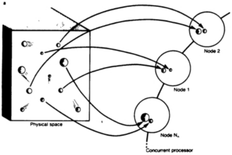
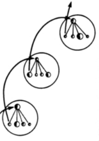
Hm 71

# Algorithms for concurrent processors

A few general techniques allow many small computers to work together efficiently and attack computationally demanding problems in fields ranging from aerodynamics to astrophysics.

Geoffrey C. Fox and Steve W. Otto



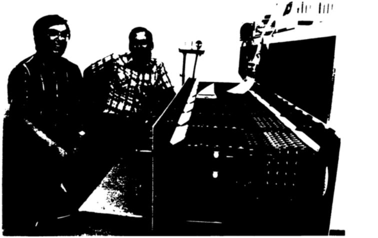


**Decomposition scheme** in algorithm that calculates the evolution of a system governed by long-range forces. On the left in a is the physical space containing the interacting masses. On the right is the concurrent processor, in a ring topology, with each circle representing a small computer, or "node." Note that masses stored in the same node do not necessarily lie close to one another in the physical space. In b we see a "snapshot" of a typical step in the long-range-force algorithm. One particle from each node is shown traveling between each pair of nodes. Once this "traveler" arrives in a node, the forces between it and all the particles residing in that node are computed. This is represented by the sets of lines drawn within the nodes. Figure 1

We are on the verge of a revolution in computing, spawned by advances in computer technology. Progress in very-large-scale integration is leading not so much to faster computers, but to much less expensive and much smaller computers—computers contained on a few chips.[1] These machines, whose cost-effectiveness is expected[2] to be staggering, will make it practical to build very-high-performance computers, or "supercomputers," consisting of very many small computers combined to form a single concurrent processor.

Concurrent processing seems a more practical route to high performance than very fast sequential processing. In fact, we anticipate machines consisting of from 10 000 to 100 000 "nodes," with each node being a small but complete individual computer of modest power capable of some ten million floating-point operations per second, or "megaflops." Such a design, which should be practical within five to ten years, offers the promise of machines that are from one thousand to ten thousand times as powerful as current supercomputers. As such "top-of-the-line" million-megaflop machines become available, so will smaller parallel processors, consisting of, say, 100 individual nodes and having a total power of some thousand megaflops and a cost of perhaps twenty thousand dollars for the basic CPU and memory (we are ignoring such essential peripherals as disks).

Geoffrey Fox and Steve Otto are at the California Institute of Technology, where Fox is professor of theoretical physics and dean for educational computing, and Otto is a postdoctoral research fellow in theoretical physics.



**"Cosmic cube."** This concurrent processor consists of 64 identical small computers connected so that each can send messages to 6 others. The 64 corners of a 6-dimensional cube, or hypercube, have this connection topology. Each small computer contains 16-bit Intel 8086 and 8087 processors with 136 kilobytes of memory—roughly the power of an IBM personal computer. The $80 000 cube has up to one-tenth the power of a Cray-1 computer, but one-hundredth the cost. Shown here with the experimental machine at Caltech are Geoffrey Fox (left) and Charles Seitz. Figure 2

This increased power will revolutionize the approach to computation in all fields of science and engineering. For instance, one will be able to solve such difficult problems as weather prediction and the dynamics of quantum field theories. It is worth noting that these and other computationally intense problems do need the huge increase of power we expect from the largest concurrent processors. Other important problems, such as those based on the three-dimensional differential equations that arise in aerodynamics and other fields, may need computers "only" in the thousand-megaflop range.

The main stumbling block to the use of concurrent processors is the difficulty of formulating algorithms and programs for them.[3] Indeed, this leads some to doubt the utility of these machines. It is our belief that concurrent processors are quite easy to use and are not specialized devices, but rather can address the vast majority of computationally intensive problems. (See the example in figure 1.) Thus, our goal in this article is to present the general techniques for using concurrent processors and to illustrate them with simple examples, some of which we have run on the machine shown in figure 2. We will confine ourselves mainly to fields of science and engineering, as opposed to, say, artificial intelligence, because these fields offer well-understood algorithms that make it possible to quantify the effectiveness of concurrent processors. However, we believe that our considerations are general and have applications in other fields.

**General features of problems.** Before moving on to specific examples, it is worth noting some general properties of computationally demanding problems. Table 1 lists several examples of problems and some of their features that we have found important to their solution on a concurrent processor. In each case, one must decompose the problem into many parts—one for each node. Typically, a problem is not demanding because its algorithm is complex in a conceptual sense. Rather, there is a relatively simple procedure (for example, partial derivatives in computing $\nabla^2$) that one must apply to a basic "unit" (a field, for example) in a "world" that consists of a huge number of such units. In finite-difference problems, the unit is a grid point in a three-dimensional world. In a study of the evolution of the universe, the unit is a galaxy and the world is the universe itself.

The first step in decomposing such a problem for a concurrent processor is to divide the world into subdomains in such a way that each node of the processor is responsible for a single region. The box on page 59 illustrates this for a simple two-dimensional problem requiring the solution of the Laplace equation $\nabla^2\varphi = 0$. If we have $N_n$ nodes and a total of $D$ units—the grid points in the figure—we find that the number $n$ of contiguous units in each node is $D/N_n$. This type of decomposition is possible only if the number of units is at least as large as the number of nodes; we will see later that in fact it is desirable for the number of units to greatly exceed the number of nodes. This constraint is easy to satisfy. Today, calculations with over a million units are commonplace, and in all fields the number of degrees of freedom in state-of-the-art calculations is steadily increasing.

There are exceptions, of course, where computationally intensive problems cannot be so decomposed. The solar system is an example. In this $N$-body gravitational problem with $N = 10$, we wish to integrate the 10 equations of motion over a very long

**Table 1. Problems and concurrent processing**

| Class of problems | Examples | Unit and world | Natural Load Balance? | Communication range | Communication topology |
|---|---|---|---|---|---|
| Finite difference equations, finite element equations, partial differential equations | Geophysics, aerodynamics | Grid point, space $(x,y,z)$ | Yes | Short | 3D mesh |
| Statistical | Lattice gauge | Spacetime $(x,y,z,t)$ | Yes | Short | 4D mesh |
| | Melting | Configuration space $(x,y,z)$ | No | Short | 3D mesh |
| | Coulomb gas | Particle number | Yes | Long | Ring |
| Time evolution of $1/r$ potential | $N$-body gravity | Particle number | Yes | Long | Ring |
| Time evolution of general dynamics | Particulate motion (sand avalanches) | Space $(x,y,z)$ | No | Short | 3D mesh |
| Fast Fourier transform | Evolution of universe, fluid dynamics | "Bit space," space $(x,y,z)$ | Yes | Long | Hypercube |
| Network simulation | Circuit simulation | Component circuit | No | Mixed** | Logarithmic graph* such as hypercube |
| | Neural network | Neuron, brain | No | Mixed** | |
| Isolated | Ray tracing (graphics), data analysis, initial condition study | Event space | Yes | None needed | |
| Image processing | Analysis of satellite data | Pixel space | Yes | Long | Hypercube |
| Artificial intelligence | Chess | Inference, decision tree | Yes | Short | Tree |
| Event-driven simulation | Industrial, economic, military ("war games") | Cars on a freeway, tanks on a battlefield | No | Mixed** | Logarithmic graph* such as hypercube |

*Maximum communication time increases as the log of the number of nodes.
**A mixture of long- and short-range connections or interactions

time $t$. This large parameter $t$ *cannot* be easily decomposed, so we can use, at most, 10 nodes for the problem. On the other hand, one usually wants to examine the results of the integration for a variety of initial conditions. One can then decompose the problems on the product space—particles and initial conditions—and so make effective use of a large concurrent processor.

## Approach to problems

Concurrent processors have many possible designs, which differ primarily in the number and nature of the nodes and their interconnection topology. The hardware that we consider for this discussion is what our Caltech colleague Charles Seitz terms[4] the ensemble or homogeneous machine. Such a machine is a collection of identical nodes, each a complete computer with its own arithmetic unit and memory. We will assume that each node can execute its own instruction stream, although this is not necessary for every application. The resulting parallel processor will then have an architecture known as "MIMD"—multiple instruction, multiple data. The nodes may even have another level of concurrency within them, such as "pipelining." Rather than assuming that the nodes are connected in any particular arrangement, we will allow the connection topology to be general and examine each problem to find the "natural" connectivity. Of particular importance is the so-called hypercube, or more precisely, Boolean hypercube topology, in which one uses $2^\gamma$ computers with the connectivity of a cube in $\gamma$ dimensions. With the number of nodes $N_n$ equal to $2^3$, for example, the 8 small computers will be connected like the corners of an ordinary 3-dimensional cube—each to three others. We will *not* assume that there is any shared memory accessible by all nodes; the simpler distributed-memory architecture seems sufficient for our applications. (See the article by James C. Browne, page 28).

It is convenient to characterize the effectiveness of a concurrent processor by the *speed-up*, *S*, defined so that the collection of $N_n$ nodes runs the same problem $S$ times faster than a single node. Furthermore, we define the *efficiency* $\epsilon$ as the speed-up per node: $S = \epsilon N_n$. We wish to examine the effects that reduce the performance of a concurrent processor and lower the efficiency from the nominally perfect value of unity. One is usually quite satisfied to find algorithms with linear speed-up, that is, with an efficiency that is independent of the number of nodes, and of reasonable size, say at least about 0.5.

Two considerations are particularly important in discussing efficiency. First, the nodes must spend some time communicating with their neighbors. This time is minimized if the internode communication demanded by the algorithm always proceeds by a "hard-wired" path. We can view communication in ensemble machines as a mail system where messages are sent between arbitrary nodes through intermediate nodes. Obviously, the "wasted" communication time is minimized if the amount of such message forwarding is small. In general, the "world" that is decomposed in a particular problem has a certain topology that dictates the appropriate hardware connectivity. The hypercube node connection scheme is attractive because it includes the ring (figure 1a) and many mesh topologies as subsets, as well as being the topology needed for the fast Fourier transform. Furthermore, the distance between arbitrary nodes grows only logarithmically with the total number of nodes. This means that the time spent forwarding is modest for problems that have an irregular structure—circuit simulations and "war games," for example. Table 2 summarizes the reduction in efficiency due to communication for the problems discussed in this article.

"Load balancing" is the second factor affecting efficiency. One needs to ensure that all the nodes have essentially identical computational loads. The efficiency is typically reduced by a factor that is approximately the ratio of the mean computing load per node to the maximum load on a node. In the example discussed in the box on page 59, we see that identical loads are

**Potential energy** between two static (heavy) quarks, as a function of their separation. The 64-node concurrent processor shown in figure 2 did this lattice gauge theory calculation in 2500 hours. No units for distance and energy are given because the setting of an absolute scale is nontrivial and requires further calculation. Figure 3

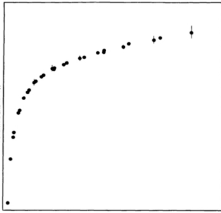


**Table 2. Algorithm efficiency and communication overhead**

| **Algorithm** | **Proportionality function** $f(n)$ |
|---|---|
| One-dimensional grid point problems | $n$ |
| Long range forces | |
| Full matrix inversion | $n^{1/2}$ |
| Full matrix eigenvalues/function | |
| Sparse matrices from two-dimensional grid-point or finite-element problems | $n^{1/2}$ |
| Two-dimensional statistical mechanics | |
| Sparse matrices from three-dimensional grid-point or finite-element problems | $n^{1/3}$ |
| Three dimensional statistical mechanics | |
| Fast Fourier transform | $\log n$ |

The communication overhead, which is the amount by which the efficiency differs from unity, is directly proportional to the ratio of $t_{comm}$ to $t_{calc}$, and inversely proportional to $f(n)$, where $n$ is the number of "units," of "world" stored in each node, $t_{comm}$ is the typical time to transmit a word along a hard-wired link between nodes and $t_{calc}$ is the typical time for a floating point calculation within a node. Note that in all cases the communication overhead tends to zero as the number $n$ of units per node tends to infinity.

achieved by assigning equal numbers of grid points to each node. For such a regular problem, this means simply giving the nodes equal areas of the "world." There is, in fact, a small deviation from perfect balancing in this example, as the number of points where the potential is fixed (and so have no work associated with them) does vary from node to node.

For homogeneous problems, it is generally easy to achieve balanced loads, but in some inhomogeneous cases, care is necessary. Consider a gravitational evolution, where we assign an equal number of stars or other celestial bodies to each node. In a region where, say, binary stars form, we may need to use a reduced time step. Then, nodes containing binary stars will have a larger load than those without. There are two strategies for combating this. If the nodes were "large" enough, each would hold roughly the same number of binaries and the load would balance—we can term this the "central limit theorem of decomposition." This example argues against decompositions that are too "fine-grained." We also see in table 2 that one needs to store a reasonable number of "units" in each node to minimize the communication overhead. A more complex solution is to rearrange the allocation of stars to processors dynamically, so that those with binaries have fewer other computational tasks. Load balancing is an important practical consideration but does not seem to be an insurmountable difficulty.

A column in table 1 shows whether or not certain algorithms achieve load balance naturally. Other columns show the communication range and the topology demanded by the algorithms.

## Short- and long-range force

It is, perhaps, most clear how to use concurrent processors for solving problems of a local nature. The canonical example is the finite-difference solution of partial differential equations, such as the one in the box on page 59. The fundamental locality of the equations implies that field variables evolve as a function of only the nearby (in physical space) variables. Therefore, the algorithm decouples in a simple way: Assign each node to a subvolume of the physical space and have it develop the variables in this subvolume alone. The algorithm for each node proceeds almost as it would on a sequential computer. The only difference is that variables in the subvolume develop under slightly more complex "boundary conditions"—when the algorithm encounters boundaries of the subvolume, there must be communication with other nodes. In this type of problem, we clearly want the interconnection topology of the nodes to match the physical space. We mean by this simply that points that are nearby in the physical space of the problem should be "nearby"—separated by few communication steps—in the concurrent processor. Two-dimensional and three-dimensional meshes and hypercubes are examples of these topologies.

Problems of a local nature are not the only ones with algorithms that make for efficient concurrent processing. To illustrate this point, we will discuss the extreme case of a completely non-local interaction: the $N$-body Newtonian gravity problem. In the direct method for calculating the evolution of this system, one simply computes for all possible pairs of particles the two-body force

$$\mathbf{F}_{i \to j} = G\frac{m_i m_j}{r_{ij}^2}$$

After computing these forces, one can use the usual time-stepping procedures to calculate how the particles move. The force part of the computation will dominate the calculation because it grows as the square of the number of masses, while the time stepping grows only linearly. Though this is a slow algorithm for the $N$-body problem, it is often used because it is the most accurate.

To perform this calculation on an $N_n$-node machine, we must first decompose the problem. The relevant "space" is not the space of particle coordinates—the long-range interaction tells us that it does not help to associate particular regions of coordinate space with the processing nodes. Instead, we decompose in particle number by making each node responsible for calculating the evolution of an equal number $n$ of the masses. Here $n$ is simply $N/N_n$, the ratio of the number of bodies to the number of nodes. We can assign particles to nodes randomly, because each node will track a particular group of particles throughout the entire evolution. At any given time step, a node's particles will be at widely scattered positions in coordinate space, as figure 1a illustrates.

Suppose now that the interconnection topology of the nodes contains a ring. Examples of such topologies are rings themselves, hypercubes and periodic meshes. In the first step of the algorithm, each node notes the mass and coordinates of one of its particles and passes the information forward in the ring by one node. Each node then computes the forces between the incoming particle and all the other particles in the node. Once this is completed, the "traveling" particles move along the ring by another node and the process repeats until the entire ring of nodes has been traversed. The entire traversal cycle is then repeated for another particle, and so on, until all particles have visited all nodes. The snapshot in figure 1b shows a typical step in the algorithm.

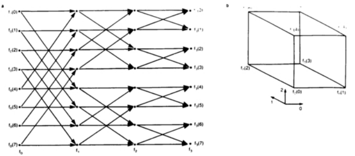
This algorithm for handling long-range forces is efficient as long as the time to communicate a particle between nodes is small in comparison to the time needed to compute the forces between the particle and all the particles in the node. This condition is easy to satisfy in practical examples—even with just a few particles per node, and hence a high degree of parallelism, the method is very efficient. In fact, the long-range-force algorithm is one of the most efficient algorithms we have found. This is illustrated in table 2, which shows that the communication overhead is proportional to $1/n$, whereas for other algorithms the overhead decreases more slowly as a function of $n$. The general point is that it is not the amount of communication that matters; rather, what matters is the amount of computation done per communication.

## Lattice gauge theory

Numerical simulations of statistical mechanical systems in thermal equilibrium and the similar methods used in lattice gauge theory are becoming increasingly important in physics. These simulations require computers with extremely high computational capacity because of the slow, statistical convergence of observables and because of the typically large number of degrees of freedom.

The interaction in this class of applications can be taken to be local, that is short ranged in the physical space of the problem. Non-local interactions do arise, for example, from a renormalization group analysis or, in the case of lattice gauge theory, from the presence of dynamical quarks. In these situations, to make the calculation tractable—even on a sequential computer—the interaction must be cut off (in the renormalization group case) or calculated by a more complex method, but one involving only local interactions (in the dynamical quark case). Once the interaction is local, the obvious decomposition onto the concurrent processor is the same as that for the finite-difference algorithm, in which each node is responsible for a subregion of the physical space and for developing the variables of that subregion only. When calculating the evolution of systems with many degrees of freedom, there is a slight complication in that one must ensure that the procedure used still satisfies the necessary constraint of detailed balance.

We have some direct experience with the statistical-mechanics algorithm. Currently, we are doing a Monte Carlo lattice gauge theory computation on the system of 64 microprocessors built at Caltech[5] and shown in figure 2. The $2^6$ nodes of this MIMD machine are wired as a 6-dimensional hypercube. An interdisciplinary team of scientists and engineers is doing forefront research problems on this machine and is designing and constructing more powerful machines. Similar projects underway at other universities include a successful Ising model processor[6] at the University of California, Santa Barbara, and a potentially high-performance machine being built by a group at Columbia University.[7] On the Caltech system, we have simulated SU(3) lattice gauge theory for a $12\times12\times12\times16$ lattice, measuring the potential energy between a pair of static (heavy) quarks. This quantity is of fundamental interest both because it can be compared with experiment and because it clearly demonstrates the nonlinear and quantum mechanical nature of quantum chromodynamics.[8] Due in part to the significant power of the 64-node machine—up to one-tenth that of a CRAY-1—we have achieved results[9] with statistics good enough to make detailed checks of the efficacy of the lattice gauge approach. (See figure 3.) As for the efficiency of the machine, for this problem it varies from 95% during measurement of the potential to 97% during calculation of the gauge field's evolution. Equivalently, the speed-up



for this 64-node machine varies from 61 to 62. We consider this realistic, state-of-the-art algorithm to be an "existence proof" that computations in statistical mechanics run with very high efficiency on a concurrent processor with mesh or hypercube connectivity.

There is another class of problems in statistical mechanics in which the degrees of freedom are not tied to an underlying lattice. Examples of such problems are the melting transition and the thermodynamics of liquids and gases. In these cases, one still decomposes in physical space, but the particles can travel from one processor to another. This makes it somewhat harder to maintain load balance and to implement the constraint of detailed balance. Members of our research group are currently working on problems of this nature.

## Matrix problems

Many scientific and engineering problems boil down to the solution of large matrix equations. These can be classified according to the structure of the matrix and the nature of the operations to be performed. We will consider here the inversion of both sparse matrices, in which most elements are zero, and "full" matrices, in which few elements are zero.

Sparse matrices arise in the solution of boundary-value problems for partial differential equations, which, of course, are common to many disciplines, such as geophysics and aerodynamics. In such a problem, the rows and columns of the matrix are labeled by the coordinates of an associated grid point (or node of a finite-element formulation). For a three-dimensional grid with $100 \times 100 \times 100$ points, the matrix $M$ is very large, namely $10^6 \times 10^6$. One usually wants to solve the boundary-value problem for one or a few different choices of the boundary conditions. Each solution involves solving the following system for $\mathbf{x}$

$$M\mathbf{x} = \mathbf{b}$$

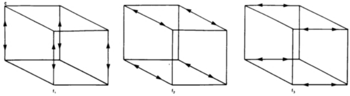
In this equation, $M$ is the matrix representing the differential operator, $\mathbf{x}$ is the field and $\mathbf{b}$ represents the particular boundary condition in question. Popular methods for solving this equation are pre-conditioned conjugate gradient and Gauss–Seidel or Gauss–Jacobi iteration techniques, and we take the latter as the simplest example. We write the matrix $M$ in the form $M = D + N$, where $D$ is a diagonal matrix and $N$ has the off-diagonal elements of $M$. The equation is solved iteratively by the recurrence relation

$$D\mathbf{x}_n = \mathbf{b} - N\mathbf{x}_{n-1}$$

The nonzero elements of the matrix $M$ relate rows and columns corresponding to nearby space points. It follows that one can implement this recurrence relation efficiently on a concurrent processor as long as the machine itself has a three-dimensional mesh connection. One must assign nodes to subregions in the fashion analogous to the scheme shown in the box on page 59, so that the topology of the concurrent processor matches that of the underlying physical space. In fact, the algorithm given in the box for solving the Laplace equation is just the Gauss iteration technique.

There are important problems that involve the manipulation of full matrices. For instance, one approach to the numerical calculation of chemical reaction dynamics described by a multichannel Schrödinger equation is dominated by successive inversion of full $N_c \times N_c$ matrices, where $N_c$ is the number of channels. Variants of the familiar Gaussian elimination techniques are the preferred algorithms. The best decomposition for such problems appears to correspond to viewing the matrix as a two-dimensional "world" whose "units" are the matrix elements.[10] One divides this world into square subregions with, for example, each of 64 nodes holding $10 \times 10$ subblocks of an $80 \times 80$ matrix. One can show that as long as the concurrent processor has at least a two-dimensional periodic mesh connection, the implementation has linear speed-up and reasonable efficiency. The communication overhead is small and the lack of exact load balancing degrades the efficiency to about 50% with a simple algorithm, while more complicated implementations can improve even this satisfactory result significantly. This analysis holds for either inversion or eigenvector problems.

In fact, full matrix problems are another example where what counts for the efficiency is not the amount of communication but rather the ratio of communication to calculation. Let us give the basic idea. A typical suboperation in a matrix algorithm is the subtraction of one row (with a certain multiplier) from all other rows. Substantial communication is involved in sending the row to be subtracted to a particular processor. However, each transmitted matrix element is used in a calculation for every row stored in the given processor. If we have $n$ matrix elements stored in each processor as a $\sqrt{n}$ by $\sqrt{n}$ submatrix, then the ratio of calculation to communication is proportional to $n/\sqrt{n}$, or $\sqrt{n}$. It is interesting to compare this with our example of



**Data flow** in an 8-point fast Fourier transform. The original function is denoted $f_0(x)$ and appears at the left in **a**, stored in natural order, $x = 0, 1, \ldots, 7$. The algorithm proceeds to the right. The function $f_1$ is the result of partially transforming the original function $f_0$ in the highest, or 3rd bit of $x$. Similarly, $f_2$ and $f_3$ are the results of transforming in the 2nd and 1st bits, respectively. The function $f_3$ is simply related (through bit reversal) to the Fourier transform of the original function $f_0$. The three-dimensional cube in **b** shows the original function $f_0$ stored consistent with the binary labeling scheme explained in the text. The axes 0, 1 and 2 refer to incrementing the lowest bit of $x$, the second bit and the third. Part **c** shows the flow of data for the 8-point fast Fourier transform, after it is mapped onto the three-dimensional cube. The three pictures $f_1$, $f_2$ and $f_3$ correspond to the three iterations shown in **a**. Figure 4

a two-dimensional Laplace equation, where we get the same result even though the underlying matrix is sparse. Note that, as table 2 shows, these problems have the same form for the communication overhead. In the case of the full matrix, we have to transmit all $n$ elements, but we do $\sqrt{n}$ calculations for each element transmitted. For the sparse matrix, we only communicate $\sqrt{n}$ of the elements, but the number of calculations is also reduced to the order of one per element. We saw the same effect in the analysis of long-range-force problems. It is clearly a rather general result that features of a problem that increase or decrease the needed communication also alter the calculational load in the same direction. It is clear from our examples that many problems with substantial communication still have low values for the crucial ratio of communication to calculation.

## Fast Fourier transform

The fast Fourier transform is one of the single most important algorithms for the sciences. Any concurrent processor, to be at all considered "general purpose," must be able to perform this algorithm with reasonable efficiency. Beyond its obvious applications in signal processing and image processing, the fast Fourier transform is useful for solving linear partial differential equations with translational invariance.

As a concrete example, we turn once again to the $N$-body gravity problem. Instead of solving for the forces directly, we first find the gravitational potential $\varphi$, which is related to the force $\mathbf{F}$ by

$$\mathbf{F} = \nabla\varphi$$

The potential satisfies the Poisson equation

$$\nabla^2\varphi = 4\pi G\rho$$

Here $\rho$ is the mass density function. We apply these results to the $N$-body problem by first laying a finite grid over the continuous, three-dimensional space in which the particles move. Next, we define a discrete mass-density function on the grid sites by averaging over particles near each site, giving, we hope, a good approximation to the mass density function $\rho$. Finally, we solve the discrete version of the Poisson equation, finding the potential $\varphi$, which we differentiate numerically to get the force $\mathbf{F}$.

The computationally intensive part of the above procedure is solving the Poisson equation. The speed of the fast Fourier transform method makes it most attractive to do this by transforming to Fourier space. In the continuum, the idea is as follows. Transform the Poisson equation to Fourier space, arriving at an equivalent equation for the transformed functions:

$$-k^2\tilde{\varphi}(\mathbf{k}) = 4\pi G\tilde{\rho}(\mathbf{k})$$

As is the case for most linear partial differential equations, the solution becomes trivial—simply divide by $k^2$:

$$\tilde{\varphi}(\mathbf{k}) = -4\pi G\tilde{\rho}(\mathbf{k})/k^2$$

Finally, transform back to real space to find the solution $\varphi(\mathbf{x})$. Because the fast Fourier transforms take a time proportional to $N\log_2 N$, where $N$ is the total number of grid points ($N = L^3$ for a $L\times L\times L$ grid), and the "divide by $k^2$" step goes as $N$, the equation is solved in a time $c_1 N\log_2 N + c_2 N$.

There are various fast Fourier transform algorithms that are appropriate depending upon whether the number $N$ of points of the transform is a prime number, a composite or an integral power of 2. The algorithm for $2^\gamma$ points is the simplest and most commonly used; here we will concentrate on it.

A discrete Fourier transform requires the evaluation of an expression such as

$$F(k) = \sum_{x=0}^{N-1} f(x)\omega^{kx}$$

$$\text{for } k = 0, 1, \ldots N-1$$

$$\omega = \exp(-2\pi i/N)$$

Instead of evaluating the above sum directly, which would take $N^2$ steps, the fast Fourier transform for $2^\gamma$ points evaluates it in a series of $\gamma$ iterations, each iterate consisting of a Fourier transform in one of the binary digits or "bits" of $x$. It turns out[11] that the transforms in each bit partially decouple, allowing the algorithm to proceed rapidly, in $N\gamma$, or $N\log_2 N$ steps.

Figure 4a illustrates the flow of data for the fast Fourier transform for $2^3$ points. At each iteration, data that differ in one and only one bit, calculated in the previous iteration, are combined. So that our concurrent processor will perform efficiently, we want it to have a connection topology that allows these data to be "near" each other. The hypercube is one possibility. A convenient scheme for labeling the nodes at the $2^\gamma$ corners of a $\gamma$-dimensional hypercube is to label each by a $\gamma$-bit binary number. The $i$th bit of the number represents the coordinate of that node in the $i$th dimension. The edges of the hypercube connect the nodes. In terms of the binary labeling scheme, we see that nodes whose labels differ in one and only one bit are connected. The applicability of this topology to the fast Fourier transform now becomes clear: If the data are stored in the $2^\gamma$ hypercube consistent

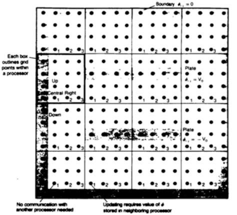
## Potential around a capacitor in a grounded box

Here we show a decomposition appropriate for a two-dimensional problem requiring the solution of Laplace's equation $\nabla^2\varphi = 0$ with particular boundary conditions: a parallel plate capacitor in a grounded box. We have set up a $16 \times 16$ grid with the field at a point $(i, j)$ denoted $\varphi_{i,j}$. Dots in colored regions denote field points where the potential $\varphi$ is fixed. Other dots denote interior points at which $\varphi_{i,j}$ is to be determined. The finite-difference approximation to Laplace's equation $\nabla^2\varphi = 0$ leads to a matrix inversion that one can solve iteratively. To each internal point $(i, j)$, we successively apply the basic algorithm

$$\varphi_{\text{central}} = (\varphi_{\text{left}} + \varphi_{\text{right}} + \varphi_{\text{up}} + \varphi_{\text{down}})/4$$

This is illustrated in the diagram on the right for one point labeled "central."

The diagram shows a possible implementation on a 16-node processor. Grid points marked 1 are updated simultaneously (one per node), then those marked 2, and so on. Note that the local nature of the differential equation means that application of the above equation needs either points stored internally to the node or points stored in processors that are adjacent in the array. This implies that a two-dimensional mesh connection for the nodes, corresponding to the nature of the domain and the local equation defined on it, is appropriate.


Boundary $\phi_{i,j} = 0$
Each box outlines grid points within a processor
Up
Central Right
Down
1
2
3
Plate $\phi_{i,j} = V_0$
Plate $\phi_{i,j} = -V_0$
No communication with another processor needed
Updating requires value of $\phi$ stored in neighboring processor

with the binary labeling scheme of the nodes, the topology is natural for the algorithm. This is shown in figures 4b and 4c, where we see that the seemingly complex topology of the fast Fourier transform maps naturally onto the cube. At each iteration of the fast Fourier transform, the data points differing by one bit are one communication step apart and the algorithm proceeds straightforwardly.

Because the Boolean hypercube is a natural topology for the binary fast Fourier transform, it is not surprising that a detailed analysis shows[10] that the transform runs with high efficiency. On our machines at Caltech, we have implemented fast Fourier transform codes for galactic dynamics and simulations of the early universe.

The fact that the hypercube is well suited for fast Fourier transforms does not mean that simpler architectures, such as meshes, cannot perform these transforms. They can, although with some complications in the structure of the algorithm and some degradation in efficiency. For more detailed considerations concerning fast Fourier transforms on meshes and hypercubes, see reference 12 and references therein.

In the past, the subject of algorithms for parallel processing was a somewhat esoteric pursuit, known to a few computer scientists. It was mainly a theoretical subject, for the simple reason that few machines existed. The technology of very-large-scale integration is rapidly changing the situation, and it will soon be possible to build machines of very high processing capability cheaply. We believe that with this motivation, scientists will learn to use the parallel processing algorithms already known, and will no doubt invent better ones. This will not only delineate the basic principles of decomposition but help the development of tools, such as languages and compilers, to make concurrent processors as easy to use as sequential machines.

## References

1. C. Seitz, J. Matisoo this issue, page 38.
2. C. A. Mead, L. A. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass. (1980), chapter 8; C. Seitz, *Proceedings of the MIT Conference on Advanced Research in VLSI*, Artech House, Dedham, Mass. (1982).
3. A general reference for algorithms for parallel processors is H. T. Kung, *Advances in Computers, volume 19*, Academic, New York (1980), page 65.
4. C. L. Seitz, J. VLSI and Computer Systems 1, no. 2, in press.
5. E. Brooks III, G. Fox, S Otto, M. Randeria, W Athas, E. DeBenedictis, M Newton, C. Seitz, Nucl. Phys. **B220** [FS8], 383 (1983); E. Brooks III, G Fox, R. Gupta, O Martin, S Otto, E. DeBenedictis, Caltech preprint CALT-68-867 (1981).
6. R. Pearson, J. Richardson, D. Toussaint, University of California, Santa Barbara, Institute for Theoretical Physics preprint number NSF-ITP-81-139 (1981)
7. N. Christ, A Terrano, Columbia University preprint CU-TP-261 (1983).
8. J. D Stack, Phys Rev D **27**, 412 (1983); N. Isgur, G. Karl, PHYSICS TODAY, November 1983, page 36.
9. E. Brooks III, G. Fox, M. Johnson, S. Otto, J Stack, P. Stolorz, W Athas, E. DeBenedictis, R. Faucette, C. Seitz, submitted to Phys. Rev. Lett., available as Caltech preprint CALT-68-1112; S Otto, J. Stack, Caltech preprint CALT-68-1113.
10. G. Fox, Caltech preprints CALT-68-939 and CALT-68-986.
11. E. O. Bingham, *The Fast Fourier Transform*, Prentice-Hall, Englewood Cliffs, New Jersey (1974).
12. R. W Hockney, C R. Jesshope, *Parallel Computers*, Adam Hilger, Bristol, England (1981).