REFERENCES AND NOTES

1. A. Newell and H. A. Simon, *Human Problem Solving* (Prentice-Hall, Englewood Cliffs, NJ, 1972).
2. E. A. Feigenbaum, B. Buchanan, J. Lederberg, in *Machine Intelligence* B. Meltzer and D. Michie, Eds. (American Elsevier, New York, 1971), vol. 6.
3. The related term "expert system" is less desirable for its lack of technical substance and tendency to suggest inappropriate standards. Calling something an expert system primarily advertises the aspiration to have it perform at the level of human experts. Although this has been accomplished in several cases, significant utility has arisen from knowledge-based systems that function as intelligent assistants and colleagues, without ever becoming expert at their task.
4. R. O. Duda and E. H. Shortliffe, *Science* **220**, 261 (1983).
5. I coined the term "inference engine" in 1974 in conscious analogy to C. Babbage's term for his pioneering 19th-century machine, the "analytical engine," but with the appropriate modification to make clear that its basic operation is inference, not calculation.
6. B. G. Buchanan and E. H. Shortliffe, *Rule-Based Expert Systems* (Addison-Wesley, Reading, MA, 1984).
7. H. E. Pople, in *AI in Medicine*, P. Szolovits, Ed. (Westview, Boulder, CO, 1982).
8. R. O. Duda and R. Reboh, in *AI Applications for Business*, W. Reitman, Ed. (Ablex, Norwood, NJ, 1984).
9. M. C. Maletz, *Artificial Intelligence 1985* (1985), p. 71
10. R. Bogen *et al.*, "*MACSYMA Reference Manual*" (Laboratory for Computer Science report, Massachusetts Institute of Technology, Cambridge, 1975).
11. *Proceedings of the AAAI Workshop on Uncertainty and Probability in Artificial Intelligence* (American Association for Artificial Intelligence, Menlo Park, CA, 1985).
12. J. Fox, C. D. Meyers, M. F. Greaves, S. Pegram, *Methods Inform. Med.* **24**, 65 (1985).
13. This explanation has been simplified to present only the points needed for what follows. A more detailed explanation can be found in (*4*) or (*6*).
14. B. K. P. Horn, *Robot Vision* (MIT Press, Cambridge, MA, 1984).
15. J. Kunz *et al.*, "A physiological rule-based system for interpreting pulmonary function results" (Computer Science Department working paper HPP-78-19, Stanford University, Stanford, CA, 1978).
16. J. S. Bennett, L. A. Creary, R. S. Engelmore, R. E. Melosh, "SACON: A knowledge-based consultant in structural analysis" (Computer Science Department report HPP-78-23, Stanford University, Stanford, CA, 1978).
17. J. S. Bennett and C. R. Hollander, *Proc. Int. Joint Conf. AI* **7**, 243 (1981).
18. W. vanMelle, "A domain independent system that aids in constructing consultation programs" (Computer Science Department report STAN-CS-80-820, Stanford University, Stanford, CA, 1980).
19. The concept that programs might be given advice rather than instructions appears quite early in the history of AI [J. McCarthy, in *Semantic Information Processing*, M. Minsky, Ed. (MIT Press, Cambridge, MA, 1968)]. This same motivation is also at the heart of some of the work on languages like Prolog.
20. This is not always easy. (i) Each equation has to be inverted; this can be difficult in complex models. (ii) It may be necessary to keep track of alternative choices: if $c = a + b$, for instance, c can be changed by changing a alone, b alone, or both together. But often it can be done and in those situations illustrates an important technique. Even where equations cannot be inverted, there is still utility in moving backward through the model to determine which quantities are relevant to the desired result.
21. Various commercial spreadsheet programs have several of these capabilities; a small, experimental program used in the MIT course on knowledge-based systems does them all (crudely).
22. R. Davis and D. B. Lenat, *Knowledge-Based Systems in AI* (McGraw-Hill, New York, 1982).
23. J. Bachant and J. McDermott, *AI Mag.* **5**, 21 (1984); R. Davis, *Artif. Intell.* **12**, 121 (1979).
24. W. J. Clancey, *Int. J. Man-Mach. Stud.* **11**, 25 (1979).
25. A. N. Campbell, V. F. Hollister, R. O. Duda, P. E. Hart, *Science* **217**, 927 (1982).
26. R. G. Smith, *AI Mag.* **5**, 61 (1984).
27. G. T. Vesdoner, S. J. Stolfo, J. E. Zielinski, F. D. Miller, D. H. Copp, *Proc. Int. Conf. AI* **8**, 116 (1983).
28. R. K. Lindsay, B. G. Buchanan, E. A. Feigenbaum, J. Lederberg, *Applications of Artificial Intelligence for Organic Chemistry* (McGraw-Hill, New York, 1980).
29. R. W. Olford and S. C. Peters, in *Artificial Intelligence and Statistics*, D. Pregibon, Ed. (Addison-Wesley, Reading, MA, 1985); D. Pregibon, *ibid.*, p. 136.
30. M. Stefik, *Artif. Intell.* **16**, 111 (1981); P. Friedland and Y. Iwasaki, *J. Automated Reasoning* **1**, 161 (1985).
31. W. T. Wipke, H. Braun, G. Smith, F. Choplin, W. Sieber, in *Computer-Assisted Organic Synthesis*, W. T. Wipke and W. J. House, Eds. (American Chemical Society, Washington, DC, 1977).
32. D. B. Lenat, *Artif. Intell.* **9**, 257 (1977); D. B. Lenat and J. S. Brown, *ibid.* **23**, 269 (1984).
33. R. Patil, "Causal understanding of patient illness for electrolyte and acid-base diagnosis" (Computer Science Department, report MIT/LCS/TR-267, Massachusetts Institute of Technology, Cambridge, 1981).
34. J. G. Carbonell, R. S. Michalski, T. Mitchell, in *Machine Learning*, R. S. Michalski, J. G. Carbonell, T. Mitchell, Eds. (Tioga, Palo Alto, CA, 1983).
35. J. McDermott, *Artif. Intell.* **19**, 39 (1982).
36. C. Forgy and J. McDermott, *Proc. Int. Joint Conf. AI* **5**, 933 (1977).
37. R. Davis, *Artif. Intell.* **15**, 179 (1980); M. R. Genesereth, *Proc. Natl. Conf. AI* (1983), p. 119; W. J. Clancey, *ibid.*, p. 74.
38. H. J. Levesque, R. J. Brachman, in *Readings in Knowledge Representation*, R. J. Brachman and H. J. Levesque, Eds. (Morgan-Kaufmann, Los Altos, CA, 1985).
39. W. Clancey, *Artif. Intell.* **27**, 289 (1985).
40. *Artif. Intell.* **24**, 1–491 (1984).
41. J. deKleer, in (*40*), p. 205; A. Stevens *et al.*, "Steamer: advanced computer-aided instruction in propulsion engineering" (report 4702, Bolt, Beranek & Newman, Cambridge, MA, 1981).
42. R. Davis, in (*40*), p. 247; M. R. Genesereth, in (*40*), p. 411.
43. J. deKleer and J. S. Brown, in (*40*), p. 7.
44. Useful comments on earlier drafts were received from B. Williams, R. Duda, W. Hamscher, M. Shirley, D. Weld, T. Malone, R. Valdes-Perez, P. Szolovits, D. Lenat, and E. Feigenbaum.

Small Shared-Memory Multiprocessors

FOREST BASKETT AND JOHN L. HENNESSY

Multiprocessors built from today's microprocessors are economically attractive. Although we can use these multiprocessors for time-sharing applications, it would be preferable to use them as true parallel processors. One key to achieving efficient parallel processing is to match the communications capabilities of the multiprocessor to the communications needs of the problem. The other key is improved parallel programming systems. If these are achieved, then efficient parallel processing can be approached from both ends by providing more communications capability in the hardware and restructuring the problem to reduce the communications requirements.

THE LABORATORY COMPUTING ENVIRONMENT FOR SCIENTISTS and engineers has been changed dramatically, first by minicomputers and, more recently, by personal computers and powerful workstations. Small multiprocessors promise to increase the speed with which computationally intensive problems can be solved in the laboratory (*1*). While such machines should be as much as one order of magnitude faster than today's minicomputers, they will probably have comparable or lower costs. These multiprocessors will be small in that the processors will number on the order

Forest Baskett is director, Western Research Laboratory, Digital Equipment Corporation, 100 Hamilton Avenue, Palo Alto, CA 94301. John L. Hennessy is associate professor, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305.

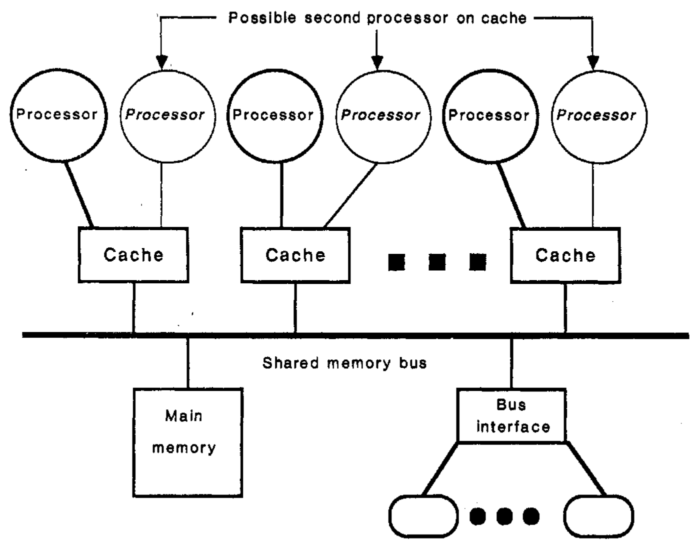

Fig. 1. Shared-memory multiprocessor.

of ten rather than 100 or more. Although there are many possible implementation technologies for the individual processors, CMOS (complementary metal oxide semiconductor) microprocessors are economically very attractive.

In this article we will discuss problems in using these multiprocessors as parallel processors. Given a parallel program, the multiprocessor can provide a better response by executing the program on multiple processors (*2, 3*). But even if none of the parallel programming problems we describe has good solutions, the kind of small multiprocessor described can easily be used in a traditional time-sharing mode to run existing applications. The multiple users of a time-sharing system can be distributed among the available processors. Only the operating system need "know" that it is running on a multiprocessor.

This ability to support time-sharing for existing applications and the promise of faster solution times for future applications makes shared-memory machines more commercially appealing than many other multiprocessor structures (*4, 5*).

A Small Shared-Memory Multiprocessor

A small multiprocessor built with today's technology would comprise 5 to 20 processors, the heart of each of which would be a 32-bit microprocessor. Each microprocessor or each pair of microprocessors would be supported by a separate high-speed memory that is a cache on a larger, common main store. Each cache would share access to the main memory system via a single common bus. One or more additional mechanisms might be provided to enable the separate caches to keep a coherent view of shared data. Data could be exchanged between processors via the common main storage. Sometimes additional high-speed processor-to-processor communication mechanisms would be available; these mechanisms can be crucial to the performance of a multiprocessor application. Figure 1 depicts a block diagram of such a small multiprocessor.

What Makes the Small Multiprocessor Attractive?

CMOS semiconductor technology has provided two key components in this multiprocessor organization: 32-bit microprocessors and caches built of fast, static memory. A cache is a local memory that keeps a copy of the most recently accessed data from main memory. A datum may be retrieved more quickly from a cache than from the main memory.

The workstation market ensures continued rapid development of 32-bit CMOS microprocessors; their high speed is well matched to the speed of the latest CMOS static memory devices. Thus, a natural tight coupling is a 32-bit microprocessor and a cache.

A large cache can support very high hit rates, the hit rate being the probability that the datum desired by the processor will be found in the cache. (Scientists and engineers will also want to know that the density of logic on CMOS integrated circuits can now support high-performance floating-point arithmetic logic.)

The cost of a laboratory computer today is dominated by the cost of the main memory, the peripherals (disks), the power, and the packaging. The microprocessor and its tightly coupled cache are cheap in comparison to the rest of the computer. The marginal cost of additional processor-cache pairs is small, and the high hit rate in a cache means that the main memory can support more than one processor-cache pair without being overloaded (*6*).

An additional critical element for achieving faster solution of individual problems on multiprocessors is new compiler technology. For multiprocessors to be easily used in the laboratory they must be easily programmed. Compiler technology has developed to the point where there are now commercial systems that can automatically decompose a traditional sequential algorithm into a form that can be executed in parallel on certain multiprocessors. While these systems currently tend to work best on well-structured, numerical problems and often need advice from the programmer, work continues in this area, and more powerful systems seem imminent.

Bolting Processors Together Is Not Adequate

If several processor-cache pairs are tied together in the most straightforward way, the performance may be extremely poor. The crucial shared resources in a shared-memory multiprocessor are the main memory and its bus. To learn about performance in this kind of computer, one looks at the point when the main memory and its bus become saturated. The saturation point is the knee in the curve of performance versus number of caches. It is the intersection point of the initial slope of the curve, which is 1, and the final slope of the curve, which is 0 (the intersection of the two asymptotes). The results show the sensitivity of the performance of these systems to the cache miss rate and the ratio of main memory speed to processor speed.

Reference rate is a more precise term than miss rate: it is the probability that a main memory reference will be generated by a single processor in a single machine cycle. The formula for the saturation point (in numbers of caches) is

$$1 + \frac{(1 - \text{reference rate}) \times (\text{cache service time})}{(\text{reference rate}) \times (\text{memory service time})}$$

If we assume that the memory service time is ten machine cycles (a reasonably aggressive value), then the results shown in Table 1 are obtained. This indicates the importance of a high cache hit rate, which is the major determinant of the reference rate, to the overall performance of the multiprocessor. It also makes clear the importance of the service time of the main memory system.

The most obvious source of main memory references are requests to satisfy cache misses. Synchronization and cache consistency traffic increase the reference rate above the simple cache miss rate, because this traffic uses the shared bus. With very large caches, interprocessor communication may dominate the bus traffic (*7*). Hit rates better than 98 percent should be attainable on a multiprocessor if multi-

Table 1. The saturation point as a function of the reference rate when the main memory service time is ten machine cycles.

Reference rate	Saturation point
0.01	10.9
0.02	5.9
0.03	4.2
0.04	3.4
0.05	2.9

programming and operating system interference can be reduced. The percentage of the traffic not originating from cache miss requests is not well understood and is probably program-dependent. Because performance is so sensitive to the reference rate, a processor-to-processor data exchange mechanism that requires one or more main memory service times and is heavily used will result in low multiprocessor performance.

If the memory system is pipelined, then the memory service time in the formula should be the reciprocal of the memory service rate rather than the latency. Thus, if pipelining the memory system can double the service rate, it nearly doubles the saturation point. Electrical bus loading constraints of current components make it difficult to build buses that are fast and support more than about ten connections. Since the saturation point analysis indicates a similar numerical constraint on the number of caches, small multiprocessors are a good engineering design point.

Interprocessor Communication Support

The first problem to overcome in designing a shared-memory multiprocessor is to create an efficient way for processors to share and communicate data. The previous section demonstrated that, if the bus is the only way for processors to communicate, then, as soon as they start trying to share substantial amounts of information, the bus and main memory become saturated. So some systems provide a separate mechanism for processors to synchronize and exchange data. Although a number of multiprocessors have been built, there has not been sufficient experience to evaluate all the alternatives to using a shared main-memory bus. The choice of the best communication mechanism is further clouded by differences among applications and the match between particular communication mechanisms and particular applications.

The major advantage to communicating via shared memory is that asynchronous communication is easy. The shared memory provides an almost infinite buffer between two communicating processors and allows a less rigid form of communication than a completely synchronous communication channel.

It is desirable that the processors have a consistent view of the shared memory. A cache coherency mechanism guarantees that the caches always reflect the most recent value of every memory location. The most common hardware cache coherency strategy is a snoopy cache, which watches the traffic on the shared bus (as well as from the processor) and tries to maintain a consistent view of shared memory by tracking updates to the memory made by other processors (*8*).

Use of a separate interprocessor communication bus may, it is hoped, relieve some of the traffic on the shared-memory bus. The major disadvantage of this separate bus is that it must be synchronous and communication must follow a rigid protocol. However, some traffic (such as scheduling requests and synchronization activity) may naturally fit this synchronous model. Of course, the separate bus removes some of the attractiveness and simplicity of the simple shared-memory model. But, if software that can use this bus in a programmer-transparent fashion is developed, then a separate bus will be an advantage in reducing memory bus contention.

The value and role of each of these communication mechanisms are not well understood. However, it appears that the choice of the mechanism can significantly affect performance. In some cases, the best choice will vary depending on the application.

Dividing a Single Task for Parallel Processing

Even if there is an efficient communication mechanism, we still have to worry about how to partition the problem. Some problems have no parallelism. Given a problem with parallelism, we must still decide how to decompose it into separate pieces, each to be executed by a single sequential processor. It would be ideal if there were no dependencies among the pieces, but usually there are, and the dependencies force a certain amount of sequential execution of the pieces. The object is to get as many of the pieces as possible to execute independently in parallel.

The challenge in this decomposition task is to balance the communication and computational requirements of each piece with the capabilities of each processor. The larger the granularity, the more computation each piece includes. For example, consider the partitioning of doubly nested DO-loops that is done in the Cray-XMP FORTRAN compiler (*9*). The XMP allows several processors to be connected for parallel execution of a task. In the best case, the outer loop can be split among the processors and the iterations of the outer loop are independent, that is, no data need to be transferred among the processors in executing separate iterations of the loop. The existence of the inner loop provides each processor with enough work to justify the overhead of splitting up the computation and resynchronizing the processors at the end. If the iterations of the outer loop had dependencies that required interprocessor communication, a different partitioning, perhaps involving fewer processors and larger granularity, might be better.

Different types of multiprocessors have different grain sizes that they can accommodate efficiently. For example, dataflow machines provide extensive hardware support for low-overhead communication and scheduling, making small grain sizes practical. The class of machines discussed in this article must achieve a delicate balance between communication cost and computation. In general, communication of a data item will require about an order of magnitude more time than a simple machine cycle on one of the processors. Because the number of processors is small and the communication overhead is high, partitioning the program into larger grains with less communication among the pieces seems prudent.

Another phenomenon motivates us to be cautious in our choice of a larger grain size: the predictability of the grain size. Whether the partitioning into parallel tasks is done by a program or by a programmer, the computation cost of a piece must be estimated. If this cost is estimated poorly, performance can be lost because the communication-computation balance is disturbed. Another problem that can occur from incorrect estimates is an inefficient scheduling of the parallel pieces. For example, assume that we partitioned a problem into n components to be executed on n processors, with the result from all processors needed before the computation could proceed. If we underestimated the size of the one of the tasks, it would become the bottleneck. Accuracy in predicting the computational requirements of a piece of code decreases as the pieces get larger, simply because the larger pieces contain more conditional activity and their behavior is less predictable.

When we partition a task, we must be careful to examine the total cost of making the task parallel. Parallel partitioning often adds to the computation work in an amount proportional to the number of subcomputations. Most real problems require additional computation to resolve the subcomputations done for each partition. For example, in a parallel simulation, this might involve reconciling of data among partitions or decisions to alter the time step. Typically, these portions of the computation have a much lower degree of parallelism. If we achieve a tenfold speedup for 90 percent of a computation but 10 percent of the computation remains serial, we have actually achieved a fivefold speedup. Great care is required to ensure that this more sequential interaction does not become the bottleneck in an attempt to exploit parallelism.

Tools for Parallel Programming

We can build tools to help decompose tasks for parallel execution. There are three basic approaches to obtaining parallelism: using a sequential language with explicit parallelism directives, using a sequential language and relying on a compiler to extract parallelism, and starting with a language that is inherently parallel. Each of these approaches presents different challenges and has different advantages.

To express parallelism, we could use a conventional programming language with directives that specify a set of separate processes, each a potential task for a processor and each a sequential program. In addition, we need to have "primitives" to control the interaction of these processes. Without these synchronization primitives, the processes may interact in an unforeseen manner, yielding incorrect, and even unpredictable, results. This approach has been the main technique used to organize operating systems for 20 years, and its main advantage lies in its simple implementation and well-understood behavior. Programming languages such as Ada and Concurrent Pascal support this style of programming directly, and other languages support it through operating system calls that create processes and synchronize them. The disadvantage of this approach is that it requires the programmer to establish the number of parallel tasks and explicitly synchronize their interaction, thereby limiting the ways in which parallelism might be used. Since each parallel activity must be clearly delineated, this approach is more attractive for large-grain parallelism with a fairly low degree of information sharing among processes.

A second approach is to extract parallelism from programs written in existing languages by using compiler techniques. Kuck *et al.* (*10*) have been using this approach on FORTRAN programs with success for many years. Clearly, extracting parallelism from programs in languages that programmers know and have used is a major advantage, especially for existing software that need not be rewritten to take advantage of a multiprocessor. The main disadvantage of this approach is that the amount of parallelism that can be successfully extracted and usefully exploited is usually limited. These techniques are most successful in extracting low-granularity parallelism and at finding a particular style of parallelism, such as a vector operation. Because of these limitations, automatic parallelism extraction is currently most feasible for scientific programs that are amenable to vectorization. By concentrating on local, low-granularity parallelism and on structured parallel operations, the problem of translating these tasks to appropriate hardware is straightforward. However, exploiting other types of parallelism or compiling for machines that do not support the appropriate structured operations may be difficult and will not yield much improvement. Many applications will not obtain any performance increase from a multiprocessor with this approach to parallelism; for example, a logic simulation program, which is inherently highly parallel, may not benefit.

The final technique that some researchers are pursuing are languages that are fundamentally parallel. These languages vary from extensions to LISP to parallel versions of PROLOG (a logic-based programming language) to single-assignment languages (value-oriented languages). The aim of all these approaches is to provide a language that allows the programmer to convey the parallel structure of the program. The task of the compiler in such an environment is to decide how much parallelism to attempt to use (as in how big the grain size should be) and how best to map the parallel structure to a particular multiprocessor. These approaches face a variety of challenges in the software area, including finding a way to compile such programs efficiently, determining how and on what basis to set the grain size, and finding algorithms for efficient scheduling and processor assignment. Because these languages are less well understood and their translation to a multiprocessor involves more automation and less human direction, we suspect that effective schemes for exploiting this approach will take longer to develop. The reward these schemes may bring is a level of machine-independence in the programming process that exceeds what is obtainable with the other approaches. Compilers could partition the problem, depending on the characteristics of the program and the architecture.

Related Efforts

Although the small, general-purpose multiprocessors may represent the most general style of multiprocessor, other approaches that have advantages in certain aspects or application areas are being pursued. Many of these efforts are aimed at the major shortcoming of the shared-memory multiprocessor: the ability to expand. To maintain expandability, large-scale multiprocessors must usually rely on a local memory architecture and communicate between processors by passing messages through a point-to-point interconnection network (*11*). An alternative approach with limited expandability is the use of a large interconnection network that can allow hundreds of processors and memories to communicate simultaneously (*12*).

Most of the large-scale machines are organized as a regular interconnected set of processor-memory modules (*13*). The massively parallel machines described by Gabriel (*14*) are such machines. The Cosmic Cube (*15*) is another example of a machine in this class; it consists of hypercube-connected processor modules, each module containing a microprocessor, a floating point chip, and its own local memory. In general, programming such machines is more difficult than programming a smaller scale, shared-memory processor. However, application-oriented programming systems could be developed that would ease the programming of larger machines, at least for certain classes of applications.

Related to the topic of general-purpose parallelism is the issue of vectorization. Vectors represent one of the simplest forms of parallelism (since there is only one instruction stream); vectorization is also attractive because it fits the computation paradigm for many scientific problems, and automatic vectorization is a viable approach. The recent development of floating-point processors with vector support will probably lead to even more multiprocessors and uniprocessors with vector hardware.

Conclusion

The combination of microprocessor and memory technology with compiler technology for parallel decomposition of single-stream programs makes it seem that small multiprocessors will have an important place in the laboratory computing environment for many scientists and engineers in the near future. While these systems are 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.