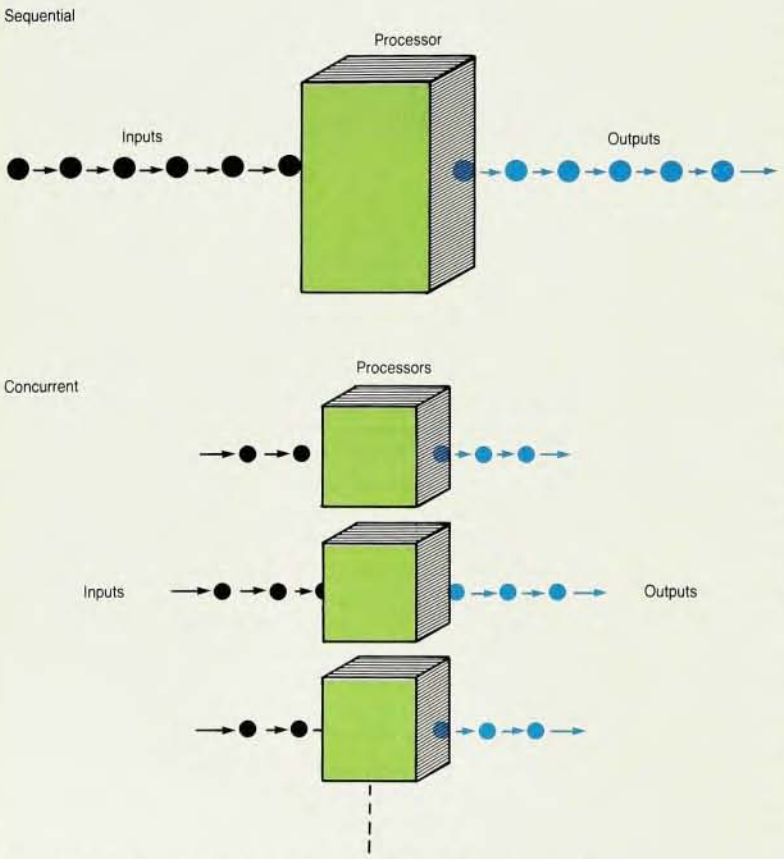
Parallel architectures for computer systems

Having several parts of a system simultaneously perform different parts of a task is an old notion that is proving more and more useful in the design of powerful computers.

James C. Browne



Sequential versus concurrent (or parallel) computing. A sequential computer works on one operation at a time, converting input information into output information; the processes it performs are governed by a program, which forms part of the input. The output can, of course, be used as input—data or even program information—at a later stage. A concurrent processor consists of several sequential processors working on separate streams of input; outputs from one processor can serve as inputs for that processor or for another processor in the system. Figure 1

People frequently do more than one thing at a time: Driving a car while listening to the radio, cooking a meal so that several dishes are ready at once, or playing two lines of melody on a piano are all familiar examples. On a larger scale, many human activities, such as building a house or complicated experimental apparatus, or putting out a magazine, are separated into what might be called "units of activity" that are performed separately by people working in parallel. On a smaller scale, our brains control separately—but in a coordinated fashion—breathing, heartbeat and several different kinds of motor activity. In each of these cases, separate units of activity are carried out by separate processors (different people or different parts of the brain, for example) that work simultaneously (at the same time, but not in lockstep) and interact to produce the final effect or product.

The traditional view of computer design, however—as promulgated by such theorists as Charles Babbage, Alan Turing and John von Neumann, and as built into computers ranging from ENIAC and UNIVAC to today's pocket calculators—is based on a single central processor performing operations sequentially to produce the desired result. In this scheme, computers that appear to perform several tasks simultaneously only do so by switching between tasks so rapidly that the user is not aware of the switches.

The distinction is, of course, not absolute. At the most elementary lev-

James C. Browne is professor of computer sciences and physics at the University of Texas at Austin.

0031-9228 / 84 / 0500-0028 / $01.00

el, even the most rigidly sequential of computers performs some tasks in a parallel: Only the purely abstract Turing machine performs its tasks one bit at a time; most computers deal with at least eight bits in parallel. And in many cases, some units of activity of a task must be performed in a specified order: You can't make the omelette until all the eggs are open. The difference is in what is called the architecture of the computer, the way it is designed to perform tasks: Does it perform several "units" of activity at once, using interacting processors, or does it use a single processor to perform one unit of activity at a time? (See figure 1.)

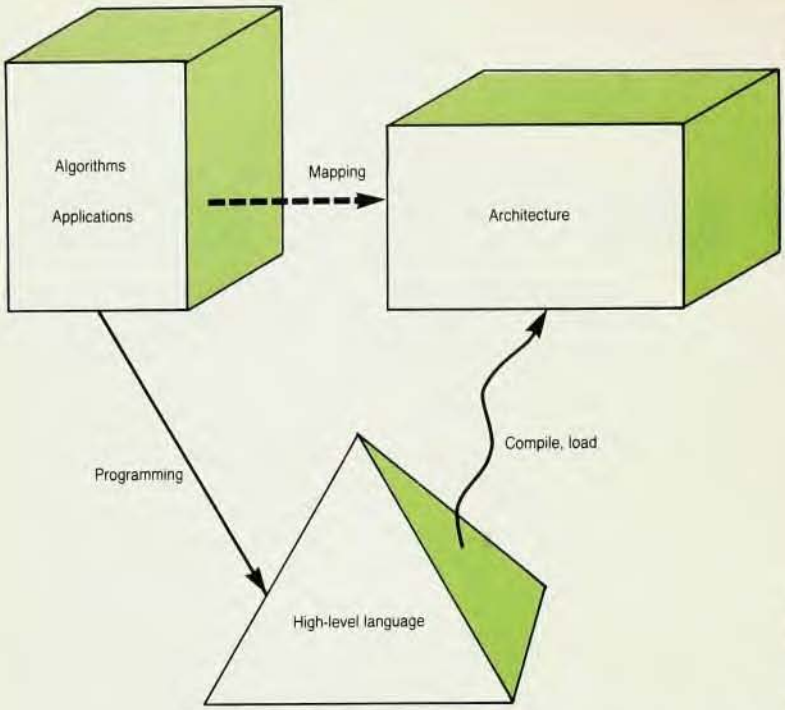
Parallel architectures are not a new idea.[1] The British cryptanalysis machine Colossus is reported to have used some parallel computations, and Vannevar Bush described some proposals along these lines in his 1945 report *Science, The Endless Frontier*. In the 1950s, groups at IBM, the University of Illinois and elsewhere were working on designs for parallel structures for numerical computations[2] and for pattern recognition.[3] Daniel Slotnick and his collaborators at IBM, at Westinghouse, and (since 1964) at Illinois produced proposals for several general-purpose computers based on parallel architectures[4]—Solomon in 1962 and Illiac-IV in 1968. Illiac-IV was designed to consist of four sets of 64-processor arrays, each array carrying out a separate stream of instructions. Because of engineering problems (its design was too advanced for the available technology), only one array was built; it was dismantled a few years after being commissioned in 1973. The simple, regular structure of signal-processing applications motivated the development of several early parallel computers, such as the Burroughs Corporation's Parallel Element Processing Ensemble (for radar tracking; built in 1972), and the Goodyear Aerospace Corporation's Staran (for image processing; several were built, starting in 1972). In England, International Computers Ltd. built the Distributed Array Processor (first operational in 1977); it contains an array of 64×64 processors, each of which operates on single-bit operands, but which operate together on multiple-bit operands. In the last few years machines have become available that perform several different operations in parallel—rather than the same operation on several sets of data. The Heterogeneous Element Processor built by Denelcor is probably the first commercially available general-purpose computer that can perform several operations concurrently.

The operating systems of today's multiprogrammed computer systems are parallel programs—often of immense complexity. The high-performance mainframes of today have much internal parallelism that is rarely visible from the user's level. In these CPUs the internal parallelism includes overlapping the execution of individual instructions and parts of instructions. For example, the vector architectures that dominate today's very-high-performance computers incorporate processors that operate on vectors by performing simultaneously the required operation on several elements, or components, of the vector. Furthermore, because one knows in advance the sequence in which operands are required by the program, one can arrange to fetch them in advance and present them to the CPU in order. Such operations can be performed while the CPU is working on other data—a special kind of parallelism called pipelining.

Potential power

Not only will parallel computation have a direct impact on science by increasing the computer power available for focusing on a given problem, it will also, by reducing the cost of large-scale computations, broaden the scope of problems to which computer-based techniques can be applied. Furthermore, I believe that the concepts of parallel structures and the ways in which algorithms and programs exploit those structures will have an effect on the way in which problems are formulated in the first place. The computer may, in fact, become as deeply involved in the formulation of problems as in their solution.

At present, one of the chief functions of high-performance computers in science is to implement—through evaluation of mathematical models—experiments on physical systems; in many cases such computer simulations are required because the real experiment would be too slow, too costly, too idealized, or impossible to realize. Thus, for example, one can model plate tectonics to obtain insights that might have taken millions of years of experimental observations; one can model the behavior of plasmas in a fusion reactor, under varieties of conditions that would require enormous resources to produce experimentally; one can model the behavior of spins in an Ising model, avoiding interactions and other complications that would obscure the relevant effects in a real system; and one can



Models of computation are structures inherent in any description of a computation. An algorithm provides a description of steps to be taken in the computation; it can be mapped onto a computer architecture whose processors and structures perform the calculation. In practice, the calculations are described in higher-level languages, in which the algorithm is represented by a program; a compiler binds the program to the architecture of the machine by expressing the program in machine language. Figure 2

model the behavior of objects near a black hole.

The more powerful the computer system, the more effective it is as an "experimental" apparatus, and the more complex are the models that it can investigate. In today's world, the degree of resolution or the complexity of the model is often severly limited by the capabilities of the available computer systems. There is currently no visible limit to the computer power that could be utilized for these problems. Clearly, while we may have reached in some cases a diminishing ratio of knowledge returned per cost of computation on today's computers, we have not yet reached a diminishing rate of return for knowledge gained from increased resolution and complexity of the models investigated.

There has historically been a substantial rate of increase in computer power, of the order of a factor of 4 or so with every new generation of high-performance computers. Each generation of these computers has also had a more or less constant cost (in real—uninflated—terms) for a typical system. This enhancement in power at more or less constant cost has come by increasing the speed of the components and by increasing the internal parallelism of computer architectures. (One should not forget the role of better algorithms as well: Frequently the improvement in the algorithms for sequential computations have created improvements in effective computer power greater than that supplied by improvements in components and architectures.) The article by Charles Seitz and Juri Matisoo in this issue (page 38) analyzes the state of component technology and explores the factors that limit development of high-performance components.

Components at the top end of the performance spectrum have not increased their speed dramatically in recent times, and while new technologies such as gallium arsenide promise to give higher speeds, revolutionary increases are not to be expected. The increase in speed one can gain by internal parallelism in the computer also appears to be slowing now, largely because of the limited span of opportunities for parallelism in the hardware, because of limits in software that make use of the internal parallelism, and because users are limited in the ability to formulate algorithms using the internal—and generally invisible—parallelism. We must now turn to architectures with explicit, problem-visible parallelism to bring about another sharp increase in absolute speed and in the ratio of power to cost.

The potential for improvement in the ratio of cost to performance, while not as obvious as the potential for improvement in raw computer power, is also considerable. Current high-performance computers are specially designed and custom-built using special high-speed components. These components are seldom used in volume, and their unit cost is very high. The potential increase in the ratio of performance to cost for parallel architectures arises from the fact that the computational elements of parallel architectures can be built from standard components. Because they are used in large numbers, these components may turn out to be very inexpensive. High-performance parallel computers may eventually be built by connecting together a large number of identical simple cells into a regular network. Such a manufacturing process is potentially subject to automation. The kind of leverage available through replication can be seen by considering that a single computing element capable of perhaps one million instructions per second, complete with adequate memory, can be built for perhaps $500 to $1000; it should thus be possible to build a computer than can perform 100 000 000 instructions per second for only hundreds of thousands of dollars. Of course, many very real practical factors still stand in the way of realizing this computational nirvana.

Models of computation

To understand parallel computer architecture, we need to discuss in very general terms what computers do:

- What are the primitive units of computation that are directly executed?
- How are these primitive units combined into larger tasks?
- Where do the inputs and outputs of the primitive units come from and go to?
- How are the operations of the computer organized in time?
- How do the processors that execute the computation communicate with each other and what is the topology of their connections?

Any model of computation must provide answers to these questions. In the case of the familiar sequential comput-

er, the primitive units are arithmetical operations, such as + or −; they are combined by performing them sequentially; the data on which operations are performed come from, and go to, memory registers; the operations are performed sequentially, with all operations timed by a master clock; and the interconnections are trivial, as there is only one processor. Today's high-performance computers, of course, embody much more complex models of computation.

Every algorithm defines an intrinsic model of computation, in that there is a computer architecture to which it can be directly mapped. Conversely, the class of algorithms a computer can execute effectively is determined by the model of computation implemented in its architecture.

Among computer architectures one can distinguish several basic varieties:

SISD: The computer applies a single stream of instructions to a single stream of data (as in a typical sequential computer).

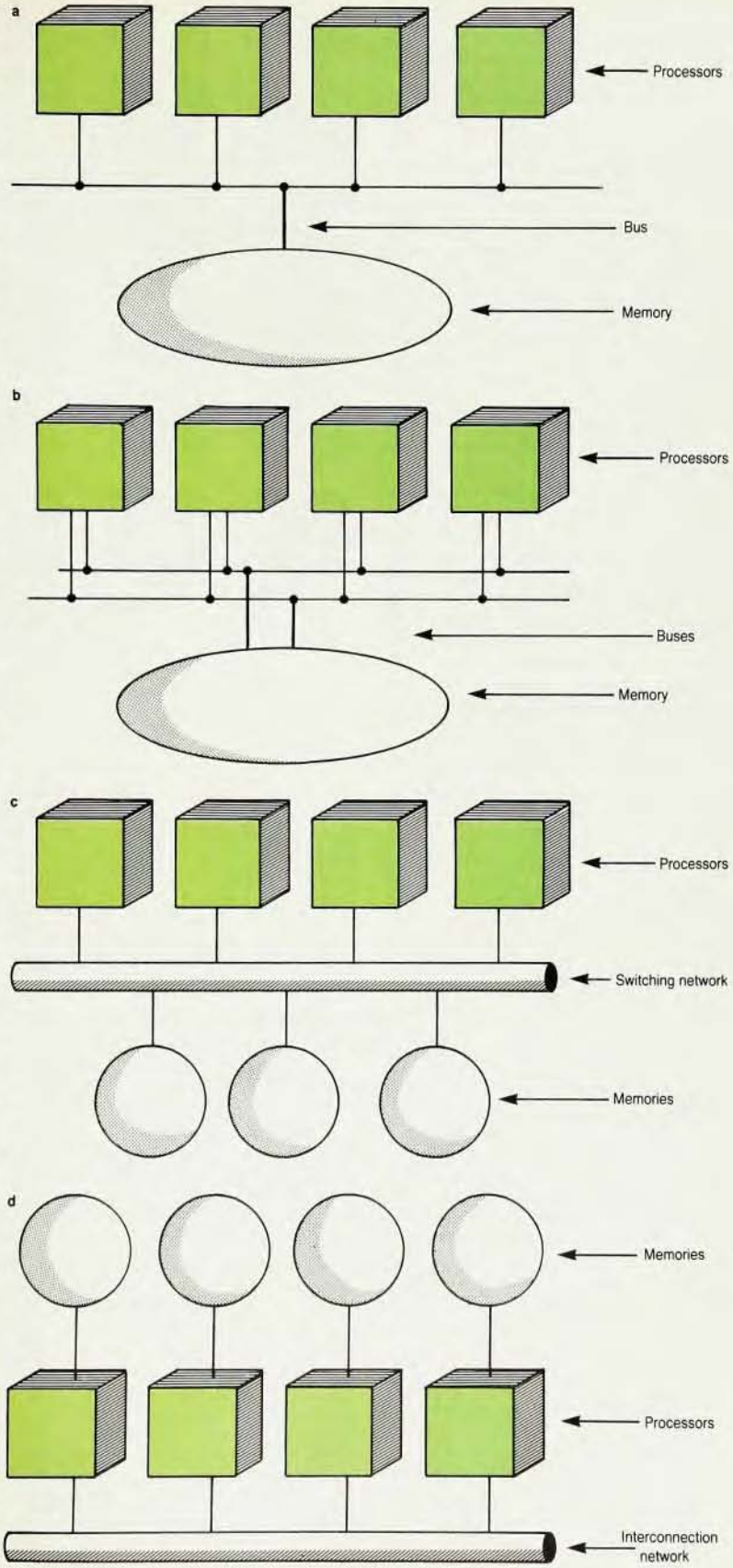
SIMD: The computer applies a single stream of instructions to multiple streams of data.

MIMD: The computer applies multiple streams of instructions to multiple streams of data.

The processors that carry out primitive units of computation are SISD systems; these can then be connected together, with appropriate sequencing and synchronization mechanisms, to form the more complex systems.

There are several ways to ensure that computations are performed in the proper order and at the proper time. With explicit synchronization, the instruction streams are regulated either by an external agent or by a mutual protocol. With implicit synchronization, instruction streams are regulated by the flow of data; a processor either waits to perform its computation until all the necessary inputs are ready, or it waits to call for inputs until the results of its computation are called for elsewhere. Processors can communicate either via switched circuits or via permanent circuits.

Today's high-performance computers generally include some parallel structures because their multiprogramming operating systems are highly parallel programs, whose unit tasks for parallel execution are the activities



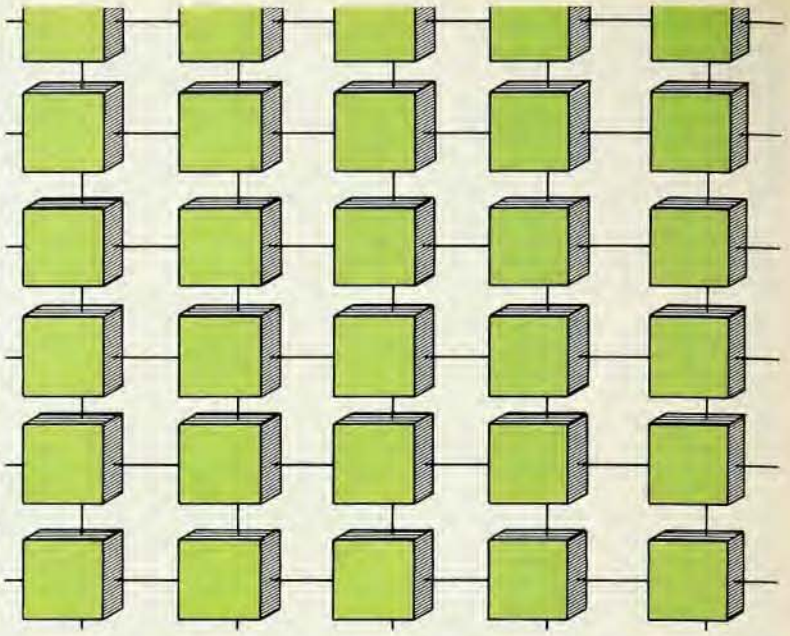
Parallel architectures can be classified according to connections between processors and memory: (**a**) single bus connecting processors to a common memory; (**b**) multiple buses connecting processors to a common memory; (**c**) multiple processors connected by a switched network to multiple memories; (**d**) processors with internal memories connected into a network. Figure 3

of separate user programs. To allow these prarallel programs to execute effectively, the architecture must provide capabilities for synchronization.

An example of the sort of problem faced by parallel processors is raised by the use of shared memory. It is vital that two different programs—or parts of a program—not simultaneously alter the value of a specific set of data, otherwise the symbols on which the computer operates would lose their consistency or meaning. A single address, in other words, must always refer to a single item, although its value may change in time. Most operating systems now avoid the problems using a TEST AND SET or TEST AND RUN operation. To use such an operation, each unit of memory carries a coordination element that can be set to either "busy" or "free." A process that requires shared data first executes a TEST AND SET operation. If the value of the coordination element is "free," it simultaneously returns that value and sets it to "busy." The process can then use the shared data; when it is done it resets the coordination element to "free"so other processes cn use the data. If the value is "busy," the process executes another TEST AND SET operation later. (The operation is, in fact, much like the operation of "busy" signals for telephones.)

Computer scientists have developed a large number of such mechanisms for coordinating processes through the use of shared data.[5] Operating systems and data-base systems make use of these mechanisms. However, the mechanisms will probably not be effective for the management of large-scale, user-level parallelism because they do not scale up to the management of a large number of processes, nor do they take account of the topology of communication. In fact, many of the problems faced by programmers for parallel architectures will be different in kind from those solved by the currently used mechanisms, such as the TEST AND SET operation. The several processors may, for example, each have an independent memory; the coupling between processors would thus not be via a shared memory but via explicit communications channels between processors. The synchronization and sequencing methods for such architectures must be developed specially for each type of system.

Typical of the sort of problem that can be efficiently solved by parallel structures is the solution of partial differential equations. In general, the equations are discretized into linear equations, and the solutions are found by iterating these step-by-step from the initial or boundary conditions. One possible way to proceed is to perform each of the operations on a given row of the set of linear equations using a separate computer. One architecture that can effectively execute such a parallel algorithm is one with nearest-neighbor connections between the computers and enough memory on each computer to store a full row of the array. The synchronization mechanism can be the notification of readiness to exchange data.



A mesh of processors. A fixed network can connect processors, or processors and memories, with each other in a multidimensional array. Usually only nearest neighbors are connected, as in this two-dimensional array. Figure 4

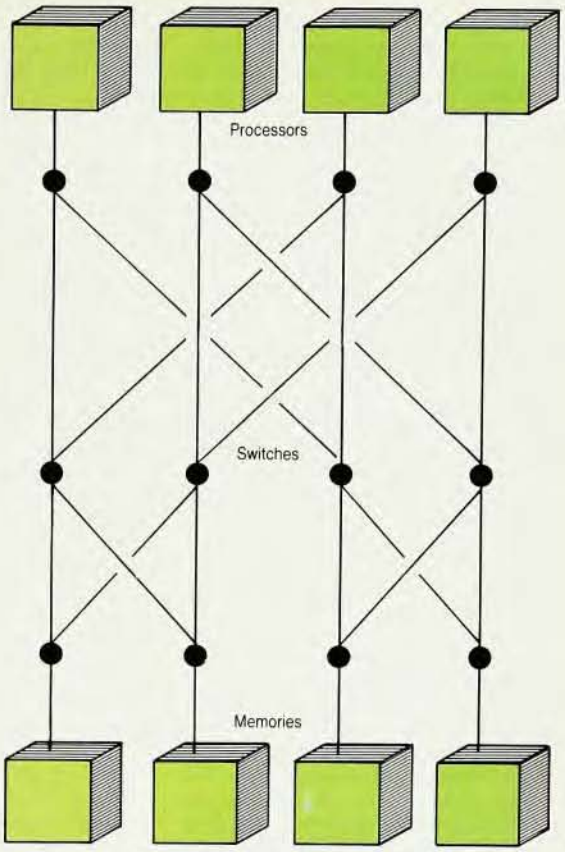
Languages and architectures

As I have mentioned, each computer architecture implements efficiently some class of algorithms—those algorithms that map onto the model of computation embodied in the architecture. However, the programmer rarely deals with the machine-language details of the execution of a program. Nearly all programming today is in some higher-level language, which, in turn, is compiled into machine instructions for executing the program.

Each higher-level language also embodies a model of computation. For example, FORTRAN specifically implements in its model of computation the execution of logical and arithmetic operations on scalars. Operations on larger arrays, such as vectors, must be explicitly assembled by the programmer from elementary operations; FORTRAN procedures, such as DO loops, are operations for composing larger operations from smaller operations.

Figure 2 illustrates the relationship between algorithms, architecture and languages. A given problem is formulated as a sequence of algorithms. Each algorithm defines some model of computation. The algorithms are expressed in a programming language, which itself embodies some spectrum of models of computation. The programs are then compiled or translated into a program for an architecture, which also implements some model of computation. This series of mappings introduces additional complexity into the process of solving computational problems. The more disparate are the models of computation of the algorithms, the languages and the architectures, the more complex are the mappings and the more difficult their expression becomes. James McGraw debates with David Kuck and Michael Wolfe on the best way to design higher-level languages to respond to these problems in their articles on pages 66 and 67.

The current programming languages available for common use in science offer very little in support of parallel structuring. New versions of FORTRAN are being proposed that include vector and parallel constructs. Manufacturers who have built computers containing parallel architectures have explicitly included parallel structuring capabilities in the compilers for some of



Switching network. An external program can set the switches so that any processor can be connected to any memory. Alternatively, by carrying an appropriate address, a packet of information traveling through the network can go between any processor and any memory. In the network shown here a single binary bit can switch each node to the appropriate state, so that the address and switching instructions can be one and the same. Figure 5

their machines. The vector extensions in the FORTRAN languages for the Control Data Corporation Cyber 205 and the Cray Research Cray-1 systems are examples. Another example is the parallel FORTRAN for HEP, which has a MIMD architecture. There are modern programming languages, such as ADA, that do include explicit parallel structuring. These languages, however, tend not to be a part of the professional background of the working computational physicists. Nor are these languages particularly well suited to the expression of parallel mathematical models of physical systems.

Connecting the elements

Both the primitive operations performed by the individual processors in a parallel architecture, as well as their organization into a network, determine the properties of the resulting computer system. In fact, the dominant element in determining the properties of a parallel computer can be the way in which the processors communicate and synchronize their operations. The individual processors in a brain (the neurons), for example, compute fairly primitive functions; the computations performed by the entire network are quite sophisticated, as anyone who has used one can affirm.

Figure 3 shows the kinds of structures that have been proposed for parallel architectures: Individual processors can be connected to a common memory by a single or multiple data bus; processors can be connected to multiple memories by a network of connections; or the individual processors can each carry their own memories, with interconnections directly among processors. Each of these schemes has advantages and disadvantages. The single common bus is simple and straightforward, but for large systems it is easily overloaded. The multiple bus is less subject to overload, but each processor's access to memory may still be degraded by conflicts with other processors. The switched network reduces the conflicting-access problem, but at the cost of a large switching network and the programs to control it.

The next obvious step is to construct individual computer systems, each complete with its own memory, and connect each processor to some of the others in the system. The common choice is to connect the processors to their nearest neighbors in a multidimensional array. The individual processors of Illiac-IV, for example, were each connected to their four nearest neighbors in a planar array, like that shown in figure 4. Arrays in three or more dimensions are, of course, also possible. Geoffrey Fox and Steve Otto discuss such interconnections in their article on page 50. A problem can arise with such systems when one assigns computations to processors, because the communication between computations required by the algorithms may not map straightforwardly upon the nearest-neighbor connections implemented by the architecture. The time taken by the communication among processors can then become a major factor, dominating the total execution time. The mapping of common algorithms of linear algebra onto simple interconnection structures is an active topic of research.[6]

To avoid both the material cost of establishing permanent connections between all processors in a system of the kind shown in figure 3**d** and the cost in execution time arising from mismatches in connectivity between the algorithms and the hardware, some designers have proposed replacing fixed interconnections with a switchable network. These networks may be based on multilevel structures, such as the one shown in figure 5, or on the creation of circuits across a mesh. However, such systems are still limited in that they may not in fact be able to provide all the connections needed to perform some algorithm efficiently. They also require extra programming effort to manage the switching network.

An additional element to be considered in the connections among proces-

sors is how to keep them in step. For systems in which processors share access to memory, the processors must all be synchronized, and their operations properly sequenced. Usually the synchronization and sequencing are under the control of the operating system of the computer. However, as I mentioned earlier, it is also possible to design architectures that do not work in lock-step. Rather than controlling the sequence of operations explicitly, the processors can be made to wait to perform their computations until they have all the necessary information to proceed (or until another part of the system requires the result that they can provide). Such data-flow systems[7] typically connect processors to "source" and "sink" memory units, where the operations and their inputs are assembled.

Real systems

Computers with parallel architectures are being realized. There are over fifty proposals for parallel architectures from university and government laboratory research projects in this country alone. Companies have been formed to develop and market parallel computer systems. Several major computer companies have research projects aimed at building systems with parallel architectures. Parallel architectures are a major element of the Microelectronics and Computer Corporation, a research consortium formed by about a dozen US computer and electronic firms (see PHYSICS TODAY, July, page 65). The famous Japanese "Fifth Generation" and "Supercomputer" projects are focused on developing parallel computer systems that will be effective in symbolic and numerical applications, respectively. There are also similar national projects in several European countries. (See the news story on page 61.)

Commercial architectures. Several major computer companies offer systems containing several CPUs working from a common memory (as in figure 3**a** or **b**). These systems are not truly parallel architectures because they generally do not support direct use of multiple processors on a single logical computation. Rather, the several CPUs are intended to allow the execution of a larger number of distinct programs in a given time—that is, to increase total system throughput.

The best-known currently available system that can execute multiple instruction streams for a single problem is the Heterogeneous Element Processor, manufactured by Denelcor, Inc. Several HEP systems are being used for experimental research in parallel formulations of problems of physics. A single HEP system may have up to 16 CPUs, each of which can use multiplexing to work on up to 100 active processes or instruction streams concurrently. The HEP is programmed in a version of FORTRAN extended to include elements of a parallel model of computation. The operating system uses an extended form of the TEST AND SET instruction to regulate the use of shared data by the instruction streams. To connect processors to memory elements, the HEP uses a network switch based on a packet routing strategy; the details of the switching arrangement are a commercial secret.

Research projects. Because I cannot describe all of the many research projects fully enough to do them justice, I will discuss only a few that illustrate the concepts I have discussed.

The ULTRA project[8] at New York University is an example of an architecture that uses a shared memory structure. The ULTRA is planned to contain some 4000 processors connected by a multi-level switching network to the memory, which also has some 4000 units. The network is "packet switched" rather than "circuit switched," so the connections are established only as needed; the nodes carry embedded processors (with memory) to resolve conflicts among requests for the same unit of memory from several processors. Instead of a TEST AND SET instruction, which—for 4000 processors—would produce unacceptably many "busy" signals in accessing memory, the ULTRA uses a FETCH AND ADD instruction. Such an instruction retrieves a value from memory and simultaneously replaces the stored value with a sum of the stored value and a value carried with the instruction. This instruction forms a central part of the operating system, for example in the organization and maintainance of queues of jobs and data. The switching nodes of the networks are designed to let several processors FETCH AND ADD simultaneously, thereby avoiding busy signals entirely. The ULTRA group has used computer models of these ideas to test them in applications to a variety of problems in mathematical physics.

The Texas Reconfigurable Array Computer, being developed[9] at the University of Texas at Austin, represents a different approach to a network architecture. Here the interconnections between processors and memory units are established by a dynamic circuit-switched network. That is, the connections act exactly as fixed buses during their existence, but the circuit paths can be created and destroyed at any time. (One can compare a circuit-switched network to a traditional telephone system, with connections made and broken as required for exchanging information; packet-switched networks behave more like a mail service, with data carrying addresses to route the packet through the system.)

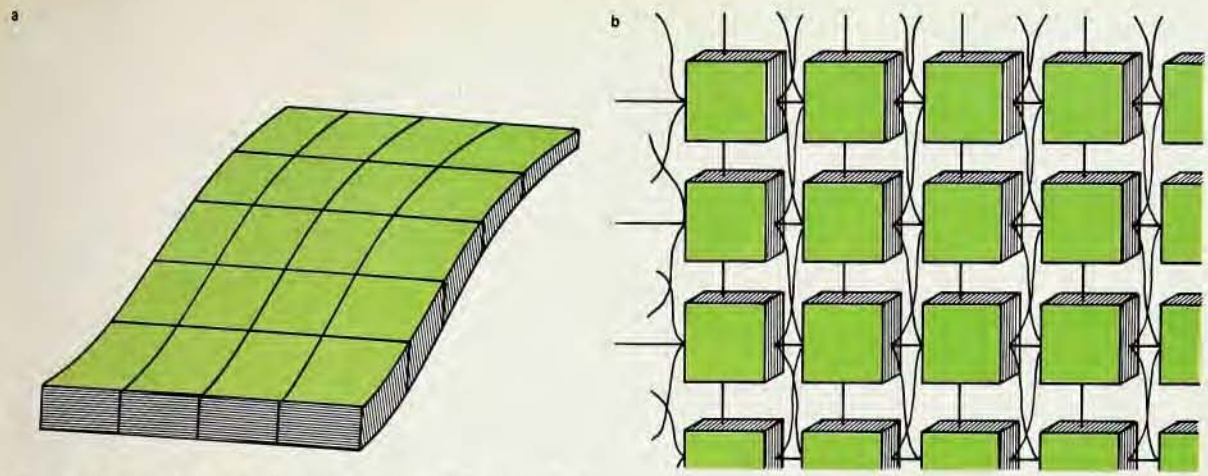
The TRAC network is of a form called a switchable tree, connecting many memory units at the "roots" to many processors at the "leaves." Because of its configuration, it is often referred to as a "banyan network." The structure allows memory units to be switched between processors in a few memory cycles, providing a communication system with very high bandwidth that also does not degrade access to memory units that are not being shared. The network avoids conflicts in access to memory, but still offers a spectrum of topologies for communication.

In addition to such switched-network machines, there are systems that incorporate fixed networks. The effectiveness of such systems, of course, depends very much on whether the particular topology of the network is appropriate to the calculation being performed. But because fixed networks are much easier to design and much less expensive to build than switched networks, they are clearly the most cost-effective architectures for situations in which algorithms arise that generally match the topology of the network. Such processors are, in a sense, intermediate between the switched-network architectures and the special-purpose computers, whose processors and connections are specifically designed for one class of problems.

Specialized processors. Some problems in mathematical modeling of physical systems have a simple and readily visible model of parallel computation. In some of these cases the computations are repeated very many times for different values of variables of parameters, but without changing the model of computation. These circumstances suggest developing special computers with architectures that embody as exactly as possible the precise model of computation required by the application.

A few such devices have been built. Jorge E. Hirsch and Douglas Scalapino describe one such device, for performing Monte Carlo simulations of the three-dimensional Ising model, in PHYSICS TODAY, May 1983, page 44. Because of the regular structure of their algorithms, signal and image processing also clearly lend themselves to specialized parallel architectures.

Another application for specialized parallel processing is structural mechanics and mechanics of continua. A typical calculation may involve determining the deformation throughout a body or structure when an external force is applied to it. By dividing the body into small cubes, say, one can turn the continuum problem into a discrete one (figure 6). The deformation of any one cube is determined by its properties (elastic and structural) and by the



Structural connections embodied in an architecture. Each cell in the solid at left is acted upon by forces from its neighbors. The computer at right contains the same nearest-neighbor connections, so that it can readily compute the response of each cell (modeled by one of the processors) when the solid is subjected to an external force, for example. Figure 6

forces exerted on it by its nearest neighbors. The mathematical model of this structure can be resolved into a set of linear equations, with each variable appearing only in equations that also contain variables for the neighboring elements. A computer that embodies this very simplified model is easy to construct: Assemble an array of ordinary microcomputers; connect each element in the array to its "neighbors" so that the network reproduces the neighborhood relations of the original system; let each microcomputer query its neighbors for relevant data to compute the values of the structural parameters for one element of the original system. All processors will then be solving the linear equations in parallel to give the response of the system to the external force.[10]

Because the cost of designing and building special-purpose electronics is dropping rapidly, we can expect to see many such computers in the future. At this time, the principal bottleneck appears to be one of getting the people who need the special-purpose computers together with the people who can design them.

Future architectures

As I have indicated, communication among the various elements of a parallel structure is one of the major features to be considered in the design. The communication itself requires computational work that must be done in addition to the computations that would have to be performed in a purely serial machine—unless, of course, the computational work has already been done in designing a fixed network for the connections. The finer the grain of the parallel structure, the more the properties and functions of the network dominate over those of the individual processors. For sufficiently fine-grained parallel structures, the computational demands of the network may dominate the computational requirements of the system, resulting in less efficient performance than comparable but coarser-grained systems. For each application and model of computation, there is some "natural" point where additional parallelism will not be effective. The problem is that we do not know what determines this point for any nontrivial applications on any nontrivial architectures. There is a vast domain of open research problems here that cry out for collaboration between computer science and the physical sciences. The few experiments that have been done are encouraging.

It is easy to identify properties of possible hardware that would yield potential improvements of two orders of magnitude in the cost-to-power ratio over the next decade or so. An even larger improvement seems likely with changes in algorithms and architectures. Realizing these potential improvements will depend on knowledge of how to structure algorithms and programs into parallel formats. Until recently, students have been taught to solve problems in a rigorous step-by-step fashion. They are taught that sequential thought patterns are the easiest to organize and analyze and thus are most likely to lead to correct answers. Such patterns of thought, may, however, be artificial and limiting. For several centuries, mathematical analysis for focused on sequential algorithms for numerical computations. To use the new computer technologies to their fullest, we must reformulate the traditional methods to maximize parallel execution: Every activity of an algorithm should proceed concurrently, except where an exchange of information is required.

References

1. For a more complete history of parallelism, see, for example, R. W. Hockney, C. R. Jesshope, *Parallel Computers*, Hilger, Bristol (1981), chapter 1. This also contains an extensive bibliography of original papers, I have therefore cited only a few here.
2. J. Cocke, D. L. Slotnick, "The use of parallelism in numerical calculations," IBM Research memorandum RC-55, 21 July 1958.
3. P. Weston, Electronics, 22 September 1961, page 46.
4. D. L. Slotnick, C. W. Borck, R. C. McReynolds, *Proc. Fall Jt. Comput. Conf. 1962*. AFIPS Conf. Proc. vol. 22, page 97; G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. J. Slotnick, R. A. Stokes, Computer, IEEE Trans. Comput. **C-17**, 99 (1968).
5. Standard texts generally survey synchronization techniques; see, for example, J. L. Petersen, A. Silbershatz, *Operating System Concepts*, Addison–Wesley, Reading, Mass. (1983).
6. T. Hoshino *et al.*, *Proc. 1983 Int. Conf. on Parallel Processing*, H. J. Siegel, l. Siegel, eds. IEEE Comput. Soc., Los Angeles (1983), page 95.
7. J. B. Dennis, Computer **13**, 48 (1980).
8. A. Gottlieb *et al.*, IEEE Trans. Comput. **C-32**, 175 (1982).
9. M. C. Sejnowski, E. T. Upchurch, R. N. Kapur, D. P. S. Charlu, G. J. Lipovski, *Proc. 1980 Natl. Comput. Conf.*, AFIPS Conf Proc. **49** (1980), page 631.
10. H. F. Jordan, P. L. Sawyer, Comput. Struct. **10**, 21 (1979). □