Parallel Scientific Computation

W. Daniel Hillis and Bruce M. Boghosian

Massively parallel computers offer scientists a new tool for computation, with capabilities and limitations that are substantially different from those of traditional serial computers. Most categories of large-scale scientific computations have proven remarkably amenable to parallel computation, but often the algorithms involved are different from those used on sequential machines. By surveying a range of examples of parallel scientific computations, this article summarizes our current understanding of the issues of applicability and programming of parallel computers for scientific applications.

Traditionally, most digital computers have been sequential in that they perform a sequence of arithmetic operations, one at a time. Today the fastest computers are parallel computers in which thousands or even tens of thousands of processors operate simultaneously. These massively parallel computers offer scientists a new tool for computation, a tool that presents an unfamiliar set of capabilities and limitations to users of traditional sequential computers.

To exploit parallelism efficiently, massively parallel computers often require different programs or even different algorithms. The development of parallel software once seemed to be an almost insurmountable problem for parallel machines. It was assumed that many or even most computations were inherently sequential in nature, and therefore that parallelism would be suitable for only a narrow range of applications. Today there are hundreds of massively parallel supercomputers in everyday use, in thousands of different applications. Most categories of large-scale scientific computations have proven amenable to parallelism, including some computations that seemed "obviously" sequential. This article gives a range of examples of parallel computations that exploit parallelism, summarizes our current understanding of which types of scientific applications are suitable for parallel machines, and discusses some of the issues involved in parallel programming.

Data Parallelism

There are several different ways of writing programs for parallel computers. One common programming method is to run a conventional sequential program on each processor of the parallel machine with explicit commands inserted to send and receive messages to and from other processors. These explicit commands are usually supplied in the form of a library of message-passing subroutines, such as the CMMD or PVM (Parallel Virtual Machine) library. This message-passing model treats the parallel computer as a network of independent communicating processors, each with its own memory (*1*). Some programmers find message passing easy to understand because the programs are written in a standard sequential programming language.

An alternate model of parallel programming is the data-parallel model, in which a single program specifies the coordinated activities of all the processors. This single program consists of a sequence of inherently parallel operations. The data-parallel model is intuitive to many scientists and mathematicians because it corresponds very closely to standard mathematical array notation. For example, in array notation, we may write the equation $A + B = C$, indicating the addition of two arrays. If the arrays are say, 1000 × 1000 elements, then the single addition operator corresponds to the addition of 1,000,000 numbers. The parallelism is implicit in the operator. This article will describe algorithms that use the data-parallel model, although equivalent algorithms could also be expressed by message passing.

The data-parallel model is a shared-memory model; that is, all data is accessible to all processors. This is in contrast to the local memory model of message passing, in which each processor can directly access only its own subset of the data. In the data-parallel model, it is the responsibility of the compiler and the operating system to assign the operation to specific processors and to make sure that the data necessary to perform those operations is available at the processor. For example, in a machine with a million processors the compiler could perform the 1000 × 1000 array addition mentioned above by assigning to each processor the task of adding one pair of array elements. If this operation were running on a machine with only 500 processors, then the compiler would need to assign the job of adding 2000 array elements to each of the processors. This assignment may be done automatically or, as will be discussed below, with hints from the programmer. Normally a data-parallel program is written without explicit reference to the number of processors in the machine, so the same program can run on machines with different numbers of processors.

Because the primitives of data-parallel programming are operations on entire arrays, data-parallel programs are often simpler than sequential counterparts. The array addition mentioned above can be written in a data-parallel language [High-Performance Fortran (*2*)] as follows

```
C = A + B                                    (1)
```

In parallel Fortran, as in mathematical array notation, the iteration over the elements of the array is implicit. In a sequential language like Fortran 77, this would need to be expressed as a sequence of scalar additions

```
DO I = 1, 1000
 DO J = 1, 1000
    C(I,J) = A(I,J) + B(I,J)                 (2)
 ENDDO
ENDDO
```

Data Parallelism and Computational Science

Just as Wigner spoke about "the unreasonable effectiveness of mathematics in the physical sciences" (*3*), there seems to be a certain natural elegance and effectiveness in the application of data parallelism to science and engineering problems. One reason is that the laws of physics themselves, by their very nature, are parallel. Indeed, they are usually expressed mathematically in data-parallel form. For example, Faraday's law

$$\frac{\partial \mathbf{B}(\mathbf{x},t)}{\partial t} = -c\nabla \times \mathbf{E}(\mathbf{x},t) \qquad (3)$$

has a dummy variable, $\mathbf{x}$, indicating that the equation holds at every point of space simultaneously. When the field changes, it does so concurrently, at each point in space.

In the simplest applications of data parallelism, this spatial parallelism is implemented by manipulating a data structure which is analogous in structure to

The authors are at Thinking Machines Corporation, Cambridge, MA 02142.

physical space. For example, in grid-based codes (such as, finite difference, finite element, and so on) arrays represent grid points of a discrete spatial lattice. When these arrays are distributed over a parallel computer, each processor is responsible for computing the physics at one or more points in space.

In other situations, the parallel index need not be over physical space. For example, the equations of motion integrated by a molecular dynamics code might have the form

$$\dot{\mathbf{x}}_i = \mathbf{p}_i/m_i \quad (4)$$

$$\dot{\mathbf{p}}_i = \sum_j \mathbf{F}_{i \leftarrow j}(\mathbf{x}_i, \mathbf{x}_j) \quad (5)$$

where the index i runs over all the molecules in the system. Once again, the fact that i is a dummy index indicates that the form of these equations is the same for every molecule in the system. Thus, the arrays in such molecular dynamics codes that are dimensioned over the number of molecules present might be distributed so that each processor is assigned to one or more molecules.

In these simple examples, the fact that the same physical law applies to each point of a space implies that a similar computation is being applied to every element of an array. This sort of operation is particularly easy to express using data-parallel notation, and is representative of the simplest sort of data-parallel algorithm. In real problems there are often singularities and boundary conditions. These are generally expressed as parallel conditionals. For example, the program fragment

```
WHERE(R.GT.0.0)
  PHI = 1/R
ELSEWHERE
  PHI = 0.0
ENDWHERE                    (6)
```

fills array PHI with the inverses of the corresponding elements of array R where those elements are positive, and with zeros elsewhere.

The simple examples mentioned above all involve spatially local interactions, but locality is not required for efficient parallel implementation. A fundamentally nonlocal equation, such as

$$\frac{\partial N(\mathbf{x},t)}{\partial t} = \int d^3x\, G(\mathbf{x},\mathbf{x}')N(\mathbf{x}',t) \quad (7)$$

is also inherently parallel since it holds for all $\mathbf{x}$. Indeed, local problems are often transformed to nonlocal representations for solution. For example, spectral methods use representations based on the spatial or temporal frequency components of the phenomenon being modeled. These methods are nonlocal, but also have a natural parallel implementation.

Nonlocal Data Communication—The Fast Fourier Transform

A simple example of a parallel algorithm with nonlocal interactions is the Fast Fourier Transform (FFT) algorithm (4), often visualized on a "butterfly network" (Fig. 1). In the illustration, the data to be transformed starts at the left-hand side, and flows to the right. When the path splits, the data is copied onto each branch. Along some of the paths (not indicated) the data must be multiplied by constants (roots of unity). Where paths coalesce, the incoming data is added. The result of the transform appears on the right.

In the illustration, the algorithm has eight inputs, and takes place in three "stages." In the first stage, each datum is moved to the pathway a distance one away on the diagram; in the second stage, each is moved a distance two away; in the final stage, each is moved a distance four away. More generally, if the FFT of N data is desired, there must be $\log_2 N$ stages of data movement, each involving the movement of N data. Thus, the algorithm has complexity $O(N \log N)$.

Consider the implementation of this transform on a parallel computer of M processors (For simplicity, let us assume here that M and N are powers of two, and that $N \geq M$.) As mentioned earlier, the compiler can distribute the data among the processors in several different ways. Let us suppose that the eight data elements ($N = 8$) on the right side of the illustration are distributed among four processors ($M = 4$), with two data elements per processor. Then

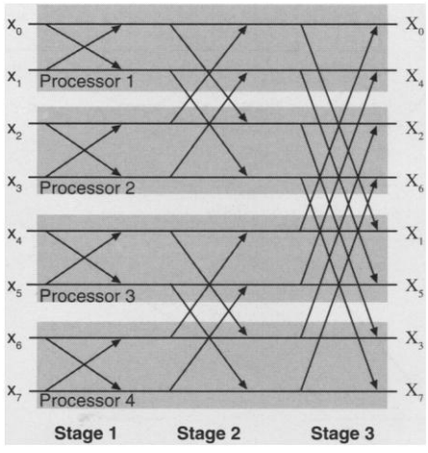

Fig. 1. The Fast Fourier Transform is an example of a parallel algorithm that requires nonlocal communication. Here x denotes the initial data, and X denotes its Fourier transform. Each stage of the algorithm consists of a parallel operation that corresponds to a single statement in a data-parallel program. Computing the transform of N data points requires $O(\log N)$ time on a parallel machine with $O(N)$ processors.

it is clear that the last two stages involve interprocessor communication, while the first stage involves only local computation (that is, the shaded boxes in the figure represent processors). Moreover, the communications patterns involved are very regular, and can be implemented easily on a variety of parallel computers (5).

More generally, there are $\log_2 M$ stages that involve interprocessor communication, and $\log_2 (N/M)$ stages that involve $O(N/M)$ local computations. The total time required thus scales as

$$\frac{N}{M}\left[k_1 \log_2 M + k_2 \log_2 \left(\frac{N}{M}\right)\right] \quad (8)$$

where k_1 is the time required for the steps which involve communication, and k_2 is the time required for those which do not. Note that when $M = 1$ (a serial computer), this reduces to $k_2 N \log N$; whereas when $M = N$ (a data-parallel implementation), this becomes $k_1 \log_2 N$.

This algorithm can be further improved by consolidating the communications steps: One can begin by performing the stages that are local, and then perform a permutation of the array so that the stages that used to involve interprocessor communication are now local, and vice versa. This has the effect of isolating the interprocessor communication required to one single step—the data permutation.

Data-parallel software libraries automate this entire process so that the details are invisible to the user (6). In a high-level data-parallel language, an FFT is invoked by a single call, with the array passed as an input. The compiler and library routines then implement the algorithm described above. This makes it straightforward to implement spectral and spectral-element codes that will run efficiently in parallel.

Data-Parallel Algorithms

Be it local or nonlocal, the simple type of spatial parallelism mentioned above is relatively easy to understand. Nevertheless, it is often preferable to use a representation of a problem that is yet more abstract. For example, it is common practice to solve partial differential equations with so-called implicit methods, in which the calculation cannot be time advanced independently at each gridpoint, but rather only together in a self-consistent fashion that requires the solution to a set of coupled linear equations at each time step. Such solution may then be carried out by iterative methods, such as the conjugate gradient algorithm (7), or by direct methods.

Because these types of methods are normally implemented on serial machines, many researchers have assumed that there is

something about them that is fundamentally sequential. In fact, implicit methods and direct solvers have natural representations in parallel machines (8).

In practice there are surprisingly few problems that are fundamentally sequential in nature. Many problems that appear to be sequential can be parallelized by the use of a different algorithm. We will explain in detail a simple example of an apparently sequential problem, called the pointer-following problem (9), that can be solved in parallel. The method used to solve this problem can be applied to many other situations that seem to require sequential processing.

The pointer-following problem begins with a chain of memory locations set up in memory as follows: An identified location, say location 0, is the first link in the chain. This memory location contains the address of the second link in the chain, or in computer science terms, a pointer to the location of the second link. The second link, in turn, contains a pointer to the third link, and so on. The last link in the chain contains the special value NULL. The problem is to find the address of the last link. Naively, it would seem that this problem requires sequential access of each link to reach the end. This would require a million steps to reach the end of a chain with a million links, and since each operation depends on the one before it, it is difficult to see how parallel processing could be used to advantage.

In fact, there is a parallel algorithm. On a parallel processor with a million processors, the millionth link can be found in 20 steps (since 20 is the ceiling of the base 2 logarithm of 1,000,000). Here is the program

```
for all k in parallel do
    A[k] := P[k]
    while A[k] =/ null and
    A[A[k]] =/ null do
      A[k] := A[A[k]]          (9)
    end while
end for
```

In the above program, P is the array of linked pointers. The program uses an algorithm called pointer-doubling (Fig. 2). The idea behind pointer doubling is to reduce the length of the list by half on each step. Each link in the chain finds the address of the link two steps down the chain. This can be accomplished for all links in the chain simultaneously. When the program terminates, a pointer to the last link is stored in each element of array A.

The pointer-doubling algorithm is an example of a log-time algorithm; an algorithm whose time for execution is proportional to the logarithm of the size of the problem—in this case, the number of links in the chain. Using algorithms similar to the pointer-doubling algorithm, one can compute any sequential chain of N associative operations in order $O(\log N)$ time. This is the basis of many fast parallel algorithms. Similar algorithms can be used, for example, to compute the subtotals of a long list of numbers or the product of a long chain of matrices.

The reason that parallel computers work well on these problems is because in each example there is a large amount of data on which it can operate simultaneously. The number of operations that can be performed simultaneously is proportional to the amount of data, hence the name data parallel. The larger the problem, the greater the amount of data, and the greater the opportunity for parallel processing. This leads to the most general signature of the type of problems that are suitable for parallel processing: Problems that have large amounts of data tend to work well in parallel. Because the opportunity for parallelism increases with the data size, larger version problems can often be run on machines with more processors in the same amount of time. Often the throughput, in terms of number of arithmetic operations per second, increases linearly with problem size and the number of processors (Fig. 3).

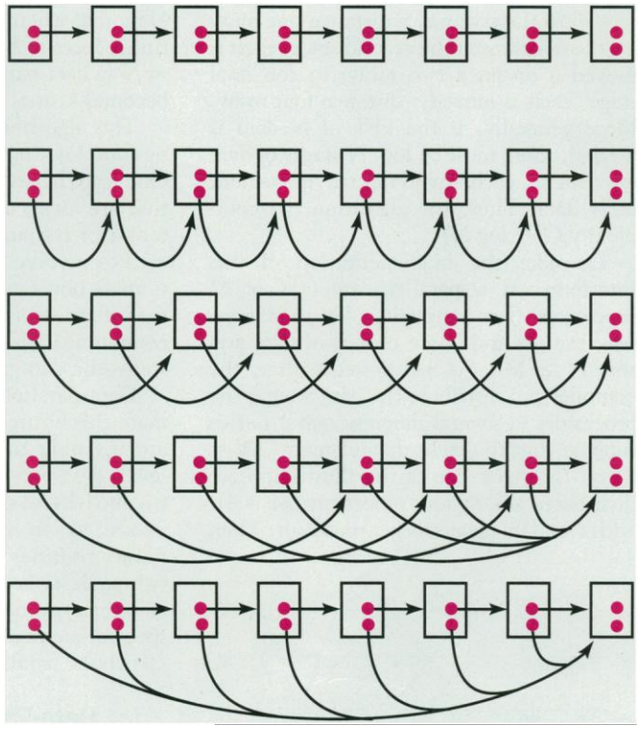

Fig. 2. Finding the end of a serially linked list is an example of an apparently sequential problem that can be computed in parallel. Each link in the chain contains a pointer to the next element. The problem is to find the end. In the parallel algorithm, each step halves the length of the list by skipping every other link.

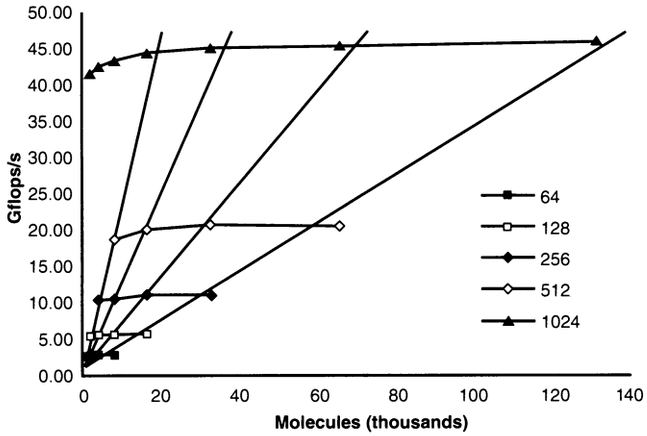

Fig. 3. The average performance of various size Connection Machines on a molecular dynamics calculation for various problems shows scaling properties characteristic of massively parallel machines. The near-linear performance curves show that larger machines solve proportionately larger problems in essentially the same time. The top curve shows the typical "roll off" in performance for smaller problem sizes. For sufficiently small problems (much smaller than those shown in the figure), a parallel machine may actually be slower than a conventional vector computer. [Data are from (*12*)]

The simple rule of thumb, that the opportunity for parallelism grows with the amount of data, also suggests the types of problems that may not be well suited for parallel computers: those in which the amount of data being operated on is very small. For example, solving an ordinary differential equation with only a few variables may be much less efficient on a parallel machine than solving a partial differential equation with 100,000 variables. Consider the classical physics problem of predicting the motion of the nine planets around the sun. It is not currently understood how to use more than about 100 processors concurrently on this problem, one for each pair-wise gravitational interaction between bodies (*10*). Does this mean that this problem is inherently sequential beyond this point? We speculate that it is not, but suitable large-scale parallel algorithms have yet to be developed.

Amdahl's Law

There is reason to be cautious in declaring any problem to be fundamentally unsuitable for parallel computers, because such claims have often been proven wrong. As recently as a decade ago, it was generally assumed by many computer scientists that massively parallel machines would be inherently inefficient on most problems. The argument was generally cast in the form called Amdahl's Law (*11*), which stated that the time required to do a computation will always be determined by the slowest part of the computation.

The argument against parallel machines went as follows: Assume that the time involved in performing a computation can be broken into two categories, s and p, where s is the time required for the operations that need to be done sequentially, and p is the time required for those that can be done in parallel. Then, assuming unlimited potential parallelism in the parallel portion of the problem, the running time for the algorithm on a machine with n processors will be at best $s + p/n$. As the number of processors increases, the time will be increasingly dominated by s. For example, if 10% of the operations are currently sequential, then even with an infinite number of processors the time required for the overall operation of the problem cannot be reduced by more than a factor of 10. By this argument, the efficiency of a parallel machine would seem to go down with the number of processors.

This argument seems compelling, yet parallel processors with tens of thousands of processors operate efficiently on a wide range of problems. Where is the flaw in reasoning? The difficulty is not with the equation, but with the assumption about the type of problem being solved. A faster computer is typically not used just to solve the same problem faster. It is more often used to solve a larger problem in roughly the same amount of time. Because the opportunity for parallelism tends to grow with the size of the problem, p tends to grow more rapidly than s with the size of the problem.

A typical situation might have p of order N, and s of order unity or order log N, where N is the amount of data in the problem. With this scaling, we see that the amount of data processed per unit time, $N/(s + p/n)$, can become arbitrarily large as n goes to infinity, as long as N increases with n. That is, as long as one looks to problems with more and more data, one will always benefit from adding processors.

A rule of thumb when writing parallel programs is that if any part of the program has a time scaling that is linear or greater in N, then there is probably an algorithm with a better scaling that uses more processors. A problem may grow either because a more complex system is being analyzed or because the same system is being analyzed to a higher degree of accuracy. Either way, the opportunity for parallelism generally increases. In problems of this sort it generally makes sense to scale the amount of processors with the data. The power of this approach is evident from a few examples:

- Adding N pairs of numbers together takes $O(N)$ time on a sequential computer, because each pair must be added together in sequence. On a parallel computer of N processors, however, it would take $O(1)$ time if the pairs were distributed one per processor.
- Adding N numbers together to get one total also requires $O(N)$ time on a sequential computer. On a parallel computer of N processors, however, the total can be computed in $O(\log N)$ time, since the data can be combined in the fashion of a binary tree, with the final result appearing at the root, in a manner similar to that of the pointer-doubling algorithm mentioned above. Each layer of the tree can be computed in parallel.
- Sometimes the time scaling can be improved by assigning more processors to the problem. For example, the multiplication of two N by N matrices takes $O(N^3)$ time on a sequential computer, $O(N)$ time on a parallel computer of N^2 processors, and $O(\log N)$ time on a parallel computer of N^3 processors.

In all of these cases, as N goes to infinity, the importance of the serial part of the computation goes to zero, so Amdahl's Law is irrelevant.

As noted earlier, problems with small amounts of data may not work well on parallel machines. In the case where the data size is fixed, the inefficiencies indicated by Amdahl's Law may turn out to be significant. For this reason, massively parallel machines are most often used to calculate larger, more detailed versions of problems than sequential machines, rather than to speed up the calculation on the same size problems. For problems that are small enough to run in a short period of time on a sequential machine, massively parallel machines often offer only modest improvements.

For problems with large data sets, massively parallel machines often perform at significantly greater speeds than those that are obtainable on sequential machines.

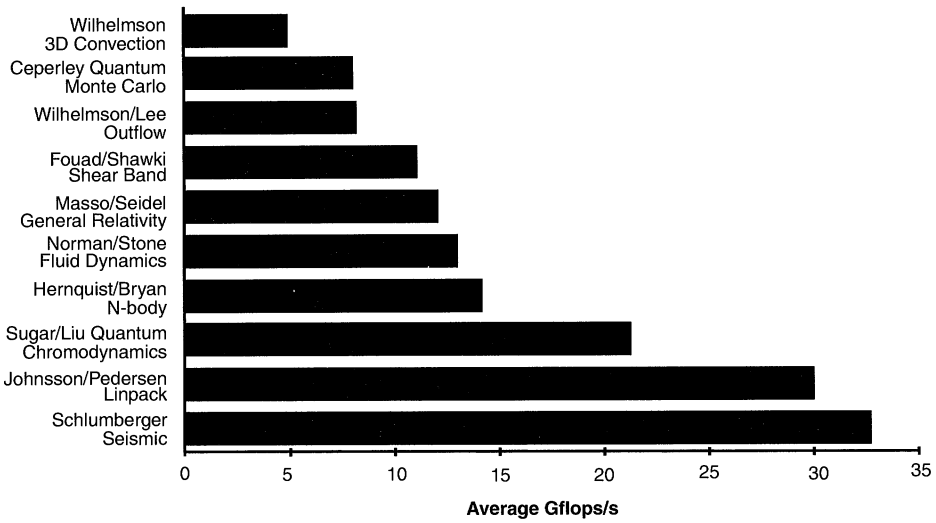

Fig. 4. This chart shows the performance, in billions of floating-point operations per second, of a variety of large-scale applications running on the 512 processor Connection Machine CM-5 at the National Science Foundation NCSA Supercomputer Center. [Courtesy NCSA]

Figure 4 shows the performance on a variety of applications that run on the 512 processor Connection Machine at the National Science Foundation's National Center for Supercomputing Applications in Illinois. On some unusual codes, massively parallel processors have generated even higher rates of sustained performance. For example, the 1024 processor Connection Machine at Los Alamos National Laboratories has recently achieved sustained performance rates of 45 Gflop/s (billions of floating point operations per second) on molecular dynamics calculations (*12*), 60 Gflop/s on a fluid flow calculation (*13*), and 60 Gflop/s on a weather calculation (*14*). For comparison, the fastest rate achievable on a single Cray vector processor is about 1 Gflop/s, and the fastest rate achievable on a shared memory vector processor is about 16 Gflop/s.

Even these high rates of sustained performance are only about half of the potential peak performance of the machines. It is very unusual for any type of computer, parallel or otherwise, to sustain its peak arithmetic rate during normal operation. The average performance of a user's code run on the University of Illinois' National Center for Supercomputing Applications' four-processor Cray Y-MP is about 0.070 Gflop/s, or 5% of its peak (*15*). It is used almost entirely in single-processor mode. Because the arithmetic units represent only a small part of the hardware resource, a well-balanced machine is not necessarily optimized to run the arithmetic units at peak speed, any more than a well-designed car is designed to run the transmission at peak torque. A computer's peak performance therefore is irrelevant to most users. What matters to users is the time required for the computation or the cost of the computation. In both of these categories, massively parallel machines routinely excel over vector machines by considerable margins. Figure 4 shows the performance of a 512 processor machine on a variety of scientific applications. Figure 5 shows the performance of various parallel machines on a standard performance benchmark (LINPACK) (*16*).

Data Communication

Another important factor determining the speed of a computation on a parallel machine is the pattern of communication. In most parallel machines each processor has its own associated memory which holds a certain subset of the data. Access to another processor's memory requires significantly more time than for accessing the processor's own local memory. This places a performance premium on having the right data, or as much of it as possible, available locally. For example, when the parallel processor performs the addition of two arrays, it is advantageous to have the corresponding elements of the two arrays stored within the same processor, so that the computation can be performed using entirely local memory reference operations. In this case, most parallel compilers will do this automatically.

The situation is more difficult when more complex operations are being performed. For example, consider the problem of multiplying two 1000 × 1000 arrays on a machine with 1000 processors. Is it better to put a single row of the array into the local memory of each processor or a single column? Or would it be better to put a subarray, say, 25 × 40, in each processor? For most multiplication algorithms the subarray allocation turns out to be preferable because it minimizes the nonlocal communication (*17*).

Today's compilers are able to do a good job of automatically assigning data to processors in many cases, but it is still often necessary for a programmer to explicitly specify the allocation to achieve good performance. Parallel languages, like High Performance Fortran, which provide simple notation for array operations, also allow optional directives specifying how these arrays are allocated. With improvements in compiler technology, the automatic allocation performed by compilers is likely to improve, but for now, understanding how data maps onto processors is an important part of writing an efficient parallel program.

In some problems there is no efficient static allocation of data to processors. In such cases it is possible for parallel processors to perform the data-to-processor mapping dynamically during the course of the computation. A good example is an adaptive grid used, for example, in fluid flow calculations based on Lagrangian grid methods. In this calculation the fluid is represented by a discrete lattice structure which changes with time. As fine structure develops in the fluid, a localized whirlpool for example, new grid elements are allocated dynamically where they are needed to model the details of the structure. New data structures allocated in memory are automatically placed in the processors containing the lattice elements with which they will communicate. If the flow of the fluid is nonuniform, the grid is likely to grow disproportionately in certain processors. Processors handling a difficult portion of the fluid will have a disproportionate amount of the data and the computational load. In order to address this problem, a parallel program for an adaptive grid algorithm typically alternates between a phase of computation and a phase of load balancing. During the load balancing phase the data is redistributed evenly across the processors. Although such algorithms are more difficult to write than a fixed allocation program, they are often able to achieve significant performance benefits by exploiting the nonuniform nature of the problem. In the last example below, we shall describe the implementation of a load-balancing algorithm for a quantum chemistry computation.

Because communication from nonlocal memory is the highest cost operation, it often makes sense to choose a parallel algorithm that minimizes communication, even if that algorithm involves doing slightly more arithmetic. For example, the standard method for solving dense systems of linear equations on sequential machines is Gaussian elimination, or in matrix terms LU decomposition. To maintain numerical stability with this method the order of variable elimination (pivoting) is normally determined dynamically. It is possible to use this algorithm on a massively parallel machine, and in fact it is often used, especially for extremely large dense matrices. Sometimes, however, it may actually be faster to use an alternate algorithm which requires fewer communication oper-

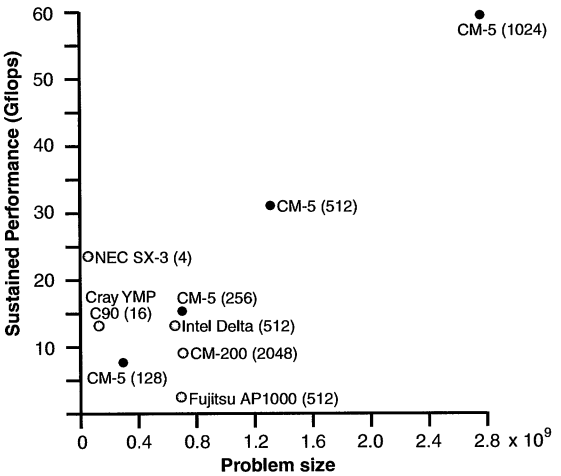

Fig. 5. The measured performance, in floating-point operations per second, of various parallel computers solving a large system of linear equations (LINPACK). The performance of different sizes of CM-5 Connection Machines illustrates a near-linear relationship between performance and problem/machine size. The horizontal axis shows the problem size in terms of the number of coefficients involved. The numbers of processors for each machine are shown in parentheses.

ations at the expense of twice as many arithmetic operations. This method, called QR factorization, is based on isolating variables by orthogonalizing the matrix. Although it involves twice as many arithmetic operations as Gaussian elimination, the communication patterns are very simple, so it may actually be faster on a parallel machine. This method also has advantages in numerical stability (*18*).

Similarly, for sparse systems of equations, often encountered in finite difference or finite element calculations, it has been noted that both direct solvers and iterative solvers are frequently used on sequential computers. The choice between these alternatives depends on many factors, but parallelization tends to favor iterative solvers since they can be implemented with a minimum of data communication (*7*).

Because the ratio of local to nonlocal references depends on the ratio of the number of processors to the total data size, applications involving large amounts of data are likely to do a larger number of local references, resulting in higher average computation rate. This effect reinforces the rule of thumb mentioned earlier that massively parallel machines are more suitable for problems with large amounts of data.

Computational Geometry and the Ising Model

This trade-off between local and nonlocal approaches is nicely illustrated by the example of the Ising model of statistical physics (*19*). This model is frequently used as a prototype for understanding the critical behavior of systems of spins, for example, magnetic domains of solids. In its simplest embodiment, it is a grid-based algorithm with one bit of data at each site that tells whether the spin at that site is up or down. Two neighboring spins on the lattice contribute energy -1 if their spins have opposite value, and $+1$ if their spins have the same value. This energy is summed over the set of neighbors of a site, and then over the entire lattice to get the lattice energy.

There are then several strategies for updating the values of the spins to march the system toward equilibrium. The simplest is the heat-bath algorithm—based on the Metropolis Monte Carlo algorithm (*20*)—in which the spins flip randomly, the difference in total lattice energy is computed, and the changes are accepted with some probability that depends on this difference. This algorithm is easily parallelizable. If the sites are distributed across processors, the energy of each site could be computed locally, and the sum of that energy over the lattice is then obtained by a log-time global reduction operation. Variants of this algorithm abound. For example, it is much more efficient to perform this algorithm on alternate checkerboard-colored sites of the lattice, and then accept or reject each change locally. Once again, this is straightforward to compute in parallel (*21*).

For large systems, however, these brute-force algorithms, based entirely on local interaction, are not the most efficient way to equilibrate a system of Ising spins. Near criticality, the spins tend to cluster on length scales that range up to the size of the entire lattice. Thus, in a local algorithm, the number of iterations required must scale as a power law in the lattice size so that information has a chance to transit the lattice many times. For this reason, Swendsen and Wang (*22*) introduced an improved algorithm that involves identifying connected clusters of like spins and then flipping the entire cluster at once. Thus, changes made to the system may extend over a large distance, and the propagation of information is not restricted to one link per iteration of the algorithm.

The problem then reduces to the efficient identification of the connected clusters of like spins on a lattice (see Fig. 6). This problem is not as obviously parallelizable as the original heat-bath algorithm or its variants. Nevertheless, it has been shown (*23*) that it does indeed parallelize.

To understand the data-parallel algorithm used for this problem, consider first the following simple algorithm for identifying the clusters: Label each site with a unique integer. Then compare a site's integer to that of all of its connected neighbors, and have it overwrite its value with the minimum of its present value and that of all of its neighbors. By iterating this last step until a steady state is reached, it is clear that we will arrive at a situation in which every site of a cluster will have the same integer, and sites in different clusters will have different integers.

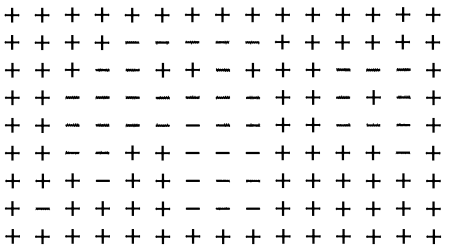

Fig. 6. This algorithm is an important improvement to the standard "heat-bath" Monte Carlo algorithm for thermalizing the Ising model. It calls for flipping entire connected blocks of like spins at each step. One part of the algorithm is thus the identification of these blocks. This is accomplished by a multigrid access pattern in which information is combined on different scales. Here, "up" spins are denoted by + signs, and "down" spins are denoted by − signs. The outlines of the blocks are shown in gray.

This method of cluster identification is clearly parallel, but it can be so slow as to mitigate the effectiveness of the Swendsen-Wang approach, since it involves only nearest-neighbor communication on the Cartesian grid (as did the heat-bath algorithm). Because of this slow propagation of information, identifying clusters on an $N \times N$ lattice may require $O(N^2)$ time for sufficiently serpentine clusters. (We shall present this technique for the two-dimensional case, but it is obviously generalizable to higher dimension.)

We would like a way to move the cluster information around faster. To do this, let us first recast the above simple algorithm in the language of matrices: Let $\Delta^{(0)}$ denote the connectivity matrix describing the clusters on the two-dimensional Cartesian grid. Thus, if we again label the sites by integers, the ith row of $\Delta^{(0)}$ contains a 1 in all columns corresponding to sites to which site i is connected (including i itself), and 0 elsewhere. Thus, there will be at most five 1's in each row.

The operation of comparing with neighbors in your cluster and iterating can then be understood as repeated multiplication by $\Delta^{(0)}$. For example, the ith row of $(\Delta^{(0)})^2$ will contain a 1 in every column corresponding to a site that is zero, one, or two

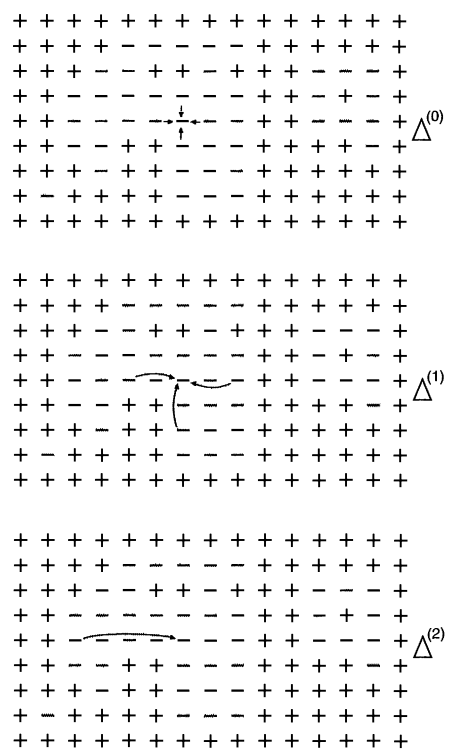

Fig. 7. Nonzero elements of the cluster connectivity matrices correspond to sites in the same cluster that are a power-of-two distance away. The figure shows the first three steps of data access for a single site.

links away, etc. For sufficiently large J, we will have $(\Delta^{(0)})^J = (\Delta^{(0)})^{J+1} = (\Delta^{(0)})^\infty$, and at that point, row i will contain a 1 in all columns corresponding to sites to which site i is connected, and 0 elsewhere. Unfortunately, as noted above, J might have to be $O(N^2)$ before convergence is obtained.

To significantly speed up this algorithm, a multigrid approach can be used. Referring to Fig. 7, let $\Delta^{(1)}$ be the connectivity matrix for sites at distance 2. More generally, let $\Delta^{(l)}$ be the connectivity matrix for sites at distance 2^l along each axis. These connectivity matrices can be obtained recursively. For example, if a site is connected to another site, a distance 2^l to the north, and that site is in turn connected to another site, a distance 2^l further to the north, then the first site knows that it is connected to the site a distance 2^{l+1} further to the north, etc.

Given these connectivity matrices at different length scales, we can arrive at $(\Delta^{(0)})^\infty$ much more quickly by successively multiplying by connectivity matrices for many length scales right up to the full size of the lattice

$$\Delta^{(0)}\Delta^{(1)}\Delta^{(2)} \ldots$$
$$\Delta^{(n)}\Delta^{(0)}\Delta^{(1)}\Delta^{(2)} \ldots$$
$$\Delta^{(n)}\Delta^{(0)}\Delta^{(1)}\Delta^{(2)} \ldots \Delta^{(n)} \ldots \quad (10)$$

where n is on the order of $\log N$ for an $N \times N$ lattice.

On a lattice of 1024^2 sites, the Swendsen-Wang algorithm thermalizes the Ising model in 667,000 times fewer iterations than the heat-bath algorithm. Of course, because of the necessity of finding the clusters in the above-described fashion, each iteration of the Swendsen-Wang algorithm takes much longer than those of the heat-bath algorithm (*19*). Nevertheless, Brower *et al.* (*23*) were able to implement the above multigrid algorithm so that each iteration took about 25 times longer than a heat-bath iteration, so that the Swendsen-Wang algorithm yielded a factor of 27,000 improvement over the heat-bath algorithm on a CM-2 Connection Machine parallel computer with 65,536 processors.

Multigrid methods are also often used for elliptic equation solvers. Because they have favorable convergence properties, they are often faster than local methods on parallel machines (*24*).

N-Body Problems

Many algorithms of computational science call for operations to be performed on each element of an array that involve all of the other elements thereof. For example, simulations of systems of gravitationally interacting masses (*25*), electrostatically interacting charges (*26*), or vortices in an incompressible fluid (*27*) involve what is known as the N-body problem, in which every particle must interact with every other one in the system.

Note that this problem generally requires $O(N^2)$ time on a sequential computer [For certain force laws—most notably the Coulomb force law—there are $O(N)$ complexity methods that employ multiple expansions for the treatment of distant particles, and these are also parallelizable, but they are outside the scope of the present discussion.] Figure 8 illustrates a data-parallel algorithm for a computer with $O(N)$ processors. Each processor is responsible for treating one or more bodies. The information corresponding to the bodies is copied, and then cyclically rotated through the array of processors so that each body has a chance to interact with every other body in $O(N)$ steps. Thus, scaling the number of processors with the number of bodies has served to reduce the time needed from $O(N^2)$ to $O(N)$.

Better time scaling is possible if one increases the number of processors as the square of the number of bodies. In this case, one might imagine the processors as representing the interactions between the bodies, rather than the bodies themselves. Once the pairwise interactions have been computed, the N interactions involving a given body can be summed in $\log_2 N$ steps. All these sums can be computed in parallel. The entire algorithm is thus completed in $O(\log N)$ time (see Fig. 8).

Thus, note that the asymptotic time scaling for this problem depends on how one is willing to scale the number of processors with the data. If one has only one processor (a sequential computer), then it takes $O(N^2)$ time. If one lets the number of processors scale as the number of bodies, then it takes $O(\log N)$ time. If one lets the number of processors scale as the square of the number of bodies, then it takes $O(\log N)$ time.

This improvement in time scaling with processor number scaling is generic to a wide class of problems in computational science. It is significant that data-parallel libraries can automate this entire process by examining the number of bodies and the number of processors available at run time to decide which algorithm is optimal. Indeed, all-to-all communications routines for exactly this type of problem are available in at least one existing data-parallel software library (*6*).

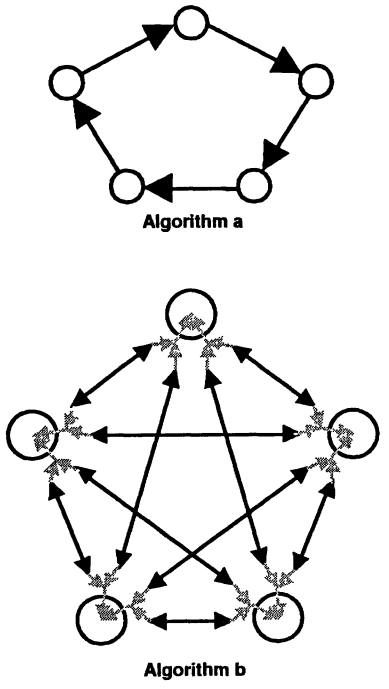

Fig. 8. Two different parallel algorithms for the N-body problem have different scaling properties. Both algorithms compute the forces on each of N bodies in parallel. The arrowheads in the figure denote processors. Algorithm a cyclically shifts the data through N processors, calculating each pairwise interaction in sequence, requiring N steps. Algorithm b uses $O(N^2)$ processors to compute all N^2 interactions at once, and then it adds all the forces together in $\log_2 N$ steps using the tree structures shown in gray.

Load Balancing Algorithms and Monte Carlo Methods for Quantum Chemistry

To understand how load-balancing operations can be expressed in the data-parallel style, we consider the example of Monte Carlo methods for quantum chemistry calculations. The accurate computation of ground-state properties of atoms, molecules, or systems thereof is a ubiquitous problem in computational chemistry. The Schrodinger equation, which governs this, can be expressed as a diffusion equation in imaginary time (*28*) or as an integral equation involving a Green's function (*29*). In either case, it may then be solved iteratively for ground state properties.

Because the configuration space may be of very high dimension, it is impossible to use grid-based algorithms. Instead, the wavefunction is represented as a weighted average over a set of point particles in the configuration space. The above-mentioned diffusive process is then affected by having the particles undergo a random walk. As they move about and sample the configuration space, weighted averages over them converge to ground-state expectation values, such as the ground-state energy.

In spite of this rather drastic change of representation, the method is still amenable to data-parallel treatment. Because of the linearity of the Schrodinger equation, the random walkers move about indepen-

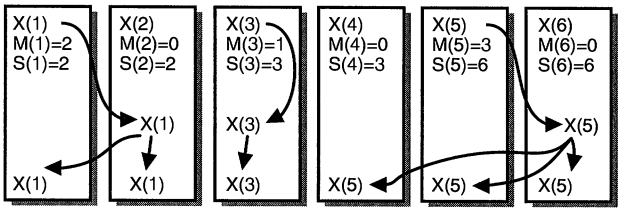

Fig. 9. To improve the variance of quantum Monte Carlo simulations, a load-balancing algorithm is frequently used that calls for assigning a weight to all walkers. As the walkers move about and sample configuration space, their weight is modified. Occasionally, low-weight walkers are killed and high-weight walkers are split. If we imagine parallelizing over the walkers present, the problem reduces to that of the dynamic allocation and deallocation of processors to tasks. The illustration shows that this problem can be handled simply using data-parallel primitives that involve the cumulative summation of the number of children of each walker, the sending of walkers to their new positions in the array, and the copying of walkers across segments of the array as many times as necessary (see text).

dently, and so their random walk can be parallelized with no interprocessor communication. Arrays containing walker attributes are simply spread over the array of processors, so that each processor represents one (or a group of) random walkers.

Like the Ising model, this naïve implementation can be improved in numerous ways. For example, tracking large numbers of low-weight random walkers is a burden to the computation because they contribute little to any expectation value. In order to reduce the statistical variance, it is preferable to load balance the computation by selectively eliminating low-weight walkers, and cloning high-weight walkers. Because we are associating processors with walkers, this killing-and-splitting process requires the dynamic allocation and deallocation of processor resources.

In the data-parallel paradigm, this killing-and-splitting algorithm for load balancing can be implemented as follows: Based on walker weight, each processor can decide how many children will be spawned by its walker(s). This number of children walkers may be zero (for walkers that will be killed), one (for walkers that will just survive), or greater than one (for walkers that will be cloned). Call this array of nonnegative integers M. By taking the cumulative sum of M, one can determine the leading array position of the walker in the desired load-balanced state. By sending walkers with nonzero M to these positions, and then copying them as necessary, the desired killing and splitting is accomplished.

To illustrate this, Fig. 9 depicts six walkers with attributes (position, weight, and so on) stored in the array **X**. Walker one will give rise to two children, walker three to one child, and walker five to three children. The other walkers will give rise to zero children; that is, they will be killed. The array **M** is shown. The cumulative sum of **M**, denoted by **S**, is then taken by the sum-scan operation—a log-time data-parallel primitive. Processors with nonzero **M** then send their walker's attributes to the array element given by **S**. Another log-time data-parallel primitive, known as the (segmented) copy-scan, then fills in the new array **X** by copying entries as necessary.

With this method, highly accurate quantum Monte Carlo computations have been carried out on numerous atomic and molecular systems. For example, the most accurate computation to date of the ground-state energy of the hydrogen molecule was carried out in this fashion (*30*).

Further refinements to the algorithm make it possible to treat larger systems of electrons, but the constraints imposed by the Pauli principle—the requirement that the electronic wavefunction be antisymmetric—turn out to necessitate further interprocessor communication. Recently, it has been shown (*31*) that these constraints can be satisfied by performing an *N*-body computation—one in which every processor exchanges information with every other. In this way, recent quantum Monte Carlo studies of the helium dimer have been accurate enough to resolve its very weakly bound ground state (*32*), which was detected experimentally in March of this year (*33*).

Conclusions

The examples above are only a sample of the types of engineering and scientific applications for which parallel computers are well suited. They range in character from simple analogs of physical processes with spatial parallelism to sophisticated mathematical models that bear only an abstract relation to physics. Methods of programming parallel computers continue to evolve, and as yet the limitations of this technology are not well understood—but it is already clear that massively parallel computers are and will continue to be an important tool for the largest and most complex scientific computations.

REFERENCES

1. G. C. Fox, *What Have We Learnt From Using Real Parallel Machines To Solve Real Problems?* (Caltech Rep. C3P-506, California Institute of Technology, Pasadena, CA, 1988).
2. *High Performance Fortran Language Specification* (Tech. Rep. CRPC-TR 92225, Center for Research on Parallel Computation, Rice University, Houston, TX, 1993).
3. E. P. Wigner, *Commun. Pure Appl. Math.* **13**, 222 (1960).
4. J. W. Cooley and J. W. Tukey, *Math. Comput.* **19**, 297 (1965).
5. S. L. Johnsson, R. L. Krawitz, R. Frye, D. MacDonald, *Cooley-Tukey FFT on the Connection Machine* (Tech. Rep. Ser. NA89-4, Thinking Machines Corporation, Cambridge, MA, 1989).
6. *CMSSL for CM Fortran, CM-5 Edition* (Thinking Machines Corp., Cambridge, MA, 1993), vols. I and II, version 3.1.
7. G. Golub and C. van Loan, *Matrix Computations* (The Johns Hopkins Univ. Press, Baltimore, MD, 1990).
8. Z. Johan, T. Hughes, K. Mathur, S. Johnsson, *Comput. Meth. Appl. Mech. Eng.* **99**, 113 (1992).
9. D. Hillis and G. Steele, *Commun. ACM* **29** (1986).
10. J. H. Applegate, *IEEE Trans. Comput.* **C-34**, 9 882 (1985).
11. J. L. Gustafson, *Commun. ACM* **31**, 5 (1988).
12. D. M. Beazley and P. S. Lomdahl, *Paral. Comput.*, in press.
13. L. Long, M. Kamon, T. Chyczewski, J. Myczkowski, *Comput. Syst. Eng.* **3**, 337 (1992).
14. G. Sabot, S. Wholey, J. Berlin, P. Oppenheimer, *Supercomputing '93 Proceedings* (in press).
15. L. Smarr, private communication.
16. J. Dongarra, *Performance of Various Computers Using Standard Linear Equations Software* (Computer Science Department, University of Tennessee, Knoxville, TN, 1993).
17. L. E. Cannon, thesis, Montana State University (1969).
18. H. Simon, Ed., *Proceedings of the Conference on Scientific Applications of the Connection Machine* (World Scientific, Singapore, 1989).
19. See, for example, L. E. Reichl, *A Modern Course in Statistical Physics* (Univ. of Texas Press, Austin, TX, 1980).
20. See, for example, M. H. Kalos and P. A. Whitlock, *Monte Carlo Methods, vol. 1, Basics* (Wiley, New York, 1986).
21. See, for example, T. Toffoli and N. Margolus, *Cellular Automata Machines* (MIT Press, Cambridge, MA, 1987), section 17.2.
22. R. Swendsen and J. S. Wang, *Phys. Rev. Lett.* **58**, 2 (1987).
23. R. C. Brower, P. Tamayo, B. York, *J. Stat. Phys.* **63**, 73 (1991).
24. O. McBryan and P. Frederickson, *Parallel Superconvergent Multigrid* (Theory Center Technical Report, Cornell University, Ithaca, NY, 1987).
25. J. Barnes and P. Hut, *Nature* **324**, 446 (1986).
26. C. K. Birdsall and A. B. Langdon, *Plasma Physics Via Computer Simulation* (McGraw-Hill, New York, 1985).
27. A. J. Chorin, *J. Comput. Phys.* **27**, 428 (1978).
28. N. Metropolis and S. Ulam, *J. Am. Stat. Assoc.* **44**, 335 (1949).
29. M. H. Kalos, *Phys. Rev.* **128**, 4, 1791 (1962).
30. C. A. Traynor, J. B. Anderson, B. M. Boghosian, *J. Chem. Phys.* **94** (no. 5) (1991).
31. J. B. Anderson, C. A. Traynor, B. M. Boghosian, *ibid.* **95**, 7418 (1991).
32. ———, *ibid.* **99**, 345 (1993).
33. F. Luo, G. C. McBane, G. Kim, C. F. Giese, W. R. Gentry, *ibid.* **98**, 3564 (1993).