Kirk-Othmer Encyclopedia of Chemical Technology.

COMPUTER TECHNOLOGY

In the last 10 years the computer industry has experienced dramatic and fundamental changes. Computer users have witnessed both vastly increased capabilities and dramatically reduced costs. Chemists have often been at the forefront of taking advantage of the new capabilities. It seems likely that the next few years hold the promise of even greater rates of change and ever-expanding capabilities. The challenge of staying current with the technology will be ever more difficult.

The years since publication of the third edition of the *Encyclopedia* (1978–1984) have brought the rise and fall of the minicomputer, the worldwide ascendancy of microprocessor-based personal computers, the emergence of powerful scientific work stations, the acceptance of scientific visualization, further advances with supercomputers, the rise and fall of the minisupercomputer, and the realization that the future lies in parallel computing.

Each of these and other phenomena could, by themselves, benefit from in-depth examination. This article focuses primarily on those computing technologies that find application in computational domains, especially within computational chemistry.

1. Personal Computers

This sentence from *ECT* 3rd ed has proven prophetic:

> An important recent development has been introduction of the microprocessor, or computer on a chip.

Microprocessor-based personal computers have since become ubiquitous in modern business practice, forever setting aside many previously accepted notions of the office. Not surprisingly, the principal gains went first to general business users, with spreadsheets and word processors becoming standard equipment. Over time, significantly faster computers have become available at ever lower prices. Thousands of software packages are available. Today there are more than 50 million personal computers in the United States alone; a $100 software package that could be sold to just 1% of the installed base yields sales of over $50 million.

Over time, the market has demanded increasingly sophisticated software. Each successive enhancement in processor speed has been consumed by software that is more complex, even if only in creating a more user-friendly interface. In the past, computer time was expensive relative to labor costs. That situation is now reversed, and spending more for a more user-friendly computer can often be easily justified in order to enhance the productivity of the vastly more expensive human being.

Whereas the general office community reaped the benefits (and endured the pain) of the rise of the personal computer, the scientific and technical market was forced to wait. Chemists used spreadsheets and suffered through word processors that did not know what a chemical structure was. However, there is now a good variety of personal productivity tools that have been designed specifically for the chemist.

Word processors that deal readily with chemical equations and mathematical formulas are now commonplace. There are several molecular-structure drawing tools, the output of which can be imported into popular word processing packages. There are numerous packages for displaying, manipulating, and analyzing scientific data, as well as many that can be used for data acquisition and laboratory instrument control. Many chemical journals regularly review new scientific software packages.

As is the case elsewhere in the industry, there are no constants. The package currently judged superior is often eclipsed by a newer version of a competing package. The only trend that remains constant is that software continues to become more complex while efforts to make computers easier to use increase. Although the benefits first accrue to the general business community, increasing interapplication cooperation, where the output of one package can be easily integrated into another package or where one package maintains live links to the data in another package, can be expected. Changing data in one package will see those changes automatically propagated to all other applications that are using that same data.

The networking of personal computers has continued and has enabled the sharing of expensive hard-copy output devices and other peripherals and has provided the significant benefit of facilitating the transfer of data between computers. Joint authorship of technical articles is a quick and easy process with networked computers. The interapplication communication that today links applications within a single computer will transparently link applications on networked computers. Voice-annotated documents are already available.

The great majority of desktop computing is performed in the DOS or Macintosh operating environments. Microsoft Windows has been an immensely popular addition to DOS. IBM is currently aggressively marketing OS/2 as the preferred desktop computing environment for the future. Microsoft has strongly supported Windows. Apple has recently formed an alliance with IBM to jointly develop object-oriented methodologies for future desktop computing. It is beyond the scope of this article to speculate on the probability of success of these initiatives, but it can be said that the industry is entering a potentially confusing and rapidly changing era.

2. Supercomputers

It is difficult to define a supercomputer. Today's supercomputer becomes the minimum expectation for tomorrow's computers. A ten-year-old supercomputer is most likely a museum piece. Although there may be no universally accepted definition of a supercomputer, there are some characteristics that all supercomputers have. A supercomputer is expensive. Its cost has stayed relatively constant over the years, typically between $1 and $30 million. Performance is the primary goal of the designers of a supercomputer; cost is a secondary consideration at best. Supercomputers utilize the fastest electronic components available, connected in ways designed to minimize transmission delays. With a large number of electronic components driven at very fast rates, the typical supercomputer generates enormous amounts of heat. Most require extensive power-supply and liquid-cooling support systems.

Supercomputers are found in many government research laboratories, intelligence agencies, universities, and a small number of industrial companies. In the United States, the National Science Foundation (NSF) has provided supercomputers to several prominent universities for both academic and industrial users. These centers provide state-of-the-art, supercomputer-tuned applications for a wide variety of disciplines, together with staffs who are very knowledgeable in optimization for supercomputer performance.

A common acronym is MFLOPS, millions of floating-point operations per second. Because most scientific computations are limited by the speed at which floating point operations can be performed, this is a common measure of peak computing speed. Supercomputers of 1991 offered peak speeds of 1000 MFLOPS (1 GFLOP) and higher.

2.1. Vector Computers

Most computers considered supercomputers are vector-architecture computers. The concept of vector architecture has been a source of much confusion.

Most computers use pipelining to maximize performance. Pipelining is analogous to an automobile assembly line, with the finished product being the result of many sequential but otherwise independent operations on the object being produced. It is inefficient to have only one car on an assembly line at one time because stages which had finished their operations on the car would then sit idle until the last stage had completed operations before they could begin work again. In computing, many operations can be decomposed into suboperations that can be performed sequentially but independently. The pipelining in the Amdahl 470V/6 supercomputer has been described as follows (1975):

> A high throughput of instructions was achieved by pipelining the processing of instructions. The execution of instructions was divided into 12 suboperations that used 10 different circuits. When flowing smoothly, a new instruction could be taken every two clock periods (or 64 nanoseconds) and therefore up to six instructions were simultaneously in different phases of execution, and could be said to be in parallel execution (1).

Clearly the key to maximum performance was being able to keep the pipeline full. This observation still holds true for today's computers.

There are many reasons that it might be difficult to keep the pipelines full. The most obvious is a data dependency, where a previously initiated computation must pass through all stages of the pipeline before the next operation can be commenced.

$$X = 2^*Y$$

$$Z = 3^*Y$$

$$W = Z^*2$$

In this example, the evaluation of X and Z can be overlapped within the multiplication pipeline, but work cannot begin on the evaluation of W until the result of the computation of Z is known. The pipeline must empty, and the result, Z, must be retrieved before the pipeline can be refilled for the evaluation of W.

A pipelined floating-point multiply unit might accomplish a floating-point multiply by performing four independent suboperations, labeled a, b, c, and d, on the operands. The suboperations can be envisioned as the four workers on a four-person assembly line. The floating-point multiply pipeline could accept a new set of operands every clock cycle. The pipeline occupancy of this code fragment would look like

clock cycle	0 1 2 3 4 5 6 7 8 9
computation of X	a b c d
computation of Z	a b c d
computation of W	a b c d

In other words, X is being computed during cycles 0–3, Z is being computed during cycles 1–4, and W is being computed during cycles 5–8. The code fragment is complete after nine clock cycles.

A modern optimizing compiler would recognize that X and Z are independent and would probably reorder the code to be

$$Z = 3^*Y$$

$$X = 2^*Y$$

$$W = Z^*F$$

Now the computation of W can begin on the fifth clock cycle, rather than on the sixth.

	0 1 2 3 4 5 6 7 8
Z	a b c d
X	a b c d
W	a b c d

The code fragment is now finished after the eighth clock cycle. Note that there are still three clock cycles during which there are idle stages in the multiplication pipeline. The compiler would look for other statements in the code that could be overlapped with those already in process.

The best way to ensure that the pipelines are full is to gather together a complete series of inputs before starting the pipeline. These are the vector registers of a supercomputer. Vector supercomputers typically contain at least eight vector registers, each holding 64 or more floating-point numbers.

Consider the following FORTRAN code fragment:

```
DO 100 I = 1, 1000
X(I) = 2.0*Y(I)
100 CONTINUE
```

On a vector computer having vector registers that hold 64 floating-point numbers, this loop would be processed 64 elements at a time. The first 64 elements of Y would be fetched from memory and stored in a vector register. Each iteration of the loop is independent of the previous iteration, so this loop can be fully pipelined, with successive iterations started every clock cycle. Once the pipeline is filled, the result, X, will be produced one element per clock cycle and will be stored in another vector register. The results in the vector register will then be stored back into main memory or used as input to a subsequent vector operation.

A significant amount of machine overhead is involved in setting up a vector operation, with maximum benefit accruing if there are a full 64 elements to compute. For short loops, typically eight elements or fewer, vector operations are often slower than doing the computations one at a time in nonvector mode.

Vectorization becomes more difficult when there are loop dependencies, or when one iteration of the loop depends on another. Consider the following:

DO 100 I =

$X(I) = X(I + N) + 3.4$

100 CONTINUE

If N is positive, the loop can be vectorized. If $N = 5$, the existing 10th element of X is needed in order to compute the new value of the fifth element. As work begins on the fifth element before the tenth element is changed, the loop can be vectorized.

If, on the other hand, N is negative, then there is a loop dependency, and it may not be possible to fully vectorize the loop. If $N = -1$, the new value of the fifth element of X needs to be known before computation of the new value of the sixth element can begin. It might be several clock ticks before the new value of $X(5)$ becomes available. On the other hand, if N is a larger negative number, for example, $N = -50$, $X(1)$ is needed to evaluate $X(51)$; then it may be possible to vectorize this loop—the new value of $X(1)$ would be known long before the computation of the new $X(51)$ is started. In the absence of knowledge of the possible values for N, most compilers would not produce vector instructions for this loop. If the programmer has knowledge of the possible values of N, explicit directives can be inserted in the code to inform the compiler of assumptions that can be made about the loop.

There are vastly more complex examples of difficult vectorization decisions. A great deal of effort has been devoted to writing vector code, or code that compilers can safely translate into vector instructions. As compilers become more sophisticated, their ability to recognize vectorization opportunities increases. The vendors of vector computers often claim that vectorization is automatic and that end users need not be concerned with it. This claim is usually true only if the end user is similarly unconcerned with achieving maximum performance. More often than not, codes that have not been written for vector architecture machines must undergo substantial restructuring in order to achieve significant performance enhancements on vector-architecture machines.

2.2. Banked Memory

Another characteristic of many vector supercomputers is banked memory. The main memory is usually divided into a small number of electronically separate banks. A given memory bank can absorb or supply operands at a much slower rate than the rate at which the central processing unit (CPU) can produce or use data. If the data can be spread across multiple memory banks, the effective memory bandwidth, or rate at which memory can absorb or supply data, is increased. For example, if a single memory bank can supply one operand every 16 clock cycles, then 16 memory banks would enable the entire memory subsystem to deliver one operand per clock cycle, assuming that the data come sequentially from different memory banks.

The memory subsystem on most supercomputers is organized to support maximum performance on loops of stride one, or when the elements of an array are accessed sequentially with no gaps. In general, the stride is defined by

DO 100 I = START, STOP, STRIDE

$X(I)$ = ...

100 CONTINUE

The most common loop has $\text{stride} = 1$. Typically $X(1)$ would be stored in memory bank 1, $X(2)$ in memory bank 2, $X(16)$ in memory bank 16, and $X(17)$ in memory bank 1. In the loop in the example, if $\text{stride} = 1$, then the elements of X can be delivered to the CPU at the maximum rate, one per clock cycle.

If, on the other hand, $\text{stride} = 2$, then the system memory may limit the speed of the calculation. During the first clock cycle, a request is sent to bank 1 for $X(1)$. During the second clock cycle, a request is sent to bank 3 for $X(3)$. During the 8th clock cycle, a request is sent to bank 15 for $X(15)$. On the 9th clock cycle, when $X(17)$

should be fetched from memory bank 1, that memory bank is still processing the previous request, because $X(1)$ initiated during the first clock cycle. The system must wait until that request is completed before initiating a new request to memory bank 1. This access pattern uses only half the elements of X. More importantly, it uses only half the memory banks available. Consequently, effective memory bandwidth is halved. The most pathological case comes with $\text{stride} = 16$, when all references come from a single memory bank; effective memory bandwidth is one-sixteenth theoretical maximum.

2.3. Other Performance Considerations

Even if a program allows main memory to supply operands at peak rate, it may not be fast enough to keep the CPU operating at its peak rate. Consider the general SAXPY

$$\text{DO } 100\ I =$$

$$Z(I) = \text{ALPHA}X^{*}(I) + Y(I)$$

The term SAXPY has arisen as a mnemonic for scalar alpha X plus Y (2). This loop requires two operands and produces one result for each iteration of the loop. In 64 bit, or 8 byte, precision (8 bytes per floating-point number), this is a total of 24 bytes of data being consumed or produced per iteration. If the memory bandwidth were 240 megabytes per second, the memory subsystem could maintain this loop at 10 million iterations per second. Each loop iteration represents two floating-point operations, a multiplication and an addition; thus running at 10 million iterations per second is only 20 MFLOPS, a small fraction of the peak performance of supercomputers. Most supercomputers have memory subsystems with much higher bandwidths, sometimes with separate pathways for read and write operations. Nevertheless, careful analysis of memory subsystem usage can be an important ingredient of any code optimization. In the preceding example, the system should look for subsequent operations to be performed on *Z(I)* while it is still near the CPU, before it is returned to main memory. The problem of optimizing CPU performance is not specific to supercomputers. However, given the enormous cost, a great deal more effort is devoted to optimizing codes on supercomputers than on other machines.

Because of the relative slowness of main memory (compared with the CPU), most computers have a much smaller, but much faster cache memory subsystem that augments main memory. The size of the cache memory and the extent to which a program can utilize the cache can be critical determinants of performance. Again, there are some common optimization techniques designed to maximize cache utilization.

FORTRAN stores doubly dimensioned arrays in memory as:

$$X(1,1)\ X(2,1)\ X(3,1)\ \ldots\ X(n,1)\ X(1,2)\ X(2,2)\ \ldots\ X(n,2)\ X(1,3)\ \ldots$$

This has important consequences for programs that use such arrays. Consider the two code fragments:

The programs are functionally identical and produce the same result. Version 1 accesses $X(1,1)$,$X(1,2)$,$X(1,3)$ …, whereas version 2 accesses $X(1,1)$,$X(2,1)$,$X(3,1)$ …. Version 2 executes much faster on most computers, because it can exploit cache memory much more effectively than can the first version. Version 2 accesses the elements of X in the same order in which they are stored in memory, or in cache. On first reference to array X, a portion of the cache is filled with contiguous elements of X. Whereas version 2 of the program accesses all the elements of X now in cache, version 1 only accesses one element before making a reference to $X(1,2)$, which, being from a potentially distant memory location, is unlikely to be already in cache; a portion of array X from $X(1,2)$ is loaded into cache, perhaps displacing the portion near $X(1,1)$. Version 1 is likely to run no faster than the speed of memory. Again, this phenomenon is characteristic of virtually all computers having cache memory and is not limited to supercomputers. Vastly more complex examples of

```
Version 1                          Version 2
REAL X(M,M)                        REAL
                                   X(M,M)
DO 200 I = 1, M                    DO 200
                                   I = 1, M
  DO 100 J = 1, M                    DO 100
                                     J = 1, M
    X(I, J) = 0.0
                                     X(I, J) = 0.0
100   CONTINUE                     100
                                   CONTINUE
200 CONTINUE                       200
                                   CONTINUE
```

optimization for maximum cache utilization exist. Because supercomputers are typically much more expensive than the labor of the people who use them, these painstaking optimizations can pay big dividends.

The hierarchy of disk, main memory, and cache, each one faster than the one before, is a general one. On the top of this pyramid are registers. Supercomputers typically contain a small number of ultrafast registers within the CPU. The registers hold scalar variables, such as loop counters, or accumulator variables. At all stages of the hierarchy, performance tuning involves maximizing the use of the faster components. The compiler usually decides which variables should be kept in registers.

2.4. Performance

The most commonly cited performance measure in the scientific computing world is the LINPACK benchmark (3). The benchmark involves diagonalizing a 100×100 double precision matrix. The 100×100 problem is moderately vectorizable in that good speedups over scalar execution can be achieved, but it generally does not permit vector computers to perform near peak performance. Table 1 contains an extract from the August 1991 LINPACK report. LINPACK data is reported as MFLOPS, so larger numbers are better. Because LINPACK is such a well-known benchmark, a great deal of effort is devoted to optimizing its performance on many computers. Successful efforts tuning LINPACK may or may not correlate with increased performance for other programs.

Performance achieved on single processor vector computers is governed by Amdahl's law (4). Once started, vector operations can be performed much faster than single arithmetic operations. As an example, consider a machine in which this speed ratio is 10:1, a very low ratio. Assume further that the program runs in 100 s without using vector instructions. If 10% of that program can be run in vector mode, and thereby speeded up by a factor of 10, execution time drops to $90 + 1 = 91$ s. If, on the other hand, 90% of the program can be run in vector mode, execution time is now $10 + 9 = 19$ s. This situation is illustrated in Figure 1.

Thus, only when substantial parts of the program can be run in full vector mode can significant overall speed improvements be achieved, and regardless of how efficiently the part of the program that can be run in vector mode is run, the nonvector part will still limit the execution time. Most real world, as opposed to synthetic benchmark, programs contain significant nonvectorizable portions. For this reason, it is now widely accepted that traditional vector supercomputers will not be the architecture that first delivers usable TERAFLOP (1000 GFLOP) performance.

Table 1. Sample LINPACK MFLOPS Ratings,[a] Aug. 1991

Computer	MFLOPS[b]	Computer	MFLOPS[b]
Cray Y-MP/16 prototype	403	Alliant FX/2800-200[c]	10
NEC SX-3/14	314	MIPS RC6280	10
Cray Y-MP/832[d]	275	Stardent 3010	10
Fujitsu VP2600/10	249	IBM RS/6000 model 320	9
Cray Y-MP/832	161	SCS-40	8
Cray Y-MP/416	121	Convex C-130	7.2
Hitachi S-820/80	107	DEC VAX 6000 (vector)	7.0
Cray 2S/4-128	107	IBM 3090/180	6.8
ETA 10-G	93	Alliant FX/2800-200	6.4
Cray Y-MP/832	90	Silicon Graphics 4D/420	6.0
IBM ES/9000 model 900 VF	60	FPS 264	5.9
Convex 3810	44	MIPS RC3360	4.5
IBM ES/9000 model 900	38	SUN SparcStation II	3.8
CDC Cyber 2000V	32	DecStation 5000/200	3.7
Alliant FX/2800-200[e]	29	Gould NP1	3.1
Cray 1S	27	IBM 370	2.5
IBM RS/6000 model 550	27	SUN SparcStation I+	1.6
DEC VAX model 9000	22	DEC VAX 6000 model 410	1.2
FPS model 522	20	SUN 4/260	1.1
Amdahl 1400	19	DEC VAX 8650	0.70
Silicon Graphics 4D/480[d]	18	SUN 3/260 (FPA)	0.47
Convex C-210	17	Apple Macintosh IIfx	0.41
Cydrome CYDRA 5	14	VAX 11/785 (FPA)	0.20
Cray 1S (1983)	12	Compaq 386/20 w/387	0.16
Multiflow Trace 7/300	11	DEC microVAX II	0.13

[a] Ref. 3.
[b] All data are for single processor execution except where noted. Higher numbers indicate better performance. As compilers change, these ratings may change.
[c] Two processors.
[d] Eight processors.
[e] Twelve processors.

2.5. Peak Performance

Every computer has a theoretical peak speed, the speed that would be achieved if all the pipelined functional units of the machine could be kept fully supplied with operands. It is a speed that is guaranteed never to be exceeded. These peak processing speeds make for interesting discussion and speculation; however, few programs allow vector computers to run at anything approaching peak speed. Recent LINPACK reports contain a section that describes the performance of a program that does allow vector computers to perform near peak speeds.

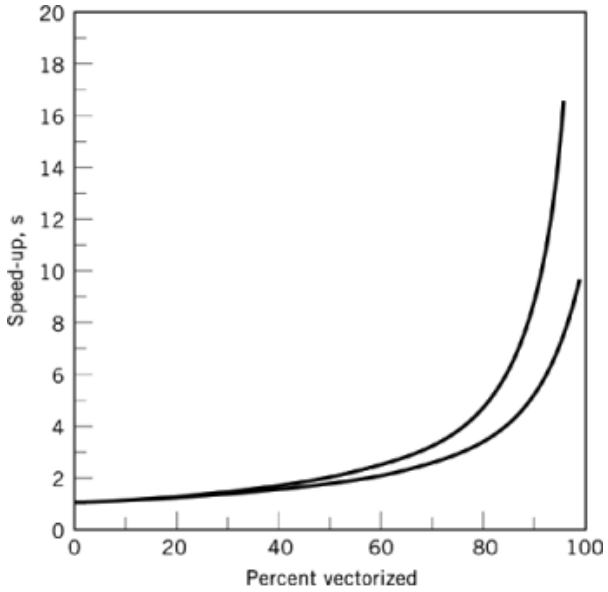

Fig. 1. Amdahl's law. Speedup as a function of the percentage of the program that can be vectorized. Lower curve vector–scalar $_{\text{speedup=10}}$; upper curve vector–scalar $_{\text{speedup=100}}$.

Supercomputers from vendors such as Cray, NEC, and Fujitsu typically consist of between one and eight processors in a shared memory architecture. Peak vector speeds of over 1 GFLOP (1000 MFLOPS) per processor are now available. Main memories of 1 gigabyte (1000 megabytes) and more are also available. If multiple processors can be tied together to simultaneously work on one problem, substantially greater peak speeds are available. This situation will be further examined in the section on parallel computers.

3. Chemistry

Many computational chemistry programs have been adapted for vector processing, often with good gains in speed. The work on adapting the programs has been done both by the original developers and by staffs at the NSF centers. Many of the optimizations are related to the use of optimized matrix manipulation routines or to recasting data structures into forms that are amenable to vectorization.

Ab initio quantum chemistry programs (5, 6) can easily be made to occupy any computer fully, and so are good candidates for a supercomputer. These programs treat individual atomic orbitals as combinations of Gaussian orbitals. Generally, the more Gaussian orbitals used in the basis set, the more accurate the results. As the problem scales as the fourth power of the number of electrons (or the sixth power when using theories that account for electron correlation), it is easy to cast a problem that requires the generation and use of several gigabytes of data. Early versions of the programs computed these two electron integrals, stored them on disk, then fetched them back into memory as they were needed for subsequent calculations. Over the years, the speed of disk drives has increased slowly, whereas CPU and compiler technologies have made much more rapid progress. It is now often more efficient to recalculate the two electron integrals rather than to wait for previously computed values to be retrieved from a disk (7).

3.1. Supercomputer Costs and Benefits

Given the great expenses associated with supercomputers, not only the initial purchase and facilities cost but also the large support staffs typically required, they are most often shared by large numbers of people, with the undesirable effect of providing to each user only a small fraction of the computer's performance. Sometimes in these situations no user gets supercomputer-class throughput. On widely shared supercomputers, large jobs are often run overnight only in order to maximize the availability of resources for daytime users. There is a great deal of debate about the ultimate value of widely shared supercomputers. Well-funded, grand-challenge, or mission-critical applications can often justify their own dedicated supercomputers; however, such projects seem to be more the exception than the rule.

4. Minicomputers

In the late 1970s and early 1980s, the typical chemistry department used a central time-shared minicomputer for virtually all its computing. These air-cooled computers required neither expensive liquid cooling nor large support staffs. They were priced within the range of typical departmental budgets and could be run according to the needs of the individuals in the department. As many as 30 users would share such a computer, often providing each user much less throughput than that provided by a personal computer. However, the cost and flexibility of these systems overwhelmed other concerns, and the minicomputer became the standard for departmental scientific computing during the 1980s, when computers still cost much more than the labor of the people using them. Painstaking efforts were undertaken to optimize computational chemistry programs for the minicomputer architectures. With computations taking days or more to run, even small percentage gains could translate to time savings of hours.

The Digital VAX rose to prominence as a departmental minicomputer and became a virtual standard in the world of chemistry. The VAX offered a user-friendly flexible environment, together with what was then considered good computational throughput. Much computational chemistry methodology was developed on the VAX.

4.1. Chemistry

The widespread availability of minicomputers and the advent of robust programs led to greatly enhanced use of *ab initio* and semi-empirical quantum chemistry techniques (8). Whereas it might once have been necessary to negotiate with computer center staff for computer time and to consult with a quantum chemistry expert in order to perform detailed structure calculations, minicomputers and easy to use quantum chemistry programs made these computations much more readily available to researchers. This trend of greater availability, enhanced ease of use, lower cost, and more power seems to be a general phenomenon, although in the case of quantum chemistry the seeming ease of use has sometimes hidden the limitations inherent to the techniques. This trend poses interesting challenges to all disciplines in which computational results can be subject to interpretation, but are easy and cheap to perform.

As the minicomputer rose to prominence so did the more widespread use of molecular mechanics as a computational technique. Whereas the quantum chemical programs deal with molecules as nuclei and electrons, the molecular mechanics paradigm treats each atom as a classical ball of a certain mass. The bonds connecting the balls are treated as classical, generally harmonic springs, and bond angles are described by similar classical terms. Through space (London), rather than through bond, interactions are typically described by Lennard-Jones potential functions.

The strength of molecular mechanics is that by treating molecules as classical objects, fully described by Newton's equations of motion, quite large systems can be modeled. Computations involving enzymes with

thousands of atoms are done routinely. As computational capabilities have advanced, so have the size and complexity of the systems modeled with molecular mechanics. Unfortunately, the execution time can scale as the square of the number of particles, but the method does hold interesting possibilities for parallel processing, which will be discussed later.

Much of the early work in molecular mechanics has been commercialized, with development continuing both in universities and by software vendors. As commercial products, these programs have been "cleaned up" with the addition of user-friendly interfaces, proper documentation, and dedicated support staff. With commercialization has also come marketing, with competitive pressures driving vendors to aggressive marketing techniques. Also, algorithmic developments that once were freely shared among colleagues are often now considered trade secrets. Chemists have certainly benefited from some aspects of the commercialization of molecular mechanics tools; however, its long-term effects are not at all clear. Again, the minicomputer was the initial enabler of what has become a pervasive technique.

The preeminent offerings in this crowded market include MacroModel (Columbia University, New York), Insight (Biosym Technologies, California), Sibyl (Tripos, Missouri), ChemX (Chemical Design, UK), BioGraf (Molecular Simulations, California), Charmm/Quanta (Polygen Corp., Massachusetts), PC Model (Serena Software, Indiana), ChemLab (ChemLab Inc., Illinois), and a large number of personal computer-based packages.

A truism of computational chemistry is that chemists will always want to model ever larger systems, or smaller systems, at ever more accurate levels of approximation. The total running time of jobs has, in general, not lowered dramatically. Computational chemists still perform calculations that take several days to complete. However, today the molecules can be much larger and the quality of the calculations better.

5. Work Stations

The mid-1980s was a turning point for minicomputers as microprocessor-based UNIX work stations began to appear. Development of the UNIX operating system began at Bell Labs in 1969. UNIX was an operating environment built for programmers by programmers. During the mid-1970s it was freely distributed to universities, many of which made significant enhancements to it. SUN Microsystems marketed a series of microprocessor-based work stations, which ran an enhanced version of UNIX from Berkeley, California. These systems offered large, high resolution, multiwindow monitors, together with computational speed that rivaled or exceeded that available on the minicomputers of the time. There began a transition that later became a stampede.

Vendors such as SUN and MIPS introduced lines of computers based on RISC (reduced instruction set computer) chips. These computers offered significant performance advantages over the CISC (complex instruction set computer) minicomputers, at least for CPU-bound work. Although there are still active debates about what RISC and what CISC are, the essence of RISC is simplicity.

The philosophy of RISC is that the CPU performs a very small number of very simple operations. Whereas a CISC-based computer might have an instruction that fetches a number from memory and updates a counter, a RISC system implements such an operation with multiple, but simple, instructions. By keeping the CPU simple, it can be more readily scaled up to ever greater speeds. The idea is that, although it might execute many more instructions than a CISC machine, it can perform its simple instructions so much faster that it gets more work done in a given time period.

The RISC versus CISC conundrum has led to the much abused and ultimately extremely confusing term MIPS (millions of instructions per second). Measures of performance that can be more directly related to a computer's ability to perform useful work should always be preferred over machine MIPS. The throughput of a computer is a function of the number of instructions to be executed, the average number of instructions that can be executed per clock cycle, and the time per clock cycle.

An additional advantage of the RISC microprocessor computers is that their implementors laid plans with a view of semiconductor and compiler technology of the 1990s. Most of the earlier CISC systems were defined

with a view of what the technology of the 1980s would bring. RISC-microprocessor-based computers hold an increasing performance advantage over CISC-based systems.

Table 2 shows timings for running version 5.0 of MOPAC (9), a semi-empirical quantum chemistry program, on a series of computers commonly used in chemistry environments. Because the table measures elapsed time, lower numbers are better. Although MOPAC 5.0 is an important program for many chemists, it is not a generally accepted industry standard performance measure. Caution should be exercised interpreting benchmarks, especially when they are run by others. Comparing the LINPACK and MOPAC benchmark data is illustrative.

Table 2. Performance According to MOPAC 5.0 Benchmark Data[a]

Computer	Cycles[b]	Elapsed time[c]
SUN 3/280 ffpa	29	39:00
SUN 4/260	29	24:18
Sparcstation 1	29	19:26
DecStation 3100	29	12:50
MIPS RC3240	29	7:08
DecStation 5000/200	29	6:18
SparcStation II	29	6:18
Silicon Graphics 4D/35	29	4:50
IBM RS/6000 model 320	29	2:40
IBM RS/6000 model 530	29	2:02
Hewlett Packard model 720	29	1:59
IBM RS/6000 model 550	30	1:19
VAX 8600	30	36:04
IBM 3090-200	30	10:08

[a] Geometry optimization of a small molecule with version 5.0 of MOPAC (9).
[b] Number of geometry optimization cycles.
[c] Computation times, min:s.

In Table 2, the computers in the MOPAC benchmark are grouped as they are to reflect differing floating-point machine representations. All these computers use 64 bits to represent a double precision floating-point number. Different behavior arises from the slightly different ways in which the bits are allocated.

Geometry optimizations in MOPAC are iterative in nature, and the program may go through a different number of cycles to achieve the same convergence criterion under different floating-point representations. In this case, the program goes through an extra cycle on both the VAX and IBM mainframe compared with most IEEE machines. Even factoring in the extra cycle it is obvious that the IEEE machines run MOPAC exceptionally well. Whereas LINPACK ranks an IBM 3090 as being significantly faster than a DecStation 5000/200, the MOPAC 5.0 benchmark shows just the opposite. The importance of well-characterized benchmarks should be clear; what is best for one person may be suboptimal for another.

The widespread confusion about MIPS and MFLOPS, together with the existence of programs that exhibit widely differing behavior on different computers, led to the formation of the SPEC group, Systems Performance Evaluation Cooperative. This group maintains the SPEC benchmark, a suite of programs having differing characteristics. Some are limited by integer performance, others by floating-point performance; some benefit from vectorization, others do not. By reporting a performance metric that is an average over several programs, it is anticipated that the SPEC rating will be more generally predictive than are benchmarks based on a single program. Of course, for those who use primarily floating-point intensive programs, the integer intensive benchmarks in the SPEC benchmark may not be of great interest; however, the purpose of the SPEC suite is to provide a general purpose rating. Note, however, that compilation is an integer intensive operation. The SPECmark rating of several well-known RISC work stations is included in Table 3.

Table 3. SPECmark Ratings[a] for Popular RISC Computers

Computer	SPEC int	SPEC fp	Rating[b]
Hewlett Packard model 730	51	100	76
IBM RS/6000 model 550	34	119	72
Hewlett Packard model 720	39	78	59
MIPS RC6280			42
IBM RS/6000 model 320	16	53	32
Silicon Graphics 4D/35			31
SUN SparcStation II	20	21	21
DEC Decstation 5000/200	18	20	19
DEC Decstation 3100			10
SUN SparcStation I			8

[a] Where available, the SPEC integer (int) rating and the SPEC floating-point (fp) rating are reported separately.
[b] Ratings are normalized to a VAX 11/780. As compilers change, SPEC mark ratings may change.

Many of the faster work stations can provide throughput similar to that observed on a crowded, shared supercomputer, especially for codes that do not benefit greatly from vectorization. The availability of such machines for less than $50,000 (much less for academic users) has once again changed concepts of what is computationally feasible. Many more people can perform computations that a few years ago were the sole domain of those with access to large-scale computing facilities, and this trend is expected to continue.

RISC work stations have been doubling in performance every two years since the mid-1980s. This growth is much more aggressive than that observed in other technologies (supercomputers, mainframes, minicomputers, etc). This trend must slow down eventually, as limits imposed by constraints such as the speed of light are approached. Computer vendors, however, are confidently predicting that the trend will continue unabated at least until the mid-1990s.

As CPU performance increases, the gap between CPU and disk and memory speeds will continue to widen. As limits of technology are approached, other techniques will be needed to gain performance advantages; more functional units, multiple processors, and so on. These approaches are discussed in the sections on minisupercomputers and parallel processing.

Technical RISC work stations have revolutionized computational science in five years.

6. Visualization

With the ever-increasing ability of more and more people to do more and more computation than ever before, the sheer volume of computational data produced can be overwhelming. As computational availability has increased, the rate-limiting step to many computational studies has become interpretation of the results, rather than waiting for the computations to complete. In any discipline, interpreting a 10-cm-thick computer printout is a daunting task. Although humans are not efficient when reading numbers sequentially, they are extremely efficient at visual interpretation because visual information is processed in a parallel fashion. Thus, if a mass of data can be presented in a meaningful visual form, the analysis of computational science can be greatly facilitated. Visual presentation not only speeds up interpretation, but also makes readily discernible insights into the results that are virtually unobtainable from a pile of paper. Visualization aids in the initial interpretation of the data and also in its communication to others.

Early visualization systems involved a special purpose graphics terminal attached to a minicomputer or mainframe time-shared host. Just as RISC work stations eclipsed mainframe and minicomputer systems for

CPU-bound work, so, too, did visualization migrate to smaller systems. The additional advantage of the work station is that the graphics subsystem can be tightly integrated with the CPU, leading to a mutually supportive combination of computing power and simultaneous visualization of the results. Just as RISC CPU power is increasing dramatically, the market is also driving a similar race for graphics performance, with ever greater drawing and rendering speeds being required. Being able to rotate a three-dimensional depiction of a molecule, perhaps colored by atomic charge, or a depiction of the electrostatic field around a molecule, can lead to great insights into detailed molecular behavior.

Two successful and widespread applications of visualization techniques in the field of chemistry are the visualization of molecular orbitals and the visualization of molecules in molecular mechanics studies.

It is now possible to "see" the spatial nature of molecular orbitals (10). This information has always been available in the voluminous output from quantum mechanics programs, but it can be discerned much more rapidly when presented in visual form. Chemical reactivity is often governed by the nature of the highest occupied molecular orbital (HOMO) and the lowest unoccupied molecular orbital (LUMO). Spectroscopic phenomena usually depend on the HOMO and higher energy unoccupied states, all of which can be displayed and examined in detail.

All the molecular mechanics programs mentioned in connection with mini-computers now run on RISC work stations with integrated graphics systems. Three-dimensional depictions of molecules can be rotated, docked with other molecules, and otherwise manipulated interactively. Important breakthroughs in understanding drug action have come about from the ability to visualize substrate molecules interacting with receptor sites. For example, there is a lock-and-key mechanism in which the substrate or inhibitor molecule must fill a well-defined cavity within the protein and interact with specific functional groups within that cavity. Molecular visualization greatly facilitates the discovery and communication of such ideas.

The challenges for visualization are at least twofold. Faster graphics hardware will be required to display and manipulate more complex data displays. More importantly, the human effort required to develop visualization systems must be reduced. It is the realm of the expert programmer to implement a usable visualization system. General purpose tools that allow the nonexpert to import data in different formats into robust visualization systems are just beginning to appear.

The evolution (revolution) of capabilities has dramatically changed the way in which chemists work, and, again, many more people can now perform and analyze many more computations than ever before, at ever diminishing costs.

7. Minisupercomputers

The so-called minisupercomputers that emerged and then declined during the 1980s were for many years some of the most interesting computers in terms of architecture. Many of the more successful aspects of their designs are expected to be incorporated into general computer design practice. The minisupercomputers were typically minicomputer-sized systems that offered performance levels of one-quarter to one-third of that available on the supercomputers of the time. As their prices were close to minicomputer prices, they were an attractive alternative for many who needed supercomputer performance but did not have supercomputer budgets. A dedicated minisupercomputer might provide more throughput than a crowded, shared supercomputer.

For all the excitement and enthusiasm of the computer architects, these computers did not meet with great success in the marketplace, and few companies remain as viable entities. One of the primary reasons for their demise seems to have been the simultaneous rise of the RISC work stations, which killed off numerous other architectural initiatives, hence the term *killer micros* (11).

Having a marketplace crowded with differing computer architectures, each perhaps requiring different strategies for achieving maximum performance, mini-supercomputer vendors laid siege to application developers to get key applications ported to their systems. Given the potentially great costs and possibly small returns

from such endeavors, these overtures were often in vain, leading to a vicious cycle of a paucity of applications inhibiting sales, and no applications because of the small installed base. The most successful porting and optimizations usually involved a partnership between the application developer and the computer vendor, but the costs were often high.

The most commercially successful of these systems has been the Convex series of computers. Ironically, these are traditional vector machines, with one to four processors and shared memory. Their Craylike characteristics were always a strong selling point. Interestingly, SCS, which marketed a minisupercomputer that was fully binary compatible with Cray, went out of business. Marketing appears to have played as much a role here as the inherent merits of the underlying architecture.

7.1. Multiflow

One of the most interesting and innovative entrants into the minisupercomputer field was Multiflow (12). The Multiflow Trace series was a VLIW, very long instruction word, architecture computer, with a great deal of fine-grained parallelism in the architecture. Each CPU contained seven independent functional units: conditional branch, two integer operations, two memory operations, and two floating-point operations. With so many functional units potentially simultaneously active during each clock cycle, a very long instruction word was needed in order to specify just what each functional unit would do on each clock cycle. Each CPU required a 256-bit instruction word, and up to four CPUs could be combined into one machine, leading to a 1024-bit instruction word. The system used banked memory, but unlike more expensive supercomputer systems that used hardware to control memory-access conflicts, the Multiflow controlled conflicts at the compiler level. The compiler became vastly more complex in order to avoid costly and complex hardware components.

The Multiflow achieved very high rates of performance using electronic components, which were relatively slow for the time. Its performance was achieved because of its ability to keep the multiple functional units busy. The compiler would aggressively rearrange code in order to achieve this goal. Once the operands for a particular instruction were available (computed or fetched from memory), that instruction could be scheduled, regardless of where it first appeared in the original code. Clearly the correctness of this approach requires a careful dependency analysis of the code.

Whereas conditional branches often destroy parallelism in more traditional computers (13), the Multiflow employed several strategies for dealing with branches. With the code fragment below, most traditional computers would need to wait until the comparison with A was complete before beginning evaluation of $D = E + F$.

$$A = B + C$$

$$IF\ (A.GT.1.0D + 03)\ D = E + F$$

$$I = I + 1$$

Because the Multiflow had multiple functional units, it would simultaneously perform both $A = B + C$ and $D = E + F$, storing both results in registers. If the result of the comparison turned out to be false, then it would not store the result D back into memory, and the calculation would be discarded. Because the two operations were done in parallel, this method took no extra time. The integer operation, $I = I + 1$, as well as four other operations, could also be performed simultaneously.

Perhaps the most unusual aspect of the compiler is that it would make compile time decisions about the most likely outcome of a comparison operation and generate code to follow that most likely path. The compiler

would insert compensation code to undo calculations already done if the branch were other than predicted. In addition, it was possible to execute a program and monitor the results of branch decisions. Armed with more accurate information about the likely outcome of conditional branches, the compiler could then generate better code, because its "guesses" would be correct more often.

Loops that could not be vectorized on conventional vector computers often performed very well under the Multiflow architecture; and, unlike vector machines, for which a person could spend a great deal of time optimizing programs, substantially less could be done on the Multiflow, as most of the work fell to the compiler anyway. All the usual optimizations for memory utilization and cache usage also applied to the Microflow. There were, of course, programs for which the compiler could not make good use of the multiple functional units, and the computer would run at the speed of just one or two individually quite slow functional units.

7.2. Others

Other architecturally interesting, but commercially unsuccessful, computers were developed by Cydrome (14), Astronautics (15), Gould, Vitesse, and others. All were creations of the ready availability of capital for computer start-up ventures in the early 1980s, but fell victim to the resulting crowded marketplace, poor marketing, and the killer micros; why work to tune a 3-MFLOPS (scalar mode) vector minisupercomputer to 10 MFLOPS performance when a killer Micro would run the original code at 10 MFLOPS with no changes at all? The minisupercomputers have retreated to the niches of highly vectorizable codes, very large memory requirements, or high input/output (I/O) requirements. As the killer micros infringe on even these areas (16), the future viability of the minisupercomputers remains an open question.

8. Parallel Processing

The vast majority of the speed increases previously described in the uniprocessor world are, in fact, parallel in nature: multiple functional units, multiple pathways to and from memory, pipelined operations, and so on. These have always been within the context of a single, tightly integrated CPU unit that executed a single sequential stream of instructions. With the speeds of such uniprocessors approaching insurmountable physical limitations, the next great leap in computational throughput can only come from parallel processing, that is, having more than one processor cooperatively working on the same problem at the same time.

Many of the problems inherent in parallel processing are illustrated by the following anecdote.

> A nobleman of the Middle Ages derived great pleasure from watching the mowing of his estate. Each month he would summon a mower from the nearby town and would then sit and watch the mowing. This typically took four hours. The nobleman reasoned that if he were to get two mowers, the mowing would be completed in two hours. This he verified the next month. He then reasoned that if he obtained the services of every mower in the country, the mowing would be completed within a matter of three seconds.

The deficiencies in this reasoning are obvious. Similarly, there often comes a point at which adding an extra processor to a problem may decrease throughput rather than increase it; imagine when the area to be mown by each person becomes smaller than a scythe swipe.

The lawn-mowing analogy is also interesting in that up to a certain point there will be a linear speedup as mowers are added. This speedup occurs because the mowers do not interfere with each other's work and have efficient mechanisms for coordinating their efforts. The lawn-mowing problem exhibits very good data parallelism.

8.1. Mutual Exclusion (MUTEX)

The idea of multiple entities all working on the same piece of work raises this issue of coordination and communication among the individual processes. A well-known example from banking is instructive. Consider two bank tellers simultaneously performing withdrawals from the same bank account. Both read the account balance and determine that the balance is $100, and so a withdrawal of $100 is allowed. Both then withdraw $100 from the account. Clearly this process needs some means by which the actions of the independent processes can be synchronized and coordinated.

The notion of an atomic operation is important for synchronization. An atomic operation is one that is indivisible. Once initiated, it will continue to completion. There are usually a large number of synchronization primitives in a parallel computer, most commonly test and set primitives, or semaphores implemented in hardware (10). A test and set operation tests the current value of a variable and optionally sets a new value, all in one indivisible operation. A semaphore forces the serialization of multiple processes around a critical section, a part of a program that must be executed by only one process at a time, such as changing the balance of a bank account, for example. When multiple processes request the same semaphore, they are automatically serialized. Of course, in systems with multiple semaphores, deadlock conditions can easily arise. For example, process 1 has control of resource $R1$ and needs resource $R2$ to continue; process 2 has control of resource $R2$ and needs resource $R1$ to continue. These situations are either avoided by careful design or must be detected and resolved as they occur. This is generally a nontrivial problem that could consume significant resources.

8.2. Parallel Languages

The computer science community generally abhors FORTRAN and has been predicting its demise for some time. A relatively new realization is that the scientific community will not be abandoning FORTRAN any time soon and that parallel computing must be fully available within a FORTRAN context. The scientific community has been slow in changing to languages that inherently express parallelism. Nevertheless, there are vigorous ongoing efforts in developing parallel programming environments and languages (17).

8.3. Performance Boundaries

Some fundamental laws work against parallel computers. For example, the same program will always run slower on a two-processor parallel computer than on a uniprocessor having double the processor speed, because the parallel computer must spend processing time synchronizing the work of its two processors, a task that the uniprocessor does not need to perform. There is also no guarantee that a program can be broken down into two computationally equal parts.

A recent victim of the killer micros was Evans and Sutherland's parallel computer development effort, halted in 1990. Their architecture combined a small number of approximately 1-MFLOPS processors into semi-independent functional units. Several of these units could, in turn, be combined to form a processor hierarchy, building up to systems that were expected to cost between 1 and 8 million dollars. With the advent of 10-MFLOPS uniprocessor killer micros, such an architecture became irrelevant and the project was halted. The RISC killer micro could deliver the same level of performance as could the combined efforts of 10 of the 1-MFLOPS processors, even with the unlikely assumption that the problem could be perfectly parallelized across 10 processors.

8.4. Speedup

The term *good performance* merits discussion. The ideal parallel computer has as many as an infinite number of processors, as much as an infinite amount of zero-latency shared memory, and all interprocessor communications require zero time. If an infinitely parallelizable problem takes T seconds to execute on one of the

processors, then it will take T/N seconds to execute on N processors. More commonly this is referred to as an N-fold speedup. Real computers and problems are different, and full N-fold speedup with N processors is impossible. One measure of the efficiency of a parallel computer is how close it comes to achieving N-fold speedup with a given program. This is a strong function of both the program and how well the parallelism inherent to the problem can be mapped onto the parallelism of the computer. In many cases problems may be too large to run on a single processor, making speedup measurements difficult.

The other kind of performance scaling is, for a constant number of processors, how does the running time change as the size of the problem increases? With more grid points, more orbitals, more molecules, and so on? Whereas a uniprocessor may exhibit m^3 behavior, where m is some parameter of the problem, a different, parallel algorithm may exhibit a better-size scaling behavior, growing slower than the serial algorithm. As with many asymptotic effects, this may only be significant for large m, but it is frequently a regime of great interest. If the inherently serial parts of the problem scale unfavorably with increased problem size, those parts may come to dominate performance, but this is rare.

8.5. Cache Coherence

Another effect that leads to reduced throughput from computers with multiple independent processors is the issue of cache coherence (18). The performance of many of these machines is critically dependent on the cache utilization of the program. For efficiency, individual processors may store frequently used variables in cache only and not write the value to memory immediately (a so-called write-back cache system). A problem arises when another processor needs to access the current value of that variable. The most current value for that variable may be in another processor's cache memory, rather than in main memory. Multiprocessors with cache memories require explicit mechanisms for ensuring cache coherency among the processors. In many such systems the trend has been to implement a separate bus used just for such interprocessor communication and synchronization operations, rather than taking up bandwidth from the main system memory bus. Nevertheless, a shared bus architecture will always be of decreasing effectiveness as more and more processes are added, because the bandwidth shared among the processors is finite.

The obvious solution to the limitations imposed by shared bus communications is to fully connect each processor to all other processors via dedicated pathways. The problem is that the number of such pathways grows rapidly, $N(N-1)/2$, where N is the number of processors. The inherent costs and complexity of such a system render it an impractical solution for large-scale parallel computing.

8.6. Types of Computers

Computers can be classified by Flynn's taxonomy (19). The three important classes are SISD: single instruction, single data; SIMD: single instruction, multiple data; and MIMD: multiple instructions, multiple data.

Within the SISD class are traditional uniprocessor computers, a single instruction stream operating on a single stream of data. SIMD machines have a single instruction stream, but multiple data streams. This class ranges from single processor vector machines, where a single (vector) instruction initiates action on multiple pieces of data, to machines having multiple identical processors, which all execute the same instruction stream but on different data streams. MIMD machines include the multiprocessor shared-memory vector supercomputers, where each processor has its own instruction stream and works on different streams of data.

Unfortunately, Flynn's classification, although commonly used, is quite restrictive when discussing parallel-architecture computers. There have been several attempts to formulate more detailed classification schemes for the great variety of parallel computers now available. None of these efforts have been entirely successful, and none appear to be in general use. A discussion of representative machines from some of the more common classes follows.

8.7. Coarse-Grained Parallelism

The coarsest grained parallelism is the situation in which separate networked computers cooperate to work on a single problem. The computers themselves could be parallel-architecture machines, but that is not important. Communication between the computers is by a network. All computers function as fully autonomous units, perhaps even running different operating systems. As data transmission through a network is usually much slower than transmission through a memory bus, this model will only be effective when relatively large chunks of the problem can be given to individual computers. If the problem requires a great deal of interprocessor communication, the computational throughput could degrade to a level limited by the speed of the network (see NETWORKS).

Many institutions have hundreds, or even thousands, of powerful work stations that are idle for much of the day. There is often vastly more power available in these machines than in any supercomputer center, the only problem being how to harness the power already available. There are network load-distribution tools that allocate individual jobs to unused computers on a network, but this is different from having many computers simultaneously cooperating on the solution of a single problem.

Currently the most widely used commercially available product to support this paradigm is LINDA (20), which maintains a distributed name space for the variables in a network program. Although LINDA successfully hides much of the complexity of network computing from the user, performance is critically dependent on the nature of the problem. Problems that can be decomposed into large, semi-independent portions will generally do well, whereas problems in which the inherent parallelism is finer grained are less likely to exhibit good performance.

Another notable implementation of multiple independent computers is the LCAP (loosely coupled array of processors) project of IBM (21). This system ties together four multiprocessor IBM 3090 Vector Facility computers in a ring topology. In addition to the local memory associated with each processor, a bank of dual-ported memory is shared between neighboring processors. This system exhibits very good performance characteristics on problems that could be mapped onto a coarse-grained architecture, but offers lesser performance when the intermachine communication requirements become greater. Interprocessor communication within a given computer can be done by fast shared memory, but communication time between processors in different computers is comparatively lengthy. Optimization for such a system involves dividing the problem into coarse-grained portions to execute on the different computers, further subdividing those portions into portions that will execute on different processors within the same computer, and then optimizing those individual portions for vector performance. Given the rather extreme costs of such a system, and the challenges of optimization, LCAP has been a valuable research tool rather than a commercial success.

The observation that certain kinds of parallel-computing architectures best support only certain kinds of problems seems to be general. The further observation that interprocessor communication can be the primary impediment to parallel performance is also general. As of this writing, any hope of a truly general purpose parallel computer seems to be remote. The best hope may lie in software efforts that describe problems at higher levels of abstraction, which can then be ported and optimized for different parallel architectures (22).

8.8. MIMD Multicomputers

Probably the most widely available parallel computers are the shared-memory multiprocessor MIMD machines. Examples include the multiprocessor vector supercomputers, IBM mainframes, VAX minicomputers, Convex and Alliant minisupercomputers, and Silicon Graphics server machines. Most of these computers have several CPUs sharing a common memory and usually a common bus. The shared memory, common bus architecture is both the best and most limiting feature of these machines. It is often quite easy to implement parallel programs on these machines, sometimes as simple as using a compiler option directing the compiler to look for opportunities for parallelism within the code. It is unlikely, though, that peak performances will be achieved

with so little human intervention and, in general, programs must be restructured to expose the parallelism inherent in the problem to the compiler. For those machines with vector instructions, multilevel optimization must be performed both for vector performance and for parallel performance. Because all processors share a common memory subsystem, and perhaps also a common path to that memory, the relative efficiency of these computers declines as more processors are added. In extreme cases, throughput may actually decrease as processors are added.

Such a decrease in efficiency, but nevertheless increasing throughput, is illustrated by the following data from LINPACK for the Cray Y-MP/832 (3).

Number of processors	MFLOPS, observed	MFLOPS, peak	Efficiency, %
1	324	333	97.3
2	604	667	90.5
4	1159	1333	86.9
8	2144	2667	80.3

This decreasing efficiency is a general characteristic of shared memory, shared bus computers. This example shows unusually high efficiency compared with many other programs. This may be because LINPACK is such a common benchmark that much effort has been devoted to optimizing it for both vector and parallel computers.

8.9. Crossbar Systems

Several systems have been built using crossbar architectures for connecting memory and individual processors. Although a crossbar does not maintain dedicated electrical pathways between all points, it can establish such paths as needed. The BBN Butterfly (23) and later Monarch (24) exemplify this approach. The Monarch consists of as many as 65,536 (2^{16}) processors communicating with half that number of memory modules linked by a pipelined crossbar switch. Although each processor has six independent functional units, the instruction word format only allows for initiation of two operations per cycle. There is no local memory associated with each processor; all memory is shared memory accessible through the crossbar. In order to avoid the complexities associated with cache coherency, the Monarch processors do not have a local data cache, although they do have a local instruction cache. To compensate for the lack of local data cache and memory, the Monarch processors each have 64 registers. The delays associated with accessing global memory through the crossbar are significant, more than 1 μs; however, the effects of this latency can be minimized by performing other instructions, for which operands are already available in the registers, while awaiting completion of a memory access. The memory subsystem is designed to spread data among as many individual memory modules as possible. An interesting feature of the machine is the ability for every processor to simultaneously read from a single memory location in one cycle. Mutual exclusion is implemented by giving individual processors the ability to gain exclusive access to a given memory location. The future of the Monarch remains in doubt; perhaps it will become another victim of the killer micros.

8.10. Linear Topology

An interconnect strategy for which the hardware cost scales linearly is to conceptually arrange the processors in a line, with a dedicated interconnect between adjacent processors. The full bandwidth of each connection is then dedicated solely to communication between the adjacent cells. The Warp computer (25) implements such an architecture, with the two ends connected via an interface unit to a host computer. The original system combined 10 identical 10-MFLOPS processors for a combined peak performance of 100 MFLOPS. Each cell

could transfer 80 megabytes per second to and from its neighboring cells, in addition to 20 million 16-bit addresses. Clearly the aggregate bandwidth available within the system is much larger than could be obtained from a single shared bus of comparable cost; however, such a system will perform poorly if a given problem is dominated by data transfers between nonadjacent processors.

The original Warp computer exhibited a wide range of performance, depending on the nature of the problem. With ten 10-MFLOPS processors, 100-MFLOPS peak, a 100×100 matrix multiply was performed at an observed 79 MFLOPS; whereas averaging a 512×512 image to produce a 256×256 image ran at 3 MFLOPS, perhaps slower than if the problem had been run on just a single processor. The critical importance of understanding the problem, the machine architecture, and the possible synergies, or lack thereof, between the two should be apparent.

8.11. Transputers

At higher levels of connectedness there is a wide variety of parallel computers. A great many parallel computers have been built using INMOS Transputer chips. Individual Transputer chips run at 2 MFLOPS or greater. Transputer chips have four communication channels, so the chips can readily be interconnected into a two-dimensional mesh network or into any other interconnection scheme where individual nodes are four-connected. Most Transputer systems have been built as additions to existing host computers and are SIMD type. Each Transputer has a relatively small local memory as well as access to the host's memory through the interconnection network. Not surprisingly, problems that best utilize local memory tend to achieve better performance than those that make more frequent accesses to host memory. Systems that access fast local memory and slower shared memory are often referred to as NUMA, nonuniform memory access, architecture.

Another characteristic of Transputer systems is that there may be a start-up phase during which the data pass from the host and are distributed throughout the processor array. Only the edge nodes are physically attached to the host system, so the time needed for the data to propagate to all processors is potentially significant. Depending on the nature of the problem, there may also be an additional, time-consuming, final collection phase during which the resulting data pass from the array back to the host.

Because of their relatively low cost and simplicity, Transputer systems can be built readily. Many systems are marketed as application accelerator boards for personal computers and work stations. A single board that turns a standard work station into a "supercomputer" (for one application at least) can be very attractive, especially in application-specific situations (26). Transputer-based systems typically cost between $1000 and $200,000, depending on the number of nodes, among other things.

8.12. Hypercube and Massively Parallel Systems

The NCUBE computer is one of a large class of hypercube topology computers. A hypercube of dimension n contains 2^n nodes (Fig. 2). Hypercube topology offers significant advantages in parallel-computer design (27). Hypercubes represent a reasonable trade-off between the number of connectors at each node, with the maximum number of interprocessor "hops" a message will need in order to pass from any one node to another being $\log_2(n)$. As this value grows slowly with n, such systems are scalable. Many other kinds of processor topology can be mapped onto a hypercube, with less-connected schemes, a mesh for example, achieved by simply not using existing connections. In order to double the number of processors in a hypercube array, it is only necessary to add one extra connection to each processor. This is a favorable implication for large-scale implementations.

The NCUBE 2 is a MIMD machine, having between 8 and 8192 processors, each with as many as 64 megabytes of local memory: in an 8192-processor machine there is a maximum of 4 megabytes of memory per processor. All memory is local. A processor needing the value of a variable not already in its own local memory must send a message to the processor whose memory contains that variable. Message passing is performed

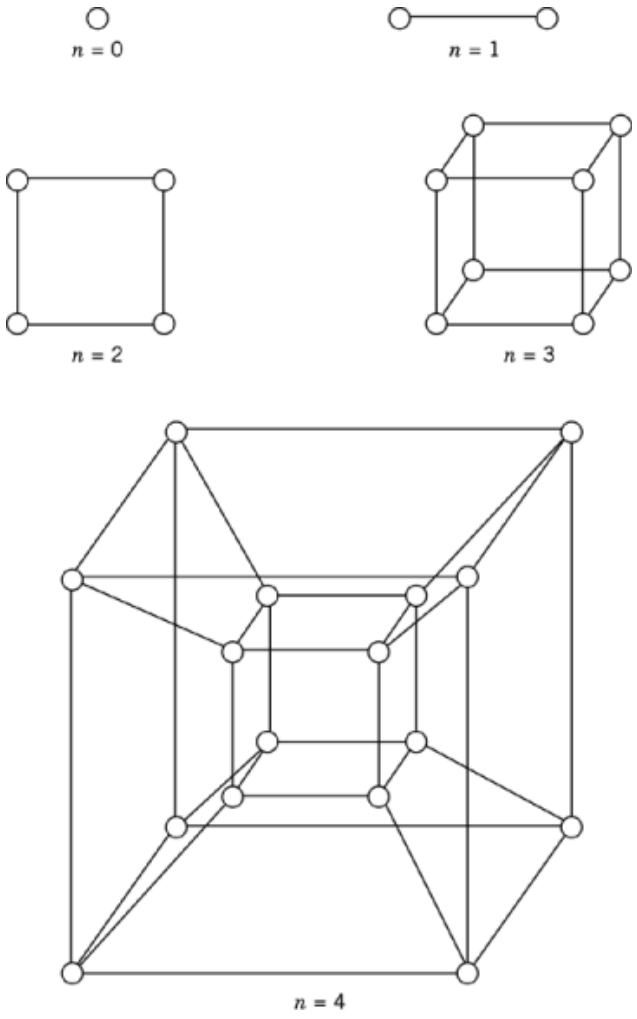

Fig. 2. Hypercubes for $_{n=0}$, 1, 2, 3, 4 (23).(Courtesy of IEEE.)

by dedicated routing units, which means that most of the overhead of message passing is not carried by the computing elements, thereby freeing them to work on computing rather than on message passing. Considerable instruction overlap is possible, so that the latencies involved with remote memory accesses can be overlapped with computation. With each processor rated at 2.4 MFLOPS, an 8196-node system has available a peak speed of 27 GFLOPS in double precision. NCUBE has been a commercial success.

The NCUBE and similar computers often exhibit very good performance on grid-based problems. There is a natural mapping from the spatial nature of the grid to the interconnectivity of the hypercube. Most such grid-based methods do not impose requirements for distant communications; the information needed to update a node value is usually a function of only nearest-neighbor grid points. Finite-element problems that do not perform optimally on vector-architecture machines may show to great advantage on a hypercube MIMD or SIMD system. An example given by NCUBE cites a fluid dynamics study in which a 128-processor system performed 1.8 times faster than a Cray X/MP, with every expectation that the problem would scale linearly when implemented on larger hypercubes. Often the parallelism inherent in a problem can be naturally

expressed in a way that maps well onto a parallel computer, whereas exposing such parallelism for a vector computer may be quite challenging.

Molecular mechanics simulations can be readily mapped onto such kinds of machine architecture by using the spatial locality of the atoms to determine their allocation to processors. Short-range van der Waals forces can usually be accurately modeled with a cut-off distance of less than one nm, so interprocessor communication requirements can also be localized.

Many problems obviously cannot be decomposed into semi-independent pieces of similar running time. Other problems can be decomposed into subproblems, the running times of which may be quite different and unknown until execution actually occurs. If these program portions are statically allocated to individual processors, or groupings of processors, there exists the possibility for having large parts of the machine idle while waiting for a hot spot to catch up. A great deal of theoretical work is devoted to the problem of dynamic load balancing in MIMD-architecture machines (28).

Whereas the NCUBE and Intel iPSC MIMD computers utilize relatively complex chips at each node, the Connection Machine (29) combines as many as 65,536 simple processors, each of which has 8K or more of local memory (a system maximum of 8 gigabytes in a fully configured system). When operating at full capacity, a Connection Machine can perform 3500 million 32-bit integer operations per second. If equipped with floating-point processors, its peak speed is 21 GFLOPS (double precision). Like the NCUBE, the Connection Machine can be logically subdivided into independent subcubes, thereby facilitating sharing by multiple users and experiments with differing numbers of processors.

The massively parallel approach adopted in the Connection Machine has been termed data parallel. Whereas a uniprocessor must sequentially step through large amounts of data, a data parallel machine moves processors to the data. Aggregate memory to processor bandwidth in the Connection Machine is more than 700 megabytes per second.

The Connection Machine computers have been quite successful, finding application in a wide variety of fields despite relatively high costs. Many image-processing algorithms exhibit good data locality and perform well on a Connection Machine as do grid-based fluid dynamics calculations. With an ability to simultaneously examine independent pieces of data, Connection Machines have found application in full text searching document retrieval areas. Partial differential equations can also be solved with very high efficiency. Protein sequence comparison is another naturally parallel problem well-suited to the Connection Machine. Molecular dynamics calculations have also been adapted to the Connection Machine with good results obtained in the evaluation of nonbonded terms (30).

As of this writing, massively parallel computers such as the Intel Delta, the Thinking Machines CM-2, and NCUBE 2 are yielding as much as 11 GFLOPS on a large-scale variant on the LINPACK benchmark (3). Interestingly, these levels of performance are achieved with much larger problem sizes than used in the traditional LINPACK benchmark.

8.13. Application-Specific Hardware

Another interesting trend is the development of application-specific hardware. These systems are usually minimally programmable, but can offer exceptional performance on the class of problem for which they are designed. A particularly interesting application-specific processor for molecular mechanics is currently being developed (31).

8.14. Trends

The parallel-computingmarketplace is one of rapid change. New companies are developing computers based on new ideas, some existing vendors are working to enhance the performance of their existing machines, and others are giving up. An end user contemplating a purchase from this market is faced with a bewildering

array of choices. Unless the user has only one application, there may be no one computer that is best for all the problems of interest. Additionally, most problems will require at least some effort, and potentially a great deal of effort, to achieve good performance on a parallel computer. Making an *a priori* prediction of the likely performance a given problem might exhibit on a particular parallel machine is often difficult. Although some problems will be obviously analogous to other problems for which good parallel algorithms and experience already exist, at other times there will be no such lead. As vectorization is a kind of parallelism, it is possible that vector-optimized code may also be well optimized for a more parallel computer. Algorithms that had been discarded as inefficient on serial computers are now being found to be optimal for various parallel architectures.

As of this writing, the parallel-processing industry is at an early stage. Further changes and maturing over the years can be expected, including more processors, faster processors, higher bandwidths, larger memories, faster I/O subsystems, and better visualization systems. The most important, but most difficult, advances that must come are in software. In all the cases previously discussed, the ready availability, ease of use, and lowered cost of a given computer, technology has led to a dramatic increase in the utilization of computations running on those kinds of machines. A similar increase should be expected with parallel computers; however, the great breakthrough has not yet arrived. Parallel computing currently remains difficult, often expensive, but frequently very rewarding.

BIBLIOGRAPHY

"Computers" in *ECT* 3rd ed., Vol. 6, pp. 701–719, by T. J. Harrison, International Business Machines Corp.

Cited Publications

1. R. W. Hockney and C. R. Jesshope, *Parallel Computers*, Adam Hilger, Bristol, UK, 1981; R. W. Hockney and C. R. Jesshope, *Parallel Computers 2*, Adam Hilger, Bristol, UK, 1988.
2. G. H. Golub and C. F. Van Loan, *Matrix Computations*, 2nd ed., Johns Hopkins University Press, Baltimore, Md., 1989, p. 4.
3. J. J. Dongarra, *Performance of Various Computers Using Standard Linear Equations Software*, Oak Ridge National Laboratory, Oak Ridge, Tenn., Aug. 1991.
4. G. Amdahl, *AFIPS Conf. Proc.* **30**, 483–485 (1967).
5. W. J. Here, L. Radom, P. R. Schleyer, and J. A. Pople, *Ab Initio Molecular Orbital Theory*, John Wiley & Sons, Inc., New York, 1986.
6. M. J. Frisch, M. Head-Gordon, and J. A. Pople, *Chem. Phys.* **141**, 189 (1990).
7. S. Obara and A. Saika, *J. Chem. Phys.* **86**, 3963 (1986).
8. G. A. Segal, *Semiempirical Methods of Electronic Structure Calculation, Modern Theoretical Chemistry*, Vols. **7** and **8**, Plenum Publishing, New York, 1977.
9. *Journal of Computer-Aided Molecular Design* **4**(1), (1990). Special issue devoted to MOPAC.
10. A. Dinning, *IEEE Computer* **22**, 66 (1989).
11. Attributed to Eugene Brooks.
12. J. J. O'Donnell, *Research and Development*, Trace/300 Series: Technical Summary, Multiflow Corp., Branford, Conn., Jan. 1990, p. 80.
13. D. W. Wall, *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 1991, p. 176.
14. B. Rau, and co-workers, *IEEE Computer* (Jan. 1989). Special issue devoted to design of high performance computers.
15. J. E. Smith, *IEEE Computer* **22**, 21 (1989).
16. B. Smith, *17th Annual Symposium on Computer Architecture*, Seattle, Wash., May 1990.
17. *Comp. J.* **34**(4), 289 (1991). Special issue on concurrent programming.
18. *IEEE Computer* (June 1990); M. D. Hill and co-workers, *18th Ann. Int. Symp. Comp. Archit.* **19**(3), 298 (1991).
19. M. J. Flynn, *IEEE Trans. Comput.* **C-21**, 948 (1972).

20. N. Carriero and D. Gelernter, *ACM Trans. Computer Systems* **4**(2) (May 1986).
21. D. Folsom, *MOTECC, Modern Techniques in Computational Chemistry*, Escom, Leiden, The Netherlands, 1990, p. 1091.
22. J. C. Browne and co-workers, *Proc. 1989 Int. Conf. Parallel Processing* **2**, 98 (1989).
23. *Butterfly Product Overview*, BBN Advanced Computers, Cambridge, Mass., 1987.
24. R. D. Rettberg and co-workers, *IEEE Computer*, 18 (Apr. 1990).
25. M. Lam and co-workers, *IEEE Trans. Computers* **C-36**, 1523 (Dec. 1987).
26. R. S. Cok, J. Gerstenberger, C. Lawrence, H. Neamtu, and D. S. Cok, *Transputing '91*, Vol. **1**, Sunnyvale, Calif., Apr. 22–26, 1991, p. 15.
27. J. P. Hayes and co-workers, *IEEE Micro*, 6 (Oct. 1986); G. C. Fox and P. C. Messina, *Sci. Am.*, 67 (Oct. 1987).
28. H. J. Siegel and M. Jeng, *Proc. 1989 Int. Conf. on Parallel Processing* **2**, 57 (1989).
29. W. D. Hillis, *Sci. Am.*, 108 (June 1987).
30. B. Murray, P. Bash, and M. Karplus, *Molecular Dynamics on the Connection Machine*, Thinking Machines Corp., Cambridge, Mass., 1989.
31. B. R. Brooks and co-workers, Molecular Graphics and Simulation Laboratory, NIH, Bethesda, Md., private communication, Aug. 1991.

General References

32. *IEEE Computer* (Sept. 1991). Issue devoted to trends expected into the year 2000.
33. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan-Kaufmann, San Mateo, Calif., 1991.
34. R. W. Counts, *J. Comp.-Aided Molec. Des.* **5**, 167 (1991).

IAN WATSON
Consultant

Related Articles

Computer-aided design and manufacturing; Computer-aided engineering