Journal of Statistical Physics, Vol. 43, Nos. 5/6, 1986

Monte Carlo Physics on a Concurrent Processor

G. C. Fox,[1] S. W. Otto,[1] and E. A. Umland[2]

We show how Monte Carlo problems of various kinds can be efficiently implemented on a parallel processor possessing the interconnect topology of a hypercube. Examples discussed include lattice gauge theory, the physics of two-dimensional melting, long-range interactions, and simulated annealing.

KEY WORDS: Monte Carlo; concurrent computation; hypercube; load balancing; Coulomb gas; lattice gauge theory; melting.

Dedicated to
Eric Alexander Umland

BS MIT, 1978
Ph.D. Rice University, 1982
Bantrell Prize Fellowship Caltech, 1983
Scientist in Caltech Concurrent Computation Program, 1985

Tragically, Eric was killed on November 17, 1985 in a light plane crash during bad weather. He was a brilliant scientist with whom it was a joy to work. His research contributions will be remembered and continued.

WATER
Ashes on the wind
ice cold, snow
winter of the heart

Blue jay cries
lost and angry
against a cold and empty
dawn
wings clipped
soul flies on alone

JoAnn Boyd Anderson

Work supported in part by DOE grant DE-FG 03-85ER25009 and the Parsons and System Development Foundations. S. W. Otto is supported by a Bantrell Research Fellowship.
[1] California Institute of Technology, Pasadena, California 91125
[2] Jet Propulsion Lab, Pasadena, California 91109.

0022-4715/86/0600-1209$05.00/0

1. INTRODUCTION

Despite the remarkable advances in computing power achieved in the latter half of this century, it seems unlikely that further order-of-magnitude progress due to improvements at the microscopic (integrated circuit) level can be maintained. Even if the seemingly inexhaustible well of cleverness of the chip designers fails to run dry, economical and fundamental physical constraints will eventually force a leveling off.

Accordingly, it is worthwhile to examine other approaches to unbounded computing power. At present, the concept of parallel processing, where many computers run parts of a problem simultaneously, appears more promising. If parallel computing technology and algorithms can be developed such that the speed-up (ratio of computing speeds of a concurrent computer to a single processor) increases sufficiently rapidly with the number of processors in the parallel machine, then increases in both calculational speed and (distributed) fast memory are in principle unlimited. The Caltech/JPL Concurrent Computation Program (C^3P)[1] has produced hardware and software capable of satisfying this criterion for a wide class of large computational problems. It is the purpose of this paper to describe our work on a particular member of this class, namely Monte Carlo physics simulations.

We begin with a discussion of the Caltech/JPL parallel processors and of the issues involved in writing efficient parallel algorithms. Several parallel processors have been constructed and are running; we describe their general features below.

The Hypercube

A Caltech/JPL parallel processor is a multiple-instruction-multiple-data (MIMD) machine. That is, each node is a complete CPU with arithmetic unit and memory. Each node can be execute instructions and access its own memory independently of the others. This is in contrast to, for example, a single-instruction-multiple-data (SIMD) machine (another common parallel architecture) which applies the same instruction in all of its nodes to different sets of data.

The nodes can communicate with each other via a message-passing system. An exterior processor, called the *Intermediate Host* (IH) serves as a data conduit between the hypercube and the outside world. The communication topology of the computer is that of a p-dimensional cube (hence the name "hypercube") with 2^p processors forming the vertices (see Fig. 1). The p links per vertex represent the possible communication channels. Thus, though the total number of nodes $(N = 2^p)$ may be large, the

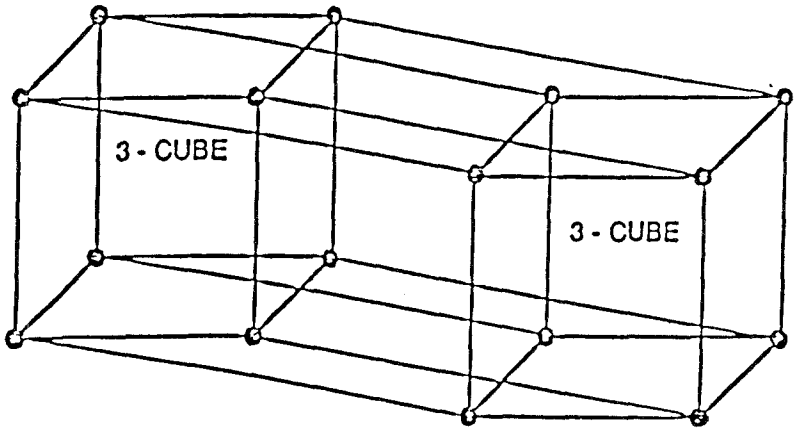

Fig. 1. Hypercube architecture: 2^p computers at the corners of a cube in p dimensions, connected along the edges of the cube.

maximum communication distance is small, equaling p. This topology appears to be an excellent compromise between the two extremes of no communication between nodes (simple design but very limited in the types of problems the machine can handle) and direct communication links between all nodes (fastest communication, but $N(N-1)/2$ links for an N-processor machine calls for complex and expensive hardware). Moreover, the hypercube includes within it the topologies of most scientific problems (e.g., meshes, rings, binary FFT). More specific information on the hardware for the various hypercubes in use or planned can be found in the Table I.

Philsophy of Parallel Algorithms

Define the efficiency of a parallel processor as

$$\varepsilon = \frac{S}{N} \tag{1}$$

where a particular problem runs S (the speedup) times faster on an N-node concurrent processor than on a single-node machine using an optimal sequential algorithm. The efficiency is less than unity for three reasons:

1. Communication overhead
   The time spent communicating between processors is "wasted." It is desirable to minimize the ratio of communication to calculation time.

Table I. Comparison of Hypercubes

	← — — Finished/Available Machines[c] — →		← Next generation →	
	Caltech(Seitz), Mark I	Caltech(JPL), Mark II	INTEL, iPSC	Caltech(JPL), Mark III
Main processor	Intel 8086, 5 Mhz	INTEL 8086, 8 Mhz	INTEL 80286, 8 Mhz	MOTOROLA 68020, 16 Mhz
Nodes/module	8	32	32	32
Maximum nodes/machine	64	128	128	1024
Memory/node (Mbytes)	0.125	0.25	0.5	4 + add-on
Floating point processor	INTEL 8087	INTEL 8087	INTEL 80287	MOTOROLA 68881, WEITEK 1164/1165 (Scalar unit)
64-bit floating point performance (flops/node)	20–30 K	30–45 K	30–45 K	100 → 1000 K[b]
Raw communication speed (Mbit/s)	4	6	10	100
Measured communication speed (including software overhead) (Mbit/s)	0.5	0.8	0.01 → 0.5[a]	15 (estimate)
Maximum machine memory (Mbytes)	8	32	64	4096
Maximum machine performance (×VAX11/780)	10	25	~30	1000 → 5000[b] (scalar)

[a] The higher bandwidth (0.5 Mbits/s) is for large packers (≥1 Kbyte) only. The lower value corresponds to small packets containing few words.
[b] The mark III vector floating point unit will give up to a factor of 10 higher performance.
[c] Commercial hypercubes from AMETEK, Floating Point Systems and NCUBE were not available for evaluation when this article was written.

2. Load balancing
   It is optimal to have each processor doing the same amount of work; otherwise some processors will be idle part of the time.
3. Nonoptimal parallel algorithm
   It may be that the most efficient parallel algorithm for a given problem is not as good as its best sequential counterpart. As an example the so-called "bitonic" sorting algorithm is very efficient in achieving load-balancing and low communication cost on the hypercube, but is poorer than modern sequential sorting schemes.[2]

In practice one attempts to balance the computational load and minimize communication time by an optimal decomposition of the problem. That is, the degrees of freedom (fields, particles, galaxies, matrix elements, etc.) of the problem are divided up into subdomains each of which is assigned to a different processor. In general, this is not a trivial task. However, for the Monte Carlo physics problems addressed in this paper (QCD, 2-D melting, long-range interactions), decomposition is relatively straightforward because the computational work associated with each degree of freedom is about the same. It follows that the optimum strategy is to attempt to maintain equal numbers of degrees of freedom in each processor to achieve load-balancing. There exists as well many situations in which the computational work per degree of freedom varies greatly. An example is a war game simulation, where the computational load involved in calculating the effects of a missile landing is significantly greater than that of advancing a soldier forward in time! Such problems are termed "inhomogeneous" and represent a difficult optimization problem for parallel implementation. A statistical approach, namely simulated annealing, to load-balancing such problems will be described in Section 3.

If we focus for the moment on homogeneous problems, we will be able to elucidate some general points concerning concurrent computation. A primary question is what problems are amenable to solution via parallel algorithms. The answer is "large" problems, large in the sense of possessing very many degrees of freedom. Small problems suffer in general from a low efficiency because communication between processors and load imbalance effects consume a significant fraction of the total running time. This is clear from a consideration of "short range" problems where a calculation for any one degree of freedom depends only on its near neighbors. Here communication between processors is clearly an edge effect and declines as a fraction of the total run time as the number of degrees of freedom per processor grows. Longer-range problems retain high efficiency despite greater communication needs, however, because they are in general much more computationally intensive as well. This example illustrates another feature of concurrent computation; parallel computers prefer "hard" problems, that is, problems that require significant calculation per degree of freedom.

It is possible to quantify some of these ideas in the following equation for the efficiency

$$\varepsilon = \left(1 - \frac{\text{const}}{f(n)} \cdot \frac{t_{\text{comm}}}{t_{\text{calc}}}\right) \tag{2}$$

Here $t_{\text{comm}}/t_{\text{calc}}$ is the ratio of typical interprocessor communication

time to calculation time and is the key hardware characteristic determining the communication overhead. The quantity $f(n)$ is a function of the number of degrees of freedom in each processor. For all problems we have studied, f is a monotonically increasing function of n. Finally, the constant depends on the amount of calculation per degree of freedom and declines as the complexity of the problem increases. Equation (2) demonstrates that the speed-up $S = \varepsilon N$ will be linear in the number of processors as long as n is kept fixed. The efficiency suffers, however, if one keeps the problem size fixed while scaling up the number of nodes in the concurrent computer.

It should now be clear why Monte Carlo physics problems are good candidates for treatment via parallel algorithms: they typically consist of large numbers of degrees of freedom (sometimes 10^6 or more) and often require quite complex computations. We point out, however, that Monte Carlo problems are but one example of a wide class of problems that we have found to be amenable to numerical solution on a parallel computer.

2. MONTE CARLO PHYSICS ON THE HYPERCUBE

Before proceeding to the several kinds of physics problems we have attacked by Monte Carlo methods on the Caltech/JPL parallel processors, we briefly describe two issues that are important in the development of convenient Monte Carlo algorithms. We first address the question of generating random numbers in parallel. This should not be done in some naive way; for instance, if one merely gives the random number generators different starting seeds in different nodes, how is one to be sure that some strong correlation doesn't develop between the various sequences? Fortunately, there exists a way for a parallel random number gnerator to easily mimic the behavior of that on a sequential machine.

The most common method for generating pseudo-random numbers is called the *linear congruential method* and is given by

$$\tau_{n+1} = (a\tau_n + b) \bmod(m) \tag{3}$$

where τ_n is a sequence of pseudo-random numbers and a, b, and m are constants. It is possible to reproduce this sequence exactly on a parallel machine. For N processors, the idea is to have every processor calculate the Nth iterate of (3). Given (3), it is easy to write down the $n+N$th member of the sequence directly in terms of the nth[3]

$$\begin{aligned} \tau_{n+N} &= (A\tau_n + B) \bmod(m) \qquad (4) \\ A &= a^N \\ B &= (1 + a + a^2 + \cdots + a^{N-1})b \end{aligned}$$

Now the idea is to have each node compute random numbers using (4) (the sum is calculated only once and stored). If the nodes are now gives a staggered start in the random number sequence, the procesors will "leap-frog" over one another and will reproduce exactly the random number sequence of a sequential machine. This is convenient for debugging complex parallel programs and also erases worries one might have regarding correlations between the random numbers at each node.

The second issue a Monte Carlo parallel programmer must face is the problem of satisfying detailed balance. That is, one is assured that a set of configurations $\{C\}$ are distributed according to the correct (Boltzmann) distribution by requiring

$$\frac{P(C \to C')}{P(C' \to C)} = \frac{e^{-S(C')}}{e^{-S(C)}} \tag{5}$$

where P is the transition probability from one configuration to the next and S is some function of configurations (e.g., the action). In order for detailed balance to hold during a configuration update, it is necessary that the previous state be well-defined. This presents some difficulty in concurrent applications, where many updates can occur simultaneously. Such a problem is particularly severe for systems with long-range interactions, where the influence of a change in a single degree of freedom is felt over great distances. We shall see that maintaining the detailed balance condition is an important constraint affecting the development of concurrent Monte Carlo algorithms to measure the properties of such systems.

Short-Range Interactions: Lattice Gauge Theory

Our first example[4] is relatively easy to implement with a parallel architecture. Lattice gauge theory models a quantum field theory on a discrete, finite spare-time lattice. The degrees of freedom are the field variables at each site. The problem is homogeneous and regular because the amount of computational work associated with each variable is the same and because sublattices of equal size can be assigned to each processor. Consequently, load-balancing is trivial. Concurrency arises from updating via Metropolis or heat bath techniques separate sites in each processor simultaneously. Since the gauge theories commonly studied possess interactions between field variables on nearest-neighbor sites only, communication and detailed balance constraints are an edge effect.

Let us examine this point in some detail. For nearest-neighbor interactions, lattice dimension d, and number of variables per processor n, the calculation time involved in a sweep of the lattice is $2dnt_{\text{calc}}$. The

communication time is $2dn^{(d+1)/d}t_{\text{comm}}$ (see Fig. 2). The ratio falls with increasing n as

$$\frac{\text{comm}}{\text{calc}} \sim \frac{1}{n^{1/d}} \frac{t_{\text{comm}}}{t_{\text{calc}}} \tag{6}$$

which derives $f(n)$ of (2). Remarkably, the communication overhead ratio *improves* as the length of interaction increases. This is because while communication time increases, so does the calculation time. In fact, for the two-dimensional case illustrated in Fig. 2, it can be shown that $f(n) \to n$ as the interaction length $\to \infty$. For the short-range lattice gauge problem considered here, detailed balance is easily satisfied since neighboring points across a processor boundary need never be updated simultaneously. The operating system (OS) is designed so that processors that get out of step for some reason (e.g., one runs slightly slower than the others) are

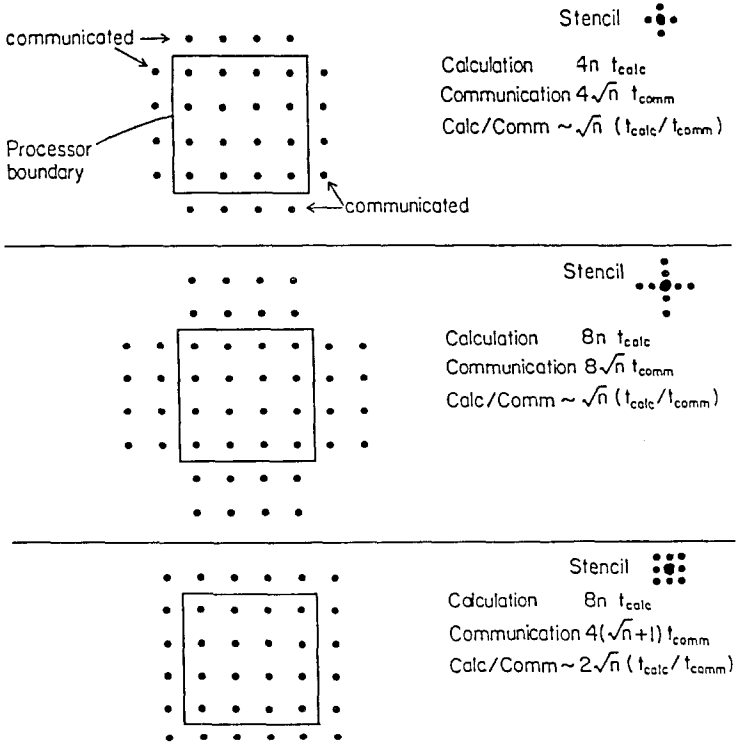

Fig. 2. Communication as an edge effect, illustrated for a two-dimensional lattice with 16 sites per processor.

resynchronized whenever communication occurs. Another characteristic of this OS is that communication occurs only between processors on *neighboring* vertices of the hypercube. It is called the Crystalline Operating System (CROS).[1]

In the Lagrangian form of lattice gauge theory, most observables require integrals over loops that can spread over several processors. Explicitly keeping track of all necessary interprocessor communications is more difficult here than for the updating problem described above. However, it is possible to construct a simple recursive algorithm which takes a list of number as input (each number giving the direction of one step in the loop) and travels around the loop. The same shape loop is necessarily calculated by all processors simultaneously, which is entirely satisfactory.

The problem we have studied most extensively is the potential as a function of distance between static quarks in the quenched approximation to lattice QCD (results are shown in Fig. 3 for a 20^4 lattice). Efficiencies on

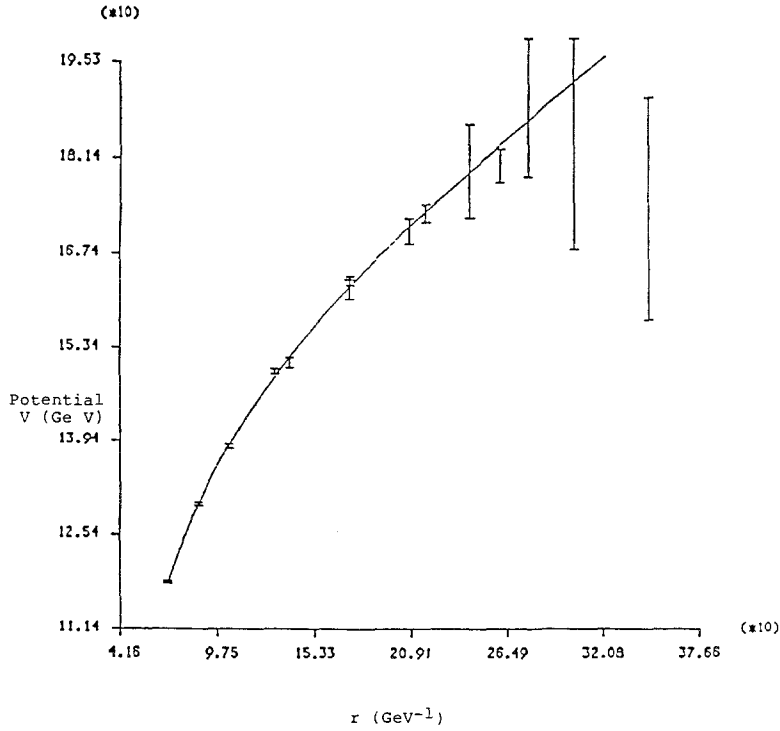

Fig. 3. $Q\bar{Q}$ potential vs distance computed on 20^4 lattice in the quenched approximation to lattice QCD.

the order of .95 were consistently obtained. This was due to the high degree of load-balancing obtained, low communication overhead, and the inherent complexity of the calculations involved (multiplication of $SU(3)$ matrices). In particular, the constant in (2) is found to be proportional to $1/m$ for $SU(m)$ gauge groups. Most problems in lattice gauge theory appear to be amenable to a parallel treatment. The kinds of problems we have or will run include the $q\bar{q}$ potential, glueball masses, renormalization group, field distributions, pseudo-fermions, and Hamiltonian and Langevin methods.

Medium Range Interactions: Two-Dimensional Melting

Here we consider an example[5] that poses two additional challenges to the concurrent programmer—how to handle an "irregular" problem, and the need to be careful in maintaining detailed balance. The problem studied is that of the solid–liquid phase transition in two-dimensions. The system consists of a collection of particles interacting through a pairwise potential. A standard Metropolis Monte Carlo procedure is employed in the simulation. As usual, the correct distribution of configurations will be attained only if particles close enough to affect one another are never updated simultaneously. This is, of course, never a problem in a sequential algorithm but require careful programming on a parallel processor.

The particles in the simulation are grouped into structures called cells. Each cell represents a fixed region of space. The size of the cell is chosen sufficiently large so that the only particles that can affect a particle during an update can be found in the same cell as the particle in question and in the eight neighboring cells. The initial configuration consists of 16 particles in each cell. The cells are distributed equally among processors; each processor can contain from 1 to 64 cells (16 to 1024 particles). It is clear that during the simulation particles can become unevenly distributed among the processors since they are free to move between cells. Computational loads are therefore unbalanced as well, though, since the density is reasonably high, the imbalance is not great. Because the computational work *per particle* remains the same, this problem falls within a class termed "irregular homogeneous."

Updates occur concurrently in all processors. Detailed balance is assured by the following "time-stamping" algorithms. When a particle in a cell in processor A is being updated and requires position information about particles in a neighboring cell in processor B in order to evaluate the potential energy, a request for this information is sent to B. We associate with each particle being updated a time. This time will be the same for *all* requests generated by this particular particle update in processor A. (The time is generated by the local clocks in each processor. The algorithm is

most efficient if one could use a global clock, but is consistent whatever the relative timing of the local clocks. The algorithm synchronizes the clocks in the processors occasionally and is not affected by small drifts in between.) When a conflict occurs with a simultaneous update in processor B, for example, a response is sent if the time stamp of the information request from A is earlier than that of update occurring in B. If the time stamp of the information request from A is later, it must wait for a response until the update in B is finished. The "earliest" update is given priority.

This rather elaborate means of resolving update conflict is necessary because of the special nature of the communication scheme used for this problem. Instead of communications synchronizing the progress of the processors as in the previous example, problems that are irregular require the ability to run asynchronously on a parallel machine for efficient operation. That is, the sending or receiving of a message should not halt operations in a processor, rather the processor should be able to complete its immediate task before attending to exterior data. Such a communication system is called "interrupt driven" (IDOS(12)). Another characteristic of this system is the ability to forward messages, since not all messages are addressed to adjacent processors in this problem.

The complicated communication structure and imperfect load balancing reduce the efficiency of the algorithm when compared to homogeneous problems. Nonetheless, efficiencies as high as 85% were achieved for the maximum size problem of 1024 particles per processor. For the minimum size problem of 16 particles per processor (1 cell), the efficiency was 50%. The hypercube prefers large problems!

Long-Range Interactions: Two-Dimensional Coulomb Gas

We come now to the problem[6] of long-range interactions, where the efficiency of a parallel computation has often been questioned. Consider a two-dimensional Coulomb gas at temperature T on a $D \times D$ square lattice with, for simplicity, free boundary conditions. On each site $\mathbf{r} = (x, y)$ is defined as an integer electric charge variable $q(\mathbf{r}) = -1, 0, +1$. The energy of each configuration $\{q(\mathbf{r})\}$ is

$$E(\{q(\mathbf{r})\}) = \sum_{\mathbf{r}} q^2(\mathbf{r}) \ln \frac{D}{c} - q(\mathbf{r})\, V(\mathbf{r}) \tag{7}$$

where the electric potential

$$V(\mathbf{r}) = \sum_{\mathbf{r}' \neq \mathbf{r}} q(\mathbf{r}') \ln \left(\frac{|\mathbf{r} - \mathbf{r}'|}{D} \right) \tag{8}$$

and c parameterizes the charge self-energy. Configurations $\{q(\mathbf{r})\}$ are generated by a heat-bath type of algorithm. Details can be found in Ref. 6. For our purposes it is sufficient to know that an exponential function of the potential V, ω, is defined so that if an update attempt

$$\Delta q = q^{\text{new}}(\mathbf{r}_0) - q^{\text{old}}(\mathbf{r}_0) \neq 0 \tag{9}$$

then at the other lattice sites $\mathbf{r}$, $\omega(\mathbf{r})$ is updated by

$$\omega(\mathbf{r}) \to \omega(\mathbf{r}) \left(\frac{|\mathbf{r} - \mathbf{r}_0|^2}{D^2} \right)^{\Delta q/T} \tag{10}$$

and *only then* does the program proceed to a new $\mathbf{r}_0$. This ordering is obviously crucial for maintaining detailed balance. Such a procedure for handling long-range forces is easily implemented on a sequential machine; we proceed to show that an efficient concurrent algorithm exists as well.

For N processors, we divide up the lattice into N domains with an equal number of sites n. Each processor is assigned a domain and stores the charges q and weights ω in the domain. The IH will manage the progress of the algorithm. All processors concurrently generate charge updates at a site $\mathbf{r}_0^i$, where i is the processor number, and send the site location and the value of Δq to the IH. Because the acceptance rate is small, many hits are made at each site. The IH steps through this data set until the first nonzero Δq is obtained. It and the site location are passed to all the processors so that the weights $\omega(\mathbf{r}^i)$ can be updated. *All other updates are discarded.* This insures that the detailed balance is maintained. This sequence is repeated with the proviso that the IH restarts the inspection of the data set for successful updates at the procesor following that whose update was successful in the prior iteration (the set is inspected cyclically, i.e., the first position follows the last). The low acceptance rate requires this procedure; if the acceptance was high the updates could be done sequentially in a single processor. For large or small acceptances, we thus find that the calculation time spent updating the $\omega(\mathbf{r})$ is much greater than that spent updating q or communicating to and from the IH (the "sequential bottleneck"), and so the time wasted is relatively small. Hence, efficiencies on the order of .95 are obtained. At constant n, therefore, speed-up is linear in n with a slope ≈ 1. A detailed analysis of the efficiency can be found in Ref. 6.

3. MONTE CARLO APPROACH TO CONCURRENT OPERATING SYSTEMS

Load-balancing affects crucially the prformance of a computation executing in parallel on a concurrent processor (CP). By "load-balance" we

refer to the amount of cpu idling occurring in the processors of the concurrent computer: a computation for which all processors are continually busy (and doing useful, nonoverlapping work) is considered perfectly balanced. This balance is nontrivial to achieve, however. The problem of distributing a computation in an efficient manner onto a CP can be fruitfully attacked via the Monte Carlo technique of simulated annealing.[7] The work described in this section is described in more detail in Ref. 8.

Operating System Model

We have some large computation which we would like to execute in parallel on the CP. To do this, of course, the computation needs to be split up into small pieces which we will call processes. The number of processes is not necessarily the same as the number of processors of the CP. Processes will need to communicate with one another in order for the computation to proceed. Assume that the processes and their communication requirements are changing with time; processes can be created or destroyed, communication patterns will move. This is the natural choice when one is considering timesharing the CP, but can also occur within a single computation. It is the task of the Operating System (OS) to manage this set of processes, moving them around if necessary, so that the CP is used in an efficient manner.

The OS performs two primary tasks. First, it must monitor the ongoing computation so as to detect bottlenecks, idling processors, and so on. Second, it must modify the distribution of processes and also the routing of their associated communication links so as to improve the situation. In general, it is very difficult to find the optimum way of doing this; in fact, this is an NP complete problem. Approximate solutions, however, will serve just as well. We will be happy if we can realize a reasonable fraction (lets say .5) of the potential computing power of the CP for a wide variety of computations. The Monte Carlo based method of simulated annealing seems to offer a way of doing this. We will see in what follows that the OS functions as a heat bath, keeping the computation "cool" and therefore near its' ground state (optimal solution).

The Physical Analogy

One may usefully think of a parallel computation in terms of a physical analogy. Treat the processes as "particles" free to move about in the "space" of the CP. The requirement of load balancing acts as a short range, repulsive "force," causing the particles, and thereby the computation, to spread throughout the CP in an evenhanded, balanced man-

ner. The situation is somewhat similar to a gas or fluid filling up a container. This analogy, though, is not complete. In a gas, the repulsive pressure which fills the container is due to the microscopic motion (velocity) of the particles, not to any true, repulsive force between them. In the case at hand, we do not want the particles (processes) to have a significant velocity—we want them to move slowly so that they "stay put" in processors long enough to do useful work. A better analogy, therefore, is that of particles interacting via a repulsive force in a system at a low temperature.

A conflicting requirement to that of load-balancing is interparticle communications; the various parts of the overall computation need to communicate with one another at various times. If the particles are far apart (distance being defined as the number of communication steps between them) large delays will occur, slowing down the computation. We therefore add to the physical model a long-range, attractive force between those pairs of particles that need to communicate with one another. This force will be made proportional to the amount of communication traffic between the particles, so that heavily communicating parts of the computation will coalesce and tend to stay near one another in the computer.

Simulated Annealing

The above can be taken as a rough description of the "Hamiltonian" of the parallel computation. The problem of executing the parallel computation in an efficient manner now becomes that of finding the ground state of this Hamiltonian. A powerful method of solving this problem is called simulated annealing.[7] This technique begins with an arbitrary initial state (in our case, an arbitrary decomposition of the processes onto the CP). The Metropolis Monte Carlo method is then applied to this starting configuration, where trial changes are made to the configuration and are accepted or rejected in the usual way. In this way, the OS functions as a heat bath at temperature T. When one starts the annealing, bringing the system in contact with the heat bath, the system is at some temperature other than T. After some amount of time (the thermalization time) the system and the heat bath reach thermal equilibrium. The annealing now consists of slowly (adiabatically) lowering T. This pulls down the temperature of the system with it (if done sufficiently slowly) so that eventually only the ground-state configuration of the system survives (remember, probability $\sim e^{-H/T}$ and $T \to 0$).

Physically, the OS functions to keep the system in thermal equilibrium, and cool.

A Good Hamiltonian

Let us be more precise and actually specify a Hamiltonian with the desired features. Think of the processors as "sites" and of the communication channels between them as bonds or links. The Hamiltonian will be a sum of terms defined on the sites and the bonds. Define

$$W_i = \text{the computational load of process } i \tag{11}$$

The OS determines this load by monitoring what happened in the recent part. Once we have the W_is we need to add to the Hamiltonian a quadratic term in the sum of W_is at the node so as to affect load-balancing

$$H_{\text{site}} = \alpha \left(\sum_i W_i \right)^2 \tag{12}$$

For α positive, this produces a short-range repulsive force between processes.

Now for communication costs. A reasonable choice for the cost function of communications seems to be the following. Define

$$c_{ij} = \text{amount of communication traffic between processes } i \text{ and } j \tag{13}$$

$$d_{ij} = \text{number of steps in computer of chosen pathway}$$

$$H_{\text{comm}} = \sum_{ij} c_{ij} d_{ij}$$

$$= \text{communication cost which impacts the system linearly}$$

$$= \text{a linear potential energy between processes} \Rightarrow \text{a constant (as a function of distance) attractive force between processes.}$$

H_{comm} is a long-distance nonlocal term, but because we have made it linear in d_{ij} (which seems a correct choice) we can actually deal with it in a local manner. Communication costs are naturally associated with the bonds of the machine. The OS will monitor the communication traffic going through a channel (bond) and then associate with this traffic an energy cost H_{bond}. Since any particular communication pathway will show up in all the bonds along its path, this will produce a linear (in distance) cost to the overall system. We have

$$H_{\text{Total}} = \sum_{\text{sites}} H_{\text{site}} + \sum_{\text{bonds}} H_{\text{bond}} \tag{14}$$

As defined so far, we can think of this Hamiltonian as describing a set of particles which repel each other at very short distances and also have a

set of rubber bands stretched between them which causes them to attract one another over long distances. There is one additional effect we would like the Hamiltonian to produce—we do not want the communication traffic to be overly high at any particular link. Physically speaking, we can think of this congestion effect as being a short-range, repulsive force between the rubber bands themselves all along their lengths. This seems to be a complex interaction to model, but can actually be done easily. We do this by completing our specification of H_{bond}. We do what we did before—measure the traffic through a link, but instead of associating a linear cost (in traffic) at this link, we use a quadratic cost. That is

$$t_{\text{link}} = \text{traffic through link} \tag{15}$$
$$H_{\text{bond}} = \gamma(t_{\text{link}})^2$$

with γ some parameter. As before, this quadratic cost causes the "rubber bands" to repel one another all along their path. Our complete Hamiltonian is that of (14), with H_{site} given by (12) and H_{bond} given by (15).

This rather nontrivial (but easily, that is, locally, computable) Hamiltonian load balances, holds down communication delays, and holds down communication traffic congestion (which effects both the startup delay and flow-through, streaming rate of communications).

A Toy Example

To illustrate a few of the ideas presented here we will present the results of the above methods applied to a simple example. The example computation to be performed on the CP is the time evoluation of a set of particles about in a two-dimensional world, interacting via a short-range force. In the terminology of the last section, we can think of each physical particle as representing one process. The short-range force means that the computational load associated with the update of a single particle is a function dependent upon the number of neighboring particles to the one in question.

The usual method of evolving a set of particles like this on a CP is shown in Fig. 4—the problem is decomposed by dividing the physical space up into equal area squares. The problem with this type of decomposition is that the particles move about, form clumps, shock waves, and so on, causing load imbalances. This type of performance degradation has been seen in actual computations on the Caltech/JPL Hypercube.[9] We model this kind of imbalance by choosing the particles of our toy example to clump somewhat toward the center of the space, as is seen in Fig. 4. The

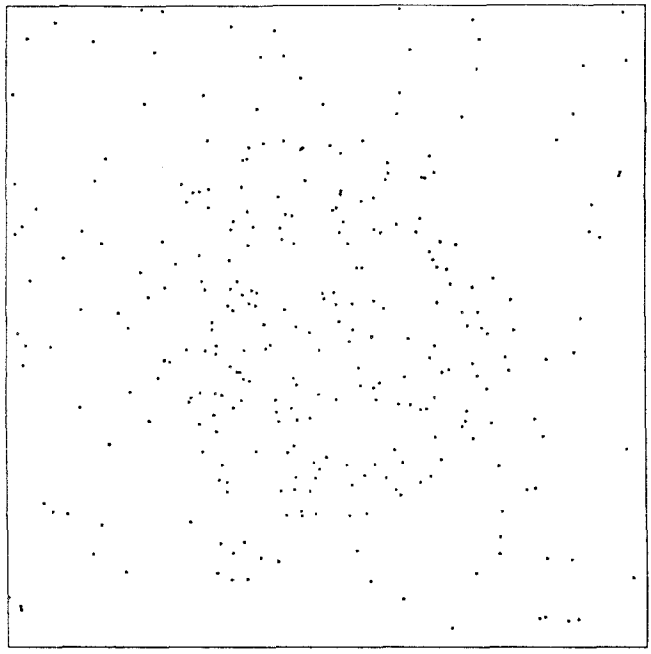

Fig. 4. The toy problem to be load-balanced: the time evolution of particles moving about in a two-dimensional world. The particles interact via a short range force. This configuration was constructed with a strong clumping toward the center in order to study load-balancing.

performance degradation that this clumping would lead to for the square decomposition is shown in Fig. 5, where it is seen that the computational load (including communication costs) per processor varies from 173 to 3 (in some arbitrary units).

The computational efficiency for the square decomposition can be estimated as follows. The average computational load for this example is 39.6, so an optimal decomposition would give each processor an amount of work slightly higher than this average. The optimal value will be somewhat higher than the average since the optimal decomposition will necessarily have greater communication costs than the square decomposition of Fig. 5. This is due to the fact that as the high load areas of the problem are divided up among many processors, more communication traffic must occur, and this is counted as part of the energy cost. For the square decomposition, however, one processor has a load of 173, and the overall computation will proceed only at the speed of the slowest processor. An estimate of the efficiency is therefore 40/173, or 0.23.

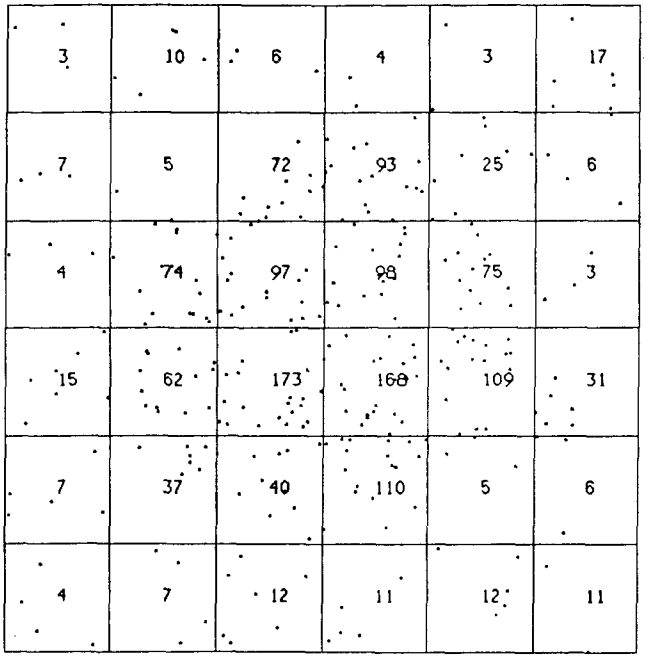

Fig. 5. The same as Fig. 4, with the usual, square decomposition overlaid. The numbers represent the total computations load in each processor. At this point, the computation is extremely unbalanced: the loads vary from a high of 173 to a low of 3.

A simulated annealing Monte Carlo was applied to the example. For ease of visualization, the processor boundaries were moved instead of moving the particles, with the constraint that the regions update by each processor remained convex quadrilaterals. After annealing, the result shows in Fig. 6 was obtained, with computational loads varying from 49 to 31, within about 20% of the optimum solution. The processors have migrated toward the center, all getting a "piece of the action" at the central clump of particles. Note that, as mentioned above, the mean load per processor increases as the annealing proceeds. The efficiency of the computation, now that the simulated annealing has taken place, is 40/49, or .82.

Continued annealing would eventually find the actual optimum, but a point of diminishing returns is quickly reached where many Monte Carlo sweeps are required to improve the situation only slightly. The Monte Carlo itself will use up computational cycles of the CP, so it is clear that in any real situation one will have to put up with some imbalance. It is worth pointing out that the simulated annealing noticeably outperformed simple iterative improvement. Iterative improvement can be though of as the

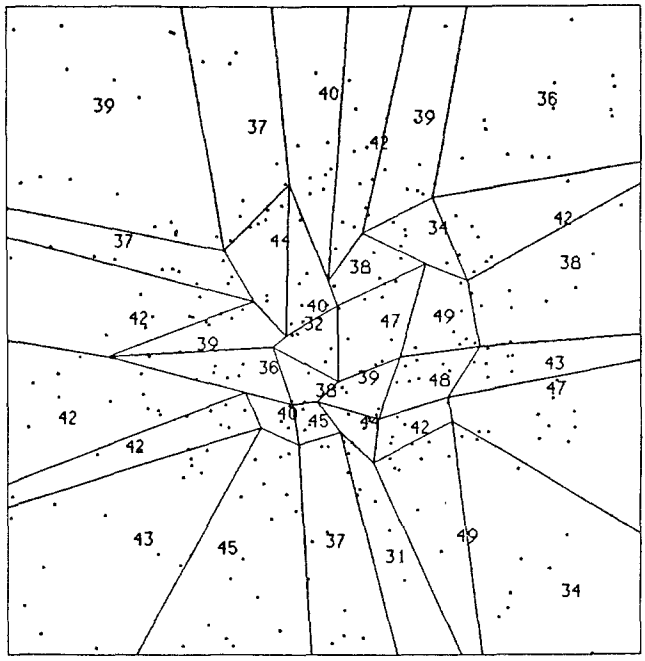

Fig. 6. The result after annealing the decomposition. The processor regions were restricted to remain quadrilaterals. Balance is now quite good: varying from 49 to 34.

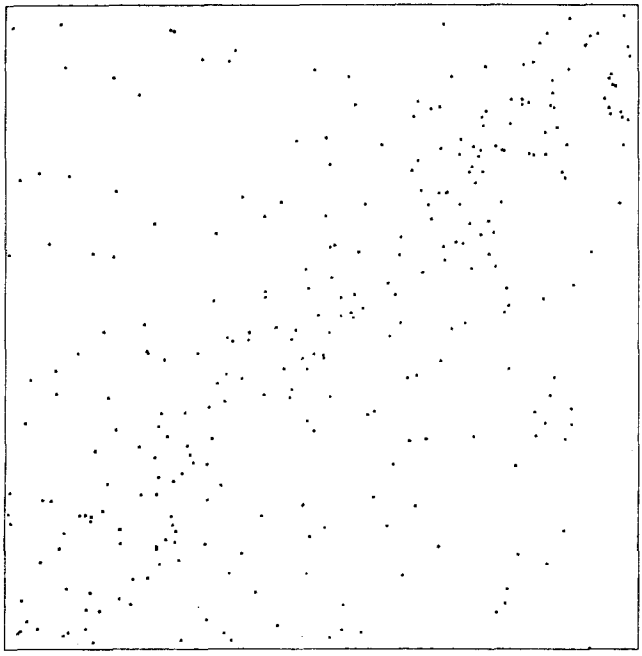

Fig. 7. The "shock wave" example.

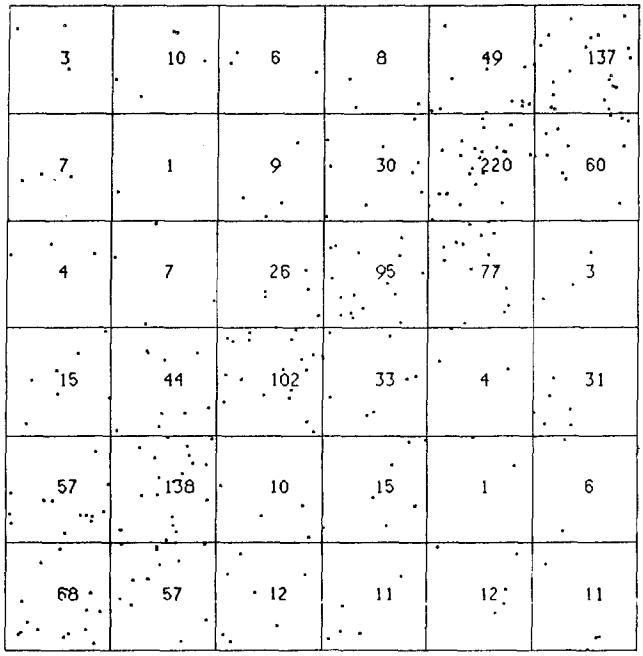

Fig. 8. As in Fig. 5, the usual decomposition produces an imbalance varying from 220 to 1.

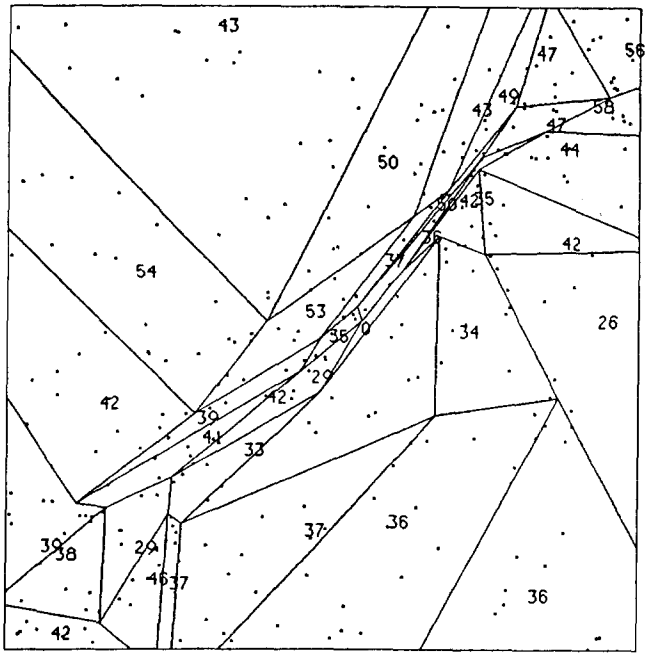

Fig. 9. The result of annealing. The maximum load has now decreased to 58, meaning that the entire computation will run at this speed. The restriction to a 6×6 connected set of quadrilaterals has constrained the annealing—a better solution could be found with a more general decomposition.

Monte Carlo algorithm with the temperature set to 0—only moves which improve the situation are accepted, all others are rejected. In terms of the energy function, this method goes only downhill and so therefore gets trapped in local minima—a phenomenon observed in the case at hand. Iterative improvement, however, can be done rapidly. Since in practice one can never reach the true optimum anyway, it may form a useful heuristic in some cases.

An irregularity of a different shape was also tried and is shown in Fig. 7. This "shock wave" example has an enhanced density of particles occurring along the diagonal of the space. The load imbalance of the naive decomposition is shown in Fig. 8. The loads vary from a high of 220 to a low of 1, corresponding to an efficiency of roughly 0.18. After annealing, the result of which is shown in Fig. 9, the processors crowd along the "shock" and change the load distribution to a max of 58 and a min of 26, or an efficiency of about 0.69.

Scattered Decomposition

We will close by describing a new decomposition technique which load-balances in a very natural way and can be understood in terms of the

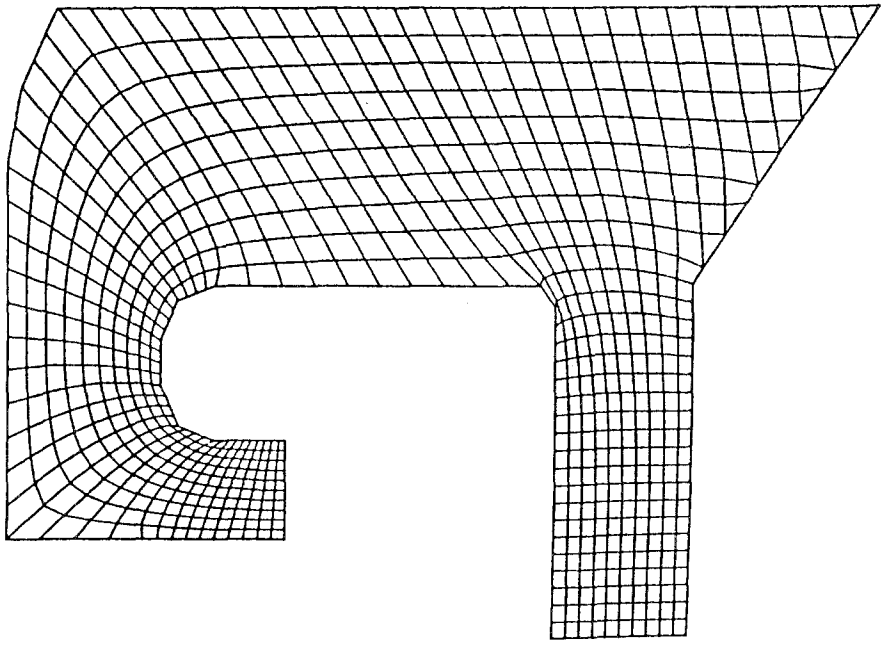

Fig. 10. The example finite element problem.

physical analogies discussed above. This decomposition is very simple to implement and seems to be effective for many types of load-balancing problems. We present the method in the context of finite element analysis of structure of a nontrivial shape (i.e., not rectangular), but it is obviously more general than this.[10]

Figure 10 shows the example computation: the finite-element analysis of a shape which doesn't map onto a hypercube or mesh-connected computer in any trivial way (that is, via a square or rectangular decomposition). Suppose, for simplicity, we wish to perform this calculation on a two-dimensional mesh of processors. The optimal decomposition, which could be found by the methods outlined previously, will look something like that shown in Fig. 11. This is all well and good, but it must be admitted that simulated annealing is a nontrivial undertaking: if we could find a simple method which gave decompositions almost as good, we would be happy. The "scattered decomposition" accomplishes this.

The decomposition is arrived at in the following way. First, take the entire problem and surround it by a large rectangle. The rectangle is subdivided into smaller rectangles, as shown in Fig. 12. Call these smaller rec-

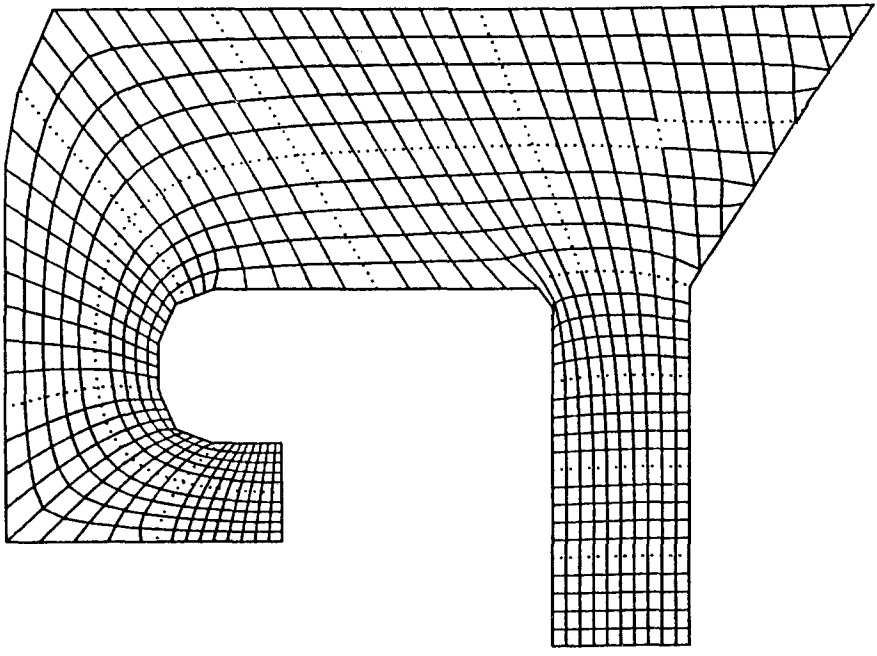

Fig. 11. A decomposition of Fig. 10 onto a 16-processor CP which is close to optimal. The dotted lines show the areas of responsibility of each processor.

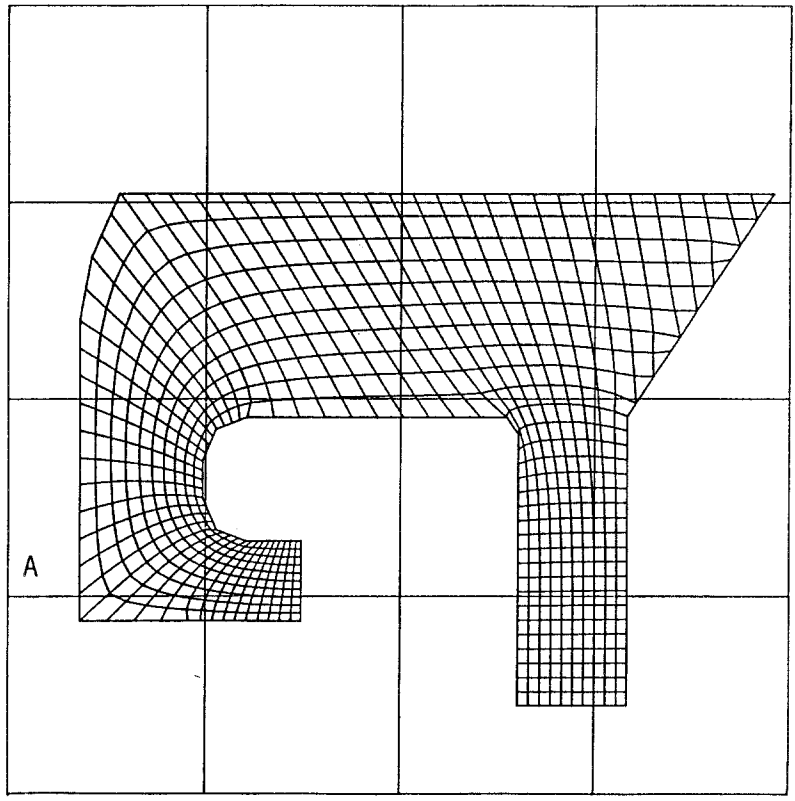

Fig. 12. The template overlay. The large squares are "templates," each of which will be decomposed onto the CP.

tangles "templates." The fundamental idea is to decompose each of the templates onto the CP by the usual square decomposition. This is illustrated for template A in Fig. 13. Where a processor region dosn't actually intersect any of the problem, a null pointer or some appropriate data structure is stored which signifies that the processor has nothing to do in this template. After each template is decomposed, the overall situation is as depicted in Fig. 14—the scattered decomposition. The point is that each processor is responsible for a scattered subset of the large rectangle; therefore, each processor will tend to have approximately the same number of intersections with the actual problem. The concurrent algorithm proceeds by cycling through the "stack" of templates, updating each according to the usual, rectangular algorithm. If the algorithm is written correctly (i.e., by not forcing resynchronization during the update of each template; for details

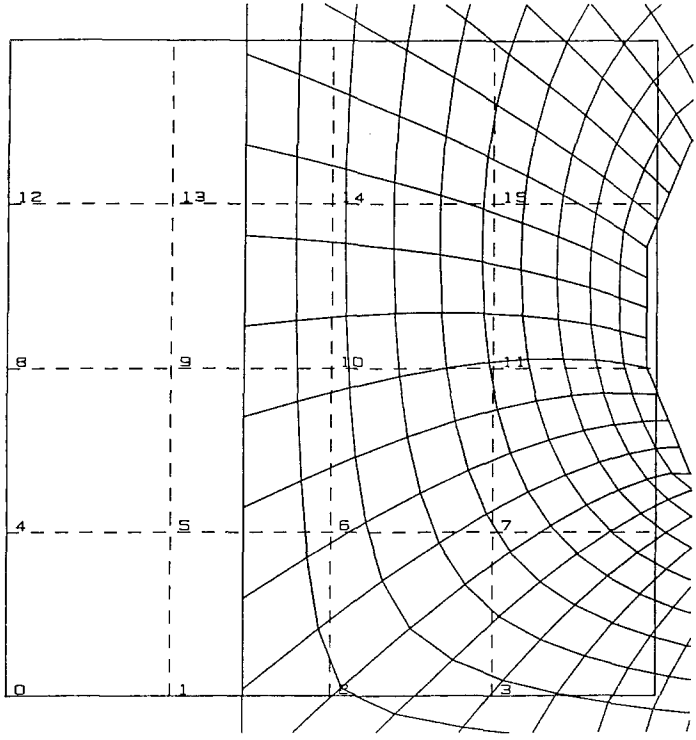

Fig. 13. A magnified view of the template marked A in Fig. 12. Each of the smaller, dotted squares is a processor of the CP: the template has been decomposed onto a 4×4 mesh of processors. The processors are responsible for the finite elements landing within their regions.

see Ref. 10), it will load-balance quite accurately for arbitrary problems! Similar ideas were used in some types of matrix algorithms.[11]

As the templates are made smaller, the load-balancing will become more accurate. The price paid, of course, is increased communication overhead. Generically, the scattered decomposition will have much more communication traffic than the optimal decomposition of Fig. 10. Often, however, communications are relatively cheap[1] and so the scattered decomposition becomes an attractive possibility. This statement is further enhanced by the fact that the communication pattern involved in the scattered case is that of the simple, two-dimensional mesh, nearest-neighbor variety. This kind of communication strategy, in contrast to general, long-distance message passing (with message forwarding), can typically be made very fast. An example is the "Crystalline Operating System" for the Caltech/JPL Hypercubes.[1]

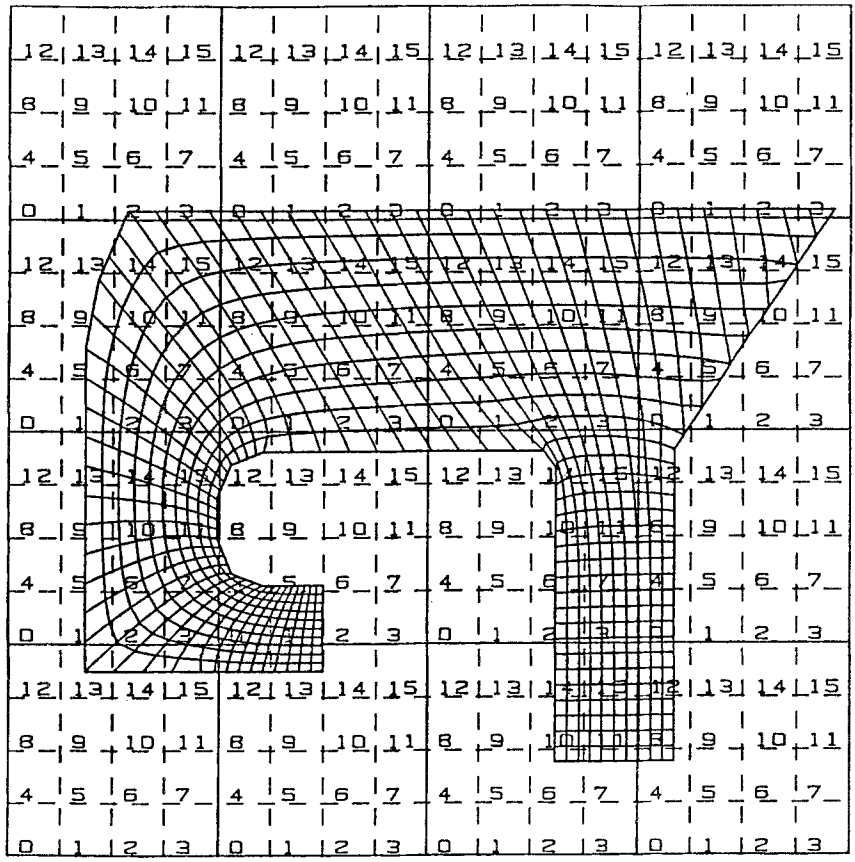

Fig. 14. The entire scattered decomposition, with processor numbers shown.

It seems fairly clear that the scattered decomposition will be a useful technique in many situations. It would be nice to relate it somehow to the physical analogies presented above since a deeper understanding would possibly result.

One of the outstanding features of the scattered decomposition is its stability. By this we mean that, as the computation changes with time (particles move, clumping occurs, etc.), the scattered decomposition is quite insensitive to these changes and will continue to load-balance rather well. Consider again the computation of Fig. 6. Suppose now that the particles move and new clumping occurs somewhere else in the physical space. If the decomposition of the space remains static, severe load imbalances will rapidly develop. Our first proposal for coping with this is to have the operating system continue to run the annealing as the computation

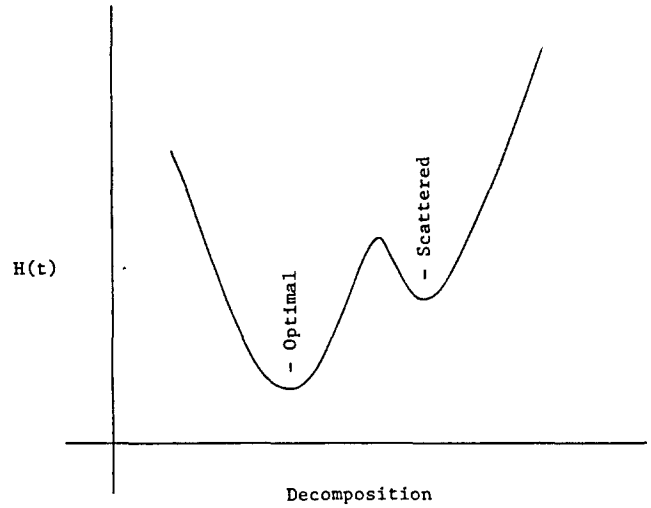

Fig. 15. A sketch of the Hamiltonian versus all possible decompositions.

progresses, and this certainly remains a viable alternative. A scattered decomposition applied to this problem will continue to load-balance for almost any pattern of clumping, however, without any annealing. Each processor "probes" all regions of the problem and so it is rather unlikely that any load imbalance will occur. We term this property stability.

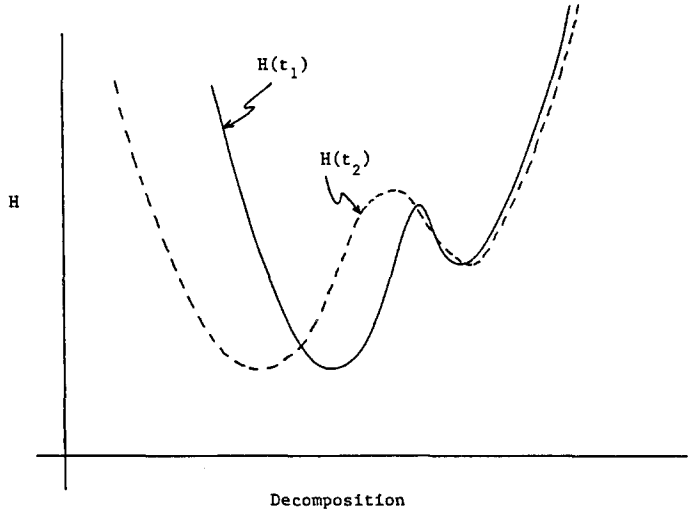

Fig. 16. The Hamiltonian at two different times. The scattered decomposition is a relatively stable minimum.

Stability can be understood in an abstract way in terms of the Hamiltonian. Figure 15 shows a schematic picture of the shape of the Hamiltonian function at some particular stage in the time evolution of the particles in Fig. 6. The horizontal axis represents the various choices of decomposition, which could be used on the problem. All of this is at time t, which is the time parameter of the particle evolution. The two decompositions, optimal and scattered, give minima, with the optimal decomposition being the global minimum (by definition). Now consider what happens to this picture as time proceeds. Something like that drawn in

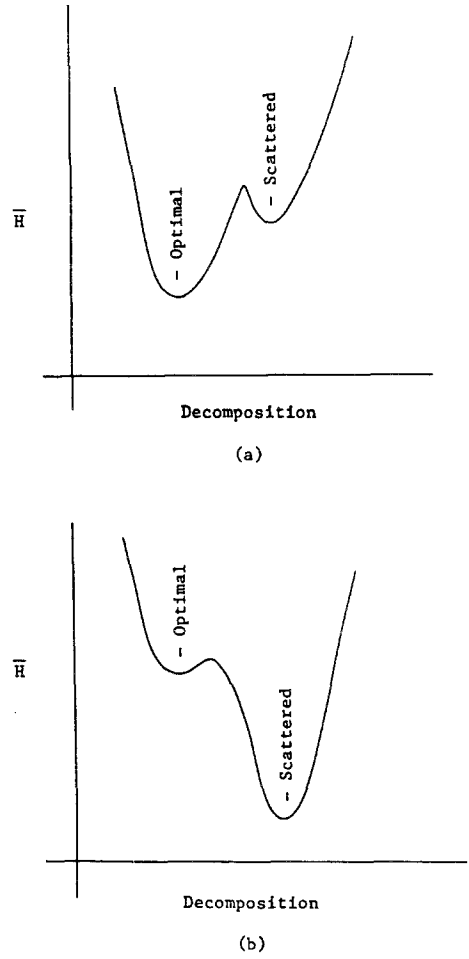

Fig. 17. The time-averaged Hamiltonian. Two scenarios are possible: the "optimal" decomposition remains the true minimum (a) or the scattered wins (b).

Fig. 16 will happen—the location of the optimal decomposition will move significantly, while the scattered minimum will move very little.

In a dynamical situation, where the characteristics of a computation are changing rapidly, the OS will not be able to "keep up" perfectly with the computation. This means that the Hamiltonian that actually matters is not the instantaneous version plotted in Figs. 15 and 16 but a time-averaged Hamiltonian $\bar{H}$

$$\bar{H}(t, T) = \int_t^{t+T} H(u)\, du$$

where the averaging time T is some natural time scale of the operating system. An interesting point is that, in terms of $\bar{H}$, the better decomposition may actually be the scattered one. Because of the rapid shifting of the optimal decomposition as a function of time, the minimum of $\bar{H}$ corresponding to this will be raised upward while the scattered minimum will remain approximately the same. As illustrated in Figs. 17 (a, b), two possible scenarios develop—the minima may or may not cross. Depending upon the parameters of the problem and upon the hardware characteristics of the CP, a "phase transition" may occur whereby the scattered decomposition actually becomes the better decomposition for $\bar{H}$.

4. CONCLUSIONS

In this paper we have demonstrated the utility of concurrent computation when the solution of otherwise intractable physics problems via the Monte Carlo method is desired. The efficient use of the MIMD machines at Caltech for problems involving interactions of arbitrary range has been achieved. It appears that problems peculiar to a parallel implementation such as satisfying the detailed balance condition, minimizing communication between processors, and balancing the computational load among all the available processors have been satisfactorily resolved. Moreover, we propose to turn the problem on its head, so to speak, and use Monte Carlo techniques to enhance the efficient operation of concurrent operating systems. For example, we showed that simulated annealing methods are a potentially powerful means of optimizing the distribution of the degrees of freedom among the processors. An analogy between a physical system consisting of particles connected by interacting rubber bands and a possible efficient operating system of a concurrent processor is remarkably appropriate. We conclude that there is a natural and useful relationship between Monte Carlo techniques and concurrent computation.

ACKNOWLEDGMENTS

We are grateful to R. Morison for his assistance in preparing many of the figures presented in this paper. We thank M. Johnson for allowing us to use results from his unpublished work on two-dimensional melting. The work on load-balancing is a collaboration with Dave Jefferson in the UCLA Computer Science Department.

REFERENCES

1. G. Fox, *Proceedings of IEEE Compucon Conference*, Feb. 28, 1984; G. Fox, *IEEE Trans. N.P.S.S.* **34** (1985); C. Seitz, *CACM*, December (1984); G. Fox, G. Lyzenga, D. Rogstad, and S. Otto, *The Caltech Concurrent Computation Program*, in Proceedings of the "1985 ASME International Computers in Engineering" Conference, August 4–8, 1985 (ASME, Boston, 1985); G. Fox and S. Otto, *Phys. Today*, May (1984).
2. E. Felten, S. Karlin, and S. Otto, *Sorting on a Hypercubic, MIMD Computer*, in C^3P Memo 92B.
3. D. E. Knuth, *The Art of Computer Programming*, vol. 2 (Addison-Wesley, Massachusetts, 1981), p. 10.
4. E. Brooks III, G. Fox, M. Johnson, S. Otto, P. Stolorz, W. Altras, E. DeBenedictis, R. Faucette, C. Seitz, and J. Stack, *Phys. Rev. Lett.* **52**:2324 (1984); S. Otto and J. Stack, *Phys. Rev. Lett.* **52**:2328 (1984).
5. M. Johnson, Ph.D. Thesis, Caltech (1986).
6. F. Fucito and S. Solomon, *Comp. Phys. Commun.* **34**:225 (1985).
7. S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, *Optimization by Simulated Annealing*, *Science* **220**:4598 (1983), pp. 671–680.
8. G. C. Fox, D. Jefferson, and S. W. Otto, *Dynamic Load Management in Distributed Systems.*
9. J. Salmon, C^3P *Memo* **78** (1984).
10. R. Morison and S. W. Otto, *Scattered Decomposition for Finite Elements.*
11. G. C. Fox, C^3P *Memo* **97** (1984).
12. M. Johnson, C^3P *Memo* **137** (1985).