most immediately useful for traditional time-sharing, they promise to provide high performance for individual applications as well. But it is not yet clear which of a variety of hardware and software structures and systems will have sufficient applicability and performance to become widespread.

REFERENCES AND NOTES

1. C. G. Bell, *Science* **228**, 462 (1985).
2. J. T. Deutch and A. R. Newton, paper presented at the 21st Design Automation Conference, Miami Beach, FL, June 1984.
3. G. C. Fox and S. W. Otto, *Phys. Today* **37**, 50 (May 1984).
4. Session on Commercial Multiprocessors, 12th Symposium on Computer Architecture, Boston, June 1985.
5. S. Frank, *Electronics* **57**, 164 (12 January 1984).
6. J. Goodman, paper presented at the Tenth Symposium on Computer Architecture, Trondheim, Norway, June 1983.
7. M. Dubois and F. A. Briggs, *IEEE Trans. Comput.* **C-31**, 1083 (1982).
8. R. H. Katz *et al.*, paper presented at the 12th Symposium on Computer Architecture, Boston, June 1985.
9. S. Reinhardt, paper presented at the Tenth Symposium on Operating Systems Principles, Orcas Island, WA, December 1985.
10. D. L. Kuck *et al.*, paper presented at the Eighth Symposium on Principles of Programming Languages, Williamsburg, VA, January 1981.
11. C. L. Seitz, *IEEE Trans. Comput.* **C-33**, 1247 (1984).
12. J. T. Schwartz, *ACM TOPLAS* **2**, 484 (1980).
13. W. D. Hillis, *The Connection Machine* (MIT Press, Cambridge, 1985).
14. R. P. Gabriel, *Science* **231**, 975 (1986).
15. C. L. Seitz, *Commun. ACM* **28**, 22 (1985).

Parallel Supercomputing Today and the Cedar Approach

DAVID J. KUCK, EDWARD S. DAVIDSON, DUNCAN H. LAWRIE, AHMED H. SAMEH

More and more scientists and engineers are becoming interested in using supercomputers. Earlier barriers to using these machines are disappearing as software for their use improves. Meanwhile, new parallel supercomputer architectures are emerging that may provide rapid growth in performance. These systems may use a large number of processors with an intricate memory system that is both parallel and hierarchical; they will require even more advanced software. Compilers that restructure user programs to exploit the machine organization seem to be essential. A wide range of algorithms and applications is being developed in an effort to provide high parallel processing performance in many fields. The Cedar supercomputer, presently operating with eight processors in parallel, uses advanced system and applications software developed at the University of Illinois during the past 12 years. This software should allow the number of processors in Cedar to be doubled annually, providing rapid performance advances in the next decade.

THE HISTORY OF PERFORMANCE GAINS IN SUPERCOMPUTERS is remarkable, yet the rate of improvement over this history has steadily declined. In a 5-year period in the 1940's, computer speeds increased by 10^3 as technology shifted from relays to vacuum tubes. ENIAC had a peak rate of about 10^3 floating-point operations per second (flops) in 1946. In the mid-1980's, after changes to transistor and then integrated circuit technology and accompanying architectural enhancements, systems are reaching peak rates of 10^9 flops for an improvement factor of 10^6 in 40 years, or an average factor of 10 every 7 years. Clock speed is the rate at which basic computer operations are performed. The Cray 2 computer (with a clock period of 4.1 nsec in 1985) has a clock speed only about three times that of the Cray 1 computer (clock period, 12.5 nsec in 1976), and this took 9 years to achieve. New materials, such as gallium arsenide devices, are not expected to increase clock speeds by more than a factor of 5 in the next 5 to 10 years.

Furthermore, clock speeds are no longer an adequate indicator of system performance. For example, the recently released Cray 2 (*1*) has a clock speed that is more than twice the speed of the Cray X-MP (*1*), and yet, because of its architecture, most initial users cannot obtain from the Cray 2 a performance equal that of the Cray X-MP. In such complex, highly concurrent systems, actual delivered performance is program- and algorithm-specific. Seemingly attractive architectural features often have low payoff in delivered system performance on actual applications, and severe system bottlenecks appear in unexpected places. Thus delivered performance to actual users is often only 5 to 15 percent of the peak performance rates quoted above, except when hand optimization and assembly language programming are used on well-suited programs.

On the optimistic side, semiconductor performance and device densities in very large scale integration (VLSI) have increased to the point where 32-bit microprocessors and high-speed 64-bit floating-point arithmetic chip-sets are available and are beginning to be used in some supercomputer systems. Memory chips with up to 10^6 bits and access times of about 100 nsec are also becoming available to system designers. These densities are expected to continue to advance in the coming decade, with some improvements in both component performance and performance-cost ratio.

In an effort to restore a high growth rate in supercomputer performance, computer designers have made the first half of the 1980's a turning point in the organization of commercially available systems. Existing companies have observed that they can no longer

The authors are at the Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Urbana, IL 61801.

primarily rely on technology to improve the speed of their systems. Furthermore, there has been such an increase in available venture capital that start-up companies, often with no strong design prejudices or constraints from prior products, have launched a number of interesting new computer organizations. These organizations, and related trends in academia, show a dramatic shift from uniprocessors toward the development of multiprocessor systems with a much tighter coupling in the development team among architecture, hardware, software, and applications expertise (*2*). Taxonomies of these systems and algorithms have been developed by a number of people (*3*).

There is clearly great appeal to an approach that allows a doubling of the peak system speed by simply doubling the number of processors. In practice, multiprocessing has been used in supercomputers by Cray Research in their two newest systems, the Cray 2 and Cray X-MP (*1*), each of which currently can have four processors. ETA Systems is planning to use eight processors in its ETA10 by 1987. Multiprocessors can be exploited by simultaneously running a different job on each processor. However, improving single-job turnaround time, the typical supercomputer mission, can only be achieved by parallel processing, that is, using parallel algorithms and restructuring the code of a single job to spread it over a number of cooperating processors.

Multiprocessor systems to date have not made parallel execution convenient, and few users have found it worth the large effort required to restructure their programs. The most critical needs in supercomputing today are to provide an easy means of achieving parallel applications code and to develop multiprocessor systems that reduce the gap between the delivered performance for this code and the peak performance of the supercomputer system. Only then will increases in peak performance through parallel processing achieve their intended effect. We believe that automatic software restructuring and use of parallel algorithms are keys to meeting these needs, and we are designing the Cedar parallel supercomputer system for this environment by incorporating multiple levels of parallelism and dynamic adaptability to run-time conditions.

This article discusses several salient issues in parallel supercomputing today regarding processor capability, shared and private memories, memory hierarchy organization and management, programming languages and program restructuring, the applications environment, and the need for new numerical and nonnumerical algorithms. For examples, we describe existing systems as well as some hardware and software projects that are currently under development.

Processors

A fundamental question is whether to use the highest performance processors available, low-cost processors, or a mid-range compromise. Supercomputer processors, as in the Cray systems, the CDC Cyber 205, Fujitsu VP-200, Hitachi S-810, and NEC SX systems, have gained speed over the past 20 years by pipelining arithmetic functions and employing vector instructions and multiple function units (*4*). This approach is limited by the diminishing returns from increasing the number of pipeline segments beyond six or eight, by the inability on average to utilize simultaneously more than two function units (for example, add and multiply), and by the large start-up overhead of vector instructions. Several identical vector units are sometimes used to increase peak performance, but this approach increases the significance of vector start-up overhead because vectors are broken into shorter pieces for each vector unit.

These supercomputer systems, which cost about $10 million, have peak performances of several hundred to 1000 megaflops. Their 4- to 20-nsec clocks lead to longer pipelines and densely packed bipolar circuits with liquid cooling. Thus, manufacturing, operating, and maintenance costs tend to be high. They require long, dense array operations with regular memory addressing to approach their peak performance. Sparse matrix operations, on the other hand, in which irregular memory addressing prevails, degrade the performance of many application packages on such machines.

At the other end of the performance spectrum, several multiprocessor superminicomputers have been introduced or announced recently, such as the Encore, Flexible, and Sequent systems (*5*), which are based on the National Semiconductor 32000 series microprocessor. These systems exploit inexpensive off-the-shelf microprocessors and standard bus designs, with no vector instructions and minimal pipelining. They use the rudimentary parallel-processing software provided by the widely adopted UNIX operating system plus language-specific synchronization techniques. The Intel iPSC abandons standard busses and operating systems in favor of a multidimensional hypercube connection between processor-memory nodes (*6*).

Superminicomputers, when configured with 8 to 32 processors, typically cost about $100,000 and have less than 2-megaflops peak performance. Thus several thousand processors would be required to reach supercomputer performance. Since it is yet to be shown what applications could readily exploit such a system without massive system coordination overhead and application development costs, systems in this class are not proven contenders in the supercomputer market today. Nevertheless, some applications with massive parallelism and a preponderance of low precision data (for example, some image and signal processing problems) have enjoyed success on the Goodyear Aerospace STARAN and MPP systems (*7*), which do contain thousands of bit-serial processors in one system. This approach (*8*) has also been used by some new logical inference projects and other projects oriented toward artificial intelligence—for example, by Thinking Machines Corporation.

In the midrange of price and performance are at least two systems, Alliant and Elxsi, that use eight to ten fairly powerful processors for multiprocessing or parallel processing. These, together with minisupercomputers built by Floating Point Systems, Star Technologies, and Convex and with those being developed by Scientific Computer Systems, Axiom, Astronautics, and others, employ a variety of pipelined, multiunit architectures for vector and scalar operations and typically have peak performance in the 10- to 100-megaflops range for less than $1 million. These systems use fairly conservative technology with 40- to 100-nsec clocks and air cooling. Those minisupercomputers that achieve less than $10,000 per megaflops (peak at 64 bits) offer the highest performance-cost ratio for supercomputer applications in today's technology. A system that includes several of these minisupercomputers can hope to achieve supercomputer performance levels.

The Cedar supercomputer is being constructed to demonstrate that parallel processing can deliver good performance across a wide range of applications. It consists of multiple clusters with a globally shared memory. Each cluster is a slightly modified Alliant FX/8 minisupercomputer (Fig. 1) with a UNIX operating system, virtual memory, eight 64-bit floating-point processors, fast interprocessor synchronization, and vector instructions. We currently have two clusters operating independently and plan to have them operating together in the third quarter of 1986; two more clusters will be added by the first quarter of 1987 for a total of 32 processors. It is our objective to double the number of processors in the Cedar system each year for the next 5 years. Parallel systems such as Cedar offer the advantage that algorithms can be much less uniform and still be executed efficiently, thereby reducing the gap between delivered and peak performance (*9*). Because we believe that the

Cedar system has a general hardware and software organization, we will use Cedar as a reference to compare ideas throughout the article. Future implementations of such systems may well employ faster technology or denser VLSI technology if and when these technologies become more cost-effective.

Memories

Supercomputer performance is usually limited by memory accessing. Supercomputer memories have become larger (the Cray 2 can have up to 256 million 64-bit words) and faster, and processors have been designed to match the fastest and widest affordable paths to memory; yet this limitation has persisted. Costs have been controlled with memory hierarchies that use small, fast memories with a high cost per bit near the processor and larger, slower memories with less cost per bit at greater distances from the processor. Blocks of information are moved from slower levels to faster levels of the hierarchy as needed. A well-designed hierarchy has an effective access time near that of the fastest memories (due to locality, the tendency of temporally near references to access physically near data) and a cost per bit near that of the largest memories. Yet the fraction of system cost invested in memory and in paths between memories and processors is still increasing.

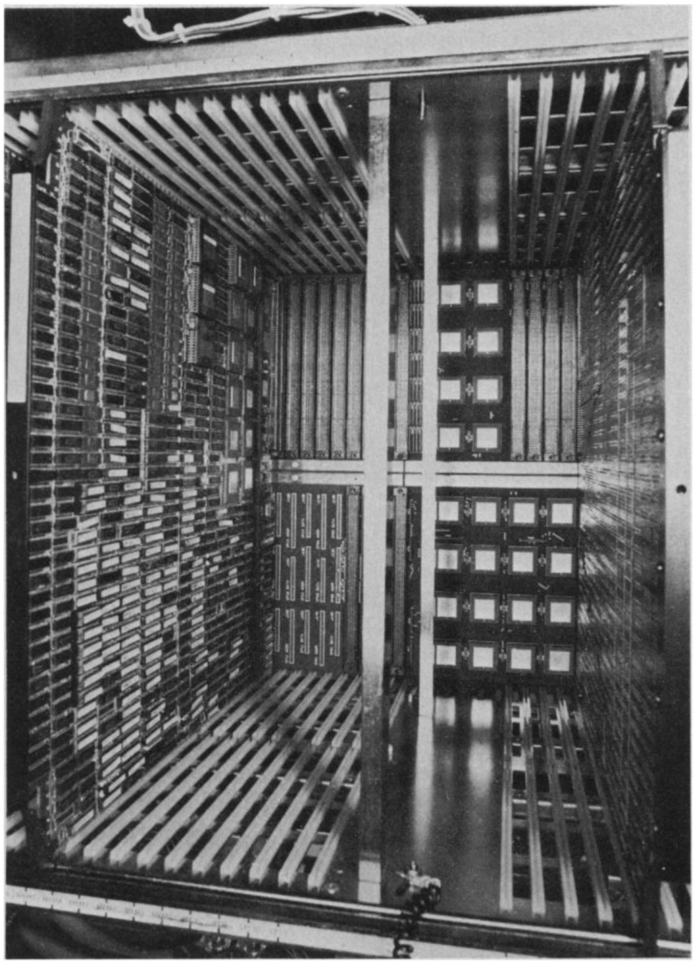


Fig. 1. The Alliant FX/8 chassis with several 18-inch boards removed to show an active backplane (rear) with a crossbar switch between eight processors and a four-port shared-cache memory (Fujitsu CMOS gate arrays of about 2000 gates). The processor board (left) contains the Weitek floating-point chip-set (two multipliers and an adder-subtractor, three packages top at midboard) that operates at about 6 megaflops (64-bit words); also visible (upper right) are two dividers (two Fujitsu gate arrays of about 8000 gates) designed by Alliant. Each single-board processor contains about 10^5 gates. The entire system is air-cooled.

Cache memories are small, fast memories that are managed by hardware and are thus transparent to system and user software. Most supercomputers do not use caches because of various inefficiencies and instead use efficiently managed vector register sets that hold small slices of large arrays for vector processing. However, data caches are appropriate if they allow noncacheable modes for large data structures that would otherwise be referenced only once in cache before being replaced, or if the system clock speed is slow enough to tolerate the small delays that caches add. Instruction caches are efficient in any case.

Virtual memory shields users from the complexity of memory hierarchies by providing a large monolithic address space and employing software and hardware support to map virtual addresses onto physical addresses and to move blocks of needed information among the levels of the hierarchy when demanded. Some supercomputers have not used virtual memory (Cray does not but CDC does) for reasons similar to those for not using cache; they have instead required application code to manage overlays of data in memory explicitly. However, recent research into loop blocking (discussed below) indicates that such supercomputer inadequacies may not endure.

In multiprocessor systems, the question of shared or private memory arises at various levels in the memory hierarchy. When multiple accesses to a shared memory must be served simultaneously, the memory is typically partitioned into modules (or banks) in such a way that simultaneous conflicting requests to a single module are rare (*10*). However, shared memories with multiple simultaneous requests must of necessity be at a greater distance from the requestors than a private memory for each requestor. Thus, although high request rates can be sustained with appropriate pipelining of packet-switched paths to memory, long access delays must be tolerated by the processor by allowing multiple outstanding requests and providing earlier fetch requests for needed information (*11*).

Registers inside the processor are one form of private memory. A private cache or local memory for each processor is often also desirable. Memory that is private to one cluster but shared among the processors of that cluster may be called partially shared memory. Sharing is almost always used at more remote levels of the hierarchy. However, some systems, such as the hypercube systems (*12*), use only private memory, wherein each memory in the system is associated with and controlled by one particular processor. Although careful algorithm design can diminish the need to share data, some sharing of data between processors is inevitable. When no shared memory is used, variables that are needed by several processors must be accessed through a message-passing system that allows one processor to request some other, possibly distant, processor to modify or transmit the requested data (*13*). Without shared memory, this process can be painfully slow.

Several alternatives exist for connecting multiple processors (or private memories) to a shared memory. A time-shared bus is the least expensive solution, but it can only serve one request at a time and therefore provides the lowest performance. When multiple requests must be served simultaneously, some form of parallel switch is required. A crossbar switch provides high performance and can simultaneously serve one request for each output port. Buffering at the switch ports can partially mask burst demands for particular output ports. However, the cost of a crossbar typically grows as the square of the number of input or output ports, and its cost can easily become prohibitive. For this reason, blocking switches are often used. These consist of several stages, where each stage is a set of small crossbars (*14*). Lower performance results since requests for distinct final output ports may be blocked as a result of contention for some crossbar's output port. Properly designed blocking switches, however, achieve nearly full crossbar switch performance at a small fraction of the hardware cost (*15*).

Whenever distinct copies of shared data are allowed to exist in the memory system a coherence problem exists: the logical consistency of these copies must be maintained. A variety of mechanisms are normally employed to insure coherence. Hardware mechanisms, such as snooping mechanisms for private caches (*16*), are typically used among multiple memories in smaller systems and within clusters. Such mechanisms (which are logically similar but more complex than the switches mentioned above) become prohibitively expensive and can limit system performance in larger systems; thus software schemes must be used at some point. The Cedar compiler will identify variables that could cause incoherence and allocate them to noncached shared memory. If there are few of these variables (as we have observed in experiments), accessing them in a slower mode should not cause much performance degradation (*17*).

Synchronization primitives are used in some form to coordinate access to shared variables. There are many types of synchronization primitives, but all may be thought of as having a key that is required to pass some comparison test before permission is given to access or update some data, the key itself, or both. Testing and modifying the key must be performed as an indivisible operation to prevent another set of operations from beginning before the set in progress is completed. Probably the simplest form of synchronization is the full-empty bit used in the HEP computer (*18*), whereby a word cannot be written if its full-empty bit indicates full and cannot be read if it is empty. Only slightly more complex is the common test-and-set form of instruction, which tests for a particular value and, if the test is true, writes a new value into the key. Still more complex is the fetch-and-add instruction used in the Ultracomputer and the IBM RP3 (*19*), which allows an arithmetic or logical operation instead of just a write and benefits from some clever combining of operations in the memory switch. Cedar uses a general and flexible synchronization mechanism (*20*) wherein each 64-bit data word may have an associated 32-bit integer key stored in the same global memory module. Each global memory module has a simple dedicated processor that performs an indivisible sequence of synchronization operations in response to a single-packet request transmitted from a processor. A transmitted key is compared with a stored key, and one of several tests is performed. If the test is successful, an arithmetic or logical operation may be performed on the stored key, and transmitted data may be stored. A return packet to the processor contains the test result and (if the test is successful) key or data values (or both) as requested.

Secondary memory is usually implemented with disks, which handle most input/output (I/O) traffic. Historically, disk traffic rates have been proportional to computational rates, regardless of the primary memory size or configuration. However, improvements in disk speeds have not kept up with processor speeds. Recently built supercomputers employ solid-state memories associated with the disk subsystem, such as the Cray SSD or caches with smart disk controllers. Alternatively or in addition, disk striping, or partitioning of very large data structures, may be used: stripes of data structures are allocated to distinct disks that are accessed in parallel to provide the massive data rates required by the supercomputer.

Supercomputers thus require a wide variety of memories distributed throughout the system. The Cedar system may be used as one illustration (Fig. 2). Effective exploitation of a complex memory

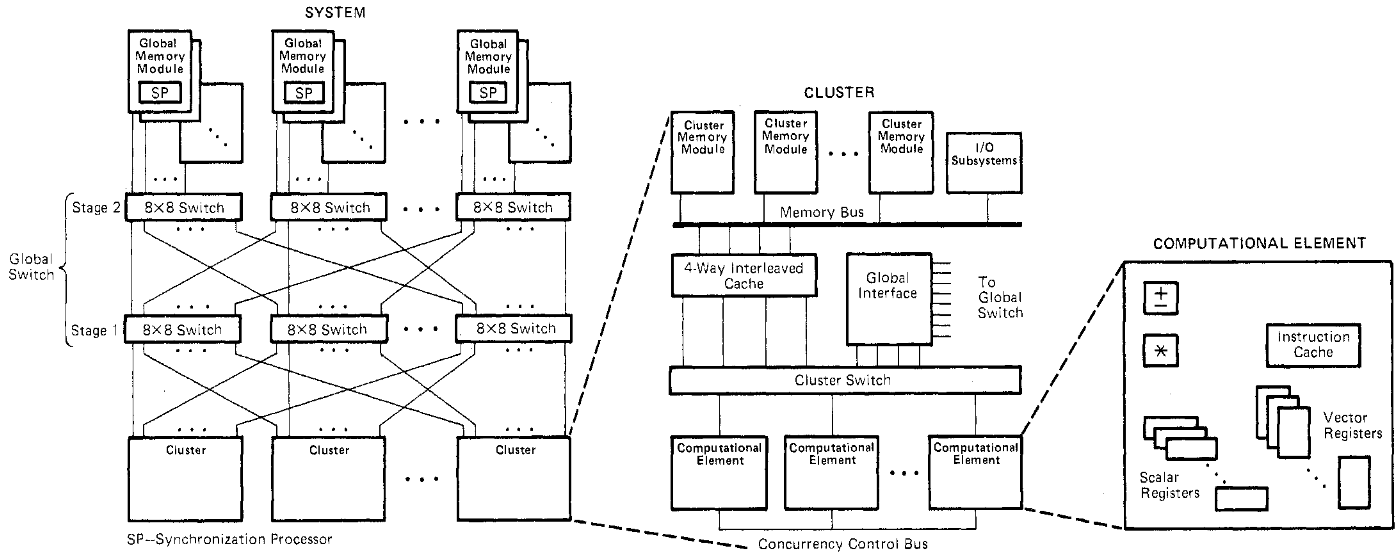


Fig. 2. In the Alliant FX/8 cluster, the eight floating-point processors each have vector registers and an instruction cache, share a concurrency control bus for fast synchronization, and share a single four-module cache with ports that are twice as fast as a processor port. This cache is backed with a shared cluster memory. The cluster memory is backed by an I/O subsystem that contains several caches, processors, disks, and other I/O devices. A hardware coherence scheme insures cache coherence. Each floating-point processor is also connected through Cedar logic to a private port of the global switch network that provides access to the shared global memory. A crossbar switch within each cluster connects each of these processors to its global switch port and to the shared cache ports. Fortran do-loops may be computed as doacross loops that are executed as follows. Cluster control provides for self-scheduled assignment of the next loop iteration to the next available processor, and fast synchronization hardware handles dependences between loop iterations within a cluster. Software handles doacross loops on multiple clusters. Each of two unidirectional intercluster global switches is fully pipelined and employs two stages, each with eight 8 × 8 unidirectional crossbars and input buffering, for system configurations up to eight clusters. The global memory contains one module per floating-point processor in the system. Each module contains two interleaved banks and a synchronization processor. The synchronization processor can perform an elaborate synchronization operation in response to a single input packet, thereby saving several round trips through the network while a memory port is locked up for each synchronization. The global memory may thus be used effectively for intercluster shared data and synchronization, for streaming long-vector accesses at high rate to the processors, and as a fast backup memory for cluster memory. Each cluster has both a computational and an I/O complex of processors so that, as more clusters are added, the peak rates for processing and I/O both grow; disk striping can be used to exploit this I/O bandwidth.

hierarchy, and thus supercomputer efficiency, can be greatly enhanced by properly structuring a program. As we shall see below, restructuring compilers are becoming more and more capable in this respect.

Software

To take better advantage of the ever more complex organizations of the latest supercomputers, users are having to write increasingly intricate programs that are fine-tuned to details of the system hardware. Thus, one of the most pressing questions associated with the powerful new concurrent processors is how to reduce application development costs, thereby making supercomputers practical for ordinary users. Answering this question means determining how to meet crucial objectives in five areas:

1) Programs: the ability to use old programs embodying sequential algorithms in old languages, as well as new programs using parallel algorithms in old or new languages.

2) Languages: new languages that allow one to express, in a well-structured form, algorithms that are amenable to parallel processing.

3) Compilers: software, which is able to exploit effectively all available architectural features, for use in developing and compiling programs in both old and new languages.

4) Algorithms: applications packages and library routines with parallel algorithms for standard problems.

5) Environments: effective programming environments for using the above software interactively, debugging programs, and graphically reviewing results of runs.

The first of the five objectives above would allow users to approach new machines without having to rewrite their programs in a new style or a new language. The ability to use old languages makes for an easy transition from an old machine to a new machine and thereby provides for architectural evolution. This approach is probably a necessary condition for the general acceptance of a new machine.

The second and third objectives, taken together, would allow users to learn and exploit a new language, especially if the program-development system could translate the old language to the new. Language evolution would occur as the user moved from familiar programs to new high-performance programs that would be easier to understand. New language features are not a sufficient condition for the success or acceptance of a new machine. New languages should permit the user to make assertions about the program that allow faster execution. In fact, the program-development software should query the user for such assertions.

Packages and library routines, mentioned in the fourth objective, have always been important to computer users. However, when integrated with program-restructuring techniques, they would help lead to the new and powerful program development systems in the fifth objective.

Debate over which programming language to use has been going on for many years and will probably continue indefinitely. As with natural languages, people become biased by what they understand and grew up with but are willing to learn a new language if their livelihood or interests depend on it. Probably the only two arguments that could effect a change of language or computer system are faster computation and easier programming. Speed is easy to measure, but ease of use is somewhat subjective, and both require the user to invest substantial time just to make a comparison. Thus, change has been slow to come.

Applicative and functional programming languages (*21*) and new languages with low- and high-level parallelism constructs (*22*) have been advocated for use with parallel processors, as have extensions to existing imperative programming languages (*23*). Our own work in program restructuring has focused on Fortran because it is so well established as a high-level language and yet is difficult to restructure automatically. Because we have achieved good results by restructuring Fortran code (*24*), we believe that the future looks bright for developing and exploiting powerful restructuring tools for a wide range of languages. We feel, however, that even for languages with explicit parallelism, automatic restructuring will always be important for advanced computer organizations. This is especially true because of the difficulty that most users have in exploiting these systems effectively (*25*).

Assuming that a user has a good algorithm to solve a problem on some machine, the remaining problem is to obtain high performance as easily as possible. A good programming environment is essential for this conversion and should include a good programming language, a powerful editor, compiler, and debugger, as well as a rich library of software packages. Using these program development tools, a programmer can transform an algorithm into a good parallel supercomputer program (the TAMPR system at Argonne and the proposed Gibbs Project at Cornell are examples).

Program development is an iterative process (Fig. 3). The user enters a program into the program-development system, which analyzes it and possibly restructures it for the target machine. The system can query the user to get additional information that can lead to a better optimization of the program. Finally, the program can be compiled and tested. Program testing on a supercomputer involves speed measurement (including bottleneck identification), which can be done at both compile- and execution-time, and numerical quality measurement, which may depend on how the program was mapped onto the system (*26*). Debugging the execution of a parallel computation can be particularly difficult without effective system aids because the logic of the program may be very complex, and the execution may be nondeterministic in that independent operations may be executed in a different order on different runs, exposing bugs in ways that are not always reproducible (*27*).

Applications programs must be handwritten or restructured by a compiler to put parallelism in the preferred form for the target supercomputer. Program restructurers can automatically effect simple algorithm changes, as illustrated in a simple vector-matrix product. Program 1 implements

$$c^T = a^T B \qquad c, a \in R^n;\ B \in R^{n \times n}$$

and is coded as follows:

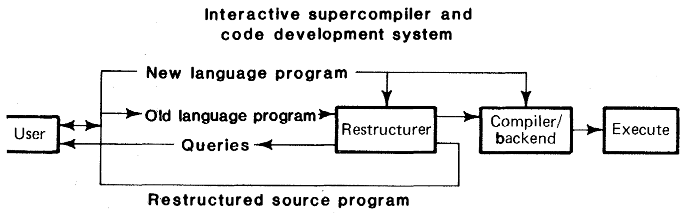
```
do j = 1, n
   $c_j = 0$
   do i = 1, n
      $c_j = c_j + a_i * b_{ij}$
   enddo
enddo
```

Program 2 implements

$$c^T = \sum_{i=1}^{n} a_i * (e_i^T B)$$

and is coded as follows:

```
do i = 1, n
   $c_i = 0$
enddo
do i = 1, n
   do j = 1, n
      $c_j = c_j + a_i * b_{ij}$
   enddo
enddo
```



ig. 3. Program development with automatic restructuring for parallel rocessing. In the source-to-source scheme illustrated here, an automatic estructurer takes a user's source program, asks questions, receives answers nd other statements, and then produces a new source program that can be ompiled and executed. The arrow returning to the user indicates that this rocess is iterative and that the new language program derived in this ianner may be maintained and compiled without further restructuring. Iowever, today's intricate supercomputers are best exploited by having the ser write an abstractly "highly parallel" program and letting the software estructure it for fast execution.

Quite different code would be generated for these two loops, and he machine architecture would dictate which to choose. Program 1 elies on reduction of vectors to a scalar (inner product) and is dvantageous in many vector processors because it requires only two ector memory accesses. Program 2 does (vector = scalar × vector - vector), which can be executed on a vector processor or spread cross the processors. However, in a vector processor, it may be less esirable than Program 1 since it requires three vector references including the store) and hence may stress the memory hierarchy iore than an inner product.

Innermost loops should contain vectors to exploit vector proces- ors, but outermost loops should contain doacross concurrency Fig. 2) that allows execution of multiple iterations in parallel. igure 4 is a simple illustration of a doacross to solve a lower iangular system $Ax = f$. If $A = [\alpha_{ij}]$, $f = \{\phi_i\}$, $x = \{\xi_i\}$, and i, = 1, 2, . . . , n, a sequential program for solving this system is iven by the following program:

```
do j = 1, n
   $\xi_j = \phi_j/\alpha_{jj}$
   if (j.eq.n) exit
   do i = j + 1, n
      $\phi_i := \phi_i - \alpha_{ij}\xi_j$
   enddo
enddo
```

The main point of a doacross is that computing each ξ_j need not ait for the completion of the whole inner iteration $i = j + 1, \ldots,$. In fact, one processor may compute ξ_j soon after another rocessor has computed $\phi_j := \phi_j - \alpha_{j,j-1}\xi_{j-1}$. To minimize the nchronization overhead in a doacross, the computation is per- ormed by blocks. For example, if $A = [A_{pq}]$, $x = \{x_p\}$, $f = \{f_p\}$, and , $q = 1, \ldots, 4$, where each block is of order $n/4$, then the doacross n two processors may be illustrated as shown in Fig. 4. Vectoriza- on can be exploited in each of the calculations shown if each rocessor has vector capabilities. Independent tasks should be ranged for simultaneous execution on multiple clusters. Also, the ata must be restructured so that it flows smoothly through the iemory hierarchy and arrives at the correct processors at the right me. Thus, loops may be blocked so that, for example, operations n large two-dimensional arrays are not carried out on whole rows r columns but rather on small rectangular blocks that can be ontained in private memory and referenced many times before the rogram accesses other blocks of the large array. The resulting code in be effective on a supercomputer with cache hardware or with ical memory that requires explicit move operations (for example, ector register loads and stores). The same ideas have been shown to be effective between main memory and secondary memory in any virtual memory system for a wide range of computations (*28*). Finally, it must be remembered that the time required to synchro- nize various parts of the computation can overwhelm any speedup obtained through parallelism. Thus the program and data restruc- turing mentioned above must be carried out with an eye to avoiding synchronization whenever possible (*29*).

How to exploit a parallel processor to speed up each job is currently a serious question in machine design as well as software design. We describe three levels of granularity in program parallel- ism that one could attempt to exploit on a correspondingly parallel machine organization. The highest level is that of separate subrou- tines, separate loops within a subroutine, and so forth. Most programmers can easily find some independent tasks at this level in their programs. However, it is unlikely that factors of more than a small constant in speedup are possible in most programs using such high-level granularity.

Medium granularity is represented by parallelism between indi- vidual loop iterations. Here, the discovery of parallelism is some- times rather intricate. For example, in weather prediction code, it is easy to see that the Northern and Southern hemispheres can be computed simultaneously, but exactly how to break a program up so that all the mesh points can be computed at once may be trickier, because there are many dependences between program variables. Nevertheless, the potential for parallelism here is enormous, because of the large number of mesh points involved. In general, the potential speedup is proportional to the loop limit or to the product of nested loop limits. Nested loops provide the potential for loop interchanging to move the appropriate type of parallelism to a nesting level that best exploits the machine organization. Experi- mentally, we have observed that medium granularity parallelism is by far the most important in program speedup (*30*).

Compared to vector machines, parallel processors can deal more efficiently with conditional branching and with random memory accessing caused, for example, by subscripted array subscripts. When program dependence graphs contain cycles with linear recurrences, fast algorithms are available for multiprocessors or properly de- signed vector machines (*31*). However, parallel processors using doacross (Fig. 4) can also deal with nonlinear recurrences by simply delaying subsequent iterations properly, whereas vector machines must execute them in scalar mode. All these cases can be compiled from sequential programs by a powerful restructuring compiler.

The lowest granularity of parallelism includes arithmetic expres- sions and blocks of assignment and control statements. Methods of speedup with optimizing compilers are well understood at this level, but the potential speedups are relatively small because of the limited complexity of individual statements or even blocks of statements. Speedup factors of 2 or 3 are possible here, as judged from extensive measurement of real Fortran programs (*32*).

Applications

The main driving force for higher supercomputing performance is the fact that some important applications in engineering and science currently consume excessive amounts of time or are infeasible to attempt at all on available vector computers. To describe physical phenomena, one must resort to simulation of complex models on the computer. The closer the model is to a physical phenomenon, the more extensive are the required computational resources. For example, to simulate certain aerodynamic flows around a three- dimensional aircraft configuration, one needs to solve time-depen- dent partial differential equations that require a capacity of fast storage and a sustained computational rate that exceed by several orders of magnitude those offered by the fastest existing vector machines (*33*). This exponential growth in the running time of time-dependent problems in three space dimensions can be easily appreciated by considering the refinement of the space mesh by halving the distance between the nodes and halving the time step, as well as by using a more realistic physical model. Such requirements can be made feasible only by parallel processors that incorporate innovative hardware organization, together with well-suited software development tools and algorithms. In addition to this three-dimensional fluid flow example, we expect to see other important uses of parallel processors in the near future (*34*). Among these are computer simulation of gauge theory and elementary particle physics, multidimensional semiconductor devices, electronic circuits, weather circulation, and oil reservoirs, as well as studies in chemical quantum dynamics and molecular scattering, seismic imaging, and dynamic structural analysis.

On the basis of experiments with automatic program restructuring for multiprocessing, we have found that potential bottlenecks in taking advantage of parallelism in these applications are most obvious from analyzing a few important building blocks. These building blocks are Monte Carlo calculations, table look-up, and various numerical linear algebra algorithms for unstructured and (to a certain extent) structured sparse problems. Some parallel algorithms, primarily in computational linear algebra, have been developed in an abstract setting (*35*); some of these are ideally suited to a parallel processor such as Alliant's FX/8 and the new Cray multiprocessors (*36*). On one cluster of Cedar, we have achieved performance rates that exceed 20 megaflops for dense matrix calculations. This performance ranges from 0.1 to 0.2 of that on one central processing unit (CPU) of the Cray X-MP. While performance on such problems is widely quoted, it seldom reflects system performance on whole applications. For example, in the eigenvalue problem of symmetric tridiagonal matrices, where vector calculations do not play a significant role, the achieved performance on one cluster is 0.5 that of one CPU of the Cray X-MP. On complete application codes, we expect the performance of one Cedar cluster relative to one Cray X-MP processor to range from 0.2 (the ratio of their peak performances for 64-bit arithmetic) to 1.

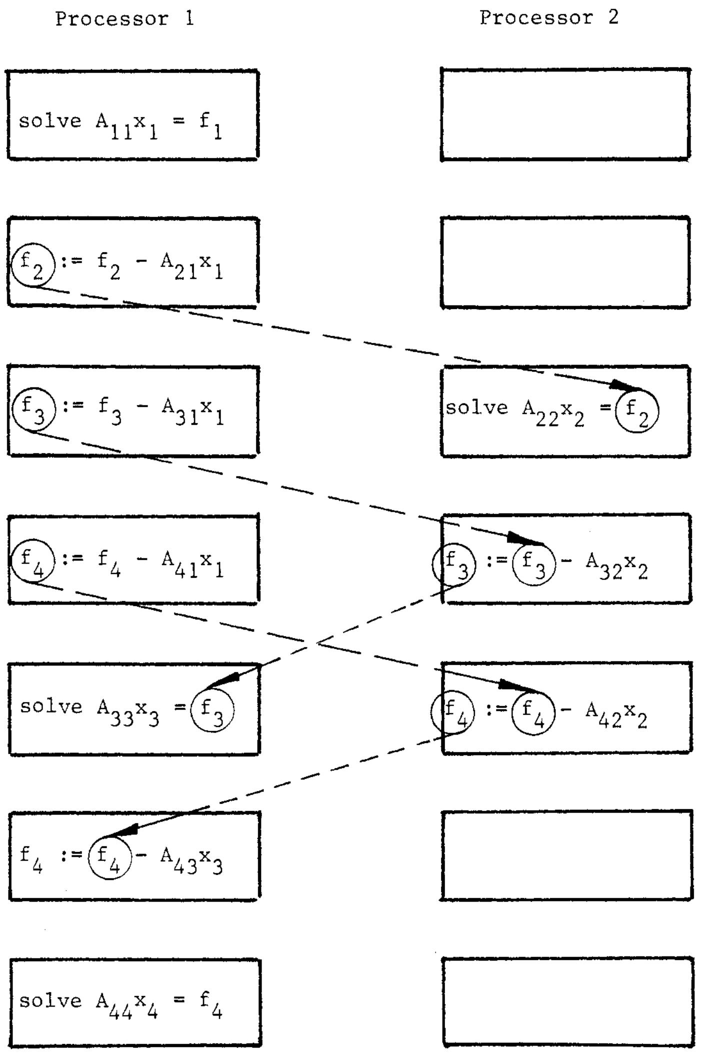


Fig. 4. Illustration of a doacross.

In general, a parallel algorithm is one where the various computational steps can be divided among a number of processors. The performance of such an algorithm depends primarily on the organization of the multiprocessor on which the algorithm is to be used. It is natural, therefore, to associate particular algorithms with major architectural features. Perhaps the most obvious question to be considered by an algorithm designer is whether the multiprocessor consists of a relatively small or large number of processors, which dictates the granularity of the parallelism to be exploited. In the former case, it is usually possible to decompose a problem into tasks that need to communicate only infrequently. For the latter case, tasks that can be executed independently in each processor are usually short-lived, and interprocessor communication may dominate. With many processors, high efficiencies are often more difficult to achieve than with few processors (*37*). For a particular application, the number of processors that can be efficiently used depends on both the degree of parallelism that can be achieved for the application and the size of the problem being solved. By using a powerful restructuring compiler on most of a program and basic algorithms with high parallelism whenever necessary, parallel processor users with large jobs will achieve high performance, even on systems with hundreds of processors (*38*). In designing parallel algorithms, it is also important to distinguish between systems with a shared memory and those with private memories only. For the private memory model, the interconnection geometry plays a vital role in the choice of an algorithm for solving a given problem (*39*). If the number of processors is large, so that communication between distant processors becomes slow, the algorithm, if possible, should be designed such that a given processor needs to communicate predominantly with those few processors in its immediate neighborhood. As the number of processors grows larger, algorithm designers on such parallel processors have increasing difficulty in achieving high speedup, except for Monte Carlo methods (*40*) and asynchronous iterative schemes (*41*).

In the Cedar architecture, we adopt the shared memory model for which algorithm design is more flexible. Furthermore, to accommodate a large number of processors without suffering from high synchronization overhead, we introduce eight-processor clusters with fast intracluster synchronization and local cluster memory. Thus, there are three levels of system parallelism at the disposal of the Cedar algorithm designer. The first step is to decompose the problem among the available clusters such that the time consumed in the computational tasks within a cluster far exceeds that consumed in intercluster communication. At the second level, the tasks within a cluster are allowed to be more communication intensive; with self-scheduled doacross (Fig. 4), the synchronization penalties are not high. At the innermost level the vector capabilities of each processor are used. Judging from our experience, we expect that Cedar will deliver more uniform speedup from application to application than has been possible on today's supercomputers.

Finally, we emphasize two points. First, in developing parallel algorithms to solve problems in the shortest time possible on a given multiprocessor, one should not sacrifice either robustness (graceful failure of algorithms on the machine) or, more important, numerical stability. Second, whenever possible, one should keep in mind the issue of portability of these algorithms, at least among members of the same class of multiprocessors. Here again, automatic program restructurers can be useful in limiting the damage to the performance of an algorithm as it is moved to another multiprocessor for which it was not originally designed (*42*).

REFERENCES AND NOTES

1. S. C. Chen, in *High Speed Computation*, J. Kowalik, Ed. (Springer-Verlag, Berlin, 1984), pp. 59–67; J. L. Larson, *Computer* **17**, 62 (1984).
2. P. B. Schneck *et al.*, *Computer* **18**, 43 (1985); J. T. Schwartz, *ACM Trans. Prog. Lang. Syst.* **2**, 484 (1980); A. Gottlieb *et al.*, *IEEE Trans. Comput.* **C-32**, 175 (1983); G. F. Pfister *et al.*, in *Proceedings of the 1985 International Conference on Parallel Processing*, D. Degroot, Ed. (IEEE, Piscataway, NJ, 1985), pp. 764–771.
3. C. Seitz, *Commun. ACM* **28**, 22 (1985); D. J. Kuck, in *Parallel Processing Systems*, D. J. Evans, Ed. (Cambridge Univ. Press, New York, 1982), pp. 193–214; M. J. Flynn, *IEEE Trans. Comput.* **C-21**, 948 (1972); J. Schwartz, paper presented at the Conference on Vector and Parallel Processors for Scientific Computation, Rome, Italy, 27 to 29 May 1985; *SIAM News* **17**, 6 (1984); J. C. Browne, *ibid.* **16**, 8 (1983); in *Computer Architecture Technical Communications Newsletter* (IEEE, Piscataway, NJ, 1984), pp. 77–191.
4. D. J. Kuck, *The Structure of Computers and Computations* (Wiley, New York, 1978).
5. J. R. Lineback, *Electronics* **58**, 32 (1985).
6. C. Seitz, *Commun. ACM* **28**, 22 (1985).
7. K. E. Batcher, in *Proceedings of the AFIPS National Computer Conference* (AFIPS Press, Montvale, NJ, 1974), pp. 405–410; in *Proceedings of the 7th Annual Symposium on Computer Architecture* (IEEE, Piscataway, NJ, 1980), pp. 168–173.
8. See R. P. Gabriel, *Science* **231**, 975 (1986).
9. D. J. Kuck *et al.*, in *Proceedings of the 1984 International Conference on Parallel Processing*, Robert M. Keller, Ed. (IEEE, Piscataway, NJ, 1984), pp. 129–138.
10. D. P. Bhandarkar, *IEEE Trans. Comput.* **C-24**, 897 (1975); F. A. Briggs and E. S. Davidson, *ibid.* **C-26**, 162 (1977); P. Budnik and D. J. Kuck, *ibid.* **C-20**, 1566 (1971); D. Chang, D. J. Kuck, D. Lawrie, *ibid.* **C-26**, 480 (1977); D. H. Lawrie and C. R. Vora, in *Proceedings of the 1980 International Conference on Parallel Processing* (IEEE, Piscataway, NJ, 1980), pp. 81–90; K. Padmanabhan and D. H. Lawrie, *ACM Trans. Comput. Syst.* **3**, 117 (1985); J. H. Patel, *IEEE Trans. Comput.* **C-30**, 771 (1981); D. W. L. Yen, J. H. Patel, E. S. Davidson, *ibid.* **C-31**, 1116 (1982); P. C. C. Yeh, J. H. Patel, E. S. Davidson, *ibid.* **C-32**, 38 (1983).
11. J. E. Smith, in *Proceedings of the 9th Annual International Symposium on Computer Architecture* (IEEE, Piscataway, NJ, 1982), pp. 112–119; A. R. Pleszkun, G. B. Sohi, B. Z. Kahhaleh, E. S. Davidson, in preparation.
12. C. Seitz, *Commun. ACM* **28**, 22 (1985); *A New Direction in Scientific Computing* (Intel, Beaverton, OR, 1985).
13. E. F. Gehringer, A. K. Jones, Z. Z. Segall, *Computer* **15**, 40 (1982); B. Lint and T. Agerwala, *IEEE Trans. Software Eng.* **SE-7**, 174 (1981).
14. V. E. Benes, *Mathematical Theory of Connecting Networks and Telephone Traffic* (Academic Press, New York, 1965).
15. V. E. Benes, *Bell Syst. Tech. J.* **162**, 499 (1983); P.-Y. Chen, P.-C. Yew, D. L. Lawrie, in *Proceedings of the 3rd International Conference on Distributed Computing Systems* (IEEE, Piscataway, NJ, 1982), pp. 622–629; N.-F. Tzeng, P.-C. Yew, C.-Q. Zhu, in *Proceedings of the 12th Annual International Symposium on Computer Architecture* (IEEE, Piscataway, NJ, 1985), pp. 368–375; K. Padmanabhan and D. H. Lawrie, *ACM Trans. Comput. Syst.* **3**, 117 (1985); A. Gottlieb *et al.*, *IEEE Trans. Comput.* **C-32**, 175 (1983); G. F. Pfister *et al.*, in *Proceedings of the 1985 International Conference on Parallel Processing*, D. Degroot, Ed. (IEEE, Piscataway, NJ, 1985), pp. 764–771; J. H. Patel, *IEEE Trans. Comput.* **C-30**, 771 (1981).
16. J. R. Goodman, in *Proceedings of the 10th Annual International Symposium on Computer Architecture* (IEEE, Piscataway, NJ, 1983), pp. 124–131; J. H. Patel, *IEEE Trans. Comput.* **C-31**, 296 (1982).
17. K. Y. Lee, W. Abu-Sufah, D. J. Kuck, in *Proceedings of the 1984 International Conference on Parallel Processing*, R. M. Keller, Ed. (IEEE, Piscataway, NJ, 1984), pp. 269–277; H. Husmann, thesis, University of Illinois, Urbana, in preparation.
18. B. J. Smith, *Proc. Int. Soc. Opt. Eng.* **298**, 241 (1981).
19. A. Gottlieb *et al.*, *IEEE Trans. Comput.* **C-32**, 175 (1983); G. F. Pfister *et al.*, in *Proceedings of the 1985 International Conference on Parallel Processing*, D. Degroot, Ed. (IEEE, Piscataway, NJ, 1985), pp. 764–771.
20. C.-Q. Zhu and P.-C. Yew, in *Proceedings of the 4th International Conference on Distributed Computing Systems* (IEEE, Piscataway, NJ, 1985), pp. 486–493.
21. J. R. McGraw, *Phys. Today* **37**, 66 (May 1984).
22. R. Taylor and P. Wilson, *Electronics* **55**, 89 (1982); C. A. R. Hoare, *Commun. ACM* **21**, 666 (1978).
23. "Fortran 8X," Doc. X3J3/S8, Version 97 (American National Standards Institute, New York, 1985; D. J. Kuck and M. J. Wolfe, *Phys. Today* **37**, 67 (May 1984).
24. D. J. Kuck *et al.*, in *Proceedings of the 1984 International Conference on Parallel Processing*, Robert M. Keller, Ed. (IEEE, Piscataway, NJ, 1984), pp. 129–138; D. J. Kuck, R. H. Kuhn, B. Leasure, M. Wolfe, in *Tutorial on Supercomputers: Design and Applications*, K. Hwang, Ed. (IEEE, Piscataway, NJ, 1984), pp. 168–178.
25. D. Kuck, D. Padua, A. Sameh, M. Wolfe, in *Proceedings of the IFIP Working Conference on The Relationship Between Numerical Computation and Programming Languages*, J. Reid, Ed. (Elsevier/North-Holland, New York, 1982), pp. 205–221.
26. J. L. Larson and A. H. Sameh, *Computing* **24**, 275 (1980).
27. T. W. Pratt, *Software* **2**, 7 (1985); J. Griffin and H. Wasserman, paper presented at the Second SIAM Conference on Parallel Processing for Scientific Computation, Norfolk, VA, 18 to 21 November 1985.
28. W. Abu-Sufah, D. J. Kuck, D. Lawrie, *IEEE Trans. Comput.* **C-30**, 341 (1981); W. Abu-Sufah, R. Lee, M. Malkawi, P. Yew, in *Proceedings of the 6th International Conference on Software Engineering* (IEEE, Piscataway, NJ, 1982), pp. 110–117.
29. Z. Li and W. Abu-Sufah, in *Proceedings of the 12th Annual International Symposium on Computer Architecture* (IEEE, Piscataway, NJ, 1985), pp. 284–291; S. Midkiff, thesis, University of Illinois, Urbana, in preparation.
30. D. J. Kuck *et al.*, in *Proceedings of the 1984 International Conference on Parallel Processing*, R. M. Keller, Ed. (IEEE, Piscataway, NJ, 1984), pp. 129–138; D. J. Kuck, R. H. Kuhn, B. Leasure, M. Wolfe, in *Tutorial on Supercomputers: Design and Applications*, K. Hwang, Ed. (IEEE, Piscataway, NJ, 1984), pp. 168–178.
31. A. H. Sameh and R. P. Brent, *SIAM J. Numer. Anal.* **14**, 147 (1977); S. C. Chen, D. J. Kuck, A. H. Sameh, *ACM Trans. Math. Software* **4**, 270 (1978); S. C. Chen and D. J. Kuck, *IEEE Trans. Comput.* **C-26**, 712 (1977).
32. R. Cytron, D. J. Kuck, A. Veidenbaum, *Comput. Phys. Commun.* **37**, 39 (1985).
33. V. Peterson, *Proc. IEEE* **72**, 68 (1984).
34. C. C. Hsiung and W. Butscher, *Parallel Computing* **1**, 113 (1984); K. C. Bowler and G. S. Pawley, *Proc. IEEE* **72**, 42 (1984); D. L. Williamson and P. N. Swarztrauber, *ibid.*, p. 56; R. P. Kendall, J. S. Nolen, P. L. Stanat, *ibid.*, p. 85.
35. D. J. Kuck and A. H. Sameh, in *Information Processing 71* (Elsevier/North-Holland, New York, 1972), pp. 1266–1272; A. H. Sameh, in *High Speed Computer and Algorithm Organization*, D. Kuck, D. Lawrie, A. Sameh, Eds. (Academic Press, New York, 1977), pp. 207–228; A. H. Sameh and D. J. Kuck, *J. ACM* **25**, 81 (1978); D. H. Lawrie and A. H. Sameh, *ACM Trans. Math. Software* **10**, 185 (1984); C. Kamath and A. Sameh, in *Proceedings of the 5th IMACS International Symposium on Computer Methods for Partial Differential Equations*, R. Vichnevetsky and R. Stepleman, Eds. (IMACS, New Brunswick, NJ, 1984), pp. 210–217.
36. A. Sameh, paper presented at the Second SIAM Conference on Parallel Processing for Scientific Computation Norfolk, VA, 18 to 21 November 1985; J. Dongarra and D. Sorensen, *ibid.*; S. Lo, B. Philippe, A. Sameh, *ibid.*; J. Dongarra and T. Hewitt, *Argonne Natl. Lab. Tech. Memo. 55* (1985).
37. J. Riganati and P. Schneck, *Computer* **17**, 97 (1984); B. Buzbee and D. H. Sharp, *Science* **227**, 591 (1985).
38. D. J. Kuck *et al.*, in *Proceedings of the 1984 International Conference on Parallel Processing*, R. M. Keller, Ed. (IEEE, Piscataway, NJ, 1984), pp. 129–138.
39. G. Fox, *Phys. Today* **37**, 50 (May 1984); A. Sameh, *Comput. Phys. Commun.* **37**, 159 (1985).
40. B. Lautrup, *Commun. ACM* **28**, 358 (1985); E. Clementi, G. Corongiu, J. Detrick, in *Proceedings of the 2nd International Conference on Vector and Parallel Processors in Computational Science*, I. Duff and J. Reid, Eds. (Elsevier/North-Holland, New York, 1985), pp. 287–294; L. Delves, *ibid.*, pp. 295–302.
41. G. Baudet, *J. ACM* **25**, 226 (1978); D. Chazan and W. Miranker, *J. Linear Algebra Appl.* **2**, 199 (1969); B. W. Wah, G. J. Li, C. F. Yu, *Computer* **18**, 93 (1985).
42. D. J. Kuck, D. Padua, A. Sameh, M. Wolfe, in *Proceedings of the IFIP Working Conference on The Relationship Between Numerical Computation and Programming Languages*, J. Reid, Ed. (Elsevier/North-Holland, New York, 1982), pp. 205–221.
43. Supported in part by Department of Energy grant DE-FG02-85ER25001, NSF grants DCR-8410110, DCR-8406916, and DCR-8509970, U.S. Air Force grant AFOSR-85-0211, and a gift from IBM.