of the matrices at each stage that the critical path has a length of  $k^2/2 + 3k/2$  floating point operations where  $k = \log_2 n$ . Such an improvement is not without cost, however. The algorithm requires approximately  $n^3/10 + O(n^2)$  operations and  $n^3/68 + O(n^2)$  processors. It is therefore typically not appropriate for an architecture with a limited number of processors such as those of interest here. For a discussion of the numerical stability of this algorithm see [187].

Note that thus far we have assumed only one right-hand side vector. The BLAS3 primitive triangular solver assumes that multiple right-hand side vectors and solutions are required. This, of course, provides the necessary data locality for high performance on a hierarchical memory system. The generalization of the algorithms above are straightforward and the blocksizes (the number and order of right-hand sides solved in a stage of the algorithm) can be analyzed in a fashion similar to the matrix multiplication primitives.

For banded lower triangular systems in which the bandwidth  $m$  (the number of subdiagonals with nonzero entries) is small, column-sweep algorithms are ineffective on vector or parallel computers. Consider such a system  $Lx = f$ , where  $L$  is partitioned as a block-bidiagonal matrix with diagonal submatrices  $L_i$  and subdiagonal submatrices  $R_{i-1}$ ,  $i = 1, \dots, n/m$ , where  $L_i$  and  $R_{i-1}$  are lower and upper triangular, respectively. Premultiplying both sides of  $Lx = f$  by  $D = \text{diag}(L_i^{-1})$  we obtain the system  $L^{(0)}x = f^{(0)}$  where  $L^{(0)}$  is block bidiagonal with identities of order  $m$  on the diagonal and matrices  $G_j^{(0)} = L_{j+1}^{-1}R_j$  on the subdiagonal, and  $f^{(0)} = Df$ . Note that we do not invert the  $L_j$ 's, but obtain  $f^{(0)}$  and  $G_i^{(0)}$  by solving triangular systems using one of the above parallel algorithms. We repeat the process by multiplying both sides of  $L^{(0)}x = f^{(0)}$  by  $D^{(0)} = \text{diag}((L_i^{(0)})^{-1})$  where

$$(L_i^{(0)})^{-1} = \begin{pmatrix} I_m & 0 \\ -G_{2i-1}^{(0)} & I_m \end{pmatrix}.$$

Now  $L^{(1)} = D^{(0)}L^{(0)}$  and  $f^{(1)} = D^{(0)}f^{(0)}$  are obtained by simple multiplication. Eventually,  $L^{(\log(n/m))} = I_n$  and  $f^{(\log(n/m))} = x$ . The required number of arithmetic operations is  $O(m^2 n \log(n/2m))$  resulting in a redundancy of  $O(m \log(n/2m))$ , e.g., see [159]. Given  $m^2 n/2 + O(mn)$  processor, however, those operations can be completed in  $O(\log m \log n)$  time.

This algorithm offers opportunities for both vector and parallel computers. At the first stage we have  $n/m$  triangular systems to solve, each of order  $m$ , for  $(m+1)$  right-hand sides except for the first system which has only one right-hand side. In the subsequent stages we have several matrix-matrix and matrix-vector multiplications, with the last stage consisting of only one matrix-vector multiplication, in which the matrix is of order  $(n/2 \times m)$ .

An alternative scheme, introduced by Chen, Kuck, and Sameh [27], may be described as follows. Let the banded lower triangular matrix  $L$  be partitioned as

$$\begin{pmatrix} L_1 & & & & \\ \tilde{R}_2 & L_2 & & & \\ & \tilde{R}_3 & L_3 & & \\ & & \ddots & \ddots & \\ & & & \tilde{R}_p & L_p \end{pmatrix},$$

where

$$\tilde{R}_j = \begin{pmatrix} 0 & R_j \\ 0 & 0 \end{pmatrix}$$

and each  $L_i$  is of order  $(n/p) \gg m$  and each  $R_i$  is upper triangular of order  $m$ . If the right-hand side  $f$  and the solution  $x$  are partitioned accordingly, then after solving the triangular systems

$$L_1 x_1 = f_1$$

and

$$L_i [U_i, g_i] = \left[ \begin{pmatrix} R_i \\ 0 \end{pmatrix}, f_i \right]$$

the original system is reduced to  $\tilde{L}x = g$  in which  $\tilde{L}$  is of the form

$$\begin{pmatrix} I_{n/p} & & & & \\ \tilde{U}_2 & I_{n/p} & & & \\ & \tilde{U}_3 & I_{n/p} & & \\ & & \ddots & \ddots & \\ & & & \tilde{U}_p & I_{n/p} \end{pmatrix},$$

where

$$\tilde{U}_j = \begin{pmatrix} 0 & U_j \end{pmatrix}.$$

Let

$$U_i = \begin{bmatrix} V_i \\ W_i \end{bmatrix}, \quad x_i = \begin{bmatrix} y_i \\ z_i \end{bmatrix}, \quad g_i = \begin{bmatrix} h_i \\ r_i \end{bmatrix},$$

in which  $W_i$  is a matrix of order  $m$  and  $r_i, z_i$  are vectors of  $m$  elements each. Thus, solving the above system reduces to solving a smaller triangular system of order  $mp$ ,

$$\begin{pmatrix} I_m & & & \\ W_2 & I_m & & \\ & \ddots & \ddots & \\ & & W_p & I_m \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_p \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_p \end{pmatrix}.$$

After solving this system by the previous parallel scheme, for example, we can retrieve the rest of the elements of the solution vector  $x$  by obvious matrix-vector multiplications. The algorithm requires approximately  $4m^2n$  operations which, given  $\tilde{p} = mp$  processors, can be completed in time  $2\tilde{p}^{-1}m^2n + 3\tilde{p}^{-1}mn + O(m^2)$ . See [189] for a discussion of the performance of this algorithm applied to lower bidiagonal systems and the attendant numerical stability properties.



**3.5.2. Distributed-memory triangular system solvers.** A large number of papers have appeared for handling triangular systems on distributed memory architectures (mainly rings and hypercubes), e.g., see Sameh [158], Romine and Ortega [151], Heath and Romine [89], Li and Coleman [122] and Eisenstat et al. [53]. Most are variations on the basic algorithms above adapted to exploit the distributed nature of the architectures. For such architectures, it is necessary to distinguish whether a given triangular matrix  $L$  is stored across the individual processor memories by rows or by columns. For example, suppose that the matrix  $[L, f]$  is stored by rows, then the above column-sweep algorithm becomes:

*Row\_Storage :*

```

do j = 1 , n
  if j is one of my row indices then
     $\xi_j = \phi_j / \lambda_{jj}$ 
    communicate(broadcast, fan-out)  $\xi_j$  to each processor
  do i = j + 1 , n
    if i is one of my row indices then
       $\phi_i = \phi_i - \xi_j \lambda_{ij}$ 
    enddo
  enddo.

```

Note first that the computations in the inner loop can be executed in parallel, and that on a hypercube with  $p = 2^\nu$  processors, the fan-out communication can be accomplished in  $\nu$  stages. If the lower triangular matrix  $L$  is stored by columns then the column-sweep algorithm will cause excessive interprocessor communication. A less communication intensive column storage oriented algorithm has been suggested in [150] and [151]. Such an algorithm is based upon the classical sequential *Row-sweep* algorithm shown above.

In implementing the column storage algorithm on an Intel iPSC hypercube, for example, information is gathered into one processor from all others via a fan-in operation  $fan\_in(\tau, i)$ . Such an operation enables the processor whose memory contains column  $i$  to receive the sum of all the  $\tau$ 's over all processors. The parallel column storage algorithm can be described as follows:

*Col\_Storage :*

```

do i = 1 , n
   $\tau = 0$ 
  do j = 1 , i - 1
    if j is one of my column indices then
       $\tau = \tau + \xi_j \lambda_{ij}$ 
    enddo
   $\eta = fan\_in(\tau, i)$ 
  if i is one of my column indices then
     $\xi_i = (\phi_i - \eta) / \lambda_{ii}$ 
  enddo.

```

Here, during stage  $i$  of the algorithm, the pseudo-routine  $fan\_in(\tau, i)$  collects and sums the partial inner products  $\tau$  from each processor, leaving the result  $\eta$  in the processor containing column  $i$ . Further modifications to the basic row- and column-oriented triangular solvers on distributed memory systems have been studied in [122], there a communication scheme which allows for ring embedding into a hypercube

is emphasized. In addition, the study in [53] has improved upon the cyclic type algorithms in [89].

**4. LU factorization algorithms.** The goal of the  $LU$  decomposition is to factor an  $n \times n$ -matrix  $A$  into a lower triangular matrix  $L$  and an upper triangular matrix  $U$ . This factorization is certainly one of the most used of all numerical linear computations. The classical  $LU$  factorization [83] can be expressed in terms any of the three levels of the BLAS, and techniques needed to achieve high performance for both shared and distributed memory systems have been considered in great detail in the literature. In this section we review some of these techniques for the  $LU$  and  $LU$ -like factorizations for dense and block tridiagonal linear systems.

**4.1. Shared-memory algorithms for dense systems.** In this subsection we consider some of the approaches used in the literature for implementing the  $LU$  factorization of a matrix  $A \in \mathbb{R}^{n \times n}$  on shared-memory multivector processors such as the CRAY-2, CRAY X-MP, and Alliant FX/8. To simplify the discussion of the effects of hierarchical memory organization, we move directly to the block versions of the algorithms. Throughout the discussion  $\omega$  denotes the blocksize used and the more familiar BLAS2-based versions of the algorithms can be derived by setting  $\omega = 1$ . Four different organizations of the computation of the classical  $LU$  factorization without pivoting are presented with emphasis on identifying the computational primitives involved in each. The addition of partial pivoting is then considered and a block generalization of the  $LU$  factorization ( $L$  and  $U$  being block triangular) is presented for use with diagonally dominant matrices. Finally, the results of an analysis of the architecture/algorithm mapping of this latter algorithm for a multivector processor with a hierarchical memory are also examined along with performance results from the literature.

**4.1.1. The algorithms.** There are several ways to organize the computations for calculating the  $LU$  factorization of a matrix. These reorganizations are typically listed in terms of the ordering of the nested loops that define the standard computation. The essential differences between the various forms are: the set of computational primitives required, the distribution of work among the primitives, and the size and shape of the subproblems upon which the primitives operate. Since architectural characteristics can favor one primitive over another, the choice of computational organization can be crucial in achieving high performance. Of course, this choice in turn depends on a careful analysis of the architecture/primitive mapping.

Systematic comparisons of the reorganizations have appeared in various contexts in the literature. Trivedi considered them in the context of virtual memory systems in combination with other performance enhancement techniques [185], [186]. Dongarra, Gustavson, and Karp [42] and more recently Ortega [137] compare the orderings for vector machines such as the CRAY-1 where the key problem is the efficient exploitation of the register-based organization of the processor and the single port to memory. Ortega has also considered the problem on highly parallel computers [137]. Papers have also appeared that are concerned with comparing the reorderings given a particular machine/compiler/library combination, e.g., see [162]. In general, most of the conclusions reached in these papers can be easily understood and parameterized by analyses of the computational primitives and the algorithms in the spirit of those in the previous section and below.

**4.1.1.1. Version 1.** *Version 1* of the algorithm assumes that at step  $i$  the  $LU$  factorization of the leading principal submatrix of dimension  $(i - 1)\omega$ ,  $A_{i-1} =$

$L_{i-1}U_{i-1}$ , is available. The next  $\omega$  rows of  $L$  and  $\omega$  columns of  $U$  are computed during step  $i$  to produce the factorization of the leading principal submatrix of order  $i\omega$ . Clearly, after  $k = n/\omega$  such steps the factorization  $LU = A$  results.

The basic step of the algorithm can be deduced by considering the following partitioning of the factorization of the matrix  $A_i \in \mathbb{R}^{i\omega \times i\omega}$ :

$$A_i = \begin{pmatrix} A_{i-1} & C \\ B^T & H \end{pmatrix} = \begin{pmatrix} L_{i-1} & 0 \\ M^T & L_2 \end{pmatrix} \begin{pmatrix} U_{i-1} & G \\ 0 & U_2 \end{pmatrix},$$

where  $H$  is a square matrix of order  $\omega$  and the rest of the blocks are dimensioned conformally. The basic step of the algorithm consists of four phases:

- (i) Solve for  $G$ :  $C \leftarrow L_{i-1}G = C$ .
- (ii) Solve for  $M$ :  $B \leftarrow U_{i-1}^T M = B$ .
- (iii) Update:  $H \leftarrow H - M^T G$ .
- (iv) Factor  $H \leftarrow L_2 U_2 = H$ .

(The arrow is used to represent the portion of the array which is overwritten by the new information obtained in each phase.) Clearly, repeating this step on successively larger submatrices will produce the factorization of  $A \in \mathbb{R}^{n \times n}$ .

In each step, solving the triangular system requires  $2\omega h^2$  operations, the update of  $H$  requires  $2h\omega^2$  and the factorization requires  $O(\omega^3)$ , where  $h = (i-1)\omega$ . Early stages of the algorithm are dominated by the factorization primitive. The later stages, where most of the work is done, is dominated by solving triangular systems with  $\omega$  right-hand side vectors. This dominance is particularly pronounced when the BLAS2 ( $\omega = 1$ ) version of the algorithm is used. Note also that when  $\omega = 1$  the use of the triangular solver allows efficient use of the vector registers on vector processors like the CRAY-1 or a single CE of the Alliant FX/8 which have single ports to memory.

**4.1.1.2. Version 2.** *Version 2* of the algorithm assumes that the first  $\xi = (i-1)\omega$  columns of  $L$  and  $\xi$  rows of  $U$  are known at the start of step  $i$ , and that the transformations necessary to compute this information have been applied to the submatrix  $A^i \in \mathbb{R}^{n-\xi \times n-\xi}$  in the lower right-hand corner of  $A$  that has yet to be reduced. The algorithm proceeds by producing the next  $\omega$  columns and rows of  $L$  and  $U$ , respectively, and computing  $A^{i+1}$ . This is a straightforward block generalization of the standard rank-1-based Gaussian elimination algorithm.

Assume that the factorization of the matrix  $A^i \in \mathbb{R}^{n-\xi \times n-\xi}$  is partitioned as follows:

$$A^i = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & I \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & A^{i+1} \end{pmatrix},$$

where  $A_{11}$  is square and of order  $\omega$  and the other submatrices are dimensioned conformally.  $L_{11}, L_{21}$  and  $U_{12}$  are the desired  $\omega$  columns and rows of  $L$  and  $U$  and identity defines  $A^{i+1}$ .

The basic step of the algorithm consists of:

- (i) Factor:  $A_{11} \leftarrow L_{11}U_{11} = A_{11}$ .
- (ii) Solve for  $L_{21}$ :  $A_{21} \leftarrow U_{11}^T L_{21}^T = A_{21}^T$ .
- (iii) Solve for  $U_{12}$ :  $A_{12} \leftarrow L_{11}U_{12} = A_{12}$ .
- (iv) Update:  $A_{22} \leftarrow A_{22} - L_{21}U_{12}$ .

Clearly, the updated  $A_{22}$  is  $A^{i+1}$  and the algorithm proceeds by repeating the above four phases.

This version of the algorithm is dominated by the rank- $\omega$  update of the submatrix  $A_{22} \in \mathfrak{R}^{(n-i\omega) \times (n-i\omega)}$ . Note that the triangular systems that must be solved are of order  $\omega$  with many right-hand sides as opposed to the large systems which are solved in Version 1. As in Version 1 the factorization primitive operates on systems of order  $\omega$ . As is well known and obvious from the analysis of the previous section, the BLAS2 version, based on the rank-1 update, is not the preferred form for register-based vector or multivector processors with a single port to memory due to poor register usage.

**4.1.1.3. Version 3.** *Version 3* of the algorithm can be viewed as a hybrid of the first two versions. Like Version 2, it is assumed that the first  $(i-1)\omega$  columns of  $L$  and rows of  $U$  are known at the start of step  $i$ . It also assumes, like Version 1, that the transformations that produced these known columns and rows must be applied elements of  $A$  which are to be transformed into the next  $\omega$  columns and rows of  $L$  and  $U$ . As a result, Version 3 does not update the remainder of the matrix at every step.

Consider the factorization:

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

where  $A_{11}$  is a square matrix of order  $(i-1)\omega$  and the rest are partitioned conformally. By our assumptions,  $L_{11}$ ,  $L_{21}$ ,  $U_{11}$ , and  $U_{12}$  are known and the first  $\omega$  columns of  $L_{22}$  and the first  $\omega$  rows of  $U_{22}$  are to be computed. Since Version 3 assumes that none of the update  $A_{22} \leftarrow A_{22} - L_{21}U_{12}$  has occurred in the first  $i-1$  steps of the algorithm, the first part of step  $i$  is to perform the update to the portion upon which the desired columns of  $L_{22}$  and rows of  $U_{22}$  depend. This is then followed by the calculation of the columns and rows.

To derive the form of the computations, suppose the update of  $A_{22}$  and its subsequent factorization are partitioned

$$A_{22} \leftarrow \begin{pmatrix} H & C^T \\ B & \tilde{A}_{22} \end{pmatrix} = \begin{pmatrix} \hat{H} & \hat{C}^T \\ \hat{B} & \hat{A}_{22} \end{pmatrix} - L_{21}U_{12}$$

and

$$\begin{pmatrix} H & C^T \\ B & \tilde{A}_{22} \end{pmatrix} = \begin{pmatrix} \tilde{L}_{11} & 0 \\ \tilde{L}_{21} & \tilde{L}_{22} \end{pmatrix} \begin{pmatrix} \tilde{U}_{11} & \tilde{U}_{12} \\ 0 & \tilde{U}_{22} \end{pmatrix},$$

where  $H$  and  $\hat{H}$  are square matrices of order  $\omega$  and the other submatrices are dimensioned conformally. Step  $i$  then has two major phases: Calculate  $H$ ,  $B$ , and  $C$ ; and calculate  $\tilde{L}_{11}$ ,  $\tilde{L}_{21}$ ,  $\tilde{U}_{11}$ , and  $\tilde{U}_{12}$ . As a result, at the end of stage  $i$ , the first  $i\omega$  rows and columns of the triangular factors of  $A$  are known.

Let  $L_{21} = [M_1^T, M_2^T]^T$  and  $U_{12} = [M_3, M_4]$ , where  $M_1$  and  $M_3$  consist of the first  $\omega$  rows and columns of the respective matrices. The first phase of step  $i$  computes

- (i)  $[H^T, B^T]^T \leftarrow [H^T, B^T]^T = [\hat{H}^T, \hat{B}^T]^T - L_{21}M_3$ .
- (ii)  $C \leftarrow C^T = \hat{C}^T - M_1M_4$ .

In the second phase, the first  $\omega$  rows and columns of the factorization of the updated  $A_{22}$  are then given by:

- (i) Factor:  $H \leftarrow \tilde{L}_{11}\tilde{U}_{11} = H$ .
- (ii) Solve for  $\tilde{L}_{21}$ :  $B \leftarrow \tilde{U}_{11}^T\tilde{L}_{21}^T = B^T$ .

(iii) Solve for  $\tilde{U}_{12}$ :  $C \leftarrow \tilde{L}_{11}\tilde{U}_{12} = C^T$ .

The work in this version of the algorithm is split between a matrix multiplication primitive, a triangular solver, and a factorization primitive; the latter two of which are applied to systems of order  $\omega$ . Note, however, that the matrix multiplication primitive is applied to a problem which has the shape of a large matrix applied to  $\omega$  vectors (or the transpose of such a problem). Hence, for  $\omega = 1$  this version of the algorithm becomes a form which uses the preferred BLAS2 primitive — matrix-vector multiplication. Although, as noted above, when  $\omega$  is nontrivial the preference for this block form over Version 2 does not necessarily follow.

**4.1.1.4. Version 4.** *Version 4* of the algorithm assumes that at the beginning of step  $i$  the first  $(i-1)\omega$  columns of  $L$  and  $U$  are known. Step  $i$  computes the next  $\omega$  columns of the two triangular factors. Consider the factorization

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

where  $A_{11}$  is a square matrix of order  $(i-1)\omega$  and the rest are partitioned conformally. By our assumptions,  $L_{11}$ ,  $L_{21}$ , and  $U_{11}$  are known.

Let  $L_\omega$ ,  $U_\omega$ , and  $A_\omega$  be the matrices of dimension  $n \times \omega$  formed of the first  $\omega$  columns of  $[0, L_{22}^T]^T$ ,  $[U_{12}^T, U_{22}^T]^T$ , and  $[A_{12}^T, A_{22}^T]^T$ , respectively. (These are also columns  $(i-1)\omega + 1$  through  $i\omega$  of  $L$ ,  $U$ , and  $A$ .) Consider the partitioning

$$L_\omega = \begin{pmatrix} 0 \\ \tilde{L} \\ G \end{pmatrix}, \quad U_\omega = \begin{pmatrix} M \\ \tilde{U} \\ 0 \end{pmatrix}, \quad A_\omega = \begin{pmatrix} \tilde{A}_1 \\ \tilde{A}_2 \\ \tilde{A}_3 \end{pmatrix},$$

where  $\tilde{L}$ ,  $\tilde{U}$ , and  $\tilde{A}_2$  are square matrices of order  $\omega$  with  $\tilde{L}$  and  $\tilde{U}$  lower and upper triangular respectively.

Step  $i$  calculates  $L_\omega$  and  $U_\omega$  by applying all of the transformations from steps 1 to  $i-1$  to  $A_\omega$  and then factoring a rectangular matrix. Specifically, step  $i$  comprises the computations:

- (i) Solve for  $M$ :  $\tilde{A}_1 \leftarrow L_{11}M = \tilde{A}_1$ .
- (ii) Update:  $[\tilde{A}_2^T, \tilde{A}_3^T]^T \leftarrow [\tilde{A}_2^T, \tilde{A}_3^T]^T - L_{21}M$ .
- (iii) Factor:  $\tilde{A}_2 \leftarrow \tilde{L}\tilde{U} = \tilde{A}_2$ .
- (iv) Solve for  $G$ :  $\tilde{A}_3 \leftarrow \tilde{U}^T G^T = \tilde{A}_3^T$ .

This version of the algorithm requires the solution of a large triangular system with  $\omega$  right-hand sides as well as a small triangular system of order  $\omega$  with many right-hand sides. The factorization kernel operates on a system of order  $\omega$ . As with Version 3 the matrix multiplication primitive operates on a problem with the shape of a large matrix times  $\omega$  vectors and the factorization of a system of order  $\omega$  and the same observations apply. This version also has the feature that it works exclusively with columns of  $A$  which can be advantageous in some Fortran and virtual memory environments.

**4.1.1.5. Partial pivoting.** Partial pivoting can be easily added to Versions 2, 3, and 4 of the algorithm. Step  $i$  of each of the versions requires the  $LU$  factorization of a rectangular matrix  $M \in \mathbb{R}^{h \times \omega}$ , where  $h = n - (i-1)\omega$ . Specifically, step  $i$  computes

$$M = \begin{pmatrix} M_1 \\ M_2 \end{pmatrix} = \begin{pmatrix} \hat{L}_{11} \\ \hat{L}_{21} \end{pmatrix} \hat{U}_{11},$$

where  $\hat{L}_{11}$  and  $\hat{U}_{11}$  are, respectively, lower and upper triangular matrices of order  $\omega$ . In the versions above without pivoting, this calculation could be split into two pieces: the factorization of a system of order  $\omega$ ,  $\hat{L}_{11}\hat{U}_{11} = M_1$ ; and the solution of a triangular system of order  $\omega$  with  $h - \omega$  right-hand sides. (These computations are: (i) and (ii) in Version 2; (i) and (ii) of the second phase of Version 3; and (iii) and (iv) of Version 4.) When partial pivoting is added to the versions of the algorithm these computations at each step cannot be separated and are replaced by a single primitive which produces the factorization of a rectangular matrix with permuted rows, i.e.,

$$PM = P \begin{pmatrix} M_1 \\ M_2 \end{pmatrix} = \begin{pmatrix} \hat{L}_{11} \\ \hat{L}_{21} \end{pmatrix} \hat{U}_{11},$$

where  $P$  is a permutation matrix. This primitive is usually cast as a BLAS2 version of one of the versions above. Note, however, a fundamental difference compared to the nonpivoting versions. The ability to split the factorization of the tall matrix into smaller BLAS3-based components in the latter case has benefits with respect to hierarchical memory usage, since  $\omega$  is usually taken so that such systems fit in cache or local memory, see [23], [67]. In the case of pivoting, these operations are performed via BLAS2 primitives repeatedly updating a matrix which can not be kept locally. As a result, the arithmetic component of time and the data transfer overhead both increase. In fact, a conflict between their reductions occurs. This situation is similar to that seen in the block version of Modified Gram Schmidt and Version 5 of the factorization algorithm, both discussed below along with a solution. (Although in the latter case, the source of difficulties is slightly different.)

The information contained in the permutations associated with each step,  $P_i$ , can be applied in various ways. For example, the permutation can be applied immediately to the transformations of the previous steps, which are stored in the elements of the array  $A$  to the left of the active area for step  $i$ , and to the elements of the array  $A$  which have yet to reach their final form, which, of course, appear to the right of the active area for step  $i$ . The application to either portion of the matrix may also be delayed. The update of the elements of the array which have yet to reach their final form could be delayed by maintaining a global permutation matrix which is then applied to only the elements required for the next step. Similarly, the application to the transformations from steps 1 through  $i - 1$  could be suppressed and the  $P_i$  could be kept separately and applied incrementally in a modified forward and backward substitution routine.

**4.1.1.6. Version 5. A block generalization.** In some cases it is possible to use a block generalization of the classical  $LU$  factorization in which  $L$  and  $U$  are lower and upper block triangular matrices, respectively. The use of such a block generalization is most appropriate when considering systems which do not require pivoting for stability, e.g., diagonally dominant or symmetric positive definite. This algorithm decomposes  $A$  into a lower block triangular matrix  $L_\omega$  and an upper block triangular matrix  $U_\omega$  with blocks of the size  $\omega$  by  $\omega$  (it is assumed for simplicity that  $n = k\omega$ ,  $k > 1$ ). Assume that  $A$  is diagonally dominant and consider the factorization:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} I & 0 \\ L_{21} & I \end{pmatrix} \begin{pmatrix} A_{11} & A_{12} \\ 0 & B \end{pmatrix},$$

where  $A_{11}$  is a square matrix of order  $\omega$ . The block  $LU$  algorithm is given by:

- (i)  $A_{11} \leftarrow A_{11}^{-1}$
- (ii)  $A_{21} \leftarrow L_{21} = A_{21}A_{11}$
- (iii)  $A_{22} \leftarrow B = A_{22} - L_{21}A_{12}$
- (iv) Proceed recursively on the matrix  $B$ .

Statements (i) and (ii) can be implemented in several ways. Since  $A$  is assumed to be diagonally dominant, explicit inversion of the diagonal blocks can be done either via the Gauss–Jordan algorithm [143] or an  $LU$  decomposition without pivoting. In the latter case, the computations in step (i) above are replaced by solving two triangular systems of order  $\omega$  with many right-hand sides. (Due to parallel processing, the Gauss–Jordan scheme, historically frowned upon, has recently been the subject of renewed interest. See [34] for a discussion of its application, with appropriate modifications, to general nonsymmetric systems of equations.)

If the Gauss–Jordan kernel is used, as is assumed below, the block  $LU$  algorithm is more expensive by a factor of approximately  $(1 + 2/k^2)$  than the classical  $LU$  factorization which requires about  $2n^3/3$  operations. In this form, the above block algorithm uses three primitives: a Gauss–Jordan inversion (or  $LU$  decomposition),  $A \leftarrow AB$ , and a rank- $\omega$  update.

Note that when  $\omega = 1$  this form of the algorithm becomes the BLAS2 version based on rank-1 updates. As with Versions 1–4, which produce the classical  $LU$  factorization, the computations of Version 5 can be reorganized so that different combinations of BLAS3 primitives and different *shapes* of submatrices are used. For example, the main BLAS3 primitive can be changed from a rank- $\omega$  update into a matrix multiplying  $\omega$  row or column vectors. As noted above, the importance of such a reorganization depends highly on the architecture in question.

**4.1.2. Performance analysis.** Gallivan et al. have applied the decoupling methodology to Version 5 [67]. Their results demonstrate many of the performance trends observed in the literature for the various forms of block methods. A summary of the important points follows.

There are two general aspects of the block  $LU$  decomposition through which the blocksize  $\omega = n/k$  influences the arithmetic time: the number of redundant operations (applicable when the Gauss–Jordan approach is used); and the relationship, as a function of  $\omega$ , between the performance of each of the primitives and the distribution of work among the primitives. The redundancy factor of  $(1 + 2/k^2)$  and the fact that the number of operations performed in the Gauss–Jordan primitive is an increasing function of  $\omega$  cause the arithmetic time component to prefer smaller blocksizes for small and moderately sized systems. For those systems, increasing  $\omega$  and therefore decreasing  $k$  clearly exacerbates the two problems noted above to such a degree that the effect is dominant compared to the reduction in data transfer overhead gained by increasing the blocksize. As the order of the system increases, however, these effects become secondary to data transfer considerations.

The data transfer overhead of the algorithm is most conveniently analyzed by writing the algorithm’s cache-miss ratio as the weighted average of the cache-miss ratios of the various instances of each primitive. The weights are the ratio of the number of operations in the particular instance of the primitive to the total number of operations required. In practice, some of the local cache-miss ratios are zero due to the interaction between the instances of the primitives; this occurs when the remaining

part of the matrix to be decomposed approaches the size of the cache and later instances of primitives find an increasing proportion of their data left in cache by earlier instances. In [67] the results are derived using the conservative assumption of no interaction between instances of primitives. Note that without a model of the data transfer properties of the primitives such an analysis at the algorithmic level is impossible. This does not imply that blocksizes cannot be set effectively based on observed performance data of the primitives for various shapes and sizes of problems. Such a *black box* tuning approach is quite useful in practice, but it does not provide any *explanation* as to why the performance is as observed. This can only be done by considering the architecture/algorithm mapping of the primitives and the implications of combining them in the manner specified by the particular version of the factorization algorithm used.

The behavior on the interval  $1 \leq \omega \leq \sqrt{CS}$ , where  $CS$  denotes cache size, roughly separates into three regimes. For small values of  $\omega$ , i.e.,  $\omega \leq 16$ , the cache-miss ratio is of the form:

$$\mu \approx \frac{1}{2\omega} \gamma_R + \eta_1,$$

where  $\eta_1$  is proportional to  $1/n$  and  $\gamma_R$  is a function of  $\omega$  which is bounded by a small constant. This result is expected since the computations are dominated by the rank- $\omega$  update which achieves a similar cache-miss ratio. In particular, it is clear that the data locality of a BLAS2 version,  $\omega = 1$ , is very poor. In the middle of the interval of interest the cache-miss ratio is of the form:

$$\mu \approx \frac{1}{\omega} \gamma_R + \eta_2,$$

where  $\eta_2$  is proportional to  $1/n$ . Finally, when  $\omega \approx \sqrt{CS}$ , the cache-miss ratio is

$$\mu \approx \frac{1}{\sqrt{CS}} \gamma_R + \eta_3,$$

where  $\eta_3$  is proportional to  $1/n$ . The ratio  $\mu$  becomes a rapidly increasing function once  $\omega$  exceeds  $\sqrt{CS}$  until it reaches, at the point  $\omega = n$ , the cache-miss ratio of the algorithm of a BLAS2-based version of the Gauss–Jordan algorithm which has a value of approximately  $\frac{1}{4}$ . The exact point where this transition to rapidly increasing occurs is dependent on the implementation of the Gauss–Jordan primitive, but, any decrease in  $\mu$  between  $\omega = \sqrt{CS}$  and the transition point is typically insignificant.

**4.1.3. Experimental results.** The various versions of the algorithms have appeared in different contexts in the literature. Here we list some representative papers and then consider in more detail the performance of Version 5 and its relationship to the trends predicted via the blocksize analysis presented above.

The column-oriented BLAS2 form of Version 4 was used by Fong and Jordan on the CRAY-1 [56]. The results of using a BLAS2 form of Version 3 on the CRAY-1 and one CPU of a CRAY X-MP were given by Dongarra and Eisenstat in [41]. Dongarra and Hewitt discuss the use of a rank-3-based approach on four CPU's of a CRAY X-MP [45]. Calahan demonstrated the power of the block form of Version 3 (with and without pivoting) on the hierarchical memory system of one CPU of a CRAY-2. Agarwal and Gustavson have extended their work which led to single CPU block algorithms for the IBM 3090 by considering parallel forms of the BLAS3 primitives and *LU* factorization on an IBM ES/3090 model 600S [2]. In particular, they discuss



the use of parallel block methods in a multitasking environment where the user is not necessarily guaranteed control of all (or any fixed subset) of the six processors in the system. Radicati, Robert, and Sguazzero have presented the results of a rank- $k$ -based code on an IBM 3090 multivector processor for one to six processors [149]. The block form of Version 4 was also considered in a virtual memory setting by Du Croz et al. in [50] and used as a model of a block  $LU$  factorization in the BLAS3 standard proposal by Dongarra et al. [39]. Finally, Dayde and Duff have compared the performance of the different organizations of the block computations on a CRAY-2, ETA-10P, and IBM 3090-200/VF.

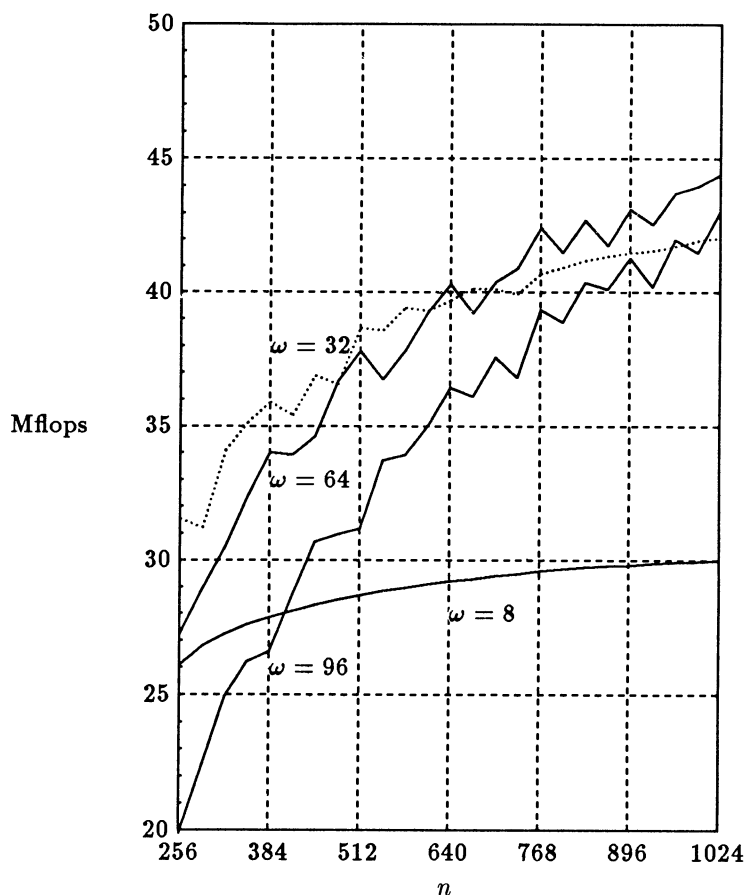


FIG. 8. Performance of block  $LU$  on an Alliant FX/8.

The performance trends for Version 5 predicted via the decoupling analysis summarized above have been verified in [67]. Figure 8 illustrates the performance of the block  $LU$  algorithm for diagonally dominant matrices for various block sizes on an Alliant FX/8 [67]. The performance was computed using the nonblock version operation count. The actual rate is, therefore, higher for the block methods.

The curves in Fig. 8 clearly show the trends predicted by the analysis above. The significant improvement over BLAS2-based routines by a small amount of blocking can be seen in the performance of the  $\omega = 8$  curve and comparing it to the 7 to 10 Mflops possible via a BLAS2-based Version 2 code or the 15 to 17 Mflops of a BLAS2-based

Version 3 code. As expected, for any fixed order of the system, performance improves as  $\omega$  is increased until an optimal is reached. For small systems, increasing beyond this value causes performance degradation due to the conflict between reducing  $\mu$  and efficiently distributing work among the primitives. For larger systems, the conflict reduces and performance is maintained until  $\omega$  exceeds  $\sqrt{CS}$ .

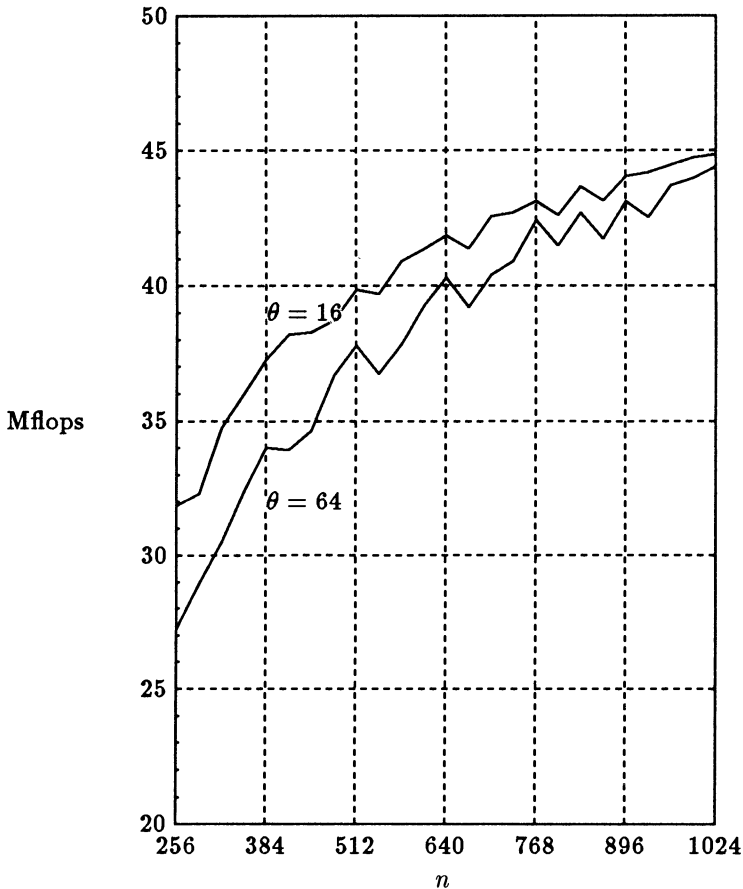


FIG. 9. Performance of double-level block LU on an Alliant FX/8.

The conflict between arithmetic time and data loading overhead minimization which produces the shifting of the preferred blocksize as a function of  $n$  can be mitigated somewhat by using a double-level blocking [67]. This conflict has been deliberately exacerbated in these experiments by using a Fortran implementation of the Gauss-Jordan primitive and assembler coded BLAS3 routines.

There are two basic approaches to double-level blocking: *inner-to-outer* and *outer-to-inner*. Both require a pair of blocksizes  $(\theta, \omega)$ . The outer-to-inner approach replaces the operation of the Gauss-Jordan primitive on a system of order  $\omega$  with a block LU factorization using the *inner* blocksize  $\theta$ . The inner-to-outer approach begins with a block LU factorization with blocksize  $\theta$  which is determined largely by the arithmetic time analysis and which is typically smaller than the single-level load analysis would recommend. Several rank- $\theta$  updates are then grouped together into a rank- $\omega$  in order to improve the data loading overhead. The decoupling methodology can be used to

show that these techniques do mitigate the conflict between reducing the arithmetic time component and the data loading overhead (see [67] for details). Figure 9 demonstrates that the use of inner-to-outer form of double-level blocking can indeed improve performance. Note that that double-level version yields performance higher than all of the single-level implementations of Fig. 8 over the entire interval.

**4.2. Distributed-memory algorithms.** Our objective here is to describe the effects that the data-storage and pivoting schemes have on the efficiency of the  $LU$  factorization of a dense matrix  $A = (\alpha_{ij})$  on distributed memory systems. The related parallel Cholesky schemes will not be discussed in this section; for an example, see Heath [88]. We also describe some  $LU$ -like factorization schemes that are useful on distributed memory and hybrid architectures.

**4.2.1.  $LU$  factorization.** A number of papers have appeared in recent years describing various parallel  $LU$  factorization schemes on such architectures, e.g., see Ipsen, Saad, and Schultz [104], Chu and George [28], Geist and Heath [77], [78], and Geist and Romine [79]. We will concentrate here only on the work of Geist and Romine.

Consider the two basic storage schemes: storage of  $A$  by rows and by columns. The row storage case is considered first. Adopting the terminology of Geist and Romine [79], we refer to the following scheme as RSRP, *Row Storage with Row Pivoting*.

RSRP:

each processor executes the following,

do  $k = 0, n - 1$

    determine row pivot

    update permutation vector

    if (I own pivot row)

        fan-out(broadcast) pivot row

    else

        receive pivot row

    for (all rows  $i > k$  that I own)

$\lambda_{ik} = \alpha_{ik} / \alpha_{kk}$

        do  $j = k + 1, n - 1$

$\alpha_{ij} = \alpha_{ij} - \lambda_{ik} \alpha_{kj}$

        enddo

enddo.

In most of the early work, row storage for the coefficient matrix was chosen principally because no efficient parallel algorithms were then known to exist for the subsequent forward and backward sweeps if the coefficient matrix were to be stored by columns. But, as discussed earlier, recent triangular solvers for distributed memory multiprocessors have removed the main reason for preferring row storage. Next, the *Column Storage with Row Pivoting* (CSRП) scheme is given by:

CSRП:

each processor executes the following

do  $k = 0, n - 1$

    if (I own column  $k$ )

```

determine pivot row
interchange
do  $i = k + 1, n - 1$ 
     $\lambda_{ik} = \alpha_{ik} / \alpha_{kk}$ 
broadcast the column just computed and pivot index
else
    receive the column just computed and pivot index
interchange
for (all columns  $j > k$  that I own)
    do  $i = k + 1, n - 1$ 
         $\alpha_{ij} = \alpha_{ij} - \lambda_{ik} \alpha_{kj}$ 
    enddo
enddo.

```

A modification of RSRP, which we refer to as RSCP, *Row Storage with Column Pivoting*, consists of searching the current pivot row for the element with maximum modulus, and then exchanging columns to bring this element to the diagonal. The RSCP algorithm can be readily seen as nothing more than the dual of algorithm CSRP. Geist and Heath [78] indicate that both RSCP and CSRP yield essentially identical speedup on an Intel iPSC hypercube. In fact, Geist and Heath conclude that, in the absence of such techniques as loop unrolling,  $LU$  factorization with partial pivoting is most efficient when pipelining is used to mask the cost of pivoting. In particular, the two schemes that can most easily be pipelined are: pivoting by interchanging rows when the matrix is distributed across the processors by columns (algorithm CSRP), and pivoting by interchanging columns when the matrix is distributed across the processors by rows (algorithm RSCP).

**4.2.2. Pairwise pivoting.** Gaussian elimination with pairwise pivoting is an alternative to  $LU$  factorization which is attractive on a variety of distributed memory architectures including systolic arrays since it introduces parallelism into the pivoting strategy.<sup>10</sup> Such a pivoting strategy dates back to Wilkinson's work on Gaussian elimination using the ACE computer with its limited amount of memory [62]. The main idea is rather simple. If  $u^T = [\mu_1, \dots, \mu_n]$  and  $v^T = [\nu_1, \dots, \nu_n]$  are two row vectors, then we can choose a stabilized elementary transformation

$$S = \begin{pmatrix} 1 & 0 \\ \alpha & 1 \end{pmatrix} P$$

so as to annihilate either  $\mu_1$  or  $\nu_1$ , whichever is smaller in magnitude. Here,  $P$  is either the identity of order 2 or  $(e_2, e_1)$  so that

$$S \begin{pmatrix} u^T \\ v^T \end{pmatrix} = \begin{pmatrix} \tilde{\mu}_1 & \tilde{\mu}_2 & \cdots & \tilde{\mu}_n \\ 0 & \tilde{\nu}_2 & \cdots & \tilde{\nu}_n \end{pmatrix}.$$

One of the many possible annihilation schemes for reducing a nonsingular matrix  $A$  of order  $n$  to upper triangular form is illustrated in Fig. 10 for  $n = 8$ . (The elements marked with  $i$  can all be eliminated simultaneously on step  $i$ .)

Such a triangularization scheme requires  $2n - 3$  *stages* in which each stage consists of a maximum of  $\lfloor n/2 \rfloor$  independent stabilized transformations. It is ideally suited

<sup>10</sup> Pairwise pivoting can also be useful on shared memory machines to break the bottleneck caused by partial pivoting discussed earlier.

$$\begin{pmatrix} * & & & & & & & & & & & & \\ 1 & * & & & & & & & & & & & \\ & 2 & 3 & * & & & & & & & & & \\ & 3 & 4 & 5 & * & & & & & & & & \\ & 4 & 5 & 6 & 7 & * & & & & & & & \\ & 5 & 6 & 7 & 8 & 9 & * & & & & & & \\ & 6 & 7 & 8 & 9 & 10 & 11 & * & & & & & \\ & 7 & 8 & 9 & 10 & 11 & 12 & 13 & & & & & \end{pmatrix}$$

FIG. 10. Annihilation scheme for  $n = 8$ .

for a ring of processors [157] or other systolic arrays [80]. Note, however, that it does not produce an  $LU$  factorization of the matrix.  $L$  is replaced by a product of matrices in which each one can be readily inverted. One possible drawback of this pivoting strategy is that the upper bound on the growth factor is the square of that of partial pivoting [168], [169]. Our extensive numerical experiments indicate that, as is the case with partial pivoting, such growth is rarely encountered in practice. In that sense, our experience contradicts some conclusions of Trefethan and Schreiber [184] indicating that some further work is required to reconcile this seeming inconsistency.

The above annihilation scheme was originally motivated by a parallel Givens reduction introduced in [161] and now used extensively in applications such as signal processing for recursive least squares computations. This parallel Givens reduction was later generalized for a ring of processors [158].

**4.2.3. A hybrid scheme.** In order to design factorization schemes for multi-cluster machines, such as Cedar, in which each cluster is a parallel computer with tightly coupled processors, we must combine the strategies outlined above for both shared and distributed memory models. Breaking the problem among the clusters so as to minimize intercluster communication while maintaining load balancing is an issue faced by users of distributed memory architectures. Cedar's advantage is the existence of a shared global memory.

The shared memory block  $LU$  algorithm and the BLAS3 primitives, discussed above, are concerned with achieving high performance on an architecture like a single Cedar cluster. While these algorithms and kernels form an invaluable building block for algorithms on the Cedar system and the conclusions of the analysis are applicable over a fairly wide range of multivector architectures, care must be taken not to generalize these conclusions too far. For example, on a single Cedar cluster (and similar architectures) routines for many of the basic linear algebra tasks encountered in practice can be designed as a series of calls to BLAS3 kernels and BLAS2-implemented algorithms thereby masking all of the architectural considerations of parallelism, vectorization, and communication. This method of algorithm design, however, cannot be generalized to all hierarchical shared memory machines. One of the main reasons for this is the fact that an algorithm designed via this method may have problems with an inappropriate choice of task granularity and the resulting excessive communication requirements. The need to introduce double-level blocking forms of the algorithm indicated the onset of such a problem on a Cedar cluster: the attempt to spread the BLAS2-implemented kernel across the processors in a cluster introduced serious limitations on the performance of the block algorithm. When this problem

becomes extreme, other forms of the algorithm must be used which typically involve reorganizing the block computations to more efficiently map the algorithm to the architecture via tasks of coarser granularity with more attention focused on minimizing the required communication. Typically this involves some notion of *pipelining* (possibly multidimensional) at the block level, e.g., see [14], [157].

An example of such a situation is the solution of a dense linear system using more than one cluster of Cedar (possibly a subset of the total number available). In this case the algorithm design must take into account that intercluster communication is rather costly. There are several possible designs for such an algorithm. One of the most straightforward is based on the outer-to-inner double-level block form presented above. The block computations can be pipelined across clusters using the necessary Cedar synchronization primitives. A second possibility uses the control structure of the pipelined Givens factorization on a ring of processors described in [158]. A block of rows rather than a single row is communicated between processors and the row rotation is replaced with a block Gaussian elimination procedure. The remainder of this section discusses another algorithm, due to Sameh [157], for solving dense linear systems on a multiple cluster architecture which requires a relatively small amount of intercluster communication. For simplicity a four-cluster Cedar is assumed.

Let  $A$ , a nonsingular matrix of order  $n$ , be partitioned as

$$A^T = (A_1^T, A_2^T, A_3^T, A_4^T)$$

where  $A_i$  resides in the  $i$ th cluster memory. The algorithm consists of two major stages. In the first stage, using a block- $LU$  scheme with partial pivoting, each  $A_i$  is factored into the form

$$P_i A_i = L_i U_i$$

for  $i = 1, 2, 3, 4$  where  $P_i$  is a permutation,  $L_i$  is unit lower triangular, and  $U_i$  is upper trapezoidal.

Assuming, without loss of generality, that each  $U_i$  has a nonsingular upper triangular part, the factorization of  $A$  may be completed in the second stage which consists of  $3n/4$  computational waves pipelined across the four clusters. These computational waves comprise three groups of  $n/4$  waves. During the  $k$ th group the latest values for the rows of  $U_k$  are used by clusters  $k+1$  to 4 in a pipelined fashion to further reduce their segments of the decomposition. It should be noted that cluster  $k$  is idle during the  $k$ th group of waves and the remainder of the algorithm since the other clusters will update the rows of  $U_k$  that it has produced and placed in global memory. (For example, cluster 1 only performs the initial reduction of  $A_1$  and is then released for other tasks within the application code of which solving the system is a part or the tasks of other users since Cedar is a multiuser system.) The first group of  $n/4$  computational waves which use the rows of  $U_1$  produced by cluster 1 is described below. The pattern of the remaining two groups follows trivially.

**Wave 1.** Let  $U_k \equiv [\mu_{i,j}^k]$ . The first row of  $U_1$  is transmitted via the global memory to cluster 2 where it is used, with pairwise pivoting, to annihilate the first element of the (possibly new due to pairwise pivoting) first row of  $U_2$ ,  $\mu_{1,1}^2$ . The updated first row of  $U_1$  is then transmitted to cluster 3 so as to annihilate  $\mu_{1,1}^3$  and then to cluster 4 where  $\mu_{1,1}^4$  is eliminated with the final version of the first row of  $U_1$  residing in global memory.

As soon as  $\mu_{1,1}^k$  is annihilated in cluster  $k$ ,  $k = 2, 3, 4$ , the nonzero portion of  $U_k$  is a  $n/4 \times (n-1)$  upper Hessenberg matrix, e.g., for  $n = 24$  it is of the form

$$\begin{pmatrix} x & x & x & x & x & x & \cdots & x \\ x & x & x & x & x & x & \cdots & x \\ & x & x & x & x & x & \cdots & x \\ & & x & x & x & x & \cdots & x \\ & & & x & x & x & \cdots & x \\ & & & & x & x & \cdots & x \\ & & & & & x & \cdots & x \end{pmatrix}.$$

The cluster then proceeds to reduce  $U_k$  to upper trapezoidal form through a pipelined Gaussian elimination process using pairwise pivoting.

**Waves**  $2 \leq j \leq n/4$ . Similar to the first wave, the  $j$ th row of  $U_1$  is transmitted to clusters 2, 3, and 4 to annihilate  $\mu_{1,j}^2$ ,  $\mu_{1,j}^3$ , and  $\mu_{1,j}^4$ , respectively. After these annihilations occur, each cluster reduces  $U_k$ , which at this point is upper Hessenberg, to upper trapezoidal form.

Note that after this first group of computational waves  $U_1$  is in its final form in global memory. The matrix  $U_2$  is in its penultimate form since it will only change due to the pairwise pivoting done by clusters 3 and 4 in the second group of computational waves. This implies that cluster 2 is now available for other work. The second and third computational groups proceed in the same way as the first did with each cluster fetching the appropriate row from the source matrix,  $U_2$  followed by  $U_3$ , transforming  $U_k$  to upper Hessenberg form and then reducing it back to an upper trapezoidal matrix. This basic form of the algorithm possesses many levels of communication and computation granularity and can be modified to improve utilization of a multicluster architecture. For example, if the whole Cedar machine were devoted to such a dense solver, simple interleaving of block rows of  $A$  would enhance load balancing among the clusters.

**4.3. Block tridiagonal linear systems.** Block tridiagonal systems arise in numerous applications — one example being the numerical handling of elliptic partial differential equations via finite element discretization. Often, solving such linear systems constitutes the major computational task. Hence, efficient algorithms for solving these systems on vector and parallel computers are of importance. Using block versions of Gaussian elimination for block tridiagonal systems seems a natural extension of the efficient dense solvers discussed above. Some of the early work may be found in [191] and the survey by Heller [90]. A more recent study of block Gaussian elimination on the Alliant FX/8 for solving such systems [11] indicates the importance of efficient dense solvers and the underlying BLAS3 as components for block tridiagonal solvers.

If the size of the blocks is small, i.e., a narrow-banded system, such forms of Gaussian elimination offer little potential vectorization and parallelization. Similar to the above discussions for banded triangular systems, a partitioning scheme, referred to as the *spike* algorithm below, for handling tridiagonal systems on vector or parallel computers was introduced in [161], where Givens reductions were used to handle the diagonal blocks. Later, Wang [192] considered the simpler problem of diagonally dominant systems and gave essentially the same form of the algorithm modified to use Gaussian elimination (made possible by the assumption of diagonal dominance) and a different method for the elimination of the *spikes*. Several studies have generalized this partitioning scheme to narrow-banded systems, e.g., see [46], [47], [120], [132] and the recent book by Ortega [137].

The main idea of this partitioning scheme may be outlined as follows. Let the linear system under consideration be denoted by  $Ax = f$ , where  $A$  is a banded diagonally dominant matrix of order  $n$ . It is assumed that the number of superdiagonals  $m \ll n$  is equal to the number of subdiagonals and that, for simplicity of presentation,  $n = pq$ . On a sequential machine such a system would be solved via Gaussian elimination, see [38] for example. The algorithm described below assumes  $p$  CPU's of a CRAY X-MP or CRAY-2, or a Cedar system with  $p$  clusters. Here, for the sake of illustration,  $p$  is taken to be 4.

Let the matrix  $A$  be partitioned into the block-tridiagonal form with block row  $[C_i, A_i, B_i]$  and conformally  $x$  and  $f$ , e.g.,

$$\begin{pmatrix} A_1 & B_1 & 0 & 0 \\ C_2 & A_2 & B_2 & 0 \\ 0 & C_3 & A_3 & B_3 \\ 0 & 0 & C_4 & A_4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{pmatrix},$$

where each  $A_i$ ,  $1 \leq i \leq p$ , is a banded matrix of order  $q = n/p$  and bandwidth  $2m + 1$  (same as  $A$ ),

$$B_i = \begin{pmatrix} 0 & 0 \\ \hat{B}_i & 0 \end{pmatrix}$$

and

$$C_{i+1} = \begin{pmatrix} 0 & \hat{C}_{i+1} \\ 0 & 0 \end{pmatrix},$$

$1 \leq i \leq p - 1$ , in which  $\hat{B}_i$  and  $\hat{C}_{i+1}$  are lower and upper triangular matrices, respectively, each of order  $m$ .

The algorithm consists of three stages.

**Stage 1.** If both sides of  $Ax = f$  were premultiplied by  $\text{diag}(A_1^{-1}, A_2^{-1}, \dots, A_p^{-1})$  we obtain a system of the form

$$\begin{pmatrix} I_q & E_1 & 0 & 0 \\ F_2 & I_q & E_2 & 0 \\ 0 & F_3 & I_q & E_3 \\ 0 & 0 & F_4 & I_q \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} g_1 \\ g_2 \\ g_3 \\ g_4 \end{pmatrix},$$

where

$$E_i = (\hat{E}_i, 0), \quad F_i = (0, \hat{F}_i),$$

in which  $\hat{E}_i$  and  $\hat{F}_i$  are matrices of  $m$  columns given by

$$\hat{E}_i = A_i^{-1} \begin{pmatrix} 0 \\ \hat{B}_i \end{pmatrix}$$

and

$$\hat{F}_i = A_i^{-1} \begin{pmatrix} \hat{C}_i \\ 0 \end{pmatrix}$$

and will, in general, be full.



In stage 1,  $\hat{E}_i$ ,  $\hat{F}_i$ , and  $g_i$  are obtained by solving the associated linear systems. In each cluster  $2 \leq k \leq 4$  we solve  $2m + 1$  linear systems of the form  $A_k v = r$ , while clusters 1 and 4 each solves  $m + 1$  linear systems of the same form. Note that no intercluster communication is needed.

The method of solution used on each cluster (Alliant FX/8) for these 4 systems with multiple right-hand sides, varies with  $m$ . For  $m < 8$  a variant of the spike algorithm is used. For  $8 \leq m \leq 16$  (approximately), block cyclic reduction is the most effective and for larger  $m$  a block Gaussian elimination is recommended [11].

**Stage 2.** Let  $\hat{E}_i$  and  $\hat{F}_i$  be partitioned, in turn, as follows

$$\hat{F}_i = \begin{pmatrix} P_i \\ M_i \\ Q_i \end{pmatrix}, \quad \hat{E}_i = \begin{pmatrix} S_i \\ N_i \\ T_i \end{pmatrix},$$

where  $P_i$ ,  $Q_i$ ,  $S_i$ , and  $T_i \in \mathbb{R}^{m \times m}$ . Also, let  $g_i$  and  $x_i$  be conformally partitioned:

$$g_i = \begin{pmatrix} h_{2i-2} \\ w_i \\ h_{2i-1} \end{pmatrix}, \quad x_i = \begin{pmatrix} y_{2i-2} \\ z_i \\ y_{2i-1} \end{pmatrix}.$$

The structure of the resulting partitioned system is such that the unknown vectors  $y_j$ ,  $1 \leq j \leq 6$  (each of order  $m$ ) are disjoint from the rest of the unknowns. In other words, the  $m$  equations above and the  $m$  equations below each of the 3 partitioning lines form an independent system of order  $6m$ , which is referred to as the *reduced system*  $Ky = h$ ,

$$\begin{pmatrix} I_m & T_1 & & & & \\ P_2 & I_m & & S_2 & & \\ Q_2 & & I_m & T_2 & & \\ & P_3 & I_m & & S_3 & \\ & Q_3 & & I_m & T_3 & \\ & & P_4 & I_m & & \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{pmatrix} = \begin{pmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \end{pmatrix}.$$

Since  $A$  is diagonally dominant, it can be shown that the reduced system is also diagonally dominant and hence there are a number of options available for solving it. Typically, it is small enough to be sent to a single Cedar cluster and solved with an appropriate algorithm.

When it is large enough to warrant a multicluster approach the reduced-system approach could be applied again. Note, however, that the bandwidth of the system has doubled compared to the original system. Block-column permutations can reduce the bandwidth back to its original value but this destroys diagonal dominance and pivoting will usually be required to solve the permuted reduced system. It is also possible to use all of the clusters to solve the reduced system via an iterative technique such as Orthomin(k) [47].

Finally, if the original linear system is *sufficiently* diagonally dominant, we can ignore the matrices  $Q_i$  and  $S_i$  as  $\|S_i\|_\infty$  and  $\|Q_i\|_\infty$  are much smaller than  $\|T_i\|_\infty$  and  $\|P_i\|_\infty$ , respectively. This results in a block-diagonal reduced system in which each block is of the form

$$\begin{pmatrix} I_m & T_k \\ P_{k+1} & I_m \end{pmatrix}$$

for  $1 \leq k \leq 3$ .

**Stage 3.** Once the  $y_i$ 's are obtained, the rest of the components of the solution vector of the original system may be retrieved as follows:

$$z_k = w_k - M_k y_{2k-3} - N_k y_{2k},$$

for  $1 \leq k \leq 4$ ,

$$y_0 = h_0 - S_1 y_2,$$

and

$$y_7 = h_7 - Q_4 y_5.$$

Provided that the  $y_i$ 's are stored in the global memory, this stage requires no inter-cluster communication.

In addition to reporting on the performance results for this algorithm on the Alliant FX/8, [11] also reports on the performance achieved on four CPU's of a CRAY X-MP/416. Using four partitions on a system of order 16384 with blocksize 32, a speedup relative to itself of 3.8 was achieved indicating an efficient use of the micro-tasking capabilities and memory system of the machine. The speedup compared to a block *LU* algorithm on one CPU was approximately 2.

There are several modifications and reorganizations possible of the spike algorithm for solving banded systems discussed above. These can be used to alter the form of the algorithm to more efficiently map to a variety of shared memory architectures. For one such alternative see [155]. Also, if the system is symmetric positive definite, Dongarra and Johnsson [46] have discussed how the algorithm can be modified to obtain a reduced system that is symmetric positive definite as well.

An analysis of the parallel and numerical aspects of a two-sided Gaussian elimination for solving tridiagonal systems has been given recently by van der Vorst [188].

The work by Johnsson [108], [109] is representative of organization of concurrent algorithms for solving tridiagonal and narrow banded systems on distributed memory machines with various connection topologies, e.g., two-dimensional arrays, shuffle-exchange networks and boolean cubes. Fox et al. have also considered the problem of banded systems on hypercubes. In [60], they provide a detailed performance analysis of the problem.

**5. Least squares.** In solving the linear least squares problem:

$$(10) \quad \min \|f - Ax\|_2,$$

where  $A$  is an  $m \times n$  matrix of rank  $n$ , ( $m \geq n$ ), it is often necessary to obtain the factorization,

$$(11) \quad QA = \begin{pmatrix} R \\ 0 \end{pmatrix},$$

in which  $Q$  is an orthogonal matrix and  $R$  is a nonsingular upper triangular matrix of order  $n$ . Such a factorization may be realized on multiprocessors via plane rotations, see [48], [158], and [161], elementary reflectors, see [16] and [158], or the Modified Gram-Schmidt algorithm, see [9]. (Although the latter algorithm is more commonly associated with the calculation of an orthogonal basis of the range of  $A$ .)

In the section concerning shared memory multiprocessors, block versions of Householder reduction and the modified Gram-Schmidt algorithm are presented, as well as a pipelined Givens reduction for updating matrix factorization. For distributed memory multiprocessors, organization of Givens and Householder reductions on a ring of processors convey the main ideas needed for implementation on hypercubes and locally connected distributed memory architectures.

### 5.1. Shared-memory algorithms.

**5.1.1. A block Householder reduction.** If  $A \equiv A_1 = [a_1^{(1)}, a_2^{(1)}, \dots, a_n^{(1)}]$ , then it is possible to generate elementary reflectors  $P_k = I - \alpha_k u_k u_k^T$ ,  $k = 1, \dots, n$ , such that forming  $P_k A_k$  produces the  $k$ th row of  $R$  and the  $(m - k) \times (n - k)$  matrix  $A_{k+1} = [a_{k+1}^{(k+1)}, \dots, a_n^{(k+1)}]$  by annihilating all but the first element in  $a_k^{(k)}$ . The two basic tasks in such a procedure are [170]: (i) generation of the reflector  $P_k$  such that  $P_k a_k^{(k)} = (\rho_{kk}, 0, \dots, 0)^T$ ,  $k = 1, 2, \dots, n$ ; and (ii) updating the remaining  $(n - k)$  columns,  $P_k a_j^{(k)} = (\rho_{kj}, a_j^{(k+1)T})^T$ ,  $j = k + 1, \dots, n$ . On a parallel computer, reflector  $P_{k+1}$  may be generated even before task (ii) for stage  $k$  is finished. While an organization that allows such an overlap is well suited for some shared memory machines and for a distributed memory multiprocessor such as a ring of processors, e.g., see [158], it does not offer the data locality needed in a hierarchical shared memory system such as that of an Alliant FX/8.

A block scheme proposed by Bischof and Van Loan [16], see also the related papers [15], [19], [36], [146], [163], offers such data locality. This scheme depends on the fact that the product of  $k$  elementary reflectors  $Q_k = (P_k, \dots, P_2, P_1)$ , where  $P_i = I_m - w_i w_i^T$ , can be expressed as a rank- $k$  update of the identity of order  $m$ , i.e.,

$$Q_k = I_m - V_k U_k^T,$$

where  $V_1 = U_1 = w_1$ ,  $V_j = (P_j V_{j-1}, w_j)$  and  $U_j = (U_{j-1}, u_j)$ , for  $j = 2, \dots, k$ .

The block algorithm may be described as follows. Let the  $m \times n$  matrix ( $m \geq n$ ) whose orthogonal factorization is desired be given by

$$A = [A_1, B],$$

where  $A$  is of rank  $n$ , and  $A_1$  consists of the first  $k$  columns of  $A$ . Next, proceed with the usual Householder reduction scheme by generating the  $k$  elementary reflectors  $P_1$  through  $P_k$  such that

$$(P_k \cdots P_2 P_1) A_1 = \begin{pmatrix} R_1 \\ 0 \end{pmatrix},$$

where  $R_1$  is upper triangular of order  $k$  without modifying the matrix  $B$ . If we accumulate the product  $Q_k = P_k \cdots P_1 = I - V_k U_k^T$  as each  $P_i$  is generated, the matrix  $B$  is updated via

$$B \leftarrow (I - V_k U_k^T) B$$

which relies on the high efficiency of one of the most important kernels in BLAS3, that of a rank- $k$  update. The process is then repeated on the modified  $B$  with another *well-chosen* block size, and so on until the factorization is completed. It may also be desirable to accumulate the various  $Q_k$ 's, one per block, to obtain the orthogonal matrix,  $Q$ , that triangularizes  $A$ .

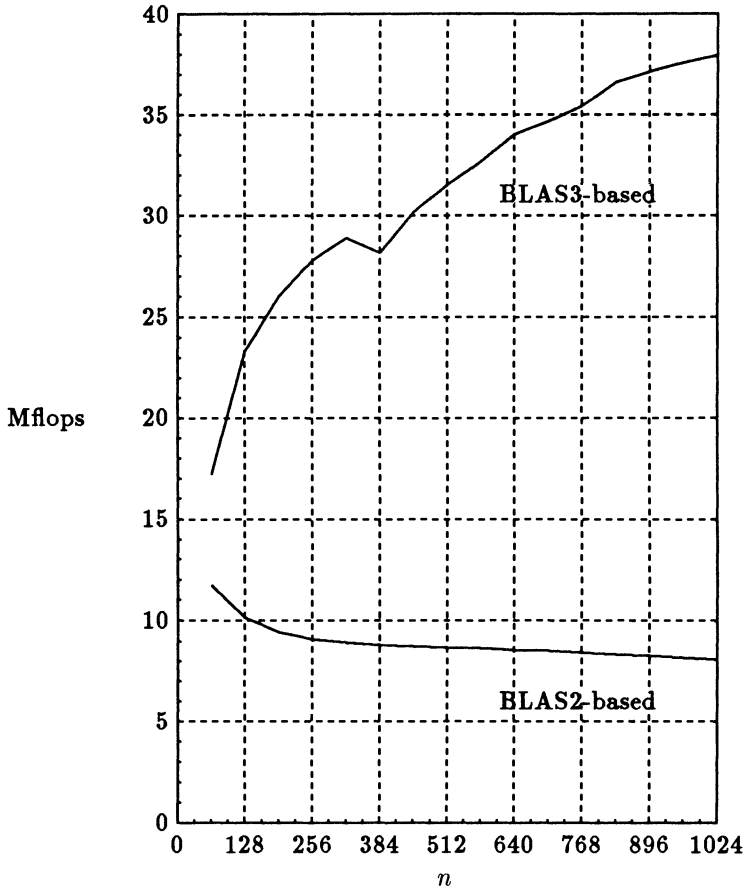


FIG. 11. Performance of block Householder algorithm on an Alliant FX/8.

It was shown in [16] that this block algorithm is as numerically stable as the classical Householder scheme. The block scheme, however, requires roughly  $(1 + 2/p)$  times the arithmetic operations needed by the classical sequential scheme, where  $p = n/k$  is the number of blocks (assuming a uniform block size throughout the factorization). Bischof and Van Loan report the performance of the block algorithm at 18 Mflops for large square matrices ( $n = 1000$ ) on an FPS-164/MAX with a single MAX board and note that an optimized LINPACK QR running on an FPS-164 without MAX boards would achieve approximately 6 Mflops. An example, of the performance achieved by a BLAS3 implementation of the block Householder algorithm (PQRDC) compared to a BLAS2 version (DQRDC) on an Alliant FX/8, [85], is shown in Fig. 11. The performance shown is computed using the nonblock algorithm operation count.

Most recently, Schreiber and Van Loan have considered a more efficient storage scheme for the product of Householder matrices [164]. They describe the *compact*  $WY$  representation of the orthogonal matrix  $Q$  which is of the form

$$Q = I + YTY^T,$$

where  $Y \in \mathbb{R}^{m \times n}$  is a lower trapezoidal matrix and  $T \in \mathbb{R}^{n \times n}$  is a upper triangular

matrix. The representation requires only  $mn$  storage locations and can be computed in a stable fashion.

**5.1.2. A block-modified Gram–Schmidt algorithm.** The goal of this algorithm is to factor an  $m \times n$  matrix  $A$  of maximal rank into an orthonormal  $m \times n$  matrix  $Q$  and an upper triangular  $R$  of order  $n$  where  $m > n$  and  $A$  is of maximal rank. Let  $A$  be partitioned into two blocks  $A_1$  and  $B$  where  $A_1$  consists of  $\omega$  columns of order  $m$ , with  $Q$  and  $R$  partitioned accordingly:

$$(A_1, B) = (Q_1, P) \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix}.$$

The algorithm is given by:

- (i)  $A_1 = Q_1 R_{11}$ ,
- (ii)  $R_{12} = Q_1^T B$ ,
- (iii)  $B_1 = B - Q_1 R_{12}$ .
- (iv) Apply the algorithm recursively to produce  $B_1 = P R_{22}$ .

If  $n = k\omega$ , step (i) is performed  $k$  times and steps (ii) and (iii) are each performed  $k - 1$  times.

Three primitives are needed for the  $j$ th step of the algorithm: a  $QR$  decomposition (assumed here to be a modified Gram–Schmidt routine — MGS); a matrix multiplication  $AB$ ; and a rank- $\omega$  update of the form  $C \leftarrow C - AB$ . The primitives allow for ideal decomposition for execution on a limited processor shared memory architecture. The BLAS2 version of the modified Gram–Schmidt algorithm is obtained when  $\omega = 1$  or  $\omega = n$ , and a double-level blocking version of the algorithm is derived in a straightforward manner by recursively calling the single-level block algorithm to perform the  $QR$  factorization of the  $m \times \omega$  matrix  $A_1$ .

Jalby and Philippe have considered the stability of this block algorithm [106] and Gallivan et al. have analyzed the performance as a function of blocksize [67]. Below, a summary of this blocksize analysis is presented along with experimental results on an Alliant FX/8 of single and double-level versions of the algorithm.

The analysis is more complex than that of the block  $LU$  algorithm for diagonally dominant matrices discussed above, but the conclusions are similar. This increase in complexity is due to the need to apply a BLAS2-based MGS primitive to an  $m \times \omega$  matrix at every step of the algorithm. As with the block version of the  $LU$  factorization with partial pivoting, this portion of each step makes poor use of the cache and increases the amount of work done in less efficient BLAS2 primitives. The analysis of the arithmetic time component clearly shows that the potential need for double-level blocking is more acute for this algorithm than for the diagonally dominant block  $LU$  factorization on problems of corresponding size.

The behavior of the algorithm with respect to the number of data loads can be discussed most effectively by considering approximations of the cache-miss ratios. For the interval  $1 \leq \omega \leq l \approx CS/m$  the cache-miss ratio is

$$\mu \approx \frac{1}{2\omega} + \eta_1,$$

where  $\eta_1$  is proportional to  $1/n$ , which achieves its minimum value  $m/(2CS)$  at  $\omega = l$ . Under certain conditions the cache-miss ratio continues to decrease on the interval

$l \leq \omega \leq n$  where it has the form

$$\mu \approx \frac{1}{2\omega} \left(1 - \frac{\gamma}{n}\right) + \frac{\omega}{2} \left(\frac{1}{n} + \frac{1}{CS}\right) + \eta_2,$$

where  $\eta_2$  is proportional to  $1/n$ , which reaches its minimum at a point less than  $\sqrt{CS}$  and increases thereafter, as expected. (See [67] for details.) When  $\omega = n$  the cache-miss ratio for the second interval is  $1/2$  corresponding to the degeneration from a BLAS3 method to a BLAS2 method. The composite cache-miss ratio function over both intervals behaves like a hyperbola before reaching its minimum; therefore the cache-miss ratio does not decline as rapidly in latter parts of the interval as it does near the beginning.

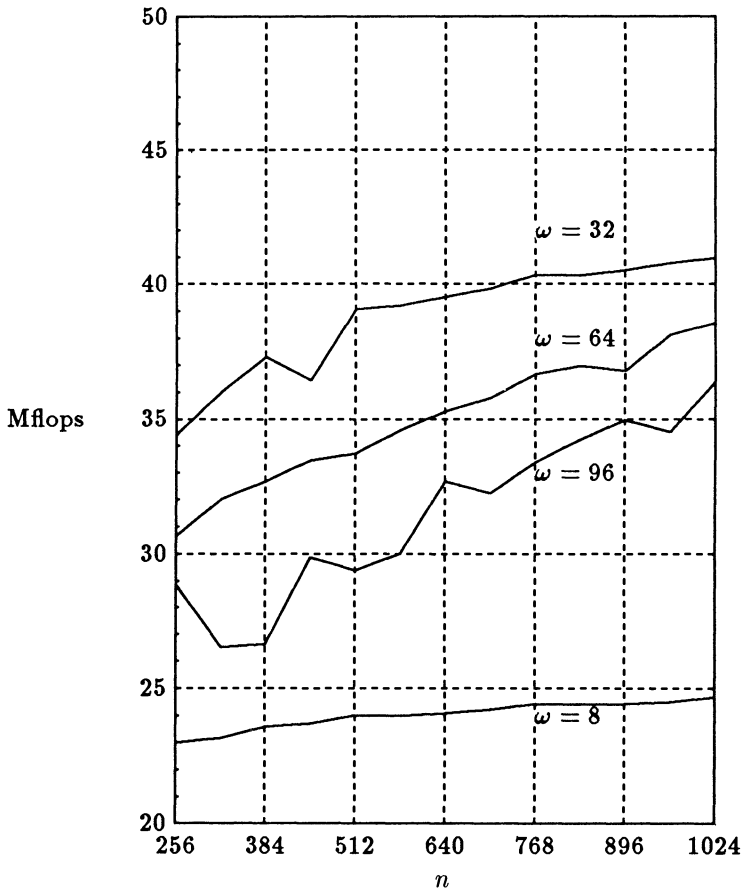


FIG. 12. Performance of one-level block MGS on an Alliant FX/8.

A load analysis of the double-level algorithm shows that double-level blocking either reduces or preserves the cache-miss ratio of the single-level version while improving the performance with respect to the arithmetic component of time.

Figures 12 and 13 illustrate, respectively, the results of experiments run on an Alliant FX/8, using single-level and double-level versions of the algorithm applied to square matrices. The cache size on this particular machine is  $16K$  double precision words.

For the range of  $n$ , the order of the matrix, shown in Fig. 12, the single-level optimal blocksize due to the data loading analysis starts at  $\omega = 64$ , decreases to  $\omega = 21$  for  $n = 768$ , and then increases to  $\omega = 28$  at  $n = 1024$ . Analysis of the arithmetic time component recommends the use of a blocksize between  $\omega = 16$  and  $\omega = 32$ . Therefore, due to the hyperbolic nature of  $\mu$  and the arithmetic time component analysis it is expected that the performance of the algorithm should increase until  $\omega \approx 32$ . The degradation in performance as  $\omega$  increases beyond this point to, say  $\omega = 64$  or  $96$ , should be fairly significant for small and moderately sized systems due to the rather large portion of the operations performed by the BLAS2 MGS primitive.

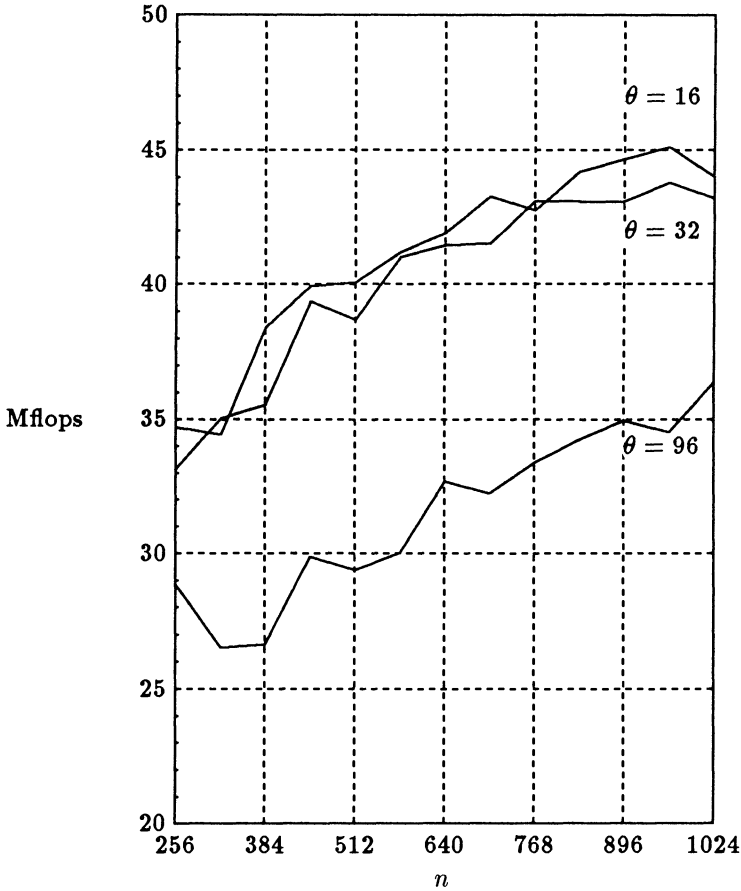


FIG. 13. Performance of two-level block MGS on an Alliant FX/8.

The results of the experiments confirm the trends predicted by the theory. The version using  $\omega = 32$  is clearly superior. The performance for  $\omega = 8$  is uniformly dismal across the entire interval since the blocksize is too small for both data loading overhead and arithmetic time considerations. Note that as  $n$  increases the gap in performance between the  $\omega = 32$  version and the larger blocksize versions narrows. This is due to both arithmetic time considerations as well as data loading. As noted above, for small systems, the distribution of operations reduces the performance of the larger blocksize version; but, as  $n$  increases, this effect decreases in importance. (Note that this narrowing trend is much slower than that observed for the block *LU*

algorithm. This is due to the fact that the fraction of the total operations performed in the slow primitive is  $\omega/n$  for the block Gram-Schmidt algorithm and only  $\omega^2/n^2$  for the block  $LU$ .) Further, for larger systems, the optimal blocksize for data loading is an increasing function of  $n$ ; therefore, the difference in performance between the three larger blocksizes must decrease.

Figure 13 shows the increase in performance which results from double-level blocking. Since the blocksize indicated by arithmetic time component considerations is between 16 and 32 these two values were used as the inner blocksize  $\theta$ . For  $\theta = 16$  the predicted outer blocksize ranges from  $\omega = 64$  up to  $\omega = 128$ ; for  $\theta = 32$  the range is  $\omega = 90$  to  $\omega = 181$ . (Recall that the double-level outer blocksize is influenced by the cache size only by virtue of the fact that  $\sqrt{CS}$  is used as a maximum cutoff point.) For these experiments the outer blocksize of  $\omega = 96$  was used for two reasons. First, it is a reasonable compromise for the preferred outer blocksize given the two values of  $\theta$ . Second, the corresponding single-level version of the algorithm, i.e.,  $(\theta, \omega) = (96, 96)$ , did not yield high-performance and a large improvement due to altering  $\theta$  would illustrate the power of double-level blocking. (To emphasize this point the curve with  $(\theta, \omega) = (96, 96)$  is included.) The curves clearly demonstrate that double-level blocking can improve the performance of the algorithm significantly. (See [67] for details.)

**5.1.3. Pipelined Givens rotations.** While the pipelined implementation of Givens rotations is traditionally restricted to distributed memory and systolic type architectures, e.g., [80], it has been successful on shared memory machines in some settings. In [48] a version of the algorithm was implemented on the HEP and compared to parallel methods based on Householder transformations. Rather than using the standard row-oriented synchronization pattern, the triangular matrix  $R$  was partitioned into a number of segments which could span row boundaries. Synchronization of the update of the various segments was enforced via the HEP's *full-empty* mechanism. The resulting pipelined Givens algorithm was shown to be superior to the Householder based approaches.

Gallivan and Jalby have implemented a version of the traditional systolic algorithm (see [80]) adapted to efficiently exploit the vector registers and cache of the Alliant FX/8. The significant improvement in performance of a structural mechanics code due to Berry and Plemmons, which uses weighted least squares methods to solve stiffness equations, is detailed in [10] (see also [144], [145]).

The hybrid scheme for  $LU$  factorization discussed earlier for cluster-based shared memory architectures converts easily to a rotation-based orthogonal factorization, see [157]. Chu and George have considered a variation of this scheme for shared memory architectures [31]. The difference is due to the fact that Sameh exploited the hybrid nature of the clustered memory and kept most of the matrix stored in a distributed fashion while pipelining between clusters the rows used to eliminate elements of the matrix. Chu and George's version keep these rows local to the processors and move the rows with elements to be eliminated between processors.

## 5.2. Distributed memory multiprocessors.

**5.2.1. Orthogonal factorization.** Our purpose in this section is to survey parallel algorithms for solving (10) on distributed memory systems. In particular, we discuss some algorithms for the orthogonal factorization of  $A$ . Several schemes have been proposed in the past for the orthogonal factorization of matrices on distributed memory systems. Many of them deal with systolic arrays and require the use of  $O(n^2)$



<u>Proc. 1</u>	<u>Proc. 2</u>	<u>Proc. 3</u>
1	—	—
21	—	—
31	2	—
41	32	—
51	42	3
61	52	43
71	62	53
4	72	63
54	—	73
64	5	—
74	65	—
—	75	6
—	—	76

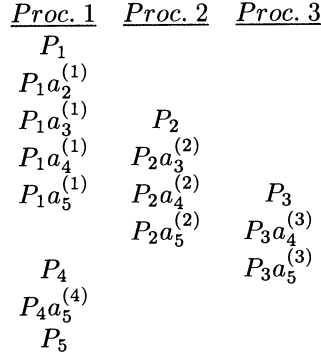
FIG. 14. *Givens reduction on a three processor ring.*

processors, where  $n$  is the number of columns of the matrix. For instance, Ahmed, Delosme, and Morph [4], Bojanczyk, Brent, and Kung [17], and Gentleman and Kung [80] all consider Givens reduction and require a triangular array of  $O(n^2)$  processors, while Luk [125] uses a mesh connected array of  $O(n^2)$  processors. Sameh [158], on the other hand, considers both Givens and Householder reduction on a ring of processors in which the number of processors is independent of the problem size. Each processor possesses a local memory with one processor only handling the input and output. Figure 14 shows the organization of Givens reduction on three processors for a rectangular matrix of seven rows and five columns on such a ring. Each column depicts the operations taking place in each processor. An entry  $ij$ ,  $j < i$ , indicates the rotation of rows  $i$  and  $j$  so as to annihilate the  $i$ th element of row  $j$ .

Recall that the classical Householder reduction may be described as follows. Let  $a_j^{(k)}$  denote the  $j$ th column of  $A_k$ , where  $A_{k+1} = Q_k A_k$  in which  $Q_k = \text{diag}(I_k, P_k)$ . Here,  $A_k$  is upper triangular in its first  $(k-1)$  rows and columns with  $P_k$  being the elementary reflector of order  $(m-k+1)$  that annihilates all the elements below the diagonal of the  $k$ th column of  $A_k$ . Then Householder reduction on the same matrix and ring architecture as above may be organized as shown in Fig. 15. Here, a  $P_k$  alone indicates generation of the  $k$ th elementary reflector.

Modi and Clarke [134] have suggested a greedy algorithm for Givens reduction and the equivalent ordering of the rotations, but do not consider a specific architecture or communication pattern. Cosnard, Muller, and Robert [32] have shown that the greedy algorithm is optimal in the number of timesteps required. Theoretical studies and comparisons of such algorithms for Givens reduction have been given by Pothén, Somesh, and Vemulapati [148] and by Elden [54]. We now briefly survey some of these algorithms that have been implemented on current commercially available distributed memory multiprocessors.

In chronological order, we begin with the work of Chamberlain and Powell [25]. In this study the coefficient matrix  $A$  is stored by rows across the processors in the usual wrap fashion and most of the rotations involve rows within one processor in a type of *divide-and-conquer* scheme. However, it is necessary to carry out rotations

FIG. 15. *Householder reflectors on a three processor ring.*

involving rows in different processors, which they call *merges*. They describe two ways of implementing the merges and compare them in terms of load balance and communication overhead. Numerical tests were made on an Intel iPSC hypercube with 32 processors based on 80287 floating point coprocessors to illustrate the practicality of their algorithms. The schemes used here are very similar the basic approach suggested originally by Golub, Plemmons, and Sameh [81] and developed further in [145]. We note that Katholi and Suter [112] have also adopted this approach in developing an orthogonal factorization algorithm for shared memory systems, and have performed tests on a 30 processor Sequent Balance computer.

Chu and George [30] have also suggested and implemented algorithms for performing the orthogonal factorization of a dense rectangular matrix on a hypercube multiprocessor. Their recommended scheme involves the embedding of a two-dimensional grid in the hypercube network, and their analysis of the algorithm determines how the aspect ratio of the embedded processor grid should be chosen in order to minimize the execution time or storage usage. Another feature of the algorithm is that redundant computations are incorporated into a communication scheme which takes full advantage of the hypercube connection topology; the data is always exchanged between neighboring processors. Extensive computational experiments which are reported by the authors on a 64-processor Intel hypercube support their theoretical performance analysis results.

Finally in this section we mention two studies which directly compare the results of implementations of Givens rotations with Householder transformations on local memory systems. Pothen and Raghavan [147] have compared the earlier work of Pothen, Somesh, and Vemulapati [148] on a modified version of a greedy Givens scheme with a standard row-oriented version of Householder transformations. Their tests seem to indicate that Givens reduction is superior on such an architecture. Kim, Agrawal, and Plemmons [113], however, have developed and tested a row-block version of the Householder transformation scheme which is based upon the *divide-and-conquer* approach suggested by Golub, Plemmons, and Sameh [81] (see also [29]). The tests by Kim, Agrawal, and Plemmons on a 64-processor Intel hypercube clearly favor their modified Householder transformation scheme.

**5.2.2. Recursive least squares.** In *recursive least squares* (RLS) it is required to recalculate the least squares solution vector  $x$  when observations (i.e., equations)

are successively added to or deleted from (10) without resorting to complete refactorization of the matrix  $A$ . For example, in many applications information continues to arrive and must be incorporated into the solution  $x$ . This is called *updating*. Alternatively, it is sometimes important to delete old observations and have their effects excised from  $x$ . This is called *downdating*. Applications of RLS updating and downdating include robust regression in statistics, modification of the Hessian matrix in certain optimization schemes, and in estimation methods in adaptive signal processing and control.

There are two main approaches to solving RLS problems; the *information matrix method* based on modifying the triangular matrix  $R$  in (11), and the *covariance matrix method* based instead on modifying the inverse  $R^{-1}$ . In theory, the information matrix method is based on modifying the normal equations matrix  $A^T A$ , while the covariance matrix method is based on modifying the *covariance matrix*

$$P = (A^T A)^{-1}.$$

The covariance matrix  $P$  measures the expected errors in the least squares solution  $x$  to (10). The Cholesky factor  $R^{-1}$  for  $P$  is readily available in control and signal processing applications.

Various algorithms for modifying  $R$  in the information matrix approach due to updating or downdating have been implemented on a 64-node Intel hypercube by Henkel, Heath, and Plemmons [92]. They make use of either plane rotations or hyperbolic type rotations.

The process of modifying least squares computations by updating the covariance matrix  $P$  has been used in control and signal processing for some time in the context of linear sequential filtering. We begin with estimates for  $P = R^{-1}R^{-T}$  and  $x$ , and update  $R^{-1}$  to  $\tilde{R}^{-1}$  and  $x$  to  $\tilde{x}$  at each recursive timestep. Recently Pan and Plemmons [140] have described the following parallel scheme.

**Algorithm** (*Covariance Updating*). Given the current least squares estimate vector  $x$ , the current factor  $L \equiv R^{-T}$  of  $P = (A^T A)^{-1}$  and the observation  $y^T x = \sigma$  being added, the algorithm computes the updated factor  $\tilde{L} \equiv \tilde{R}^{-1}$  of  $\tilde{P}$  and the updated least squares estimate vector  $\tilde{x}$  as follows:

1. Form the matrix vector product

$$(12) \quad a = Ly.$$

2. Choose plane rotations  $Q_i$ , to form

$$(13) \quad Q_m \cdots Q_1 \begin{bmatrix} -a \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ \delta \end{bmatrix}, \quad \delta = \sqrt{1 + \|a\|_2^2},$$

and update  $L$

$$(14) \quad Q_m \cdots Q_1 \begin{bmatrix} L \\ 0^T \end{bmatrix} = \begin{bmatrix} \tilde{L} \\ u^T \end{bmatrix}.$$

3. Form

$$(15) \quad \tilde{x} = x - \frac{1}{\delta} u(\sigma - y^T x).$$

As the recursive least squares computation proceeds,  $\tilde{L}$  replaces  $L$ ,  $\tilde{x}$  replaces  $x$ , a new equation is added, and the process returns to step 1. An efficient parallel implementation of this algorithm on the hypercube distributed-memory system making use of bidirectional data exchanges and some redundant computation is given in [93]. Steps 1 and 3 are highly parallelizable and effective implementation details of step 2 on a hypercube are given in [93].

Table 2 shows the speedup and efficiency on an iPSC/2 hypercube (4 MB of memory for each processor) for a single phase of the algorithm on a test problem of size  $n = 1024$ . One complete recursive update is performed. Here, the speedup is given by,

$$\text{speedup} = \frac{\text{time on 1 processor}}{\text{time on } p \text{ processors}},$$

with the corresponding efficiency,

$$\text{efficiency} = \frac{\text{speedup}}{p}.$$

An alternative hypercube implementation of the RLS scheme of Pan and Plemmons [140] has been given by Chu and George [31].

TABLE 2  
*Speedup and efficiency on the iPSC/2 for a problem of size  $n = 1024$ .*

Number of Processors	Speedup	Efficiency
$p$		
1	1	1
4	3.90	0.98
16	15.06	0.94
64	48.60	0.76

## 6. Eigenvalue and singular value problems.

**6.1. Eigenvalue problems.** Solving the algebraic eigenvalue problem, either standard  $Ax = \lambda x$ , or generalized  $Ax = \lambda Bx$ , is an important and potentially time-consuming task in numerous applications. In this brief review, only the dense case is considered for both the symmetric and nonsymmetric problems. Most of the parallel algorithms developed for the dense eigenvalue problem have been aimed at the standard problem. Algorithms for handling the generalized eigenvalue problem on shared or distributed memory multiprocessors are very similar to those used on sequential machines. Reduction of the symmetric generalized eigenvalue problem to the standard form is achieved by a Cholesky factorization of the symmetric positive definite matrix  $B$  which is well-conditioned in most applications. This reduction process can be made efficient on shared memory multiprocessors, for example, by adopting a block Cholesky scheme similar to the block  $LU$  decomposition discussed earlier to obtain the Cholesky factor  $L$  of  $B$  and to explicitly form the matrix  $L^{-1}AL^{-T}$  using the appropriate BLAS3. For the nonsymmetric generalized eigenvalue problems where the matrix  $B$  is known to be often extremely ill-conditioned in many applications, there is no adequate substitute to Moler and Stewart's  $QZ$ -scheme [136]. On a shared memory multiprocessor, the most efficient stage is the initial one of reducing  $B$  to the upper triangular form. Dispensing thus with the generalized eigenvalue problems, the

remainder of the section will be divided between procedures that depend on reduction to a condensed form, and Jacobi or Jacobi-like schemes for both the symmetric and nonsymmetric standard eigenvalue problems.

**6.1.1. Reduction to a condensed form.** We start with the nonsymmetric case. For the standard problem the first step, after balancing, is the reduction to upper Hessenberg form via orthogonal similarity transformations. These usually consist of elementary reflectors which could yield high computational rates on vector machines provided appropriate BLAS2 kernels are used. On parallel computers with hierarchical memories, block versions of the classical scheme, e.g., see [16], [44], [86], yield higher performance than BLAS2-based versions. Such block schemes are similar to those discussed above for orthogonal factorization, and their use does not sacrifice numerical stability. Block sizes can be as small as 2 for certain architectures. For the sake of illustration we present a simplified scheme for this block reduction to the upper Hessenberg form, where we assume that the matrix  $A$  is of order  $n$  where  $n = k\nu + 2$ .

```

do j = 1, k
  do i = (j-1) $\nu$  + 1, j $\nu$ 
    Obtain an elementary reflector  $P_i = I - w_i w_i^T$ 
    such that  $P_i$  annihilates the last  $n - i - 1$ 
    elements of the  $i$ th column of  $A$ 
    Construct:
     $U_i = (U_{i-1}, w_i)$ 
     $V_i = (P_i V_{i-1}, w_i)$ 
     $Y_i = (Y_{i-1}, A w_i)$ 
     $z_i = V_i^T e_{i+1}$ 
    if  $i = j\nu$  go to 10
     $a_{i+1} = (I - V_i U_i^T)(a_{i+1} - Y_i z_i)$ 
  enddo
10   $A(j\nu + 1 : n) = (I - V_{j\nu} U_{j\nu}^T)(A(j\nu + 1 : n) - Y_{j\nu} Z_{j\nu})$ 
enddo.
```

Here,  $Z_m$  consists of the last  $(n - m)$  rows of  $V_m$ . This block scheme requires more arithmetic operations than the classical algorithm using elementary reflectors by a factor of roughly  $1 + 1/k$ . Performance of the block scheme on the Alliant FX/8 is shown in Fig. 16 [86]. The performance shown is based on the operation count of the nonblock algorithm.

The next stage is that of obtaining the eigenvalues of the resulting upper Hessenberg matrix via the  $QR$ -algorithm with an implicit shifting strategy. This algorithm consists mainly of chasing a bulge represented by a square matrix of order 3 whose diagonal lies along the subdiagonal of the upper Hessenberg matrix. This in turn affects only 3 rows and columns of the Hessenberg matrix, leaving little that can be gained from vectorization, and to a lesser extent, parallelization. Stewart has considered the implementation of this basic iteration on a linear array of processors [172]. More recently, a block implementation with multiple  $QR$  shifts was proposed by Bai and Demmel [6] which yields some advantage for vector machines such as the Convex C-1 and Cyber 205.

If we are seeking all of the eigenvectors as well, the performance of the algorithm is enhanced since the additional work required consists of computations that are

amenable to vector and/or parallel processing; that of updating the orthogonal matrix used to reduce the original matrix to Hessenberg form.

Similarly, the most common method for handling the standard dense symmetric eigenvalue problem consists of first reducing the symmetric matrix to the tridiagonal form via elementary reflectors followed by handling the tridiagonal eigenvalue problem. Such reduction can be achieved by a minor modification of the above block reduction to the Hessenberg form. On 1 CPU of a CRAY X-MP, with an 8.5 *ns* clock, a BLAS2 implementation of Householder tridiagonalization using rank-2 updates (see [43]) yields a computational rate of roughly 200 Mflops for matrices of order 1000 (see Fig. 17 [87]). The performance of Eispack's TRED2 is also presented in the figure for comparison. Figure 18 shows a comparison of the performance of this BLAS3-based block reduction with a BLAS2-based reduction on the Alliant FX/8 [86]. As before, the performance is computed based on the nonblock version operation count.

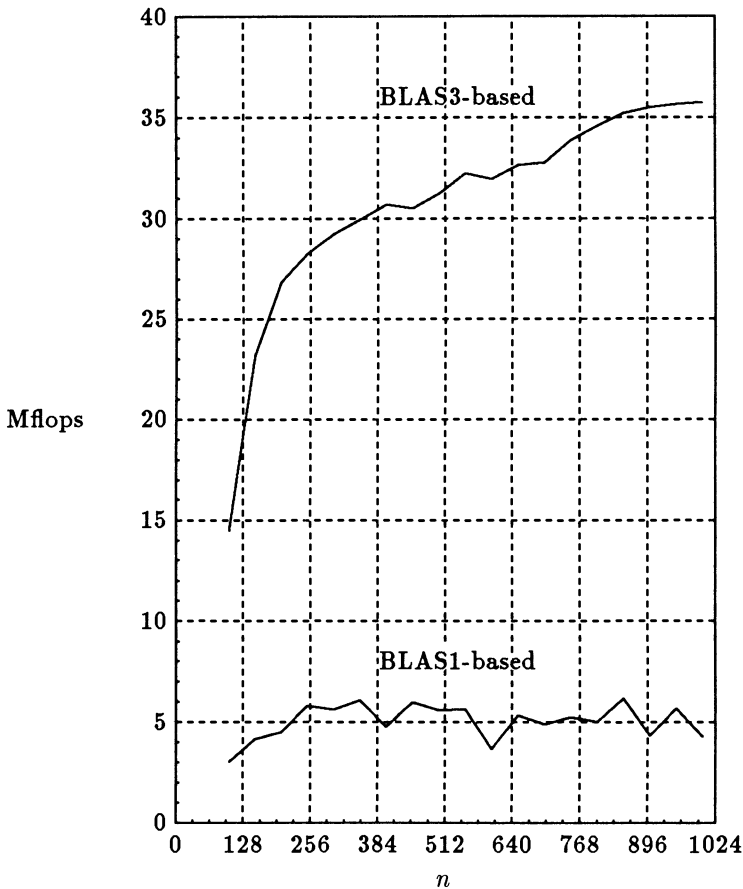


FIG. 16. Reduction to Hessenberg form on Alliant FX/8.

Once the tridiagonal matrix is obtained two approaches have been used, on sequential machines, for obtaining its eigenvalues and eigenvectors. If all the eigenvalues are required a  $QR$ -based method is used. The classical procedure is inherently sequential, offering nothing in the form of vectorization or parallelism. Recently, Dongarra and Sorensen [49], adapted an alternative due to Cuppen [33] for the use on multipro-

cessors. This algorithm obtains all the eigenvalues and eigenvectors of the symmetric tridiagonal matrix.

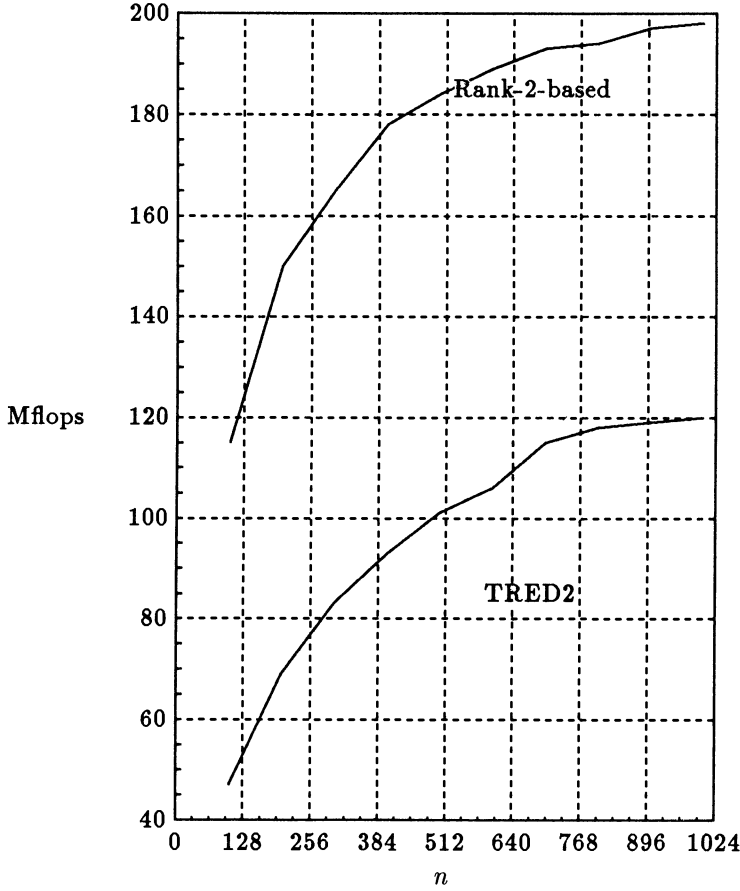


FIG. 17. Reduction to tridiagonal form on CRAY X-MP (1 CPU).

In its simplest form, the main idea of the algorithm may be outlined as follows. Let  $T = (\beta_i, \alpha_i, \beta_{i+1})$  be the symmetric tridiagonal matrix under consideration, where we assume that none of its off-diagonal elements  $\beta_i$  vanishes. Assuming that it is of order  $2m$ , it can be written as,

$$T = \begin{pmatrix} T_1 + \tau e_m e_m^T & \beta e_m e_1^T \\ \beta e_1 e_m^T & T_2 + \tau e_1 e_1^T \end{pmatrix},$$

where each  $T_i$  is tridiagonal of order  $m$ ,  $\tau$  is a “carefully” chosen scalar, and  $e_i$  is the  $i$ th column of the identity of order  $m$ . This in turn can be written as,

$$T = \text{diag}(T_1, T_2) + \gamma v v^T$$

in which the scalar  $\gamma$  and the column vector  $v$  can be readily derived. Now, we have two tasks: namely obtaining the spectral decomposition of  $T_1$  and  $T_2$ , i.e.,  $T_i = Q_i D_i Q_i^T$ ,  $i = 1, 2$ , where  $Q_i$  is an orthogonal matrix of order  $m$  and  $D_i$  is diagonal. Thus, if  $Q = \text{diag}(Q_1, Q_2)$  and  $D = \text{diag}(D_1, D_2)$ , then  $T$  is orthogonally similar to a rank-1

perturbation of a diagonal matrix, i.e.,

$$QTQ^T = D + \rho zz^T,$$

where  $\rho$  and  $z$  are trivially obtained from  $\gamma$  and  $z$ . The eigenvalues of  $T$  are thus the roots of

$$\phi(\lambda) = 1 + \rho z^T (D - \lambda I)^{-1} z$$

and its eigenvectors are given by,

$$u_i = \tau (D - \lambda_i I)^{-1} z,$$

where  $\tau = \|D - \lambda_i I\|_2$ .

This module may be used recursively to produce a parallel counterpart to Cuppen's algorithm [33] as demonstrated in [49]. For example, if the tridiagonal matrix  $T$  is of order  $2^k m$ , then the algorithm will consist of obtaining the spectral decomposition of  $2^k$  tridiagonal matrices each of order  $m$ , followed by  $k$  stages in which stage  $j$  consists of applying the above module simultaneously to  $2^{k-j}$  pairs of tridiagonal matrices in which each is of order  $2^{j-1} m$ .

If eigenvalues only (or all those lying in a given interval) or selected eigenpairs are desired, then a bisection-inverse iteration combination is used, e.g., see Wilkinson and Reinsch [195] or Parlett [141]. Such a combination has been adapted for the Illiac IV parallel computer, e.g., see [118] and [102], and later for the Alliant FX/8, see [123]. This modification depends on a multisectioning strategy in which the interval containing the desired eigenvalues is divided into  $(p - 1)$  subintervals where  $p$  is the number of processors. Using the Sturm sequence property we can simultaneously determine the number of eigenvalues contained in each of the  $(p - 1)$  subintervals. This is accomplished by having each processor evaluate the well-known linear recurrence leading to the determinant of the tridiagonal matrix  $T - \mu I$  or the corresponding nonlinear recurrence so as to avoid over- or underflow, e.g., see [141]. This process is repeated until all the eigenvalues, or clusters of computationally coincident eigenvalues, are separated. This "isolation" stage is followed by the "extraction" stage where the separated eigenvalues are evaluated using a root finder which is a hybrid of pure bisection and the combination of bisection and the secant methods, namely the ZEROIN procedure due to Brent and Dekker, see [58]. If eigenvectors are desired, then the final stage consists of a combination of inverse iteration and orthogonalization for those vectors corresponding to poorly separated eigenvalues.

This scheme proved to be the most effective on the Alliant FX/8 for obtaining all or few of the eigenvalues only. Compared to its execution time on one CE, it achieves a speedup of 7.9 on eight CE's, and is more than four times faster than Eispack's TQL1, e.g., see [167] or [195], for the tridiagonal matrix [-1,2,-1] of order 500 with the same achievable accuracy for the eigenvalues. Even if all the eigenpairs of the above tridiagonal matrix are required, this multisectioning scheme is more than 13 times faster than the best BLAS2-based version of Eispack's TQL2, 27 times faster than Eispack's pair Bisect and Tinvit, and five times faster than its nearest competitor, parallel Cuppen's procedure [49], with the same accuracy in the computed eigenpairs. For matrices with clusters of poorly separated eigenvalues, however, the multisectioning algorithm may not be competitive if all the eigenpairs are required with high accuracy. For example, for the well-known Wilkinson matrices  $W_{127}^+$ , e.g.,



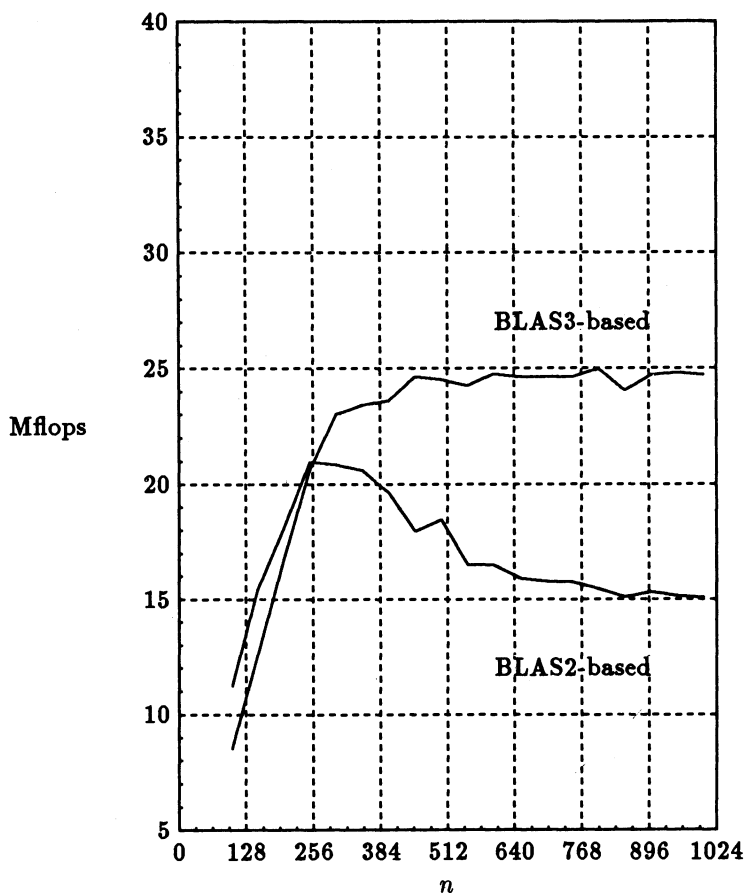


FIG. 18. Reduction to tridiagonal form on Alliant FX/8.

see [194], which have pairs of very close eigenvalues, the multisectioning method requires roughly twice the time required by the parallel Cuppen's procedure in order to achieve the same accuracy for all the eigenpairs.

Further studies by Simon [166] demonstrate the robustness of the above multisectioning strategy compared to other bisection-inverse iteration combinations proposed in [8]. Also, comparisons between the above multisectioning scheme and parallel Cuppen's algorithm have been given by Ipsen and Jessup on hypercubes [103] indicating the effectiveness of multisectioning on distributed memory multiprocessors for cases in which the eigenvalues are not pathologically clustered.

**6.1.2. Jacobi and Jacobi-like schemes.** An alternative to reduction to a condensed form is that of using one of the Jacobi schemes for obtaining all the eigenvalues or all the eigenvalues and eigenvectors. Work on such parallel procedures dates back to the Illiac IV distributed memory parallel computer, e.g., see [152]. Algorithms for handling the two-sided Jacobi scheme for the symmetric problem, which are presented in that work, exploit the fact that independent rotations can be applied simultaneously. Furthermore, several ordering schemes of these independent rotations are presented that minimize the number of orthogonal transformations (i.e., direct sum of rotations) within each sweep. Much more work has been done since on this parallel two-sided

Jacobi scheme for the symmetric eigenvalue problem. These have been motivated primarily by the emergence of systolic arrays, e.g., see Brent and Luk [18]. A most important byproduct of such investigation of parallel Jacobi schemes is a result due to Luk and Park [126], where they show the equivalence of various parallel Jacobi orderings to the classical sequential cyclic by row ordering for which Forsythe and Henrici [57] proved convergence of the method.

Also, in [152] a Jacobi-like algorithm for solving the nonsymmetric eigenvalue problem due to Eberlein [51], has been modified for parallel computations, primarily for the Illiac IV. More recent related parallel schemes, aimed at distributed memory multiprocessors as well, have been developed by Stewart [171] and Eberlein [52] for the Schur decomposition of nonsymmetric matrices.

Unlike the two-sided Jacobi scheme, for the symmetric eigenvalue problem, the one-sided Jacobi scheme due to Hestenes [94] requires only accessing of the columns of the matrix under consideration. This feature makes it more suitable for shared memory multiprocessors with hierarchical organization such as the Alliant FX/8. This procedure may be described as follows. Given a symmetric nonsingular matrix  $A$  of order  $n$  and columns  $a_i$ ,  $1 \leq i \leq n$ , obtain through an iterative process an orthogonal matrix  $V$  such that

$$AV = S$$

where  $S$  has orthogonal columns within a given tolerance. The orthogonal matrix  $V$  is constructed as the product of plane rotations in which each is chosen to orthogonalize a pair of columns,

$$(a_i, a_j) \begin{pmatrix} c & -s \\ s & c \end{pmatrix} = (\tilde{a}_i, \tilde{a}_j)$$

where  $i < j$ , so that  $\tilde{a}_i^T \tilde{a}_j = 0$  and  $\|\tilde{a}_i\|_2 > \|\tilde{a}_j\|_2$ . This is accomplished as follows, if

$$\begin{aligned} \beta &> 0 \\ c &= \sqrt{(\beta + \gamma)/2\gamma} \\ s &= \alpha/(2\gamma c) \end{aligned}$$

otherwise,

$$\begin{aligned} s &= \sqrt{(\gamma - \beta)/2\gamma} \\ c &= \alpha/(2\gamma s) \end{aligned}$$

Here,  $\alpha = 2a_i^T a_j$ ,  $\beta = \|a_i\|_2^2 - \|a_j\|_2^2$ , and  $\gamma = \sqrt{\alpha^2 + \beta^2}$ . Several schemes can be used to select the order of the plane rotations. Shown below is the pattern for one sweep for a matrix of order  $n = 8$  an annihilation scheme related to those recommended in [152],

$$\begin{array}{cccccccc} * & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ & * & 5 & 4 & 3 & 2 & 1 & 8 \\ & & * & 3 & 2 & 1 & 8 & 7 \\ & & & * & 1 & 8 & 7 & 6 \\ & & & & * & 7 & 6 & 5 \\ & & & & & * & 5 & 4 \\ & & & & & & * & 3 \\ & & & & & & & * \end{array}$$

where each sweep consists of  $n$  orthogonal transformations each being the direct sum of no more than  $\lfloor n/2 \rfloor$  independent plane rotations. An integer  $k = 8$ , for example, denotes that the column pairs (2,8), (3,7), (4,6) can be orthogonalized simultaneously by 3 independent rotations. After convergence of this iterative process, usually in a few sweeps, the matrix  $V$  yields a set of approximate eigenvectors from which the eigenvalues may be obtained via Rayleigh quotients. If the matrix  $A$  is positive-definite, however, then its eigenvalues are taken as the 2-norms of the columns of  $S$ . Note that if  $A$  is not known to be nonsingular, we treat the eigenvalue problem  $\hat{A}x = (\lambda + \alpha)x$ , where  $\hat{A} = A + \alpha I$ , with  $\alpha$  being the smallest number chosen such that  $\hat{A}$  is positive definite. On an Alliant FX/8, this Jacobi scheme is faster than algorithms that depend on tridiagonalization, with the same size residuals, for matrices of size less than 150 or for matrices that have few clusters of almost coincident eigenvalues.

Finally, a block generalization of the two-sided Jacobi scheme has been considered by Van Loan [190] and Bischof [13] for distributed memory multiprocessors. The convergence of cyclic block Jacobi methods has been discussed by Shroff and Schreiber [165].

**6.2. Singular-value problems.** Several algorithms have been developed for obtaining the singular-value decomposition on vector and parallel computers. The most robust of these schemes are those that rely first on reducing the matrix to the bidiagonal form, i.e., by using the sequential algorithm due to Golub and Reinsch [82]. The most obvious implementation of the reduction to the bidiagonal form on a parallel or vector computer follows the strategy suggested by Chan [26]. The matrix is first reduced to the upper triangular form via the block Householder reduction, suggested in the previous section, leading to the achievement of high performance. This is then followed by the chasing of zeros via rotation of rows and columns to yield a bidiagonal matrix. The application of the subsequent plane rotations has to proceed sequentially but some benefit due to vectorization can still be realized.

Once the bidiagonal matrix is obtained a generalization of Cuppen's algorithm (e.g., see [107]) may be used to obtain all the singular values and vectors. Similarly, a generalization of the multisectioning algorithm may be used to obtain selected singular values and vectors.

Luk has used the one-sided Jacobi scheme to obtain the singular-value decomposition on the Illiac IV [124] and block variations of Jacobi's method have been attempted by Bischof on IBM's LCAP system [13].

For tall and narrow matrices with certain distributions of clusters of singular values and/or extreme rank deficiencies, Jacobi schemes may also be used to efficiently obtain the singular-value decomposition of the upper triangular matrix resulting from the orthogonal factorization via block Householder transformations. The same one-sided Jacobi scheme discussed above has proved to be most effective on the hierarchical memory system of the Alliant FX/8. Such a procedure results in a performance that is superior to the best vectorized version of Eispack's or LINPACK routines which are based on the algorithm in [82]. Experiments showed that the block-Householder reduction and the one-sided Jacobi scheme combination is up to five times faster, on the Alliant FX/8, than the best BLAS2-version of LINPACK's routine for matrices of order  $16000 \times 128$  [12].

**7. Rapid elliptic solvers.** In this section, we review parallel schemes for rapid elliptic solvers. We start with the classical Matrix Decomposition (MD), and Block-Cyclic Reduction (BCR) schemes for separable elliptic P.D.E.'s on regular domains. This is followed by a Boundary Integral-based Domain Decomposition method for

handling the Laplace equation on irregular domains that consist of regular domains; examples of such domains are the right-angle or T-shapes.

Efficient direct methods for solving the finite-difference approximation of the Poisson equation on the unit square have been developed by Buneman [20], Hockney [96], [97], and Buzbee, Golub, and Nielson [22]. The most effective sequential algorithm combines the block cyclic reduction and Fourier analysis schemes. This is Hockney's *FACR(l)* algorithm [97]. Excellent reviews of these methods on sequential machines have been given by Swarztrauber [177] and Temperton [182], [183]. In [177] it is shown that the asymptotic operation count for *FACR(l)* on an  $n \times n$  grid is  $O(n^2 \log_2 \log_2 n)$ , and is achieved when the number  $l$  of the block cyclic reduction steps preceding Fourier analysis is taken approximately as  $(\log_2 \log_2 n)$ . Using only cyclic reduction, or Fourier analysis, to solve the problem on a sequential machine would require  $O(n^2 \log_2 n)$  arithmetic operations.

Buzbee [21] observed that Fourier analysis, or the matrix decomposition Poisson solver (MD-Poisson solver), is ideally suited for parallel computation. It consists of performing a set of independent sine transforms, and solving a set of independent tridiagonal systems. On a parallel computer consisting of  $n^2$  processors, with an arbitrarily powerful interconnection network, the MD-Poisson solver for the two-dimensional case requires  $O(\log_2 n)$  parallel arithmetic steps [160]. It can be shown, [142] and [173], that a perfect shuffle interconnection network is sufficient to keep the communication cost to a minimum. Ericksen [55] considered the implementation of *FACR(l)*, [97], and *CORF*, [22], on the ILLIAC IV; and Hockney [98] compared the performance of *FACR(l)* on the CRAY-1, Cyber-205, and the ICL-DAP.

**7.1. A domain decomposition MD-scheme.** We consider first the MD-algorithm for solving the 5-point finite difference approximation of the Poisson equation on the unit square with a uniform  $n \times n$  grid, where for the sake of illustration we consider only Dirichlet boundary conditions. The multiprocessor version algorithm presented below can be readily modified to accommodate Neumann and periodic boundary conditions.

Using natural ordering of the grid points, we obtain the well-known linear system of order  $n^2$ :

$$\begin{pmatrix} T & -I & & & \\ -I & T & -I & & \\ & & \ddots & \ddots & \ddots \\ & & & -I & T & -I \\ & & & -I & T \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-1} \\ u_n \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ v_{n-1} \\ v_n \end{pmatrix},$$

where  $T = [-1, 4, -1]$  is a tridiagonal matrix of order  $n$ .

This parallel MD-scheme consists of 3 stages [154]:

**Stage 1.** Each cluster  $j$ ,  $1 \leq j \leq 4$  (a four cluster Cedar is assumed), forms the subvectors  $f_{(j-1)q+1}, f_{(j-1)q+2}, \dots, f_{jq}$  of the right-hand side, where  $q = n/4$ . Next each cluster  $j$  obtains  $\hat{g}_j^T = (g_{(j-1)q+1}^T, \dots, g_{jq}^T)$ , where  $g_k = Qf_k$ , in which  $Q = [(2/[n+1])^{1/2} \sin(lm\pi/[n+1])]$ ,  $l, m = 1, 2, \dots, n$ , is the eigenvector matrix of  $T$ . This amounts to performing in each cluster  $q$  sine transforms each of length  $n$ . Now

we have the system

$$\begin{pmatrix} M & E & & \\ E^T & M & E & \\ & E^T & M & E \\ & & E^T & M \end{pmatrix} \begin{pmatrix} \hat{v}_1 \\ \hat{v}_2 \\ \hat{v}_3 \\ \hat{v}_4 \end{pmatrix} = \begin{pmatrix} \hat{g}_1 \\ \hat{g}_2 \\ \hat{g}_3 \\ \hat{g}_4 \end{pmatrix},$$

where each cluster memory contains one block row. Here,  $\hat{v}_j^T = (v_{(j-1)q+1}^T, \dots, v_{jq}^T)$  with  $v_k = Qu_k$ ,  $M = [-I_n, \Lambda, -I_n]$  is a block tridiagonal matrix of order  $qn$ , and

$$E = \begin{pmatrix} 0 & 0 \\ -I_n & 0 \end{pmatrix}.$$

This system, in turn, may be reduced to,

$$\begin{pmatrix} I_{qn} & F & & \\ G & I_{qn} & F & \\ & G & I_{qn} & F \\ & & G & I_{qn} \end{pmatrix} \begin{pmatrix} \hat{v}_1 \\ \hat{v}_2 \\ \hat{v}_3 \\ \hat{v}_4 \end{pmatrix} = \begin{pmatrix} \hat{h}_1 \\ \hat{h}_2 \\ \hat{h}_3 \\ \hat{h}_4 \end{pmatrix},$$

where  $\hat{h}_j^T = (h_{(j-1)q+1}^T, \dots, h_{jq}^T)$ ,  $F$  and  $G$  are given by:  $M\hat{h}_j = \hat{g}_j$ ,  $1 \leq j \leq 4$ ,  $MF = E$ , and  $MG = E^T$ . Observing that  $M$  consists of  $n$  independent tridiagonal matrices  $T_k = [-1, \lambda_k, -1]$  each of order  $q$ , where  $\lambda_k = 4 - 2\cos(k\pi/[n+1])$ ,  $k = 1, 2, \dots, n$ , the right-hand side of the above system is obtained by solving in each cluster  $j$  the  $n$  independent systems

$$T_k r_k = s_k,$$

for  $k = 1, 2, \dots, n$ , where  $\hat{e}_i^T s_k = e_k^T g_{(j-1)q+i}$ , and  $\hat{e}_i^T r_k = e_k^T h_{(j-1)q+i}$ , for  $i = 1, 2, \dots, q$ , and  $1 \leq j \leq 4$ . Here,  $\hat{e}_i$  and  $e_i$  are the  $i$ th columns of  $I_q$  and  $I_n$ , respectively.

The matrices  $F$  and  $G$  can be similarly obtained by solving, in each cluster  $j$ , the independent systems  $T_k c_k = \hat{e}_1$ , and  $T_k d_k = \hat{e}_q$ , for  $k = 1, 2, \dots, n$ . Since  $T_k$  is a Toeplitz matrix, however, we have  $c_k = J d_k$ , where  $J = [\hat{e}_q, \dots, \hat{e}_1]$ , see [111] for example. As a result, in order to obtain  $F$  and  $G$  we need only solve in each cluster the  $n$  systems  $T_k d_k = \hat{e}_q$ ,  $k = 1, 2, \dots, n$ . Hence,  $F$  and  $G$  are of the form,

$$F = \begin{pmatrix} \Gamma_q & 0 \\ \vdots & \vdots \\ \Gamma_1 & 0 \end{pmatrix},$$

and

$$G = \begin{pmatrix} 0 & \Gamma_1 \\ \vdots & \vdots \\ 0 & \Gamma_q \end{pmatrix},$$

where  $\Gamma_i = -\text{diag}(\gamma_i^{(1)}, \dots, \gamma_i^{(n)})$ , in which  $\gamma_i^{(k)} = \hat{e}_i^T c_k$ , for  $i = 1, 2, \dots, q$ , and  $k = 1, 2, \dots, n$ .

**Stage 2.** From the structure of (7.1) it is seen that the three pairs of  $n$  equations above and below each partition are completely decoupled from the rest of the  $n^2$

equations [161]. This reduced system, of order  $6n$ , consists of interlocking blocks of the form:

$$\begin{pmatrix} I_n & \Gamma_1 & 0 & 0 \\ \Gamma_1 & I_n & 0 & \Gamma_q \\ \Gamma_q & 0 & I_n & \Gamma_1 & 0 & 0 \\ & & \Gamma_1 & I_n & 0 & \Gamma_q \\ & & \Gamma_q & 0 & I_n & \Gamma_1 \\ & & & & \Gamma_1 & I_n \end{pmatrix}.$$

This system, in turn, comprises  $n$  independent pentadiagonal systems each of order 6, which can be solved in a very short time.

**Stage 3.** Now, that the subvectors  $v_{kq}$ ,  $v_{kq+1}$ ,  $k = 1, 2, 3$ , are available, each cluster  $j$  obtains

$$v_{(j-1)q+i} = h_{(j-1)q+i} - (\Gamma_i v_{(j-1)q} + \Gamma_{q-i+1} v_{jq+1})$$

for  $i = 2, 3, \dots, q-1$ , where  $v_0 = v_{4q+1} = 0$ . Finally, each cluster  $j$  retrieves the  $q$  subvectors  $u_{(j-1)q+i} = Qv_{(j-1)q+i}$ , for  $i = 1, 2, \dots, q$ , of the solution via  $q$  sine transforms, each of length  $n$ .

Note that one of the key computational kernels in this algorithm is the calculation of multiple sine transformations. In order to design an efficient version of this kernel it is necessary to perform an analysis of the influence of the memory hierarchy similar to that presented above for the block  $LU$  algorithm. Such an analysis is contained in [74].

**7.2. A modified block cyclic reduction.** The discretization of the separable elliptic equation

$$(16) \quad a(x) \frac{\partial^2 u}{\partial x^2} + b(x) \frac{\partial u}{\partial x} + c(x)u + \frac{\partial^2 u}{\partial y^2} = f(x, y)$$

with Dirichlet boundary conditions and a five-point stencil on a naturally ordered  $n \times m$  grid defined on a rectangular region leads to a system of the form  $\mathcal{A}u = f$ . In this case  $\mathcal{A}$  is the  $n$  block tridiagonal matrix  $\text{diag}[-I, A, -I]$ , where  $A, I$  are respectively tridiagonal and identity matrices of order  $m$ . Block cyclic reduction (BCR) dates back to the work of Hockney and was presented in [22] in its stabilized form due to Buneman. The work in [176], [178], [180] resulted in the development of FISHPAK, a package based on BCR for the solution of (16) and extensions thereof. BCR is a rapid elliptic solver (RES) having sequential computational complexity  $O(nm \log n)$ . Assuming that  $n = 2^k - 1$ , the idea of the method for reduction steps  $r = 1, \dots, k-1$  is to combine the current  $2^{k-r+1} - 1$  vectors into  $2^{k-r} - 1$  ones, and then solve a system of the form

$$p_{2^{r-1}}(A)X = Y$$

where  $Y \in \mathbb{R}^{m \times (2^{k-r-1})}$  and  $p_{2^{r-1}}(A)$  is a Chebyshev polynomial of degree  $2^{r-1}$  in  $A$ . Since its roots  $\lambda_i^{(r-1)}$  are known, it can be written in product form, where each factor is tridiagonal. Hence the system to be solved becomes

$$(17) \quad \prod_{i=1}^{2^{r-1}} (A - \lambda_i^{(r-1)} I) [x_1 | \dots | x_{2^{k-r-1}}] = [y_1 | \dots | y_{2^{k-r-1}}].$$

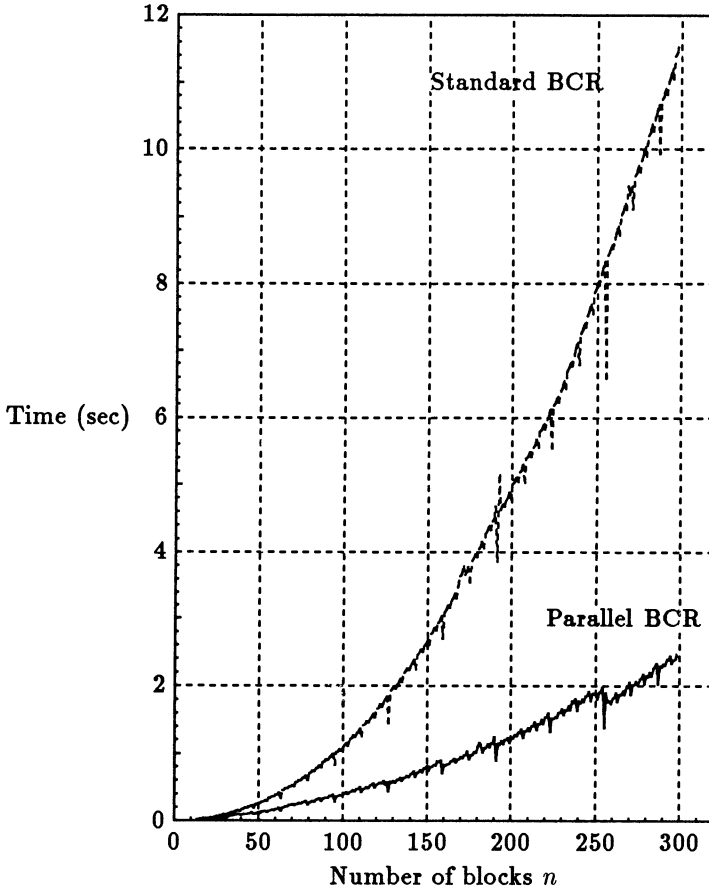


FIG. 19. *Parallel and standard BCR on  $n \times n$  grid on Alliant FX/8.*

Clearly as  $r$  increases, the effectiveness of a parallel or vector machine to handle (17) decreases rapidly.

A parallel version of BCR was recently discovered [70], [181]. In summary, the method is based in expressing the matrix rational function  $[p_{2^{r-1}}(A)]^{-1}$  as a partial fraction, i.e., as a linear combination of the  $2^{r-1}$  components  $(A - \lambda_i^{(r-1)}I)^{-1}$

$$(18) \quad [x_1 | \cdots | x_{2^{k-r}-1}] = \sum_{i=1}^{2^{r-1}} \alpha_i^{(r-1)} (A - \lambda_i^{(r-1)}I)^{-1} [y_1 | \cdots | y_{2^{k-r}-1}].$$

Coefficients  $\alpha_i^{(r-1)}$  are equal to  $1/(p'_{2^{r-1}}(\lambda_i^{(r-1)}))$  and can be derived analytically. Figure 19 shows the performance of the parallel and standard BCR on the Alliant FX/8.

For a discussion of parallel BCR on distributed memory machines see [73], [179]. Partial fraction decomposition can also be applied to the parallel solution of parabolic equations. See [71], [72] for details.

**7.3. Boundary integral domain decomposition.** A new method (BIDD) was recently proposed for the solution of Laplace's equation [68], [69]. The method is

characterized by the decoupling of the problem into independent subproblems on subdomains. An approximation  $\hat{u}$  to the solution  $u$  is sought as a finite linear combination of  $N$  fundamental solutions [128]  $\phi_j(z) = -\frac{1}{2\pi} \log |z - w_j|$  of  $\nabla^2 u = 0$ :

$$(19) \quad \hat{u}(z) = \sum_{j=1}^N \sigma_j \phi_j(z)$$

For a given set of  $N$  points  $w_j$  lying outside the domain,  $\sigma \in \mathbb{R}^N$  is computed to minimize  $\|g - G\sigma\|_\rho$  for some norm  $\rho$ .  $G \in \mathbb{R}^{\nu \times N}$  is the influence matrix consisting of fundamental solutions based at  $w_j$  for each boundary point.  $g \in \mathbb{R}^\nu$  consists of boundary values for  $u$ . Once  $\sigma$  has been computed, the solution at any  $\mu$  points on the domain is  $\hat{u} = H\sigma$ , with  $H \in \mathbb{R}^{\mu \times N}$  being the influence matrix for the  $\mu$  points. Choosing these  $\mu$  points to be subdomain boundary points, we can compute the solution by applying the elliptic solvers most suitable for each subdomain.

## REFERENCES

- [1] *IBM Engineering and Scientific Subroutine Library Guide and Reference*, IBM, 1986.
- [2] R. AGARWAL AND F. GUSTAVSON, *A parallel implementation of matrix multiplication and LU factorization on the IBM 3090*, in *Aspects of Computation on Asynchronous Parallel Processors*, M. H. Wright, ed., North-Holland, Amsterdam, 1989, pp. 217–221.
- [3] A. AGGARWAL, B. ALPERN, A. CHANDRA, AND M. SNIR, *A model for hierarchical memory*, in *Proc. 19th ACM Symp. Theory of Computing*, 1987, pp. 305–314.
- [4] H. AHMED, J. DELOSME, AND M. MORPH, *Highly concurrent computing structures for matrix arithmetic and signal processing*, *Computing*, 15 (1982), pp. 65–82.
- [5] J. ARMSTRONG, *Algorithm and performance notes for block LU factorization*, in *Proc. Intl. Conf. Par. Processing*, D. Bailey, ed., IEEE Computer Society Press, 1988, pp. 161–164.
- [6] Z. BAI AND J. DEMMEL, *On a block implementation of Hessenberg multishift QR iterations*, Tech. Rep., Courant Institute of Mathematical Sciences, New York University, New York, 1988.
- [7] D. BAILEY, *Extra high speed matrix multiplication on the CRAY-2*, *SIAM J. Sci. Statist. Comput.*, 9 (1988), pp. 603–607.
- [8] H. BERNSTEIN AND M. GOLDSTEIN, *Optimizing Givens' algorithm for multiprocessors*, *SIAM J. Sci. Statist. Comput.*, 9 (1988), pp. 601–602.
- [9] M. BERRY, K. GALLIVAN, W. HARROD, W. JALBY, S. LO, U. MEIER, B. PHILIPPE, AND A. SAMEH, *Parallel numerical algorithms on the Cedar system*, in *CONPAR 86*, Lecture Notes in Computer Science, W. Handler et al., eds., Springer-Verlag, Berlin, 1986.
- [10] M. BERRY AND R. PLEMMONS, *Algorithms and experiments for structural mechanics on high performance architectures*, *Comput. Methods Appl. Mech. Engrg.*, 64 (1987), pp. 487–507.
- [11] M. BERRY AND A. SAMEH, *Multiprocessor schemes for solving block tridiagonal linear systems*, *Intl. J. Supercomputer Appl.*, 2 (1988), pp. 37–57.
- [12] ———, *Parallel algorithms for the singular value and dense symmetric eigenvalue problems*, Tech. Rep. CSRD Rept. 761, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1988.
- [13] C. BISCHOF, *Computing the singular value decomposition on a distributed system of vector processors*, Tech. Rep. TR86-798, Dept. of Computer Science, Cornell University, Ithaca, NY, 1986.
- [14] ———, *A pipelined block QR decomposition algorithm*, in *Proc. of Third SIAM Conf. on Par. Processing for Scientific Computing*, G. Rodrigue, ed., Society for Industrial and Applied Mathematics, Philadelphia, 1988.
- [15] ———, *QR factorization algorithms for coarse-grained distributed systems*, Tech. Rep. TR 88-939, Dept. of Computer Science, Cornell University, Ithaca, NY, 1988.
- [16] C. BISCHOF AND C. VAN LOAN, *The WY representation for products of Householder matrices*, *SIAM J. Sci. Statist. Comput.*, 8 (1987), pp. s2–s13.



- [17] A. BOJANCZYK, R. BRENT, AND H. KUNG, *Numerically stable solution of dense systems of linear equations using mesh-connected processors*, SIAM J. Sci. Statist. Comput., 5 (1984), pp. 95–104.
- [18] R. BRENT AND F. LUK, *The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays*, SIAM J. Sci. Statist. Comput., 6 (1985), pp. 69–84.
- [19] O. BRONLUND AND T. L. JOHNSEN, *QR-factorization of partitioned matrices*, Comput. Methods Appl. Mech. Engrg., 3 (1974), pp. 153–172.
- [20] O. BUNEMAN, *A compact non-iterative Poisson solver*, Tech. Rep. Report 294, Stanford University Institute for Plasma Research, Stanford, CA, 1969.
- [21] B. BUZBEE, *A fast Poisson solver amenable to parallel computation*, IEEE Trans. Comput., C-22 (1973), pp. 793–796.
- [22] B. BUZBEE, G. GOLUB, AND C. NIELSON, *On direct methods for solving Poisson's equation*, SIAM J. Numer. Anal., 7 (1970), pp. 627–656.
- [23] D. CALAHAN, *Block-oriented local-memory-based linear equation solution on the CRAY-2: uniprocessor algorithms*, in Proc. Intl. Conf. Par. Processing, IEEE Computer Society Press, New York, 1986, pp. 375–378.
- [24] D. CALAHAN, W. AMES, AND E. SESEK, *A collection of equation solving codes for the CRAY-1*, Tech. Rep. SEL 133, University of Michigan, Ann Arbor, MI, 1979.
- [25] R. CHAMBERLAIN AND M. POWELL, *QR factorization for linear least squares on the hypercube*, Tech. Rep. CCS 86/10, Chr. Michelsen Institute, Bergen, Norway, 1986.
- [26] T. CHAN, *An improved algorithm for computing the singular value decomposition*, ACM Trans. Math. Software, 8 (1982), pp. 72–83.
- [27] S. CHEN, D. KUCK, AND A. SAMEH, *Practical parallel band triangular system solvers*, ACM Trans. Math. Software, 4 (1978), pp. 270–277.
- [28] E. CHU AND A. GEORGE, *Gaussian elimination with partial pivoting and load balancing on a multiprocessor*, Parallel Comput., 5 (1987), pp. 65–74.
- [29] ———, *A balanced submatrix merging algorithm for multiprocessor architectures*, Tech. Rep. CS-88-45, Faculty of Mathematics, University of Waterloo, Waterloo, Canada, 1988.
- [30] ———, *QR factorization of a dense matrix on a hypercube multiprocessor*, Tech. Rep. ORNL/TM-10691, Oak Ridge National Lab., Oak Ridge, TN, 1988.
- [31] ———, *Updating and downdating the inverse of a Cholesky factor on a hypercube multiprocessor*, Tech. Rep. CS-88-46, Dept. of Computer Science, University of Waterloo, Waterloo, Canada, 1988.
- [32] M. COSNARD, J. MULLER, AND Y. ROBERT, *Parallel QR decomposition of a rectangular matrix*, Numer. Math., 48 (1986), pp. 239–249.
- [33] J. CUPPEN, *A divide and conquer method for the symmetric tridiagonal eigenproblem*, Numer. Math., 36 (1981), pp. 177–195.
- [34] T. DEKKER AND W. HOFFMAN, *Rehabilitation of the Gauss-Jordan algorithm*, Tech. Rep. TR86-28, Dept. of Mathematics, University of Amsterdam, Amsterdam, 1986.
- [35] J. DEMMEL, J. DONGARRA, J. D. CROZ, A. GREENBAUM, S. HAMMARLING, AND D. SORENSEN, *Prospectus for the development of a linear algebra library for high-performance computers*, Tech. Rep. TM-97, Mathematics and Computer Science Div., Argonne National Laboratory, Argonne, IL, 1987.
- [36] G. DIETRICH, *A new formulation of the hypermatrix Householder-QR decomposition*, Comput. Methods Appl. Mech. Engrg., 9 (1976), pp. 273–280.
- [37] J. DONGARRA, *Workshop on the Level 3 BLAS*, Tech. Rep. TM-89, Mathematics and Computer Science Div., Argonne National Laboratory, Argonne, IL, 1987.
- [38] J. DONGARRA, J. BUNCH, C. MOLER, AND G. W. STEWART, *LINPACK User's Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1979.
- [39] J. DONGARRA, J. D. CROZ, I. DUFF, AND S. HAMMARLING, *A proposal for a set of Level 3 basic linear algebra subprograms*, Tech. Rep. TM-88, Mathematics and Computer Science Div., Argonne National Laboratory, Argonne, IL, 1987.
- [40] J. DONGARRA, J. D. CROZ, S. HAMMARLING, AND R. HANSON, *A proposal for an extended set of Fortran basic linear algebra subprograms*, Tech. Rep. TM-41, Mathematics and Computer Science Div., Argonne National Laboratory, Argonne, IL, 1984.
- [41] J. DONGARRA AND S. EISENSTAT, *Squeezing the most out of an algorithm in CRAY Fortran*, ACM Trans. Math. Software, 10 (1984), pp. 219–230.
- [42] J. DONGARRA, F. GUSTAVSON, AND A. KARP, *Implementing linear algebra algorithms for dense matrices on a vector pipeline machine*, SIAM Rev., 26 (1984), pp. 91–112.
- [43] J. DONGARRA, S. HAMMARLING, AND L. KAUFMAN, *Squeezing the most out of eigenvalue solvers on high-performance computers*, Tech. Rep. TM-46, Mathematics and Computer Science Div., Argonne National Laboratory, Argonne, IL, 1985.

- [44] J. DONGARRA, S. HAMMARLING, AND D. SORESENSEN, *Block reduction of matrices to condensed form for eigenvalue computations*, Tech. Rep. TM-99, Mathematics and Computer Science Div., Argonne National Laboratory, Argonne, IL, 1987.
- [45] J. DONGARRA AND T. HEWITT, *Implementing dense linear algebra algorithms using multitasking on the CRAY X-MP/4*, SIAM J. Sci. Statist. Comput., 7 (1986), pp. 347–350.
- [46] J. DONGARRA AND L. JOHNSON, *Solving banded systems on a parallel processor*, Parallel Comput., 5 (1987), pp. 219–246.
- [47] J. DONGARRA AND A. SAMEH, *On some parallel banded system solvers*, Parallel Comput., 1 (1984), pp. 223–235.
- [48] J. DONGARRA, A. SAMEH, AND D. SORESENSEN, *Implementation of some concurrent algorithms for matrix factorization*, Parallel Comput., 3 (1986), pp. 25–34.
- [49] J. DONGARRA AND D. SORESENSEN, *A fully parallel algorithm for the symmetric eigenvalue problem*, SIAM J. Sci. Statist. Comput., 8 (1987), pp. s139–s154.
- [50] J. DU CROZ, S. NUGENT, J. REID, AND D. TAYLOR, *Solving large full sets of linear equations in a paged virtual store*, ACM Trans. Math. Software, 7 (1981), pp. 527–536.
- [51] P. EBERLEIN, *A Jacobi-like method for the automatic computation of eigenvalues and eigenvectors of an arbitrary matrix*, J. Soc. Indust. Appl. Math., 10 (1962), pp. 74–88.
- [52] ———, *On the Schur decomposition of a matrix for parallel computation*, IEEE Trans. Comput., C-36 (1987), pp. 167–174.
- [53] S. EISENSTAT, M. HEATH, C. HENKEL, AND C. ROMINE, *Modified cyclic algorithms for solving triangular systems on distributed memory multiprocessors*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 589–600.
- [54] L. ELDEN, *A parallel QR decomposition algorithm*, Tech. Rep. Lith-MAT-R-1988-02, Linköping University, Linköping, Sweden, 1987.
- [55] J. ERICKSEN, *Iterative and direct methods for solving Poisson's equation and their adaptability to Illiac IV*, Tech. Rep. CAC Doc. 60, Center for Advanced Computations, University of Illinois, Urbana, IL, 1972.
- [56] K. FONG AND T. JORDAN, *Some linear algebra algorithms and their performance on CRAY-1*, Tech. Rep. LA-6774, Los Alamos National Laboratory, Los Alamos, NM, 1977.
- [57] G. FORSYTHE AND P. HENRICI, *The cyclic Jacobi method for computing the principal values of a complex matrix*, Trans. Amer. Math. Soc., 94 (1960), pp. 1–23.
- [58] G. FORSYTHE, M. MALCOLM, AND C. MOLER, *Computer Methods for Mathematical Computations*, Prentice-Hall, Englewood Cliffs, NJ, 1977.
- [59] G. FOX, *Domain decomposition in distributed and shared memory environments*, in Lecture Notes in Comput. Sci. 297: Proc. 1987 Intl. Conf. Supercomputing, Springer-Verlag, Berlin, 1987, pp. 1042–1073.
- [60] G. FOX, M. JOHNSON, G. LYZENGA, S. OTTO, J. SALMON, AND D. WALKER, *Solving Problems on Concurrent Processors*, Vol. 1, Prentice-Hall, Englewood Cliffs, NJ, 1988.
- [61] G. FOX, S. OTTO, AND A. HEY, *Matrix algorithms on a hypercube I: matrix multiplication*, Parallel Comput., 4 (1987), pp. 17–31.
- [62] L. FOX, E. GOODWIN, J. MICHEL, F. OLVER, AND J. WILKINSON, *Modern Computing Methods*, First edition, Philosophical Library, New York, 1961.
- [63] K. GALLIVAN, D. GANNON, AND W. JALBY, *Strategies for cache and local memory management by global program transformation*, J. Parallel Dist. Computing, 5 (1988), pp. 587–616.
- [64] K. GALLIVAN, D. GANNON, W. JALBY, A. MALONY, AND H. WIJSHOFF, *Behavioral characterization of multiprocessor memory systems*, in Proc. 1989 ACM SIGMETRICS Conf. on Measuring and Modeling Computer Systems, ACM Press, New York, 1989, pp. 79–89.
- [65] K. GALLIVAN, W. JALBY, A. MALONY, AND H. WIJSHOFF, *Performance prediction of loop constructs on multiprocessor hierarchical memory systems*, in Proc. 1989 Intl. Conf. Supercomputing, ACM Press, New York, 1989, pp. 433–442.
- [66] K. GALLIVAN, W. JALBY, AND U. MEIER, *The use of BLAS3 in linear algebra on a parallel processor with a hierarchical memory*, SIAM J. Sci. Statist. Comput., 8 (1987), pp. 1079–1084.
- [67] K. GALLIVAN, W. JALBY, U. MEIER, AND A. SAMEH, *Impact of hierarchical memory systems on linear algebra algorithm design*, Intl. J. Supercomputer Appl., 2 (1988), pp. 12–48. Presented at the Level 3 BLAS Workshop, Argonne National Laboratory, January 1987.
- [68] E. GALLOPOULOS AND D. LEE, *Boundary integral domain decomposition on hierarchical memory multiprocessor*, in Proc. 1988 Intl. Conf. Supercomputing, ACM Press, New York, 1988, pp. 488–499.
- [69] ———, *Fast Laplace solver by boundary integral-based domain decomposition*, in Proc. of

- Third SIAM Conf. on Par. Processing for Scientific Computing, G. Rodrigue, ed., Society for Industrial and Applied Mathematics, Philadelphia, 1988.
- [70] E. GALLOPOULOS AND Y. SAAD, *Parallel block cyclic reduction algorithm for the fast solution of elliptic equations*, *Parallel Comput.*, 10 (1989), pp. 143–160. Also presented at 1987 Int'l. Conf. on Supercomputing, Athens, Greece.
  - [71] ———, *Efficient parallel solution of parabolic equations: explicit methods*, Tech. Rep., Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, June 1989. To be presented at the Fourth SIAM Conf. Parallel Processing for Scientific Computing.
  - [72] ———, *Efficient parallel solution of parabolic equations: implicit methods*, Tech. Rep., Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, June 1989. To be presented at the Fourth SIAM Conf. Parallel Processing for Scientific Computing.
  - [73] ———, *Some fast elliptic solvers for parallel architectures and their complexities*, *Int'l. J. High Speed Comput.*, 1 (May 1989), pp. 113–141.
  - [74] D. GANNON AND W. JALBY, *The influence of memory hierarchy on algorithm organization: programming FFTs on a vector multiprocessor*, in *The Characteristics of Parallel Algorithms*, L. Jamieson, D. Gannon, and R. Douglass, eds., MIT Press, Cambridge, 1987, pp. 277–301.
  - [75] D. GANNON, W. JALBY, AND K. GALLIVAN, *On the problem of optimizing data transfers for complex memory systems*, in *Proc. 1988 Intl. Conf. Supercomputing*, ACM Press, New York, 1988, pp. 238–253.
  - [76] D. GANNON AND J. VAN ROSENDALE, *On the impact of communication complexity on the design of parallel numerical algorithms*, *IEEE Trans. Comput.*, C-33 (1984), pp. 1180–1195.
  - [77] A. GEIST AND M. HEATH, *Parallel Cholesky factorization on a hypercube multiprocessor*, Tech. Rep. ORNL-6190, Oak Ridge National Laboratory, Oak Ridge, TN, 1985.
  - [78] ———, *Matrix factorization on a hypercube*, in *Hypercube Multiprocessors 1986*, M. T. Heath, ed., Society for Industrial and Applied Mathematics, Philadelphia, 1986, pp. 161–180.
  - [79] A. GEIST AND C. ROMINE, *LU factorization algorithms on distributed memory multiprocessor architectures*, *SIAM J. Sci. Statist. Comput.*, 9 (1988), pp. 639–649.
  - [80] W. GENTLEMAN AND H. T. KUNG, *Matrix triangularization by systolic arrays*, in *Proc. SPIE 298, Real Time Signal Processing*, San Diego, CA, 1981, pp. 19–26.
  - [81] G. GOLUB, R. PLEMMONS, AND A. SAMEH, *Parallel block schemes for large-scale least squares computations*, in *High Speed Computing, Scientific Applications and Algorithm Design*, R. Wilhelmson, ed., University of Illinois Press, Urbana, IL, 1988, pp. 180–195.
  - [82] G. GOLUB AND C. REINSCH, *Singular value decomposition and least squares solutions*, *Numer. Math.*, 14 (1970), pp. 403–420.
  - [83] G. GOLUB AND C. VAN LOAN, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 1983.
  - [84] F. GUSTAVSON, private communication.
  - [85] W. HARROD, *Programming with the BLAS*, in *The Characteristics of Parallel Algorithms*, L. Jamieson, D. Gannon, and R. Douglass, eds., MIT Press, Cambridge, 1987, pp. 253–276.
  - [86] ———, *A block scheme for reduction to condensed form*, Tech. Rep. CSR D Rept. 696, Center for Supercomputing Research and Development, University of Illinois, Argonne, IL, 1988.
  - [87] P. HARTEN, private communication.
  - [88] M. HEATH, *Parallel Cholesky factorization in message passing multiprocessor environments*, Tech. Rep. ORNL-6150, Oak Ridge National Lab., Oak Ridge, TN, 1985.
  - [89] M. HEATH AND C. ROMINE, *Parallel solution of triangular systems on distributed memory multiprocessors*, *SIAM J. Sci. Statist. Comput.*, 9 (1988), pp. 558–588.
  - [90] D. HELLER, *Some aspects of the cyclic reduction algorithm for block tridiagonal linear systems*, *SIAM J. Numer. Anal.*, 13 (1976), pp. 484–496.
  - [91] ———, *A survey of parallel algorithms for numerical linear algebra*, *SIAM Rev.*, 20 (1978), pp. 740–777.
  - [92] C. HENKEL, M. HEATH, AND R. PLEMMONS, *Cholesky downdating on a hypercube*, in *Hypercube Multiprocessors 1988*, G. Fox, ed., ACM Press, New York, 1988, pp. 1592–1598.
  - [93] C. HENKEL AND R. PLEMMONS, *Recursive least squares on a hypercube multiprocessor using the covariance factorization*, Tech. Rep., Dept. Computer Science, North Carolina State University, Raleigh, NC, 1988. *SIAM J. Sci. Statist. Comput.*, 11 (1990), to appear.
  - [94] M. HESTENES, *Inversion of matrices by biorthogonalization and related results*, *J. Soc. In-*

- dust. Appl. Math., 6 (1958), pp. 51–90.
- [95] N. HIGHAM, *Exploiting fast matrix multiplication with the Level 3 BLAS*, Tech. Rep. TR 89-984, Dept. of Computer Science, Cornell University, Ithaca, NY, 1989.
  - [96] R. HOCKNEY, *A fast direct solution of Poisson's equation using Fourier analysis*, JACM, 12 (1965), pp. 95–113.
  - [97] ———, *The potential calculation and some applications*, Methods Comput. Phys., 9 (1970), pp. 135–211.
  - [98] ———, *Optimizing the FACR(l) Poisson solver on parallel computers*, in Proc. Intl. Conf. Par. Processing, IEEE Computer Society Press, New York, 1982.
  - [99] ———, *Problem related performance parameters for supercomputers*, in Performance Evaluation of Supercomputers, J. Martin, ed., Elsevier Science Publishers B.V., Amsterdam, 1988, pp. 215–235.
  - [100] R. HOCKNEY AND C. JESSHOPE, *Parallel Computers*, First edition, Adam Hilger, Bristol, 1981.
  - [101] J. HONG AND H. T. KUNG, *I/O complexity: the red-blue pebble game*, in Proc. 13th ACM Symp. Theory of Computing, 1981, pp. 326–333.
  - [102] H. HUANG, *A parallel algorithm for symmetric tridiagonal eigenvalue problems*, Tech. Rep. CAC Doc. 109, Center for Advanced Computation, University of Illinois, Urbana, IL, 1974.
  - [103] I. IPSEN AND E. JESSUP, *Solving the symmetric tridiagonal eigenvalue problem on the hypercube*, Tech. Rep. RR-548, Dept. of Computer Science, Yale University, New Haven, CT, 1987.
  - [104] I. IPSEN, Y. SAAD, AND M. SCHULTZ, *Complexity of dense linear system solution on a multiprocessor ring*, Linear Algebra Appl., 77 (1986), pp. 205–239.
  - [105] W. JALBY AND U. MEIER, *Optimizing matrix operations on a parallel multiprocessor with a hierarchical memory system*, in Proc. Intl. Conf. Par. Processing, IEEE Computer Society Press, New York, 1986, pp. 429–432.
  - [106] W. JALBY AND B. PHILIPPE, *Loss of orthogonality in a Gram-Schmidt process*, Tech. Rep., IRISA, Rennes, France, 1987.
  - [107] E. JESSUP AND D. SORENSEN, *A parallel algorithm for computing the singular value decomposition of a matrix*, Tech. Rep. TM-102, Mathematics and Computer Science Div., Argonne National Laboratory, Argonne, IL, 1987.
  - [108] L. JOHNSON, *Solving narrow banded systems on ensemble architectures*, ACM Trans. Math. Software, 11 (1985), pp. 271–288.
  - [109] ———, *Solving tridiagonal systems on ensemble architectures*, SIAM J. Sci. Statist. Comput., 8 (1987), pp. 354–392.
  - [110] L. JOHNSON AND C. T. HO, *Algorithms for multiplying matrices of arbitrary shapes using shared memory primitives on Boolean cubes*, Tech. Rep. YALEU/DCS/TR-569, Dept. Computer Science, Yale University, New Haven, 1987.
  - [111] T. KAILATH, A. VIEIRA, AND M. MORF, *Inversion of Toeplitz operators, innovations and orthogonal polynomials*, SIAM Rev., 20 (1978), pp. 106–119.
  - [112] C. KATHOLI AND B. SUTER, *QR factorization of a rectangular matrix*, Tech. Rep. TR88-07, University of Alabama at Birmingham, Birmingham, AL, 1988.
  - [113] S. KIM, D. AGRAWAL, AND R. PLEMMONS, *Recursive least squares filtering for signal processing on distributed memory multiprocessors*, Tech. Rep., Dept. of Computer Science, North Carolina State University, Raleigh, NC, 1988. Inter. J. Parallel Proc. (1990), to appear.
  - [114] M. KNOWLES, B. OKAWA, Y. MURAOKA, AND R. WILHELMSON, *Matrix operations on ILLIAC IV*, Tech. Rep. ILLIAC IV Doc. 118, Dept. of Computer Science, University of Illinois, Urbana, IL, 1967.
  - [115] D. KUCK, *ILLIAC IV software and application programming*, IEEE Trans. Comput., C-17 (1968), pp. 758–770.
  - [116] ———, *The Structure of Computers and Computations*, Vol. 1, John Wiley, New York, 1978.
  - [117] D. KUCK, E. DAVIDSON, D. LAWRIE, AND A. SAMEH, *Parallel supercomputing today and the Cedar approach*, Science, 231 (1986), pp. 967–974.
  - [118] D. KUCK AND A. SAMEH, *Parallel computations of eigenvalues of real matrices*, in Proc. IFIP Congress 1971, North-Holland, Amsterdam, 1972, pp. 1266–1272.
  - [119] D. LAWRIE, *Access and alignment of data in an array processor*, IEEE Trans. Comput., C-24 (1975), pp. 1145–1155.
  - [120] D. LAWRIE AND A. SAMEH, *The computations and communication complexity of a parallel banded system solver*, ACM Trans Math. Software, 10 (1984), pp. 185–195.
  - [121] C. LAWSON, R. HANSON, D. KINCAID, AND F. KROGH, *Basic linear algebra subprograms*

- for Fortran use, ACM Trans. Math. Software, 5 (1979), pp. 308–323.
- [122] G. LI AND T. COLEMAN, *A parallel triangular solver on a distributed memory multiprocessor*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 485–502.
  - [123] S. LO, B. PHILIPPE, AND A. SAMEH, *A multiprocessor algorithm for the symmetric tridiagonal eigenvalue problem*, SIAM J. Sci. Statist. Comput., 8 (1987), pp. s155–s165.
  - [124] F. LUK, *Computing the singular value decomposition on the Iliac IV*, ACM Trans. Math. Software, 6 (1980), pp. 524–539.
  - [125] ———, *A rotation method for computing the QR decomposition*, SIAM J. Sci. Statist. Comput., 7 (1987), pp. 452–549.
  - [126] F. LUK AND H. PARK, *A proof of convergence for two parallel Jacobi SVD algorithms*, 1987. IEEE Trans. Comput., to appear.
  - [127] N. MADSEN, G. RODRIGUE, AND J. KARUSH, *Matrix multiplication by diagonals on a vector/parallel processor*, Inform. Process. Lett., 5 (1976), pp. 41–45.
  - [128] R. MATHON AND R. JOHNSTON, *The approximate solution of elliptic boundary-value problems by fundamental solutions*, SIAM J. Numer. Anal., 14 (1977), pp. 638–650.
  - [129] O. MCBRYAN AND E. VAN DE VELDE, *Hypercube algorithms and implementations*, SIAM J. Sci. Statist. Comput., 8 (1987), pp. s227–s287.
  - [130] J. MCCOMB AND S. SCHMIDT, *Engineering and scientific library for the IBM 3090 vector facility*, IBM Systems Journal, 27 (1988), pp. 404–415.
  - [131] A. MCKELLAR AND E. COFFMAN JR., *Organizing matrices and matrix operations for paged memory systems*, Comm. ACM, 12 (1969), pp. 153–165.
  - [132] U. MEIER, *A parallel partition method for solving banded systems of linear equations*, Parallel Comput., 2 (1985), pp. 33–43.
  - [133] W. MIRANKER, *A survey of parallelism in numerical analysis*, SIAM Rev., 13 (1971), pp. 524–547.
  - [134] J. MODI AND M. CLARKE, *An alternative Givens ordering*, Numer. Math., 43 (1984), pp. 83–90.
  - [135] C. MOLER, *Matrix computations on distributed memory multiprocessors*, in Hypercube Multiprocessors 1986, M. T. Heath, ed., Society for Industrial and Applied Mathematics, Philadelphia, 1986, pp. 181–195.
  - [136] C. MOLER AND G. W. STEWART, *An algorithm for generalized matrix eigenvalue problems*, SIAM J. Numer. Anal., 10 (1973), pp. 241–256.
  - [137] J. ORTEGA, *Introduction to Parallel and Vector Solution of Linear Systems*, Plenum, New York, 1988.
  - [138] J. ORTEGA AND R. VOIGT, *Solution of partial differential equations on vector and parallel computers*, SIAM Rev., 27 (1985), pp. 149–240.
  - [139] J. ORTEGA, R. VOIGT, AND C. ROMINE, *A bibliography on parallel and vector numerical algorithms*, Tech. Rep. ORNL/TM-10998, Oak Ridge National Laboratory, Oak Ridge, TN, 1989.
  - [140] C. PAN AND R. PLEMMONS, *Least squares modifications with inverse factorizations: parallel implications*, Tech. Rep., Dept. of Computer Science, North Carolina State University, Raleigh, NC, 1987. Comput. Appl. Math., to appear.
  - [141] B. PARLETT, *The Symmetric Eigenvalue Problem*, Prentice-Hall, Englewood Cliffs, NJ, 1980.
  - [142] M. PEASE, *An adaption of the fast Fourier transform for parallel processing*, JACM, 15 (1968), pp. 252–264.
  - [143] G. PETERS AND J. WILKINSON, *On the stability of Gauss-Jordan elimination with pivoting*, Comm. ACM, 18 (1975), pp. 20–24.
  - [144] R. PLEMMONS, *A parallel block scheme applied to computations in structural analysis*, SIAM J. Algebraic Discrete Methods, 7 (1986), pp. 337–347.
  - [145] R. PLEMMONS AND R. WHITE, *Substructuring methods for computing the nullspace of equilibrium matrices*, Tech. Rep., Center for Research in Sci. Comp., North Carolina State University, Raleigh, NC, 1988.
  - [146] R. PLEMMONS AND S. WRIGHT, *An efficient parallel scheme for minimizing a sum of Euclidean norms*, Linear Algebra Appl., 121 (1989), pp. 71–85.
  - [147] A. POTHEEN AND P. RAGHAVAN, *Orthogonal factorization on a distributed memory multiprocessor*, Tech. Rep. CS-87-24, Pennsylvania State University, Computer Science Dept., University Park, PA, 1987.
  - [148] A. POTHEEN, J. SOMESH, AND U. VEMULAPATI, *Orthogonal factorization on a distributed memory multiprocessor*, in Hypercube Multiprocessors 1987, M. T. Heath, ed., Society for Industrial and Applied Mathematics, Philadelphia, 1987, pp. 587–596.
  - [149] G. RADICATI, Y. ROBERT, AND P. SGUAZZERO, *Dense linear systems Fortran solvers on the IBM 3090 vector multiprocessor*, Parallel Comput., 8 (1988), pp. 377–384.

- [150] C. ROMINE, *The parallel solution of triangular systems on a hypercube*, in Hypercube Multiprocessors 1987, M. T. Heath, ed., Society for Industrial and Applied Mathematics, Philadelphia, 1987, pp. 552–559.
- [151] C. ROMINE AND J. ORTEGA, *Parallel solution of triangular systems of equations*, Parallel Comput., 6 (1988), pp. 109–114.
- [152] A. SAMEH, *On Jacobi and Jacobi-like algorithms for a parallel computer*, Math. Comp., 25 (1971), pp. 579–590.
- [153] ———, *Numerical parallel algorithms – a survey*, in High Speed Computer and Algorithm Organization, D. Kuck, D. Lawrie, and A. Sameh, eds., Academic Press, New York, 1977, pp. 207–228.
- [154] ———, *A fast Poisson solver for multiprocessors*, in Elliptic Problem Solvers II, G. Birkhoff and A. Schoenstadt, eds., Academic Press, New York, 1984, pp. 175–186.
- [155] ———, *On two numerical algorithms for multiprocessors*, in High Speed Computation, J. Kowalik, ed., Springer-Verlag, Berlin, 1984, pp. 311–328.
- [156] ———, *Numerical algorithms on the Cedar system*, presented at Second SIAM Conference on Parallel Processing, 1985.
- [157] ———, *On some parallel algorithms on a ring of processors*, Comput. Phys. Comm., 37 (1985), pp. 159–166.
- [158] ———, *Solving the linear least squares problem on a linear array of processors*, in Algorithmically-Specialized Parallel Computers, Academic Press, West Lafayette, IN, 1985, pp. 191–200. Proc. 1982 Purdue Univ. Workshop.
- [159] A. SAMEH AND R. BRENT, *Solving triangular systems on a parallel computer*, SIAM J. Numer. Anal., 14 (1977), pp. 1101–1113.
- [160] A. SAMEH, S. CHEN, AND D. KUCK, *Parallel Poisson and biharmonic solvers*, Computing, 17 (1976), pp. 219–230.
- [161] A. SAMEH AND D. KUCK, *On stable parallel linear systems solvers*, JACM, 25 (1978), pp. 81–91.
- [162] H. SAMUKAWA, *Programming style on the IBM vector facility considering both performance and flexibility*, IBM Systems Journal, 27 (1988), pp. 453–474.
- [163] R. SCHREIBER AND B. PARLETT, *Block reflectors: theory and computation*, SIAM J. Numer. Anal., 25 (1988), pp. 189–205.
- [164] R. SCHREIBER AND C. VAN LOAN, *A storage-efficient WY representation for products of Householder transformations*, SIAM J. Sci. Statist. Comput., 10 (1989), pp. 53–57.
- [165] G. SHROFF AND R. SCHREIBER, *On the convergence of the cyclic Jacobi method for parallel block orderings*, Tech. Rep. 88-11, Dept. of Computer Science, Rensselaer Polytechnic Institute, Troy, NY, 1988.
- [166] H. SIMON, *Bisection is not optimal on vector processors*, SIAM J. Sci. Statist. Comput., 10 (1989), pp. 205–209.
- [167] B. SMITH, J. BOYLE, Y. IKEBE, V. KLEMA, AND C. MOLER, *Matrix Eigensystem Routines: EISPACK Guide*, Second edition, Springer-Verlag, Berlin, 1976.
- [168] D. SORENSEN, *Analysis of pairwise pivoting in Gaussian elimination*, IEEE Trans. Comput., C-34 (1985), pp. 274–278.
- [169] J. STERN, *A fast Gaussian elimination scheme and automated roundoff error analysis for SIME machines*, Master's thesis, Dept. of Computer Science, University of Illinois, Urbana, IL, 1979.
- [170] G. W. STEWART, *Introduction to Matrix Computations*, Academic Press, New York, 1973.
- [171] ———, *A Jacobi-like algorithm for computing the Schur decomposition of a nonhermitian matrix*, SIAM J. Sci. Statist. Comput., 6 (1985), pp. 853–864.
- [172] ———, *A parallel implementation of the QR-algorithm*, Parallel Comput., 5 (1987), pp. 187–196.
- [173] H. STONE, *Parallel processing with the perfect shuffle*, IEEE Trans. Comput., C-20 (1971), pp. 153–161.
- [174] H. S. STONE, *High-Performance Computer Architecture*, First edition, Addison-Wesley, Reading, 1987.
- [175] V. STRASSEN, *Gaussian elimination is not optimal*, Numer. Math., 13 (1969), pp. 354–356.
- [176] P. SWARZTRAUBER, *A direct method for the discrete solution of separable elliptic equations*, SIAM J. Numer. Anal., 11 (1974), pp. 1136–1150.
- [177] ———, *The methods of cyclic reduction, Fourier analysis and the FACR algorithm for the discrete solution of Poisson's equation on a rectangle*, SIAM Rev., 19 (1977), pp. 490–501.
- [178] P. SWARZTRAUBER AND R. SWEET, *Algorithm 541: efficient Fortran subprograms for the solution of separable elliptic partial differential equations*, ACM Trans. Math. Software, 5 (1979), pp. 352–364.

- [179] ———, *Vector and parallel methods for the direct solution of Poisson's equation*, J. Comput. Appl. Math., 27 (1989), pp. 241–263.
- [180] R. SWEET, *A cyclic reduction algorithm for solving block tridiagonal systems of arbitrary dimension*, SIAM J. Numer. Anal., 14 (1977), pp. 707–720.
- [181] ———, *A parallel and vector variant of the cyclic reduction algorithm*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 761–765.
- [182] C. TEMPERTON, *Direct methods for the solution of the discrete Poisson equation: some comparisons*, J. Comput. Phys., 31 (1979), pp. 1–20.
- [183] ———, *On the FACR(l) algorithm for the discrete Poisson equation*, J. Comput. Phys., 34 (1980), pp. 314–329.
- [184] L. TREFETHAN AND R. SCHREIBER, *Average-case stability of Gaussian elimination*, Tech. Rep. RUU-CS-84-7, Dept. of Mathematics, MIT, Cambridge, MA, 1988.
- [185] K. TRIVEDI, *Prepaging and applications to structured array problems*, Tech. Rep. UIUCDCS-R-74-662, Dept. of Computer Science, University of Illinois, Urbana, IL, 1974.
- [186] ———, *On the paging performance of array algorithms*, IEEE Trans. Comput., C-26 (1977), pp. 938–947.
- [187] N. TSAO, *On the accuracy of solving triangular systems in parallel*, Tech. Rep. ICOMP-88-19, NASA Lewis Research Center, Cleveland, OH, 1988.
- [188] H. VAN DER VORST, *Analysis of a parallel solution method for tridiagonal linear systems*, Parallel Comput., 5 (1987), pp. 303–311.
- [189] H. VAN DER VORST AND K. DEKKER, *Vectorization of linear recurrence relations*, SIAM J. Sci. Statist. Comput., 10 (1989), pp. 27–35.
- [190] C. VAN LOAN, *The block Jacobi method for computing the singular value decomposition*, Tech. Rep. TR85-680, Dept. of Computer Science, Cornell University, Ithaca, NY, 1985.
- [191] R. VOIGT, *The influence of vector computer architecture on numerical algorithms*, in High Speed Computer and Algorithm Organization, D. Kuck, D. Lawrie, and A. Sameh, eds., Academic Press, New York, 1977, pp. 229–244.
- [192] H. WANG, *A parallel method for tridiagonal equations*, ACM Trans. Math. Software, 7 (1981), pp. 170–183.
- [193] H. A. G. WIJSHOFF, *Data Organization in Parallel Computers*, Kluwer, Boston, 1989.
- [194] J. WILKINSON, *The Algebraic Eigenvalue Problem*, Clarendon Press, Oxford, 1965.
- [195] J. WILKINSON AND C. REINSCH, *Handbook for Automatic Computation: Linear Algebra*, Vol. 2, Springer-Verlag, Berlin, 1971.
- [196] C. Q. ZHU AND P. C. YEW, *A scheme to enforce data dependences on large multiprocessor systems*, IEEE Trans. Soft. Engrg., SE-13 (1987), pp. 726–739.