Advanced Computer Architectures: An Overview

The vector processing and multiprocessing architectures of a computer not only accelerate computational speeds but also offer access to larger central memories.

James A. Vegeais, Alan B. Coon, and Mark A. Stadtherr, Univ. of Illinois, Urbana, Ill. 61801

Today, the best-known and most widely-used advanced architecture machines are the so-called supercomputers, as typified by machines such as the Cray-1, CDC Cyber 205, Cray X-MP, and Cray-2. The Cray-1's peak speed is more than an order of magnitude faster than that of large mainframe computers like the CDC Cyber 175 or the IBM 3081, approximately three orders of magnitude faster than the VAX 11/780, a very popular superminicomputer, and roughly four orders of magnitude faster than the IBM PC/AT. Depending on the number of processors available, the Cray X-MP and Cray-2 can increase computational throughput by yet another order of magnitude or more relative to the Cray-1. It should be noted that these figures reflect approximate peak speeds that may not be realized in practice unless the program and algorithm used are able to take advantage of the supercomputer architecture.

J. A. Vegeais, *who holds a B.S.Ch.E. degree from the Univ. of Illinois at Chicago, is now completing his Ph.D. in chemical engineering at the Univ. of Illinois at Urbana-Champaign, where he earned his M.S. degree in chemical engineering.*

A. B. Coon, *who holds a B.S.Ch.E. degree from the Univ. of Texas at Austin, is now completing work on a Ph.D. in chemical engineering at the Univ. of Illinois at Urbana-Champaign, where he earned his M.S. degree in chemical engineering.*

M. A. Stadtherr *holds a B.Ch.E. degree from the Univ. of Minnesota and a Ph.D. degree in chemical engineering from the Univ. of Wisconsin. Now on chemical engineering faculty at the Univ. of Illinois at Urbana-Champaign, he has authored or coauthored a number of articles on process flowsheeting and optimization.*

Supercomputers

There were only a handful of supercomputers 10 years ago; as recently as four years ago about half of the supercomputers in use belonged to government facilities. This has rapidly begun to change. Today there are well over 100 supercomputers in use. In addition to their use by government laboratories and agencies, supercomputers are used commonly in the petroleum industry for reservoir simulation (*1–4*) and in other industries for nuclear reactor research and design (*5*), aerodynamic and structural design for automobiles and aircraft (*6*), VLSI design (*7*), weather forecasting (*8*), and even motion picture production (*9*). Several chemical companies have purchased time on supercomputers in the past and, more recently, have begun to purchase supercomputers of their own.

The availability of supercomputers has increased recently in academia as well. With help from the National Science Foundation and others, several universities have acquired supercomputers. Supercomputing centers set up by the National Science Foundation provide supercomputer time, training and support as well as a computing environment for researchers from different disciplines to share knowledge about common computational problems. One of these centers is located at the University of Illinois (*10*).

While supercomputers have been defined as the fastest computers available at any specific time (*11*) or as computers that are only one generation behind the computing requirements of leading edge efforts in science and engineering (*12*), the characteristic that distinguishes today's supercomputer from other computers is the extensive use of some form of parallelism. Since computers were first being manufactured, most of the increases in computational speed have been due simply to increased clock speed. However, it appears that a limit to the clock speed is now being approached. For computers with extremely fast clock speeds, the dimensions of the machine become an important consideration. Since electrical signals cannot travel faster than the speed of light, a very high clock speed requires that the maximum distance a signal must travel be very small. For example, a signal can travel only about 30 centimeters in 1 nanosecond (ns).

Therefore, a computer with a 1 ns clock (which would be about a factor of four faster than the clock cycle of a processor in a state-of-the-art supercomputer such as the Cray-2) must be no more than about a cubic foot (0.028 m^3) in size. This, of course, causes tremendous wiring and cooling problems. Consequently, it has become necessary for computer manufacturers to look to the parallel processing of instructions and data to significantly increase the speed of their computers. This parallelism generally manifests itself in some form of vector processing or multiprocessing architecture. For instance, machines such as the Cray X-MP or Cray-2 use a vector multiprocessing architecture.

Despite all of the current interest in supercomputers, their availability is still limited by their high cost, typically in excess of $5 million and perhaps much more. This may put the supercomputer out of the price range of all but the largest concerns. However, machines with advanced computer architectures that cost one or two orders of magnitude less than this are now available, and these machines offer substantial performance improvements over similarly-priced conventional machines. Some of these advanced architecture machines use architectures very similar to the Cray-1 and may even be software-compatible with the Cray-1. Others use much different architectural concepts. Assuming that their architectures can be exploited effectively by the user, these minisupercomputers (or personal supercomputers) appear to offer significantly better price/performance ratios than the conventional technology.

Chemical engineering applications

For the chemical engineer, advanced architecture machines will provide three overlapping opportunities (*13*):

1. To solve problems involving the modeling and analysis of complex physical phenomena that were previously intractable or at least computationally infeasible.
2. To greatly increase engineering design productivity in areas requiring large-scale computation.
3. To use complex models in real-time applications.

On traditional computers, one may need to limit the problem in terms of dimensionality, resolution or physical assumptions. The supercomputer reduces these limitations. It enables the chemical engineer to solve problems involving the modeling of a combination of phenomena such as simultaneous chemical reaction, fluid flow, heat transfer and mass transfer in much more detail and with greater accuracy than previously possible.

Advanced computer architectures can be used to increase the productivity of design engineers. Many computer-aided design (CAD) problems may be solved without the use of a supercomputer, but CAD offers the design engineer very rapid feedback on the results of design changes. In steady state process flowsheeting and optimization, the use of advanced architectures will provide not only the ability to more realistically handle complex phenomena, but also will have truly interactive design capabilities.

The third category is currently the least developed of the three. One likely chemical engineering application is process control. Real time simulations of entire plant complexes using advanced computer architectures will provide a powerful tool for on-line optimization of complete plant operations.

Advanced computer architectures

As noted earlier, the key element in advanced computer architectures is parallelism (*14–16*). There are many different ways in which a computer can be made to operate in parallel (*17–19*). Perhaps the most widely used taxonomy involves the four architectures proposed by Flynn (*20*): single-instruction single-data (SISD), single-instruction multiple-data (SIMD), multiple-instruction single-data (MISD), and multiple-instruction multiple data (MIMD).

In the SISD computer (including most conventional computers), only one instruction can be executed at a time and the instruction can operate only on one datum. In an SIMD while only one instruction can be executed at a time each instruction performs the same operation simultaneously on many different data. For example, a single instruction can compute the sum of two vectors. All (or at least some) of the additions of the elements of the two operand vectors can be done simultaneously. In the MIMD computer, different instructions perform different operations on many data simultaneously.

Some computers do not fit readily into one of these categories. For example, a "pipelined" computer has been considered by various people to be an SISD computer (*20*), "The characteristic that distinguishes today's supercomputer from other computers is the extensive use of some form of parallelism."

an SIMD computer (*21*), and an MISD computer (*17, 22*). The MISD category is considered by some not to be a useful category (*23*) because they regard it as impractical and there are no real implementations. For these reasons, perhaps a better and simpler way of classifying advanced architectures is as vector processors, multiprocessors, and vector multiprocessors (*24*).

The vector processing category includes SIMD and pipelined computers since both facilitate the processing of identical operations on large vectors or arrays of numbers. For example, a floating-point operation involves several steps. Without pipelining, all the steps needed to complete one operation would be performed before starting the first step of the next operation. Therefore, the computer works on one operation at a time. On the other hand, a particular operation in a highly pipelined computer is performed in several "stations," each of which is only one step of the overall operation. Since all stations work concurrently, the computer can perform each step of the operation on different data at the same time, just as in an assembly line. Examples of this type of computer are the highly-pipelined Cyber 205 (*25*) and Cray-1 (*26*), whose operations differ somewhat. The Cyber 205 operates most efficiently on very long vectors, while the Cray-1 operates efficiently even on relatively short vectors. Rudimentary forms of pipelining can also be found in some conventional machines.

SIMD computers are composed of an array of separate processors, each performing the same operation at the same time but on different data. Perhaps the best-known high-performance machines of this type are the ILLIAC IV (*27*), now retired, and the Massively Parallel Processor (MPP) (*28*) built for NASA by Goodyear Aerospace primarily for image processing. The ILLIAC IV, which had an array of 64 processors, could add two 64-element vectors in about the same time as a scalar addition, since all 64 scalar additions needed to add the vectors could be done in parallel.

The multiprocessing category covers architectures that use an array of scalar processors, each capable of executing different instructions at the same time on different data. Since the processors can execute different instructions, they are not necessarily synchronized. Examples of commercial multiprocessors are the BBN Butterfly (*29*) and the Intel iPSC (*30*). There are many other experimental prototype machines of this type as well as those in various stages of commercialization. Some conventional machines can also be thought of as "loosely-coupled" multiprocessors (*23*).

The vector multiprocessing category is essentially a combination of vector processing and multiprocessing. A number of processors can run in parallel, each of which by itself is a powerful vector processor and generally of the pipelined type. Most state-of-the-art supercomputers such as the Cray X-MP (*31*) and Cray-2 (*32*) fall into this category, and plans for newer machines suggest that this architecture will continue to dominate the high end of the supercomputer market for at least the short-to-medium term. Currently available state-of-the-art machines have up to four vector processors in parallel, and this number is expected to grow significantly over the next several years.

Multiprocessing

In all of these machines, the extent to which their computing capability can be exploited depends largely on how well the software is tailored to that type of architecture. In a vector processor, this would depend on the extent to which a problem, code, or algorithm can be "vectorized," that is, put into a form in which as many operations as possible are done on vectors. Since the architectures of these machines vary greatly from one another, it is necessary to know more about them to write efficient software for a particular machine.

The individual processors used in advanced architectures range from low-priced, relatively slow microprocessors such as the 80286 processors in the Intel iPSC to extremely fast, custom-designed processors such as those in the Cray-2. The early supercomputers (e.g., Cray-1) were generally composed of one high-speed pipelined processor. The most recent generation of supercomputers (e.g., Cray-2) includes machines with several high-speed processors. Early prototype multiprocessors, however, were generally composed of microprocessors due to their cost, availability, and reliability. Recently, commercial machines composed of microprocessors have begun to appear. It has been discovered that, in many applications, many slower, inexpensive microprocessors working in parallel can achieve the same computational speed as a single fast one. The microprocessors currently used most often are the 8086/80286 and the 68000. So, in a sense, these computers can be thought of as arrays of PC or Macintosh computers.

While the processors in these multiprocessing machines are microprocessors that operate sequentially, many of the custom-made and all of the fastest supercomputer processors include other levels of parallelism within their processors. For example, the Cray-2 computer currently has up to four processors. Because there are multiple pipelines within each of these processors, it is possible to simultaneously operate more than one processor on a program and more than one pipeline within each processor (*18, 23*).

Memory access

Multiprocessors also differ greatly in the way they access

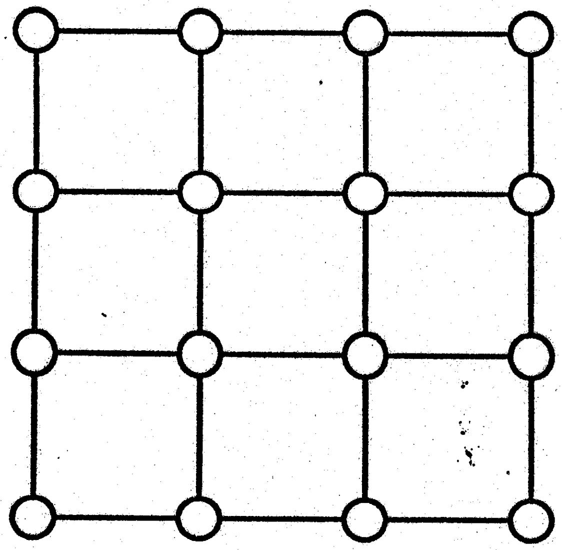

Figure 1. Nearest-neighbor interconnection scheme.

memory. In some computers, each processor has its own local memory. If a processor needs data from another, the data must be passed from the memory of one processor to that of the other processor. At the other extreme, some computers have a global memory: all of the processors access the same memory. In this case it is unnecessary to transfer data between processors. Normally, the memory is divided in such a way that only one processor can access a particular portion of memory at a time. Because of potential conflicts among different processors trying to access the same memory bank at the same time, machines with shared memory have used relatively few processors.

Other machines have memories that lie somewhere between local and global. For instance, a computer can have a number of separate memories that can be assigned dynamically to the different processors through some sort of interconnection network as a program is executed. The memory is global in the sense that it can be accessed directly, though not simultaneously, by all processors, but it is not truly global since it comprises a set of separate memories.

Some computers have more than one level of memory. Often there will be a relatively small fast memory, used to keep only those variables and portions of code that are currently needed, and one or more levels of slower memory. Other multiprocessors have both local and global memories. The presence of the global memory allows data to be shared without requiring data transfers between local memories that could be very time-consuming or could tie up the processors. The local memory can be accessed more quickly than global memory (*18, 23, 33, 34, 35*).

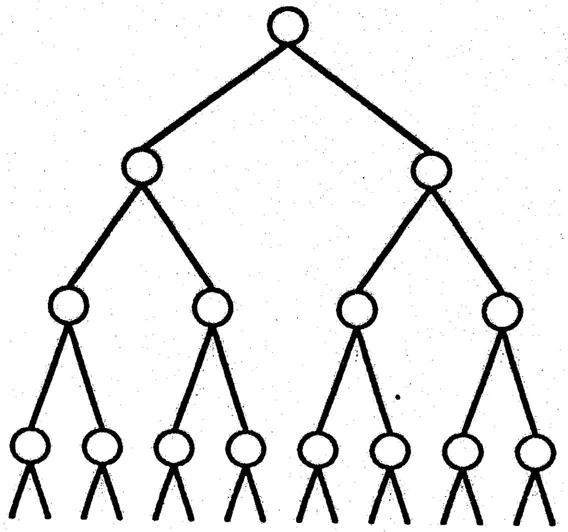

Figure 2. Tree interconnection scheme.

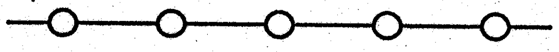

Figure 3. Linear interconnection scheme.

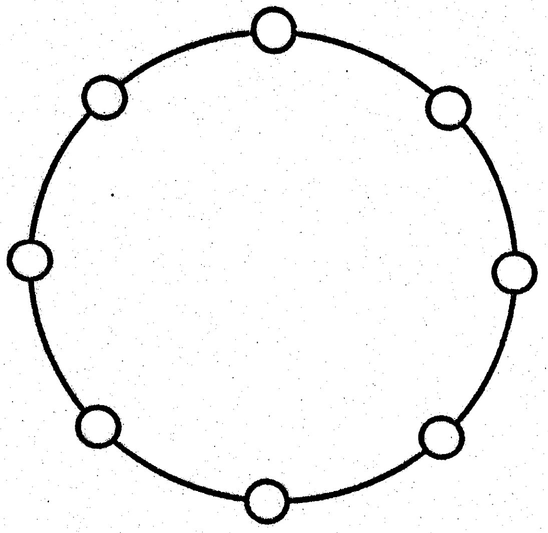

Figure 4. Ring interconnection scheme.

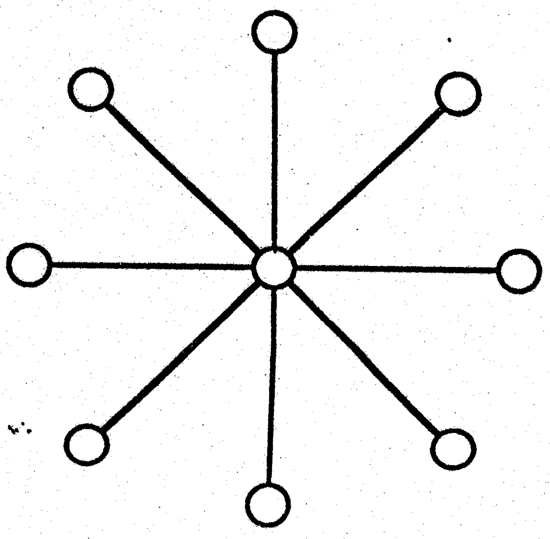

Figure 5. Star interconnection scheme.

"With the proper use of the advanced architecture, minisupercomputers offer significantly better price/performance ratios than the conventional technology."

Interconnection schemes

The interconnections between the individual processors in a multiprocessing computer also vary a great deal. Connections can be classified as either static or dynamic. Static schemes have fixed connections between processors, while dynamic schemes allow for switching of interconnections while a program is being executed.

In the ideal static scheme, each processor would be connected to all others to facilitate the transfer of data from one processor to another. The number of connections needed for this is $N(N-1)/2$ where N is the number of processors. This number is not very large for small N. However, 16 processors would require 120 interconnections, 64 processors would require 2,016, and 256 processors would require 32,640. In practice, these large numbers of connections are not feasible because of insufficient space within the computer. Because of this, other interconnection schemes have been devised that attempt to allow for efficient transfer of data between processors with fewer interconnections between the processors.

One interconnection scheme that was initially popular is the nearest-neighbor mesh, Figure 1. This is a two-dimensional rectangular grid where each processor is connected to the four nearest processors. Two-dimensional grid problems often adapt well to this sort of architecture since solutions at one grid point normally depend on the values at the nearest neighbors—exactly those processors that are directly connected.

Another architecture that has attracted some interest is the tree architecture, Figure 2. This type of architecture seems to lend itself nicely to the decomposition of problems: a root-node processor could distribute part of a problem to each of its branch-node processors. These processors would, in turn, decompose the problem further and the parts would be processed by the next level in the tree.

Many other interconnection schemes that have been proposed include: the linear, Figure 3; ring, Figure 4; star, Figure 5; and variations of these. Perhaps the architecture receiving the most interest currently, though, is the hypercube (or N-cube). The hypercube scheme connects 2^N processors so that each can communicate directly with other N processors. The three-dimensional case is shown in Figure 6. One reason for the popularity of the N-cube is that many other interconnection schemes, such as ring or linear, can be thought of as a subset of a hypercube. For example, if an algorithm is designed to work well with a ring-connected array, it will often work well on a hypercube.

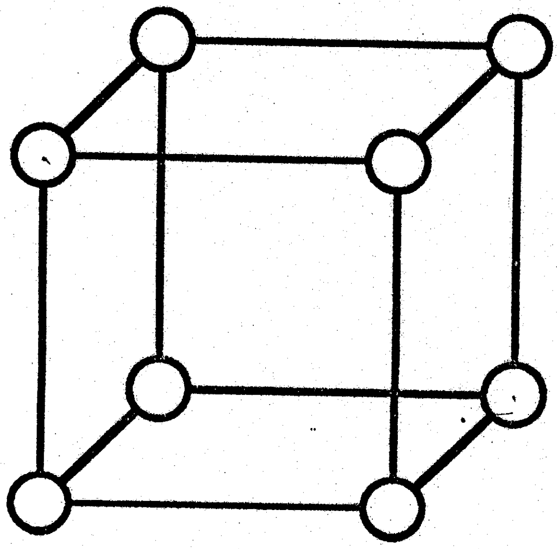
Figure 6. Three-dimensional hypercube interconnection scheme.

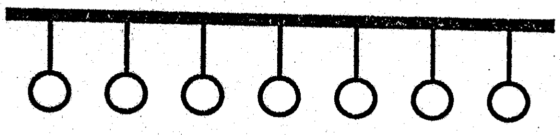
Figure 7. Bus interconnection scheme.

One dynamic scheme of connecting the processors is to connect them to a single pathway, a bus, Figure 7. All transfers of data must occur over this pathway. Unfortunately, the bus can be used only to pass one datum at a time. Hence, this connection scheme is suitable only for a very small number of processors, where there would not be as much contention for the single pathway.

It is not necessary that dynamic interconnections be between processors. Interconnections between processors and separate memories, like those mentioned above, are sometimes used instead. One such scheme is the crossbar switch, Figure 8, which essentially connects all processors to all memories. This switch allows every processor to be connected to a different memory simultaneously. Other switching networks have been used as well, such as the banyan network, omega network (Figure 9), and Batcher network. More information on interconnection networks can be found in (*18, 23, 36, 37, 38*).

Other considerations

Nearly all of the advanced architecture computers available today run a high-level language, usually Fortran. Special instructions are often added to both high- and low-level languages to better take advantage of the architecture. Of course, vector computers have vector instructions to per-

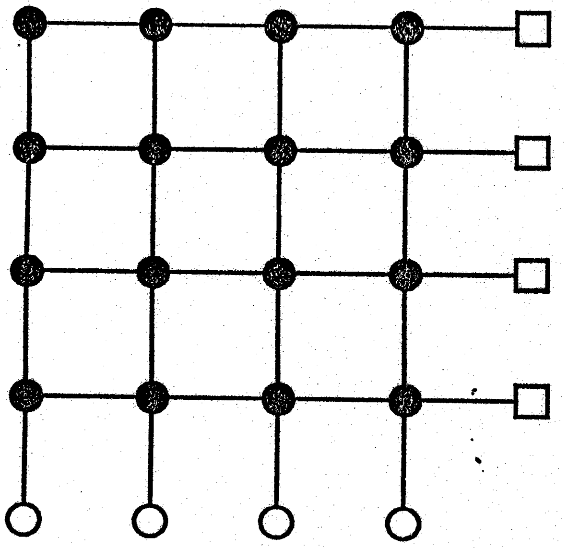

Figure 8. Crossbar interconnection scheme.

form in one instruction that which would normally be a series of instructions in the form of a DO loop on a sequential computer. Other statements are often added also. For example, the Cray computers have a vectorizing compiler (*39*) that attempts to vectorize as much of the code as possible. The compiler, however, is prevented from vectorizing if loops contain data dependencies. For instance, in the following code, the compiler would assume that the elements of the array computed in the second line depend on values of elements computed within the DO loop, even though no such data dependency actually exists.

```
    DO 10 I = 1, N
    A (I + N) = A (I) + B (I)
10  CONTINUE
```

Thus, the loop would not normally vectorize for this reason. A special compiler directive is provided that forces vectorization to occur despite an apparent data dependency in the loop. For instance, the loop:

```
CDIR$ IVDEP
      DO 10 I = 1, N
      A (I + N) = A (I) + B (I)
   10 CONTINUE
```

would vectorize because the additional statement instructs the compiler to ignore the assumed data dependency.

In multiprocessors, additional instructions are needed to allow parts of programs to be synchronized or to allow a precedence order between parts of code. For example, the Cray X-MP has a library routine (EVWAIT) that causes execution of code to wait until another routine (EVPOST) is called (*40*). Routines exist on some other multiprocessors to allow for the transfer of data from one processor to another.

Measuring performance

The traditional way of measuring the performance of a computer has been how many MIPS (millions of instructions per second) it could perform. Parallel computers have rendered this measure rather meaningless, because one single instruction can perform many operations, tens of thousands in some cases. A more meaningful measure of a computer's speed is how many MFLOPS (millions of floating point operations per second) it can perform. Unfortunately, this measure can also be misleading because the number of MFLOPS a computer actually performs depends greatly on how well the code is written to take advantage of the architecture of that particular computer.

It is not unusual for MFLOPS rates to vary well over an order of magnitude for different programs. Manufacturers often list an MFLOPS rating for their computers. Often this is a peak rate and computation cannot be sustained at this rate for a long period of time. By comparing the MFLOPS rate for a particular code with the peak or the sustainable MFLOPS rate, however, one can get a good idea of how well a particular code takes advantage of the computer's architecture. One very popular benchmark for comparing computer performance is the solution of linear equation systems using the LINPACK routines (*41*).

Speedup is another factor used to measure the performance of a particular algorithm. On a multiprocessor,

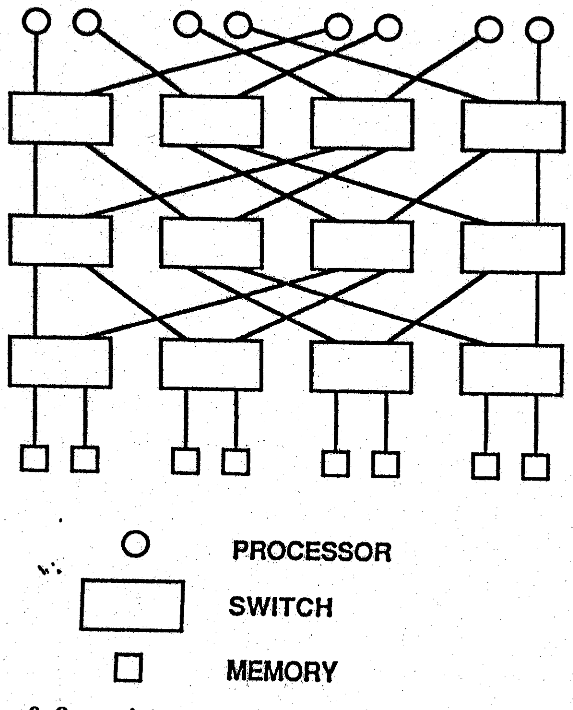

Figure 9. Omega interconnection network.

speedup is defined as the time it takes to complete a job using only one processor, divided by the time the job requires using P processors. The best possible speedup would be P, indicating that the algorithm could be performed in P independent parts of equal size. The efficiency is the speed-up divided by the number of processors.

If the fraction of code that is able to be performed in parallel on P processors is f, the maximum speedup is $P/(P - fP + f)$. This relationship, known as Amdahl's law (*42*), is derived from a simple model of parallel processing whose assumptions include: independence of parallel tasks; a negligible amount of overhead time incurred in initiating the parallel tasks; and negligible amounts of time for data organization and interprocessor communication and synchronization. Figure 10 shows a plot of speedup against the fraction of code that is parallelizable, for some values of P. For significant speedup, significant portions of the code need to be run in parallel. This becomes even more important as the number of processors increases. Note, for instance, that with 64 processors, just 5% nonparallelized code will result in a maximum speedup of 15.42 (or a

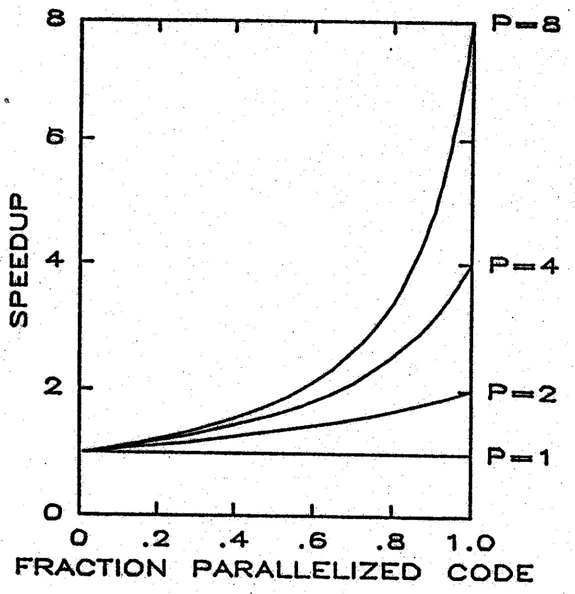

Figure 10. A plot of Amdahl's law for several values of P.

maximum efficiency of only 24%). It should be remembered that Amdahl's law does not take into account any overhead from running the job in parallel. This would further reduce the speedup and efficiency.

Speedup is a measure that is also used with pipelined computers. In this case it is the ratio of the time it takes for a job to execute without vectorization to the time it takes for a job to execute with vectorization. For pipelined computers, one can obtain an equation identical to Amdahl's law. In this case, however, f refers to the fraction of the code that vectorizes and P to ratio of peak vector speed to peak scalar speed.

Programming and algorithmic considerations

As mentioned earlier, the programmer must consider the characteristics of an advanced architecture before writing a program that runs well on a computer with that particular architecture. The concerns vary somewhat for the different classes of computers.

In a vector processor, one must be concerned with writing code so that as much of the code as possible consists of vector operations. Vectorizing compilers can now do a good job of producing code that is vectorized, although some improvement can often be made by coding in assembly language. Unfortunately, the compiler does not know as much about the purpose of the code as does the programmer. Therefore, it is still necessary for the programmer to write code in such a way that the compiler can recognize as much vectorizable code as possible.

Vectorization can be inhibited by many different factors. First, compilers generally vectorize only DO loops. Loops created with IF statements will not vectorize and should be avoided. Vectorization may not take place if certain statements are within the loops. For example, loops with IF statements, subroutine calls, or statements with recursion will not normally vectorize nor will loops with irregular addressing. Strategies that avoid some of these problems vary from simply removing a statement from a loop to splitting a loop into multiple loops to using special vector functions such as scatter/gather (*43–49*).

Even when a loop can be vectorized, it may not execute as fast as it might if the code were altered. For example, loops may often be speeded up by unrolling the loop to reduce the number of memory references (*50*). Nested loops may often be speeded up by changing the order of the loops so that the longest loop is vectorized.

It should be noted that codes that vectorize on one computer may not vectorize on another vector computer or may not show as much of a speed increase if they do vectorize. For an example, Cray computers can operate only on variables that are in the computer's registers. Since each register can hold only 64 words at once, performing operations on vectors of length greater than 64 requires that the computer split the vector into smaller vectors of length 64. The Cyber 205, however, operates directly on variables in memory. Although the start-up time for such memory operations is large compared to that for register operations, the Cyber 205 can operate on extremely long vectors with only one vector operation, avoiding the overhead that is associated with the initialization of the additional vector operations.

Programming for a multiprocessor essentially entails writing code that is split into many separate tasks, some of which can be executed simultaneously. On a shared-memory computer this often results in some sort of task queue.

The multiprocessing category covers architectures that use an array of scalar processors, each capable of executing different instructions at the same time on different data.

The next task in the queue is begun when a processor becomes available. On a multiprocessor without global memory, however, it is often necessary to specify not only the tasks but also to determine which processor should do each task. It is also necessary to determine in which local memory variables will be kept.

Algorithms can be either synchronous or asynchronous. Various degrees of synchronization exist. The most synchronized approach would be to use an MIMD computer as an SIMD computer by programming so that all processors perform the same operation at the same time.

A slightly less synchronized method would be for all processors to start different tasks at the same time. As processors complete their tasks, they become idle until the last processor finishes its task. They may all begin on their next task then. This method is often easy to program as it involves a sequence of parallel steps. In such an algorithm it becomes important to keep the time for the completion of a task about the same for all processors to minimize the amount of time that processors remain idle while waiting for the other processors to finish.

A more efficient method is to decompose the problem but not require that all tasks start simultaneously. This allows processors that have completed their tasks to begin new tasks immediately, if the necessary data are available. In this case it is necessary to synchronize only so that a processor does not attempt to use an operand that has not yet been calculated by another processor.

This synchronization can be done by several different methods. One is by causing a processor to wait until a certain necessary event has occurred. Another is to cause a processor to wait for the completion of a certain task. In this case, the size of the separate tasks in the code (or granularity) becomes a critical factor in obtaining maximum efficiency. In the case of very small granularity, machines are being developed for which this synchronization can be done on the machine level, in what is known as a data flow machine (*23, 51*). For larger granularity, the synchronization is usually done by the programmer or the compiler.

Another major concern to the multiprocessor programmer is how to structure the data in the multiprocessor. This is normally not a problem for a shared-memory multiprocessor, but is extremely important in machines with local memory, such as computers with a hypercube architecture. If not done properly, the memory transfer time could dominate the total execution time of the program.

The structuring of a program into several parallel sets of instructions that are capable of executing concurrently is called multitasking (*40*). When these parallel sets of instructions consist of large sections of code this process is referred to as macrotasking, while microtasking is the partitioning of code into parallel tasks at the DO loop level. An ideal program is one that could be macrotasked into several independent tasks, one for each available processor and each requiring the same amount of time for completion, so that its efficiency approaches one.

In reality, an efficiency of one (i.e., a speedup of *P*) is rarely possible for several reasons (*52*). Multitasking is invoked at the cost of a certain amount of overhead (which is associated with the use of operations that are not necessary for single processor implementations, such as defining and distributing tasks) so that even an ideally structured program cannot attain an efficiency of one, although its efficiency would approach one. Also, the tasks of a multitasked program may depend on results from other tasks, causing processors to remain idle until the necessary results are available. Finally, some programs may not have any identifiable parallel structure or may have only a small portion of instructions that can be performed in parallel. It should be noted, however, that there are unusual circumstances under which efficiencies greater than one may occur (*53, 54*). In such cases, the assumptions upon which Amdahl's law is based are not valid.

To execute efficiently, vector multiprocessors obviously require algorithms and programs that exploit both vector operations and parallelism. However, the need for long vectors and the need for several independent tasks can sometimes be competing demands. For example, in the implementation of a nested dissection algorithm on a vector computer, the factorization time can be decreased by stopping the dissection process short of completion (*55*). The incomplete dissection yields longer vectors than the original scheme, but it also yields fewer submatrices that can be factored in parallel.

Some algorithms have high efficiencies for a small number of processors but exhibit a sharp decline in efficiency once the number of processors is increased. Peters (*56*) reports that the solution of matrices arising from finite element calculations on a square domain will yield a speedup of about five with eight processors, or an efficiency of 0.63. However, no further speedup is obtained if more processors are used, regardless of the number of unknowns involved. This indicates that after the parallel portion of the algorithm is split into as many as eight separate tasks, the sequential portion begins to dominate the total time needed to perform the algorithm. Further efforts to exploit the architecture of a vector multiprocessor with such an algorithm should be concentrated on vectorizing the algorithm or code, particularly those portions that cannot be multitasked.

In conclusion

The ever-increasing demand for more computing power has manifested itself not only in the popularity of single and multiple vector processors with extremely fast clock speeds, but also in the recent advent of innovative configurations of arrays of microprocessors. The two predominant trends in advanced architectures are the use of a few extremely powerful processors in parallel and the use of very many microprocessors in parallel. It is not certain yet whether there will continue to be a market for both classes of machines. The former type of machine has been firmly established as a valuable scientific tool. However, developments within the latter class indicate that its members may be economical alternatives to expensive machines like the Cray-2. In either case, an understanding of the parallel nature of both the architecture and the algorithm is necessary to fully utilize its capabilities. And as computational requirements continue to grow, it seems inevitable to use parallel architectures in all branches of science and engineering. ■

Acknowledgment

Our research in the development of strategies for effectively applying advanced computer architectures in chemical process flowsheeting and optimization is being supported by the National Science Foundation under grant number DMC-8520663.

Literature cited

1. **Nolen, J. S., and P. L. Stanat,** "Reservoir Simulation on Vector Processing Computers," SPE Middle East Technical Conference, Bahrain (1981).
2. **McDonald, A. D.,** "Vector Computer Applications in Reservoir Simulation," Science and Engineering Symposium, Cray Research, Inc. (1985).
3. **Levesque, J. M.,** *Soc. Pet. Eng. J.*, 25, p. 275 (1985).
4. **Absar, I.,** *Simulation*, 44, p. 247 (1985).
5. **Fuss, D., and C. G. Tull,** *Proc. IEEE*, 72, p. 32 (1984).
6. **Gloudeman, J. F., and J. C. Hodge,** The Adaptation of MSC/NASTRAN to a Supercomputer, in *Parallel and Large-scale Computers: Performance, Architecture, Applications*, Ed., M. Ruschitzka, *et al.* (1982).
7. **Vladimirescu, A., and D. O. Pederson,** "Circuit Simulation on Vector Processors," *IEEE 1982 Int. Conf. on Circuits and Computers*, p. 172 (1982).
8. **Dickinson, A.,** *Computer Phys. Commun.*, 26, p. 459 (1982).
9. **Demos, G., M. D. Brown, and R. A. Weinberg,** *Proc. IEEE*, 72, p. 22 (1984).
10. "Profile," National Center for Supercomputing Applications, University of Illinois, p. 1 (1985).
11. **Hwang, K.,** *"Supercomputers: Design and Applications,"* IEEE Computer Society (1984).
12. **Lincoln, N. R.,** *IEEE Trans. on Computing*, C-31, p. 349 (1982).
13. **Stadtherr, M. A., and J. A. Vegeais,** *Chem. Eng. Prog.*, 81(9), p. 21 (1985).
14. **Levine, R. D.,** *Scientific American*, 246, p. 118 (1982).
15. **Lerner, E. J.,** *High Technol.*, 5, p. 20 (1985).
16. **Kuhn, R. H., and D. A. Padua,** "Tutorial on Parallel Processing," IEEE Computer Society Press (1981).
17. **Handler, W.,** *Proc. 1977 International Conf. on Parallel Processing*, p. 7 (1977).
18. **Kuck, D. J.,** "The Structure of Computers and Computations," John Wiley & Sons (1978).
19. **Ortega, J. M., and R. G. Voigt,** *SIAM Review*, 27, p. 149 (1985).
20. **Flynn, M. J.,** *Proc. IEEE*, 54, p. 1901 (1966).
21. **Flynn, M. J.,** *IEEE Trans. on Computers*, C-21, p. 948 (1972).
22. **Thurber, K. J.,** "Large Scale Computer Architecture—Parallel and Associative Processors," Hayden Book Co. (1976).
23. **Hwang, K., and F. A. Briggs,** "Computer Architecture and Parallel Processing," McGraw-Hill (1984).
24. **Bucher, I. Y.,** *Proc. ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, p. 151 (1983).
25. **Zink, B., and T. Putnam,** "Cyber 205 User Guide," Purdue University Computing Center (Sept. 5, 1984).
26. **Russell, R. M.,** *Commun. of the ACM*, 21, p. 63 (1978).
27. **Hord, R. M.,** "The ILLIAC IV The First Supercomputer," Computer Science Press (1982).
28. **Batcher, K. E.,** *IEEE Trans. on Computers*, C-29, p. 836 (1980).
29. "Butterfly Parallel Processor Overview," BBN Laboratories, Inc., Cambridge, Mass. (1985).
30. **Moler, C.,** "Programming the Intel Personal Supercomputer," SIAM Meeting (Oct., 1985).
31. **Chen, S. S.,** *et al.*, "Cray XMP: A Multiprocessor Supercomputer," submitted for publication, in "Vector and Parallel Processors: Architecture, Applications, and Performance Evaluation," Myron Ginsberg, Ed. (1985).
32. *Cray Channels*, 7, p. 2 (1985a).
33. **Cheung, T., and J. E. Smith,** "An Analysis of the Cray X-MP Memory System," *IEEE Trans. on Computers* (1984).
34. **Lawrie, D. H., and C. R. Vora,** *IEEE Trans. on Computers*, C-31, p. 435 (1982).
35. **Dubois, M., and F. A. Briggs,** "Effects of Cache Coherency in Multiprocessors," *IEEE Trans. on Computers*, C-31 (1982).
36. **Feng, T-Y.,** *Computer*, 14, p. 12 (1981).
37. **Wu, C-L, and T-Y Feng,** *IEEE Trans. on Computers*, C-29, p. 694 (1980).
38. **Thurber, K. J.,** *Proc. National Computer Conference*, p. 909 (1974).
39. "Cray-1 Computer Systems FORTRAN (CFT) Reference Manual," Publication SR-0009, Cray Research, Inc., Mendota Heights, Minn. (1984).
40. "Multitasking Users Guide," Publication SR-0222, Cray Research, Inc., Mendota Heights, Minn. (1985b).
41. **Dongarra, J. J.,** "Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment," Technical Memorandum No. 23, Argonne National Laboratory, Argonne, Ill. (1986).
42. **Amdahl, G. M.,** *AFIPS Conf. Proc.*, 30, p. 483 (1967).
43. **Levesque, J. M.,** "Application of the Vectorizer for Effective Use of High Speed Computers," in "Exploring Applications of Parallel Processing to Power System Analysis Problems," EPRI Special Report El-566-SR, Palo Alto, Ca. (1977).
44. **Levesque, J. M., and B. Q. Brode,** "Optimization for the CDC Cyber 203 and 205—Dusting Off the Dusty Decks," ECODU-30/VIM-33, Manchester, England (1980).
45. **Higbie, L.,** "Vectorization and Conversion of Fortran Programs for the Cray-1 (CFT) Compiler," Cray Research Publication 2240207 (1979).
46. **Higbie, L.,** *Datamation*, 29, p. 180 (1983).
47. **Arnold, C. N.,** "Vector Optimization on the Cyber 205, *Proc. 1983 Int. Conf. on Parallel Processing* (1983).
48. **Simon, H.,** "Supercomputer Vectorization and Optimization Guide," Report ETA-TR-22, Boeing Computer Services, (1984).
49. **Dembart, B.,** "Vectorization Using Gather and Scatter, Mathematics and Modeling," Technical Report ETA-TR-26, Boeing Computer Services (1985).
50. **Dongarra, J. J., and S. C. Eisenstat,** "Squeezing the Most Out of an Algorithm in Cray Fortran," Report ANL/MCS-TM-9, Argonne National Laboratory (1983).
51. **Dennis, J. B.,** *Computer*, 13(11), p. 48 (1980).
52. **Faber, V., O. M. Lubeck, and A. B. White, Jr.,** *Parallel Computing*, 3, p. 259 (1986).
53. **Parkinson, D.,** *Parallel Computing*, 3, 261 (1986).
54. **Douglas, C. C., and W. L. Miranker,** "Constructive Interference in Parallel Algorithms," IBM Research Report RC 11742 (#52722) (1986).
55. **George, A., W. G. Poole, Jr., and R. C. Voigt,** *Comp. and Maths. with Appls.*, 4, p. 287 (1978).
56. **Peters, F. J.,** *Parallel Computing*, 1, p. 99 (1984).