The Use of Physics Concepts in Computation

Geoffrey Fox

CRPC-TR92198
February, 1992

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

THE USE OF PHYSICS CONCEPTS IN COMPUTATION

GEOFFREY C. FOX
Northeast Parallel Architectures Center
Syracuse University
111 College Place
Syracuse, New York 13244-4100, USA

ABSTRACT

We view computers and problems to be simulated on them as physical systems. We show how the concepts of dimension, topology, size, temperature, adiabatic systems and phase transitions can be valuable in general computations. A space-time picture of computation allows one to discuss the load balancing and decomposition of processes as a theory of interacting strings. The mathematics underlying these ideas allows a quantitative description of the performance of parallel machines for different problems. We also describe a theory of the architecture or overall structure of problems. This approach can be naturally applied to parallelizing compilers which can be thought as mapping temporal constructs in "sequential" languages into spatial parallelism. We consider that current languages (C, Fortran, ADA ...) as "wrong" (incomplete) as they do not properly express the spatial and temporal structure of problems.

1. Introduction

After obtaining my Ph.D. in physics in 1967, the first 14 years of my postgraduate career were centered on the study of theoretical, phenomenological and experimental physics in a fashion that usually made essential use of the computer in simulation or data analysis. Since 1981 on the other hand, I have studied the hardware, software and applications of parallel computers in a fashion that sometimes used physics as the application[1–5] but often not. However, although much of my recent work has been in computer science and in applications outside physics, I have found my training as a physicist very valuable. Physics taught me the value of confronting theory with experiment and trained me in sophisticated problem solving techniques. However in this article, I wish to review how the concepts of physics were found to be directly applicable to the theory and phenomenology of parallel computing. Several references[3,5,6–15] contain more detail on these physics analogies. We will not give a detailed discussion of the status and issues in parallel computing here—this is covered in references[3,16–20]. It will be sufficient to view parallel computing as technologically inevitable and currently divided into three classes or architectures.

1) *Distributed Memory MIMD or Multicomputers:*
—a collection of independent computers each with processor and memory, connected by some sort of network that allows messages to be sent between the individual computers.

2) *Shared Memory MIMD or Multiprocessors:*
—as above except that either logically or physically, the network connects a set of processors to a single shared memory which each may address.

3) *Distributed Memory SIMD:*
—as 1), except each processor executes an identical instruction stream.
Examples of these architectures are given in Table 1.

Table 1: Simple Classification of High Performance Concurrent Computers

Memory Structure	Grain Size	Control	Examples
Distributed	Coarse	MIMD	Hypercubes: (Ametek, FPS, Intel, NCUBE) Transputer Arrays: (CSA, Inmos, Levco, Meiko, Parsytec, Topologix, Transtech) Crossed buses: Suprenum (GMD, Germany) Routing Mesh: Intel i860 based "Touchstone", Intel iWARP, Symult, Connection Machine CM-5 Large high speed network of workstations
Distributed	Coarse	SIMD	IBM GF11 Columbia QCD Machine APE (Rome)
Distributed	Small	SIMD	AMT DAP 510 Connection Machine CM-1, CM-2 Goodyear MPP, ICL DAP, Maspar (DEC mpp)
Shared -Many Processors	Coarse	MIMD	BBN Butterfly, CEDAR IBM RP3, NYU Ultracomputer
Shared -Modest Number of Processors	Coarse	MIMD	Alliant FX-80, CRAY X-MP, CRAY Y-MP CRAY 2, Convex, ELXSI, Encore ETA-10, Flex, IBM 3090VF, Kendall Square SSI, Sequent, Silicon Graphics

We note that, just as in physics, locality is a critical issue in high performance computing. We need to ensure that the data needed for a computation is available for the arithmetic unit. Delays increase as the data is placed in memory which get slower or further array from the arithmetic unit. Locality underlies the design and use by compilers of caches in "ordinary" computers and the nature of the networks used to link the individual computer nodes in a parallel system. Matching the problem locality to the computer locality is a key to good performance.

Parallel computers are complex entities used to simulate complex problems. While physics has developed several qualitative and quantitative methods to understand large systems, other fields, in particular computer science, have not. Thus, it is not surprising that physics concepts are helpful in a theory of computation and indeed may get more important as the computers and the problems they simulate get larger and more complicated. The basic framework for this point of view is introduced in Section 2, which defines a general space-time analogy for both computers and problems. In Section 3, we discuss the spatial structure of problems with the concepts of problem size, topology and dimension. We also discuss adiabatic problems where the operating system acts as a heat bath keeping the problem at its natural temperature. As a function of temperature, we find phase transitions. In Section 4, we move to the temporal structure where dynamic problems require an underlying theory of strings, which is illustrated with examples from computing and navigation. The particle theory of Section 3 and string theory of Section 4 can be evaluated by either Monte Carlo or deterministic minimization methods for the underlying "energy function". Both Section 3 and Section 4 contain a performance analysis based on the underlying space or space-time structure of computer and problem. In Section 5, we note that both problems and computers have architectures and the relation of these enables one to understand which problems are suitable for which computers. In the final section, we apply these ideas to compilers viewed as mapping one space-time system into another.

2. Complex Systems and the Space-Time Picture

2.1 Problems and Computers

A *complex system* is a large collection of, in general, disparate members. Those members have, in general, a dynamic connection between them; a dynamic complex system evolves by a statistical or deterministic set of rules which relate the complex system at a later time to its state at an earlier time. Table 2 lists some interesting complex systems from a wide variety of fields: biology, computer science, physics, mathematics, engineering, and various aspects of the real world. Some examples are also illustrated in Figure 1. One particularly important class of complex systems is that of the *complex computer*. In the case of the hypercube, such as the NCUBE-1,2 or other multicomputers such as the Intel Paragon or Thinking Machines CM-5, the basic entity in the complex systems is a conventional computer and the connection between members is a communication channel implemented either directly in VLSI, on a PC board, or as a set of wires or optical fibers. In another well-known complex computer, the brain, the basic entity is a neuron and an extremely rich interconnection is provided by axons and dendrites.

Mapping one complex system onto another is often important. Solving a problem consists of using one complex system, the *complex processor*, to "solve" another complex system, the *complex problem.* In building a house, the complex processor is a team of masons, electricians, and plumbers, and the complex problem is the house itself. In this article, we are mainly interested in the special case where the complex processor is a complex computer and then modeling or simulating a particular complex problem involves mapping it onto the complex computer. In this case, the map of the complex problem onto the complex computer involves *decomposition.* We can consider the complex problem as an *algorithm* applied to a *data domain.* We divide the data domain into pieces which we call *grains* and place one grain in each node of the concurrent computer. This is illustrated in Figure 1 for several examples; particles interacting with a medium range force, particles interacting with nearest neighbor forces, finite difference solutions to a second order differential equation, multiplication of power series, and finally, matrix algorithms. In each case, the data domain is a set of fundamental entities which we will term *members.* The algorithm defines a possibly dynamic *interconnection* which converts the domain into a graph. As shown in Figure 1, even though each member lies in a single processor, the decomposition does not respect the interconnects and whereas some of the connections are internal, some connections are interprocessor. In Figure 1, the latter are those connections "cut" by the processor boundaries.

Let us contemplate decomposition or the mapping of a complex problem on to a multicomputer, somewhat philosophically. In the map,

Complex Problem	$\rightarrow$	Complex Computer
Members	map into	memory locations
Internal Connections	map into	arithmetic operations
Internode or "cut" connections	map into	communication followed by arithmetic operations

In Section 3, we will be considering topological properties of complex systems which correspond to the map

Complex Problem	$\rightarrow$	**Topological Structure**
Members	map into	Points in a Space
Connections	map into	Geometric (nearest neighbor) structure

In the optimal decomposition studies in Section 3 and Section 4, we will be considering dynamic properties of complex systems for which it will be useful to consider the map

Table 2: Complex Systems
Fundamental Members and Interconnection

Field	Problem	Algorithm	World	Member or Degree of Freedom	Connection or Communication
Biology	Intelligence	Unknown	Brain	Neuron	Axon, Dendrite
Computer Science	PC Board Layout	Optimization	PC Board	Chip	Trace, Wire
Physics (Cosmology)	Big Bang	Einstein's Laws	Universe	Galaxy	Gravity (Full Interconnect)
Applied Mathematics	Differential Equation	Finite Difference	R^n	$f(\underline{x}_i)$	Local Differential Operator
Social Science	Society	Unknown	Earth	Person	Conversation Roads Telephones
Construction	Building Great Wall of China	Brick Laying	Wall	Bricks	Mortar
Structural Analysis	Stress Calculation	Finite Element	Building	Nodal Points	Next to Nearest Neighbor
Condensed Matter	2D Melting	Monte Carlo	2D Solid or Liquid	Molecules	Short Range Forces
High Energy Physics	Lattice Gauge Theory	Monte Carlo	4D World $\approx$ proton	Quark and Gluon Field Values	Local Lagrangian
Granular Physics	Formation of Ripples	Time Evolution	Desert	Sand Grain	Contact
Data Analysis	Image Processing	Convolution	2D Pixel Space	Pixel	Defined by Convolution
Defense	War Games	Event Driven Simulation	Battle of Hastings	Archers Arrows Knights	Movement Launch of Weapons
Artificial Intelligence	Computer Algebra	Simplification	Expression	Variables Coefficients	Laws of Arithmetic

Complex Problem	$\longrightarrow$	Discrete Physical System
Members	map into	Particles or Strings
Connections	map into	Force Between Particles or strings

We see that different classes of complex system realize their members and interconnection in different ways. We find it very useful to map general systems into a particular class which have a particular choice for members and interconnects. To be precise, complex systems have interconnects that can be geometrical, generated by forces, electrical connection (e.g., wire), structural connection (e.g., road), biological channels or symbolic relationships defined by the laws of arithmetic. We map all these interconnects into electrical communication in the hypercube implementation. On the other hand, in the simulated annealing approach to load balancing described in Section 3.4, we map all these interconnects to forces.

2.2 Space-Time Picture

The above discussion was essentially static and although this is an important case, the full picture requires consideration of dynamics. We now "define" space and time for a general complex system.

We associate with any complex system a *data domain* or *"space"*. If the system corresponds to a real or simulated physical system then this data domain is a typically three-dimensional space. In such a simulation, the system consists of a set of objects labelled by index i and is determined by the positions $\underline{x}_i(t)$ at each time t. As shown in Figure 2, the data domain consists of a set of interconnected nodes and this forms what we call the *computational graph.* This is defined by a time slice of the full complex system.

Other complex systems have more abstract data domains:

1) In a computer chess program, the data domain or "space" is the pruned tree-like structure of possible moves.
2) In matrix problems, the data domain is either a regular two-dimensional grid for full matrices or an irregular subset of this for sparse matrices.
3) In a complex computer defined in Section 2.1, the computational graph of a multicomputer is formed by the individual nodes with the interconnection of the graph determined by the topology (architecture) of the multicomputer. We could enrich this complex system by looking with finer resolution into the computer node itself which can be considered as a set of connected components—chips or transistors depending on detail required.
4) In the sequential neural compiler considered in Section 4.3, the space of the underlying complex system consists of possible locations of variables, i.e., of the memory, cache, registers and CPU of the computer.

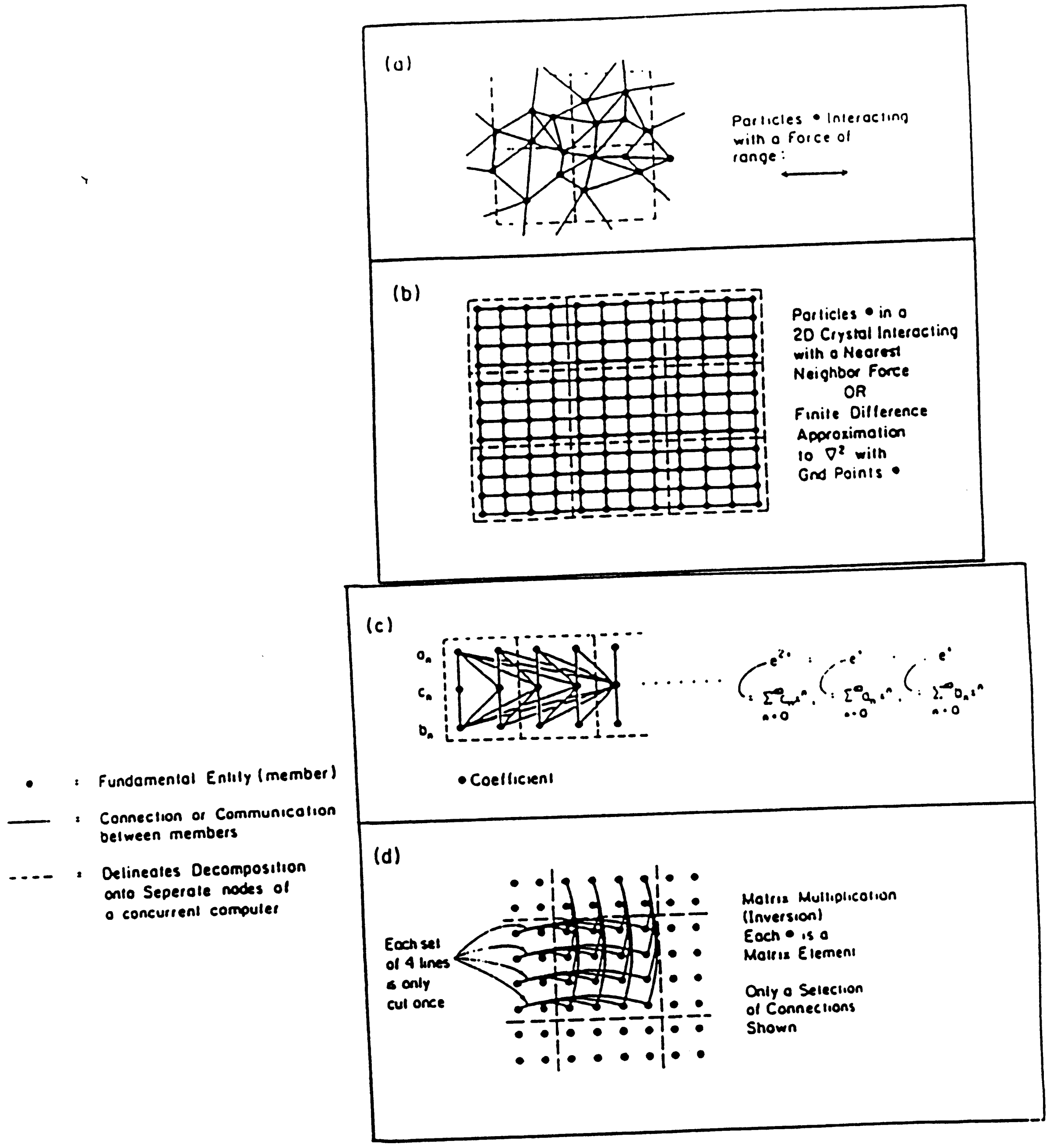

(a)
Particles • Interacting with a Force of range :
(b)
Particles • in a 2D Crystal Interacting with a Nearest Neighbor Force OR Finite Difference Approximation to ∇^2 with Grid Points •
(c)
a_n
c_n
b_n
• Coefficient
• : Fundamental Entity (member)
: Connection or Communication between members
: Delineates Decomposition onto Seperate nodes of a concurrent computer
(d)
Each set of 4 lines is only cut once
Matrix Multiplication (Inversion) Each • is a Matrix Element
Only a Selection of Connections Shown

Figure 1. Four *complex systems*: **a) particles interacting with a medium range force; b) finite difference approximation to a two-dimensional partial differential equation such as $\nabla^2\phi = 0$. This complex system is isomorphic to a two-dimensional crystal interacting with nearest neighbor forces; c) the multiplication of two polynomials; d) matrix algorithms such as multiplication and inversion. Only a sample of the interconnections for d) are shown. In reality, any element is connected with all other elements in the same row and in the same column. The dashed lines divide complex systems into grains in separate processors.**

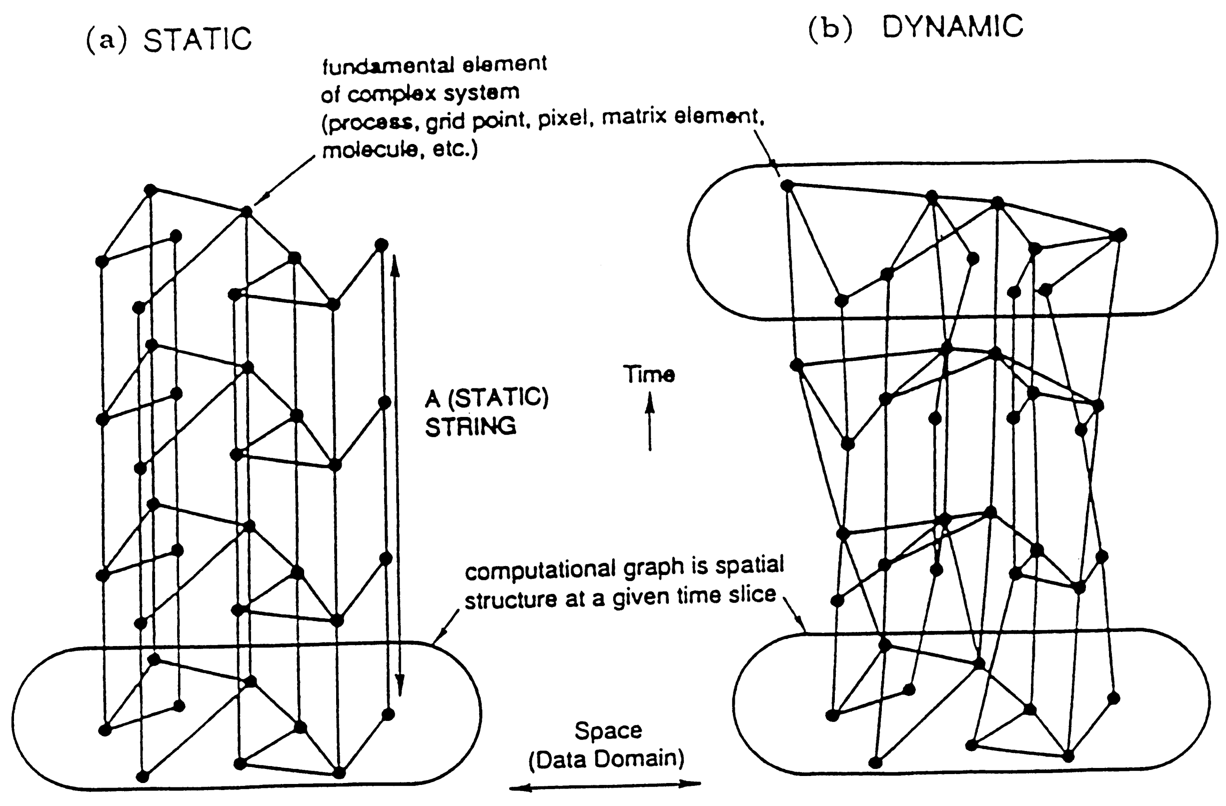

(a) STATIC
(b) DYNAMIC
fundamental element of complex system (process, grid point, pixel, matrix element, molecule, etc.)
A (STATIC) STRING
Time
computational graph is spatial structure at a given time slice
Space (Data Domain)

Figure 2. (a) Static and (b) Dynamic Complex Systems Defined in "Space"-"Time"

In a physical simulation, the complex system evolves with time and is specified by the nature of the computational graph at each time. If we are considering a statistical physics or Monte Carlo approach, then we no longer have a natural time associated with the simulation. Rather, the complex system is evolved iteratively or by Monte Carlo sweeps. We will find it useful to view this evolution or iteration label similarly to time in a simple time stepped simulation. We thus consider a general complex system defined by a data domain which is a structure given by its computational graph. This structure is extended in "time" to give the "space"-"time" cylinders shown in Figure 2. In our previous examples

1) Chess: time labels depth in tree
2) Matrix Algebra: time labels iteration count in iterative algorithms or "eliminated row" in a traditional full matrix algorithm such as Gaussian elimination.
3) The time dependence of a complex computer is just the evolution given by executed instructions. Natural SIMD machines give an essential static or synchronous time dependence whereas MIMD machines can be very dynamic. We will later discuss in Section 5, an interesting class of problems and a corresponding way of using MIMD machines, called loosely synchronous. These are microscopically dynamic or temporally irregular but become synchronous when averaged over macroscopic time intervals.

4) In the *neural* compiler, time labels clock cycles on the target computer and it labels lines of code or steps in the directed graph in the software to be mapped onto the computer.

In many areas, one is concerned with mapping one complex system into another. For instance, simulation or modeling consists of a map

$$\text{Nature (or system to be modelled)} \xrightarrow[\text{theory}]{\text{map}} \text{Idealization or Model} \qquad (2.1)$$

This map would often be followed by a computer simulation which can be broken up into several maps shown in Figure 3.

$$\begin{tikzcd}
\text{Nature} \arrow[r, "\text{Brilliant}", "\text{Idea}"'] & \text{Theory} \arrow[r] & \text{Model} \arrow[dll] \\
\text{Numerical Method} \arrow[dr, "\text{High Level Software}"] & & \\
& \text{Virtual Computer or Virtual Problem} \arrow[dr, "\text{Low Level Software}"] & \\
& & \text{Real Computer}
\end{tikzcd}$$

Figure 3. Computation and Simulation as a Series of Maps

Nature, the model, the numerical formulation, the software, and the computer are all complex systems. Typically one is interested in constructing the maps to satisfy certain goals such as agreement of model with effects seen in nature or running the computer simulation in a minimum time. In these cases, one gets a class of optimization problems associated with the complex systems. One recurring theme will be the use of simulated annealing or neural network methods to address these optimization problems. These are methods to minimize the energy function introduced in Section 3.4 as associated with the general physical system given by the space-time analogy. The energy function is the analytic form that expresses the goal described above. Typically in studying performance, the energy function would be the execution time of the problem on a computer. For software engineering, the energy function would have also reflected user productivity.

3. Spatial Structure of Problems and Computers

3.1 Performance Model of Homogeneous Multicomputers

Figure 4 shows crude diagrams for four computer architectures including the two MIMD parallel machines described in Section 1. We introduce the additional complication that most high performance sequential or parallel machines have a hierarchy of memories. We will show in Section 6 that this hierarchy corresponds to the temporal structure of computers as defined in the space-time picture of Section 2. Here we will consider the simple class typified in Figure 4(c) of homogeneous multicomputers where the "only" structure is a "spatial" distribution of nodes and associated memory.

In Figure 5, we introduce three basic parameters t_{calc}, t_{mem}, and t_{comm} which we show define a good model of the performance of the computers shown in Figure 4. t_{calc} represents the typical time to perform a floating point application including any overheads such as memory ("cache") access and indexing. t_{mem} and t_{comm} are effectively communication parameters. t_{comm} is the time taken to transmit a 32-bit (64-bit if this unit of arithmetic) word between nodes of a hypercube and t_{mem} the time taken to send a word back and forth between "cache" and main memory. As described above, we will only use t_{mem} in Section 6. We now need to make several comments and caveats on these parameters.

First, we note that these three parameters are a very incomplete description of the hardware. The analysis of Fox[8] uses the "cache" and node memory sizes as well. Here we will only need the node grain size n_{node}, abbreviated n, which is the total number of basic entities (e.g., matrix elements in a matrix problem or particles in a dynamics problem) that can be held in the node memory of a multicomputer. The performance of a system will also depend crucially on whether communication (either node to node or "cache" to memory) is concurrent or sequential with other operations.

Multicomputers, like the NCUBE-1,2, have a scalar processor and the speed t_{calc} of typical floating-point operations will not depend drastically on circumstances although, even here, factors of two variations can be expected. Pipelined or vector machines like the CM-5, Intel Paragon Multicomputer, or CRAY X-MP Supercomputer can expect very different values of t_{calc} on different applications. When in doubt, one can use the smallest possible value of t_{calc}, as this will be the pacing value for the performance analysis. The techniques described in Section 6.1 are in some sense designed to improve t_{calc} by ensuring minimal "cache" misses.

All three parameters t_{calc}, t_{comm}, and t_{mem} depend on the size of the operation performed, i.e., on the size of the vector (t_{calc}) or size of the "message" (t_{comm}, t_{mem}). We will ignore the startup time for small vectors and messages even though these are usually important. The techniques needed to minimize startup effects are interesting but are outside the scope of this overview.

In discussing multicomputers, we have listed t_{comm} as the node to node transmission time; messages between nodes that are not directly connected in the interconnection topology will be characterized by a larger value of t_{comm}. We will accommodate this by reflecting this as an application dependent effect which would be seen for

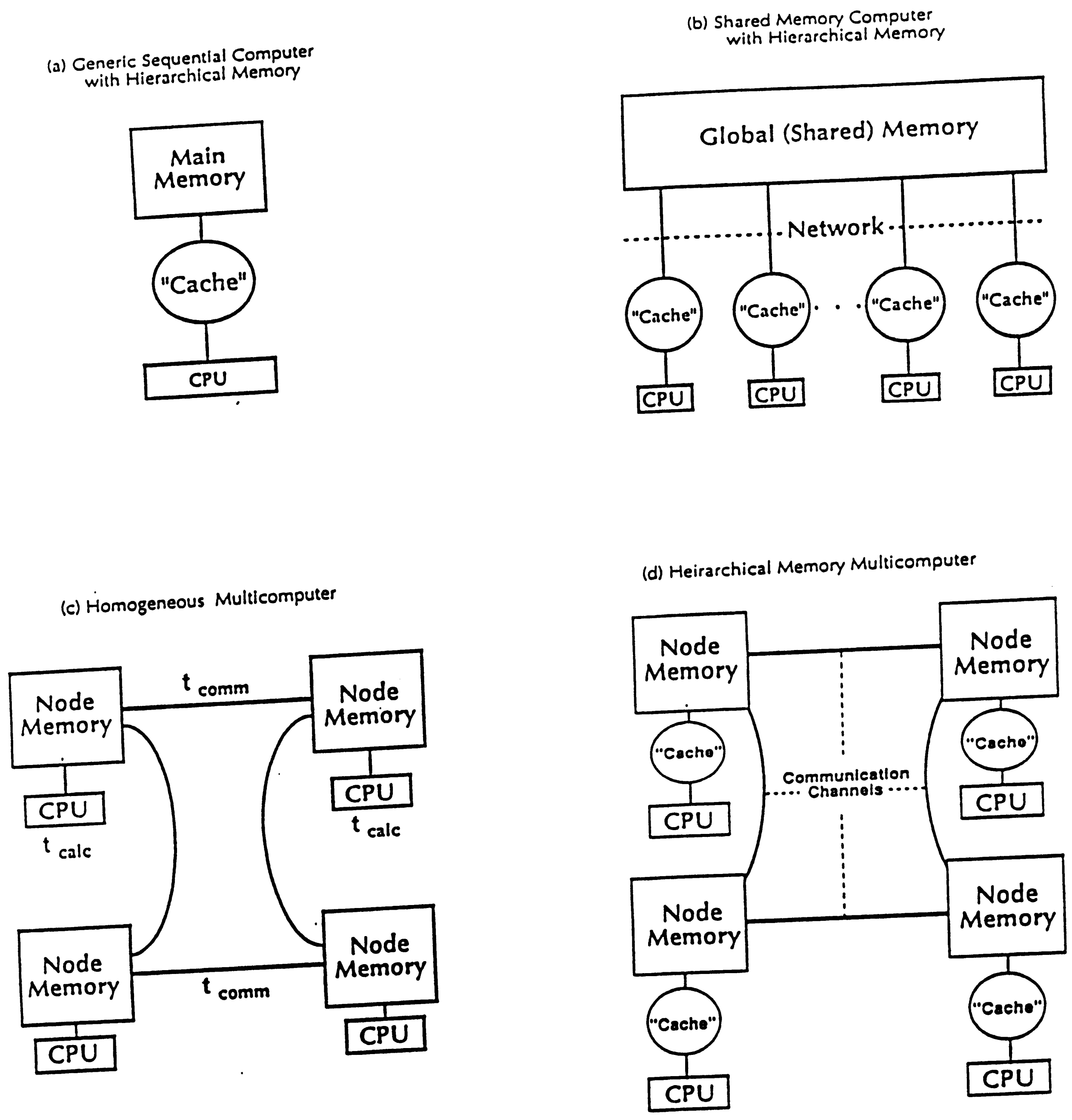


Figure 4. Block diagram of the machine architectures considered in this paper. We allow either a hardware controlled cache or user local memory to be the lowest level of the memory hierarchy. We ignore the important issues concerning the network connecting the global memory in (b) to the local "cache" and CPUs of the shared memory architecture, and the networks interconnecting the CPUs in (c) and (d).

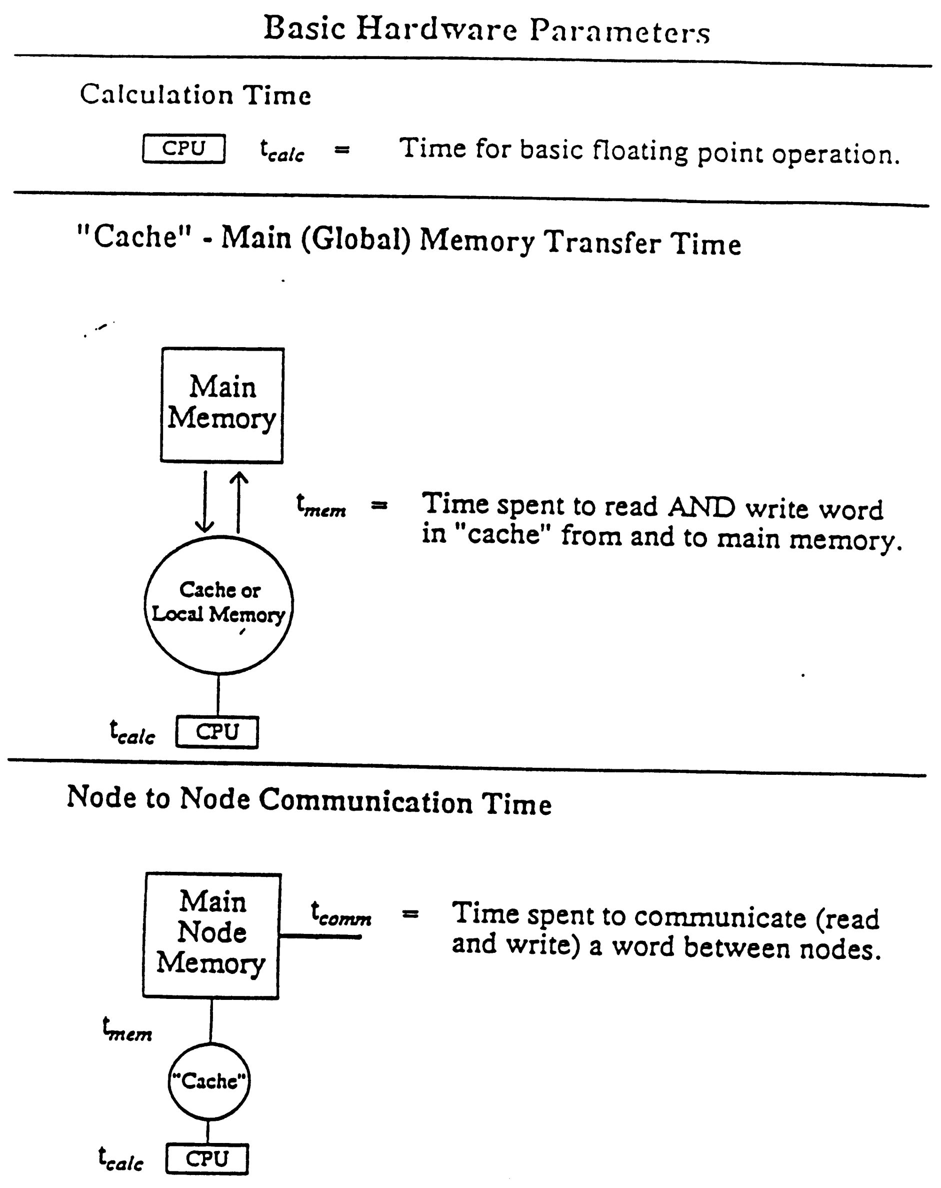

Basic Hardware Parameters
Calculation Time
CPU
t_calc = Time for basic floating point operation.
"Cache" - Main (Global) Memory Transfer Time
Main Memory
t_mem = Time spent to read AND write word in "cache" from and to main memory.
Cache or Local Memory
t_calc
CPU
Node to Node Communication Time
Main Node Memory
t_comm = Time spent to communicate (read and write) a word between nodes.
t_mem
"Cache"
t_calc
CPU

Figure 5. Definitions of the Three Hardware Parameters t_{calc}, t_{mem}, and t_{calc} discussed in Section 3.1.

algorithms like the fast Fourier transform on a hypercube as a $1/2 \log N_{\text{proc}}$ factor in the communication overhead f_c defined later in Equation (3.1). We will use N_{proc} as the number of physical processors in a parallel machine throughout this paper.

Finally, we summarize many of the caveats by noting that a performance analysis in terms of simple parameters such as t_{comm}, t_{mem}, t_{calc} is usually accurate but the simplifications imply that the parameters are not universal but need to be adjusted by different, but usually modest and understandable factors for each application[3,21,22].

We briefly introduced in Section 2, the picture of computation as a map which is shown in more detail in Figure 3. We will return to software issues in Section 6, but we can here just consider the map of a problem on a computer with the structure of Figure 4(c). Consider a very simple problem such as that shown in Figure 6 with a regular two-dimensional array of grid points to be mapped on a machine with $N_{\text{proc}} = 16$ nodes. The problem is to be divided into 16 domains or grains each with n_{grain} problem members (here members are grid points and $n_{\text{grain}} \sim 16$). Good performance uires that each processor does (roughly) equal amounts of work and also that we inimize the communication needed between nodes. We will return to the first (load balancing) issue in Section 3.4 but here we will just discuss communication which leads to an overhead f_C defined as

$$f_C = \frac{\text{amount of communication}}{\text{amount of calculation}} \tag{3.1}$$

where both calculation and communication are measured in units of time (seconds).

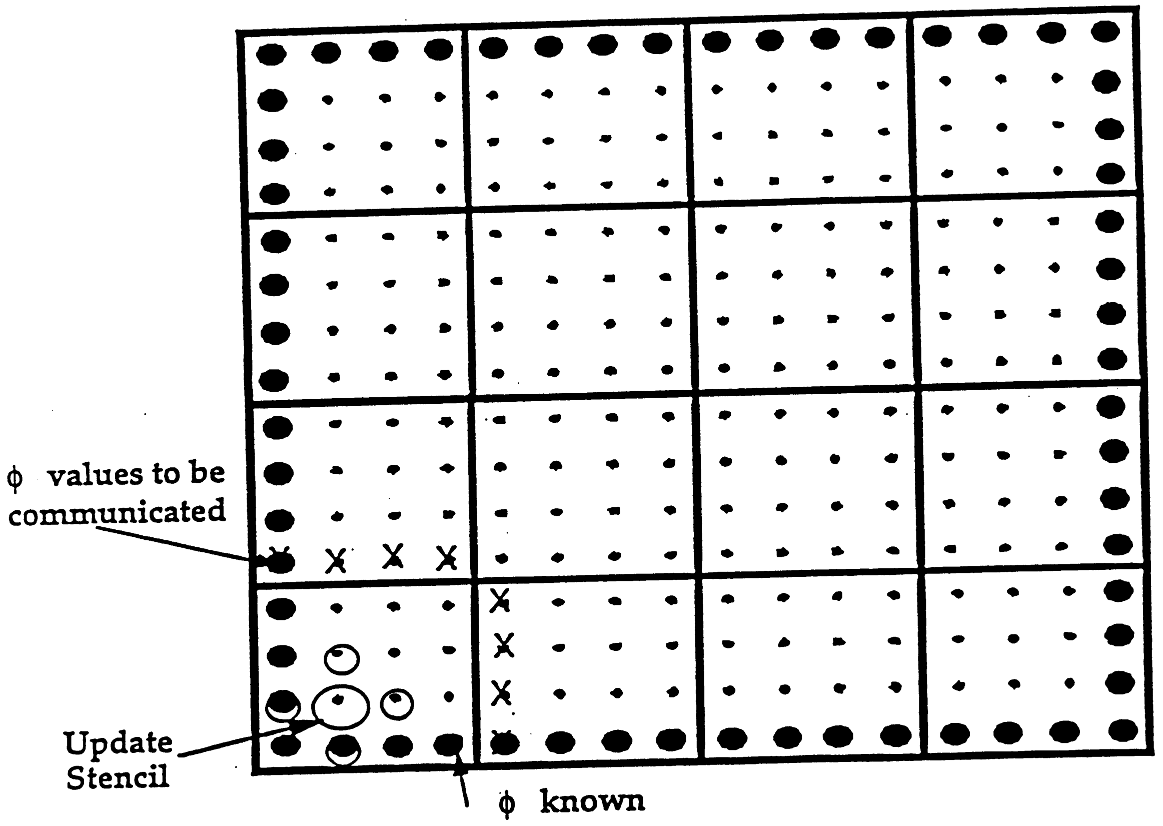


Figure 6. Domain decomposition of a simple 256 point square region onto 16 processors arranged in a two-dimensional grid. A local stencil such as that coming from a finite difference approximation to ∇^2 is shown.

In terms of f_C, one can easily show that the speedup of S on a multicomputer with N_{proc} nodes is given by

$$S = \frac{N_{\text{proc}}}{1 + f_C} \tag{3.2}$$

where the parallel machine runs a factor S faster than a single node. Onc often uses parallel efficiency ε given by

$$S = \varepsilon \; N_{\text{proc}} \tag{3.3}$$

To illustrate these concepts, we show the efficiency measured for a Quantum Chromodynamics Monte Carlo in Figure 9. This application realized 600 megaflops on the 128 node Mark IIIfp hypercube[23,24].

Returning to Figure 6, assume that this was a finite difference mesh for the solution of Laplace's equation

$$\nabla^2 \phi = 0 \tag{3.4}$$

Each processor contains $n = n_{\text{node}}$ grid points (the grain) arranged as a $\sqrt{n}$ by $\sqrt{n}$ square array. If we consider a standard iterative solution, then a typical algorithm would involve successive application of the replacement

$$\phi^{\text{new}}_{x,y} = \frac{1}{4}\left(\phi^{\text{old}}_{x-1,y} + \phi^{\text{old}}_{x+1,y} + \phi^{\text{old}}_{x,y-1} + \phi^{\text{old}}_{x,y+1}\right) \tag{3.5}$$

This is illustrated in Figure 7(a). We see that applying Equation (3.5) to all points in each node requires $4n$ arithmetic operations. The application of Equation (3.5) also requires communication for the boundary values in each node and this needs a total of $4\sqrt{n}$ numbers to be communicated. (Here, the four comes because a square has four edges.) This communication takes a total time of $4\sqrt{n}\, t_{\text{comm}}$ using the notation introduced above. Thus we see that for this example the communication overhead becomes

$$f_C = \frac{1}{\sqrt{n}} \cdot \frac{t_{\text{comm}}}{t_{\text{calc}}} \tag{3.6}$$

The result in Equation (3.6) and its generalizations have been confirmed in many applications both in our work[3] and by other groups where the Sandia analysis[25] is particularly elegant.

We note that in writing Equation (3.6), we have assumed t_{comm} has no extra factor due to routing on congested links, i.e., that the parallel computer has a two-dimensional mesh or richer interconnect. This assumption would be true for the hypercube machines.

In Figure 7(b) and Figure 7(c), we illustrate how f_C is changed as one varies the update stencil, i.e., as one changes Equation (3.5). It is important to realize that the simple nearest neighbor algorithm (Equation (3.5)) is not the optimal problem for machines with mesh interconnects. The stencil in Figure 7(b) involving higher order differences and next to nearest neighbor connections, shows an identical value of f_C to Figure 7(a). In Figure 7(c), we introduce off-diagonal terms into the stencil and find in fact that f_C is reduced. The essential point about the simple algorithm in Figure 7(a) is that not only does it have minimum communication but also minimum calculation; the overhead f_C is not especially small.

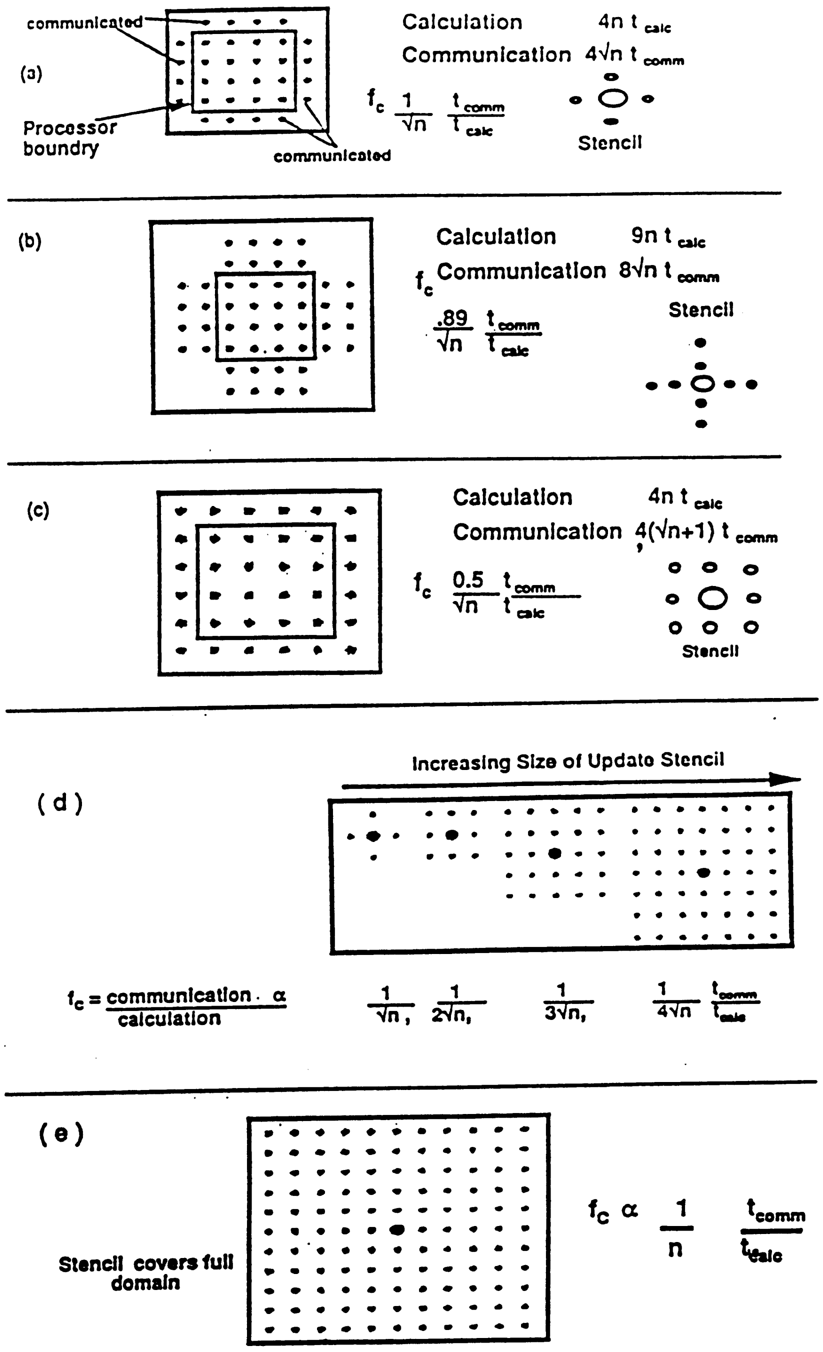

communicated
(a)
Processor boundry
communicated
Calculation 4n t_{calc}
Communication $4\sqrt{n}\, t_{comm}$
$f_c \; \frac{1}{\sqrt{n}} \; \frac{t_{comm}}{t_{calc}}$
Stencil
(b)
Calculation 9n t_{calc}
Communication $8\sqrt{n}\, t_{comm}$
$f_c \; \frac{.89}{\sqrt{n}} \; \frac{t_{comm}}{t_{calc}}$
Stencil
(c)
Calculation 4n t_{calc}
Communication $4(\sqrt{n}+1)\, t_{comm}$
$f_c \; \frac{0.5}{\sqrt{n}} \; \frac{t_{comm}}{t_{calc}}$
Stencil
(d)
Increasing Size of Update Stencil
$f_c = \frac{\text{communication}}{\text{calculation}} \cdot \alpha$
$\frac{1}{\sqrt{n}},\; \frac{1}{2\sqrt{n}},\; \frac{1}{3\sqrt{n}},\; \frac{1}{4\sqrt{n}} \; \frac{t_{comm}}{t_{calc}}$
(e)
Stencil covers full domain
$f_c \; \alpha \; \frac{1}{n} \; \frac{t_{comm}}{t_{calc}}$

Figure 7. Five stencils starting with that in Figure 6. We show an increasing "range" of stencil which would correspond to an increasing force range if a particle dynamics problem or higher order differencing for a finite different problem. We show total calculation per node and internode communication, as well as their ratio, f_C.

This point is explored further in Figure 7(d). We show square $(2l+1) \times (2l+1)$ stencils of increasing size. Such a stencil implies that

$$\phi^{\text{new}}_{x,y} = f\left(\phi^{\text{old}}_{x+i,\,y+j};\quad -l \leq i \leq l;\quad -l \leq j \leq l\right) \tag{3.7}$$

In this case, we find (for a linear function f in Equation (3.7)) that

$$f_C \sim \frac{1}{(l+1)\sqrt{n}}\,\frac{t_{\text{comm}}}{t_{\text{calc}}} \tag{3.8}$$

As we increase l, eventually the stencil covers the full two-dimensional domain. We can view the progression in Figure 7(d) (Equation (3.7) as l increases) as that coming from interacting particles with a steadily increasing range of force. (To be precise, this would lead to circular and not square stencils.) In the long range force limit—the stencil covering the full domain—shown at the bottom in Figure 7(e), we find[3] that the overhead f_C becomes

$$f_C \propto \frac{1}{n}\,\frac{t_{\text{comm}}}{t_{\text{calc}}} \tag{3.9}$$

Comparing Equation (3.8) and Equation (3.9), we find a pretty limit such that as l increases in Equation (3.8), we see that $(l+1)\sqrt{n}$ smoothly turns into the denominator n in Equation (3.9).

3.2 Information Dimension

We would like to generalize the above discussion to an arbitrary complex system for which we write[3,7]

$$f_C = \frac{\text{constant}}{n^{\frac{1}{d}}}\,\frac{t_{\text{comm}}}{t_{\text{calc}}} \tag{3.10}$$

where we introduced a fundamental static property of the complex system—its *information dimension*, d. In the first example of Figure 7(a), the decomposition dimension d was 2, and this was identical to the *geometric dimension* of the underlying *domain* being decomposed. However, in the last example of the long range force, we find $d = 1$, independent of the geometry of the domain. In Figure 7(e), we showed a two-dimensional example but one would also find $d = 1$ for the long range force in any geometric dimension. So we see that the information dimension, d is in general unequal to the geometric dimension; it also need not be an integer, and it shares this property with the fractal dimension introduced by Mandelbrot[26].

Figure 8 and Figure 9 show the measured performance of a simple matrix multiplication and QCD applications on a hypercube. Figure 8 shows[30] the overhead in the form given by Equation (3.10) with dimension $d = 2$. Figure 9[23] obscures this

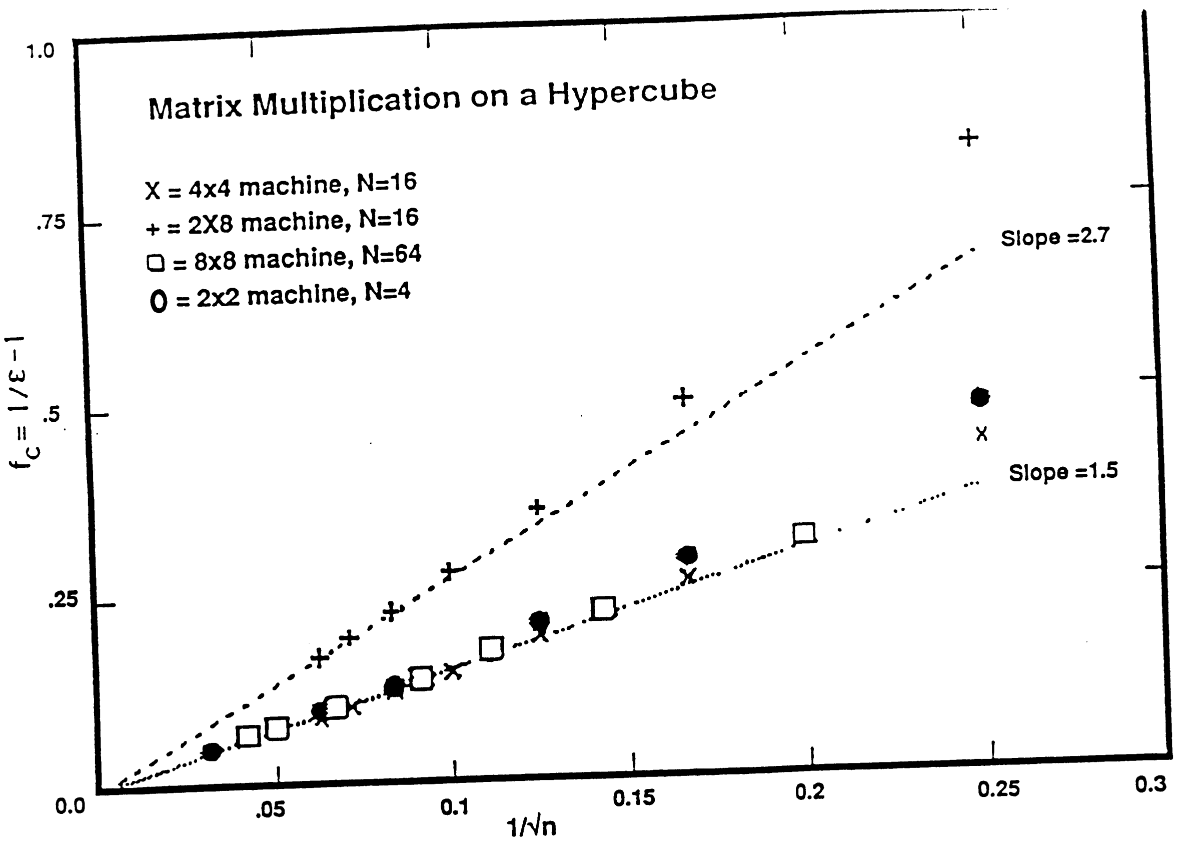


Figure 8. Matrix Multiplication on a Hypercube

analytic form but shows constant efficiency at constant grain size n, i.e., this result illustrates that Equation (3.10) depends only on n and not N_{proc}.

A good example of nontrivial information dimension comes in the simulation of electronic circuits. A well-known phenomenological rule which has been used in the packaging of circuits by IBM[27] relates the number[28,29] of output lines (pinouts) to a power ($\approx 0.5 \rightarrow 0.7$) of the number of internal components. This power is approximately independent of the size of the circuit. This immediately gives the overhead f_C for simulating such a circuit on the hypercube. The number of internal components which we call the *grain* size n again, is proportional to the calculational load in the simulation; the number of pinouts from the part of the physical circuit held in one node translates directly into the internode communication on the hypercube. Rent's Rule now takes the form (Equation (3.10)) with a non-integer value of d that is approximately three.

Rent's Rule illustrates some general features: When we simulate a physical system on a multicomputer, the communication on the multicomputer is related to physical connections between components in the physical system. These connections could be wires (for circuits), roads (for transportation) and sound waves (for human society). It is clear that the performance of a multicomputer is related to a very important property of the underlying physical system. Analogously, the proper functioning of a

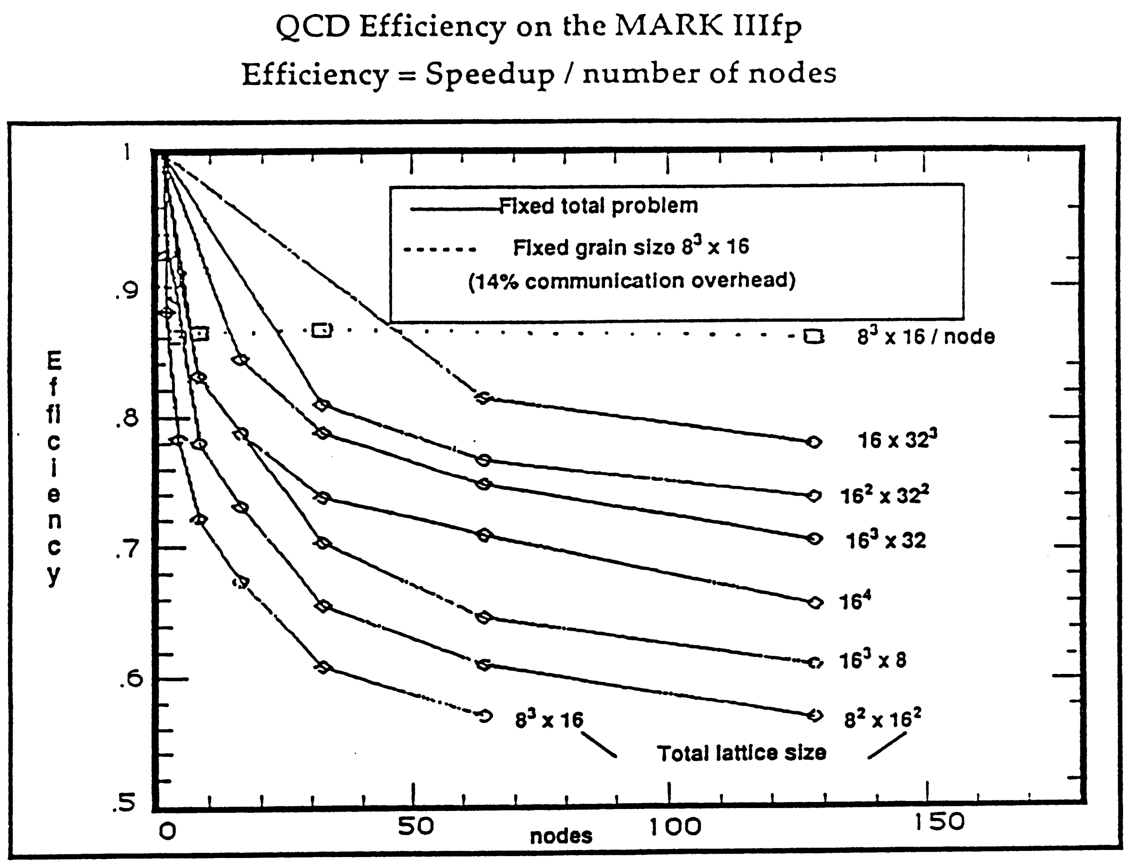


Figure 9. QCD Efficiency on the Mark IIIfp

city is dependent on an adequate density of roads to communicate between sections (grains in decomposition) within the city. One can speculate that the information dimension is a generally useful characterization of the internal communication within a physical system. One may speculate on the validity of theorems such as "Los Angeles needs a freeway system where the dimension d is greater than some critical value" or "such and such a society will function effectively if the communication between (human) members has a dimension d greater than some other critical value". This discussion is related to that in Section 2 of the mapping of complex systems on to each other.

Another interesting feature of Rent's Rule is that the decomposition dimension d for electronic circuits is approximately independent of the grain size. Such complex systems are *self similar* at different grain sizes. This point has been most emphasized by Mandelbrot[26]. In general, one must expect to find a dimension d depending on grain size, but it would be interesting to study more examples to see if self-similarity is the rule or the exception.

In the first part of this section, we showed that long range force problems exhibited a decomposition dimension that was smaller than the conventional geometric decomposition. Rent's Rule shows that electronic circuits have a decomposition dimension that is larger than the natural geometric value corresponding to a two-dimensional layout. It is interesting to find mathematical models that exhibit this effect. We have explored two possibilities. One considers members whose probability of connection is governed by a function $p(r)$ that depends on the distance r between the two members. Giving $p(r)$ a power law behavior at large r allows one to generate general fractional dimensions. Another model is shown in Figure 10. Consider[7] a two-dimensional *complex system* that consists of several types of members. For simplicity, assume that each type is only connected to other members of the same type and that this connection is of the simple nearest neighbor type shown in Figure 1(b) and Figure 7(a).

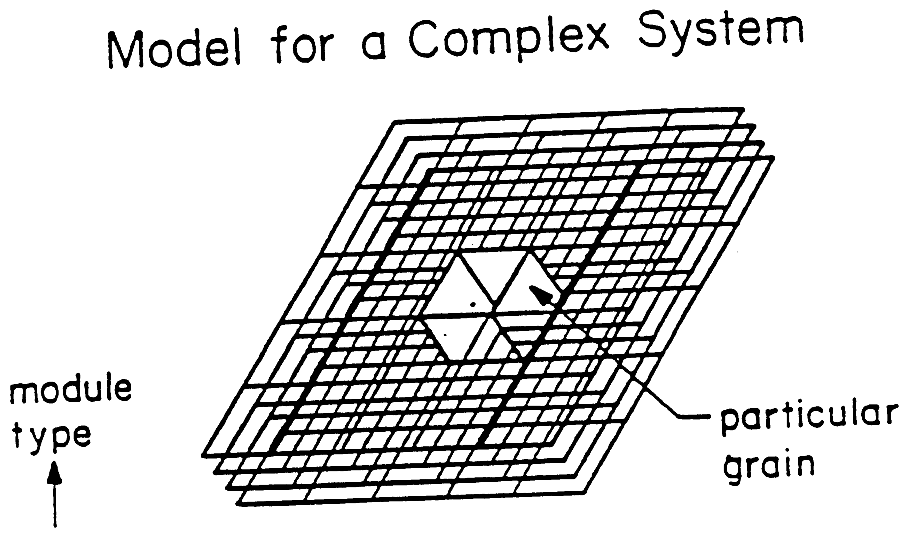


Figure 10. A Possible Model for a Fractional Dimension discussed in Section 3.

We might consider this as an oversimplified model for society; the types of members could be criminals, students, police, scientists, aristocracy, etc. The dominant communication is nearest neighbor within each type. As one increases the grain size at which the system is decomposed, the number $N_{\text{type}}(n)$ of relevant types will gradually increase. Suppose it happened that

$$N_{\text{type}}(n) \sim n^{\gamma} \tag{3.11}$$

and each type has approximately the same number of members. Then one can immediately derive a value of the decomposition dimension as

$$d = \frac{2}{1-\gamma} \tag{3.12}$$

d is always greater than two and takes the value three for $\gamma = \frac{1}{3}$. If this model had anything to do with Rent's Rule, then one could interpret the types as simple basic modules (e.g., memory or arithmetic units) from which the complicated circuit was made up.

Let us now consider another less speculative issue. If we consider the Fast Fourier Transform (FFT) on a hypercube multicomputer, then the result (Equation (3.9)) is modified[3].

$$\frac{\text{communication}}{\text{calculation}} \sim \frac{\text{constant}}{\log n} \frac{t_{\text{comm}}}{t_{\text{calc}}} \tag{3.13}$$

This can be considered as a special case of Equation (3.11) with the limit $d = \infty$; in other words the complex system corresponding to the FFT has infinite decomposition dimension. The FFT maps well onto the hypercube, and in this formalism, the good FFT map is partly due to the *complex computer* formed by the hypercube also having infinite dimension. The hypercube with N_{proc} nodes has dimension $\log_2 N_{\text{proc}}$ which does become infinite as N_{proc} itself gets infinite. To be precise, the *constant* in Equation (3.13) increases logarithmically with N_{proc}, the number of processors. In our initial formulation of dimension, we have chosen to ignore such logarithmic behavior; this point needs to be addressed in greater depth.

Another interesting *complex system* is formed by the various matrix algorithms illustrated in Figure 1(d). The members of this system are the matrix elements themselves. Each member is connected to every other member in the same row or column. There is no nearest neighbor structure at all, and yet this system has the same decomposition dimension $d = 2$ as the simple system of Figure 1(b) and Figure 7(a). This can easily be seen if we consider, as in Figure 1(d), each grain as a $\sqrt{n}$ by $\sqrt{n}$ array. Then every member communicated to this grain (node) is involved in $\sqrt{n}$ calculations corresponding to the $\sqrt{n}$ members in the grain that are associated with the row or column of the transmitted member. Thus we can directly show that the ratio

$$\frac{\text{calculation}}{\text{communication}} \sim \sqrt{n} \frac{t_{\text{calc}}}{t_{\text{comm}}} \tag{3.14}$$

which corresponds to an information dimension of two.

Let us view the dimension as a property of the underlying graph which consists of connected members as pictured in Figure 1. Then we can derive the equations:

$$\begin{aligned} \text{calculation} &= \text{Number of connections} \\ &\quad \text{made to members inside the grain.} && (3.15a) \\ \text{communication} &= \text{Number of connections} \\ &\quad \text{cut by grain boundaries.} && (3.15b) \end{aligned}$$

In Equation (3.15a), each connection is counted with no restriction. However, in Equation (3.15b), connections from a given external member are only counted once, even if they land on distinct internal members. This restriction is crucial; without it both the long range force and matrix problems would have infinite dimension rather than the values of one or two described above. In multicomputer implementations, the restriction implies that even if one needs a given external member in more than one internal calculation, one only communicates it once. This is clearly possible with careful software. A hardware cache is designed to automate this type of reuse of variables—usually when the "communication" is between different levels of a memory hierarchy as in the architecture in Figure 4(a). However, as reviewed in Section 6.1 and described in detail in Fox[8], communication between memory hierarchies has many analogies with communication between processors. Current parallel computing methodologies implement the "caching function" with software. Compilers for many sequential RISC computers handle caching with software and as mentioned in Section 6.2, this approach is now being applied successfully to parallel computers. The hardware solution to "parallel caches" is difficult to implement for large N_{proc} as one needs to ensure the caches are "coherent", i.e., are properly updated when the base variables are changed—perhaps on a processor "far" from caches in which it is stored.

Our current implementations suggest an interesting relation between f_C and the dimensions d_c and d_p of the *complex computer* and *complex problem*, respectively. In matrix multiplication[30], corresponding to the experimental measurements shown in Figure 8, one finds

$$f_C = \frac{1}{\sqrt{n}} \frac{t_{\text{comm}}}{t_{\text{calc}}} \tag{3.16a}$$

for a two-dimensional decomposition $d_c = d_p = 2$. On the other hand, if we decompose the matrix on to a ring with $d_c = 1$ by storing complete rows and columns in each node, then we get

$$f_C = \frac{\sqrt{N_{\text{proc}}}}{2} \cdot \frac{1}{\sqrt{n}} \frac{t_{\text{comm}}}{t_{\text{calc}}} \tag{3.16b}$$

Equation (3.16b) corresponds to $d_c = 1$ and $d_p = 2$. We have used as before N_{proc} to denote the number of nodes in the concurrent computer.

We can generalize these results as follows: if $d_c \geq d_p$, then

$$f_C = \frac{\text{constant}}{n^{\frac{1}{d_p}}} \frac{t_{\text{comm}}}{t_{\text{calc}}} \tag{3.17a}$$

while if $d_c < d_p$, then

$$f_C = N_{\text{proc}}^{\left(\frac{1}{d_c} - \frac{1}{d_p}\right)} \frac{\text{constant}}{n^{\frac{1}{d_p}}} \frac{t_{\text{comm}}}{t_{\text{calc}}} \tag{3.17b}$$

The N_{proc} dependence in Equation (3.17b) implies that cases with $d_c < d_p$ (dimension of computer less than that of problem) will not scale properly (speed up S proportional to N_{proc} for fixed n) as one increases the number of nodes. Equation (3.17a) corresponds to scaling behavior we have found on multicomputers. Equation (3.17) quantifies the reasons to prefer an architecture like the hypercube with a high dimension. We have only demonstrated Equation (3.17) in simple examples (matrices, partial differential equations) and need to explore its validity in more general problems.

In this section, we have concentrated on the dimension of problems and the computer as it enables one to quantify the performance of a particular decomposition. Closely related to this is the topology of computer interconnection which has been discussed in great detail in the computer architecture literature[3,21,22].

3.3 Physical Analogy

In the previous two subsections, we described static spatial properties of complex systems which were relevant for computation. These included size, topology (geometric dimension) and the information dimension. We will find new ideas when we consider problems that are spatially irregular and perhaps vary slowly with time. A simple example would be a large scale astrophysical simulation as shown in Figure 11 where the use of a parallel computer required that the universe be divided into domains which due to the gravitational interactions will change as the simulation evolves.

Load Balancing can affect crucially the performance of a computation executing on a parallel machine. 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-non-overlapping work) is considered perfectly balanced. This balance is often not trivial to achieve, however. The problem of distributing a computation in an efficient manner into a parallel machine can be fruitfully attacked via simulated annealing and other physical optimization methods[12,13,34–37].

As described in the previous section, a key to parallel computing is to split the underlying spatial domain into grains which each correspond to a process as far as the operating system is concerned. We will take a naive software model where there is one process associated with each of the fundamental members of the simulated system, i.e., with each "particle" in Figure 11 or each grid point in Figure 6. This is not practical with current software systems as it gives high context switching and other overheads. However, it captures the essential issues.

The 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 parallel computer, but can also occur within a single computation. It is the task of the operating system to manage this set of processes, moving them around if necessary, so that the parallel computer is used in an efficient manner.

The operating system performs two primary tasks. First, it must monitor the ongoing computation so as to detect bottlenecks, idling processors and so on. Secondly, 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

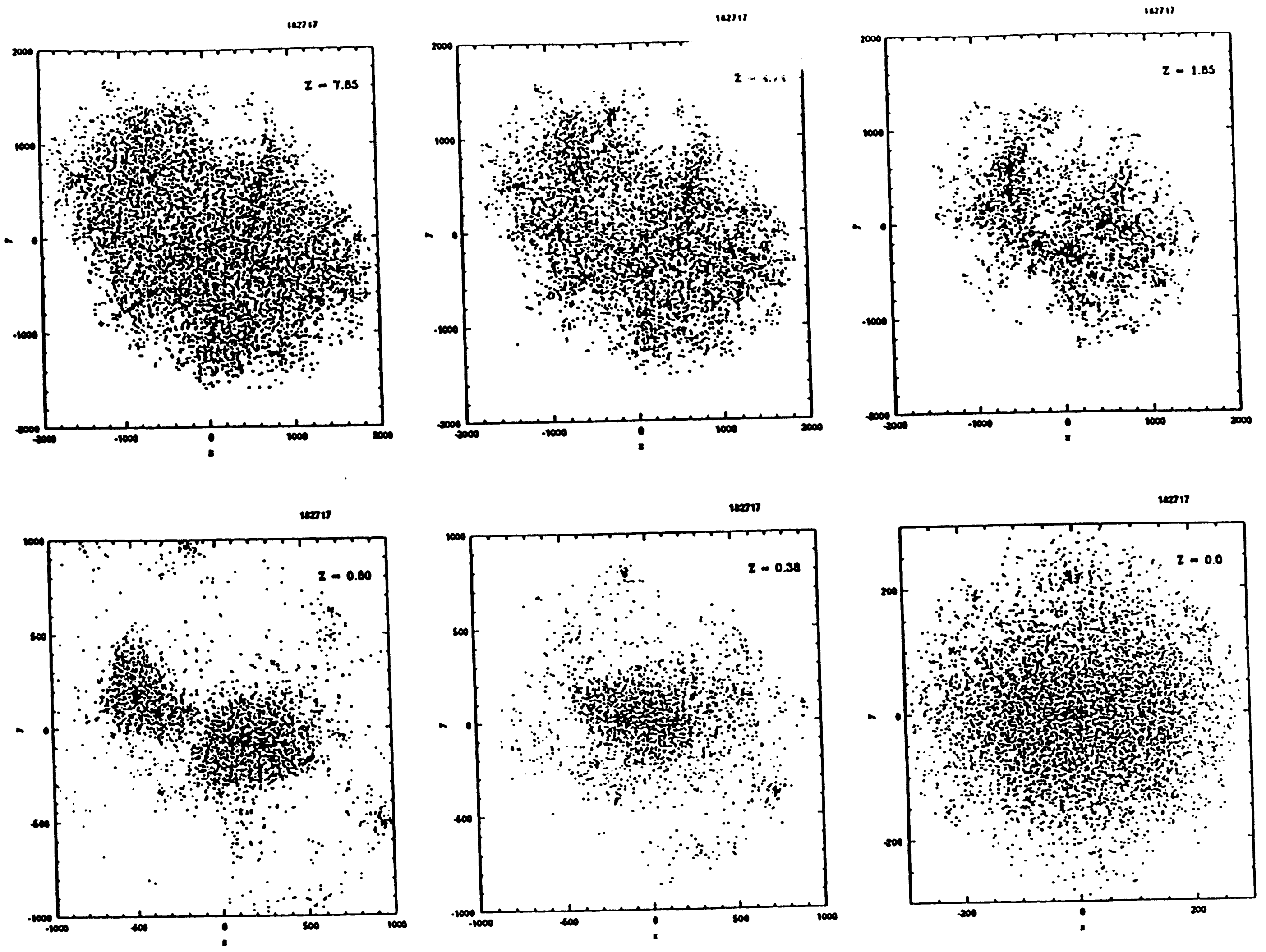


Figure 11. Results of a hypercube simulation of "evolution of universe"[31] on a 64-node JPL Mark IIIfp showing a cluster of 20,000 particles that eventually form a galaxy. This simulation uses a parallel[32] Barnes-Hut[33] algorithm using dynamic redistribution of particles between nodes of the parallel machine at each time step.

realize a reasonable fraction (let's say 80%) of the potential computing power of the parallel machine for a wide variety of computations. We will see in what follows that the operating system functions as a "heat bath", keeping the computation "cool" and therefore near its "ground state" (optimal solution).

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 parallel machine. The requirement of load balancing acts as a short range, repulsive "force", causing the particles, and thereby the computation, to spread throughout the parallel computer in an evenhanded, balanced manner. 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 sufficiently long so as to do useful work. A better analogy, therefore, is that of particles interacting via a repulsive force with the 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 separating 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 which 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.

We have described qualitatively above a "Hamiltonian" for parallel computation which is illustrated quantitatively in the next section, the operating system must try to minimize, and if possible to find the ground state. We already noted that exact minimization is not necessary—we have already "wasted" some computational power using convenient high level languages—we can surely afford to lose another 20% to load imbalance, so we can think of the operating system as a heat bath which keeps the computation as cool as possible. Most scientific simulations change slowly with time and redistribution of processes by the operating system can be gradual. Thus, we can think of the computation as being in adiabatic equilibrium at a temperature T_{problem} which reflects the ease of finding a reasonable minimum. T_{problem} will be larger for those problems which change more rapidly and where the operating system does not have "time" to find as good an equilibrium.

3.4 Load Balancing

We can now make the above discussion precise with a specific example[38]. We can view the spatial structure of the problem as a graph (computational graph in Figure 2) for which the load balancing or decomposition for parallel computers becomes a min-cut and equal weight graph partitioning problem. Consider for definiteness the finite element problem shown in Figure 12 where the strains in a plate require the unequal distribution of elements indicated in the figure. Suppose we wish to simulate this system on a 16 node parallel computer with a two-dimensional mesh (or more generally hypercube) topology. If the elements had been uniformly distributed, then the equal area decomposition shown in Figure 13(a) would be appropriate. However, although this would give modest and local node-to-node communication, it does lead to severe load imbalance shown in Figure 13(b). We need to distribute the elements so that we minimize the appropriate sum of communication and calculation. The relative weights of these two terms depends on the characteristics of the target hardware.

Formally, one needs to minimize

$$\max_{\text{nodes } i} C_i \tag{3.18}$$

where C_i is the total computation time for calculation and communication. We choose to replace this mini-max problem by a least squares[35] minimization of

$$E = \sum_i C_i^2 \tag{3.19}$$

Suppose $m(m')$ label the nodal points of the computational graph. Then

$$C_i = \sum_{m \epsilon i} \left[\sum_{\substack{m' \\ \text{linked} \\ \text{to } m}} Comm\,(m, m') + Calc\,(m) \right] \tag{3.20}$$

where it takes time $Calc\,(m)$ to simulate m and time $Comm\,(m', m)$ to communicate necessary information from m' to m. If we consider the case where we can neglect the quadratic communication terms, then

$$C_i^2 \approx \text{const.} \sum_{\substack{m,\, m' \\ \text{for } m \text{ in } i \\ \text{and } m' \text{ linked} \\ \text{to } m}} Comm\,(m, m') \tag{3.21a}$$

$$+ \sum_{\substack{m,\, m' \\ m,\, m' \\ \text{in } i}} Calc(m)\; Calc(m') \tag{3.21b}$$

The two terms in Equation (3.21) have a straightforward interpretation with the physics analogy where m, m' are particles which interact, as described in Section 3.3, if they are linked in the computational graph. We consider these particles as moving in a space formed by the nodes and linkage of the parallel computer. Then Equation (3.21a) is an attractive long range force which is minimized when m and m' are

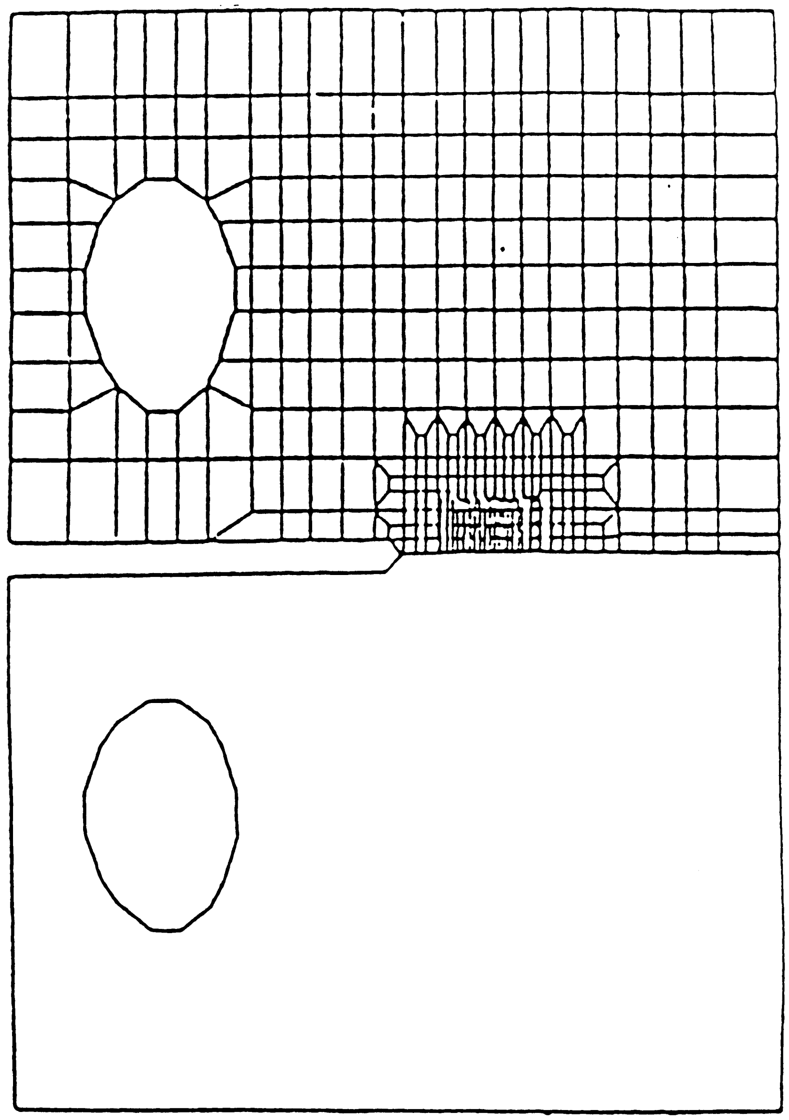


Figure 12. A Finite Element Mesh Generated by the Structural Analysis System NASTRAN. The mesh is symmetric about the horizontal division and only the top half is shown in detail.

in the same node. It is a function of the distance in the "computer space" between m and m'; this function depends on the hardware topology and the nature of message routing on the computer. Equation (3.21b) represents a repulsive short range potential which is maximized and only nonzero when m and m' are in the same node i. It is the load balancing repulsion described in Section 3.3.

Typical results for the problem of Figure 12 and Figure 13 are shown in Figure 14. We have succeeded in dividing the mesh into roughly equal collections of nodal points; moreover each set is approximately square so as to minimize edge the communication effects which are proportional to perimeter (edge) of domain stored in each processor.

Decomposition onto a hypercube naturally suggests a neural network formalism

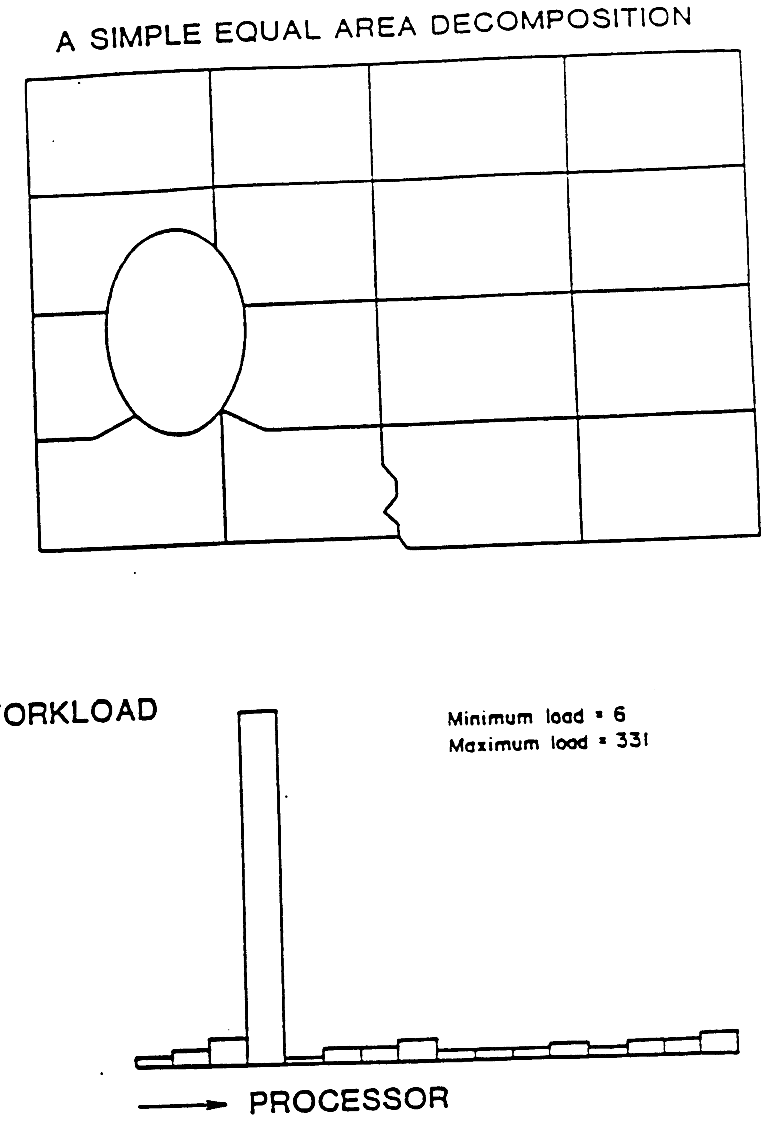


Figure 13. (a) A simple equal area decomposition of the mesh shown in Figure 12 for a 16 node machine; (b) The unequal workload corresponding to distribution shown in Figure 13(a).

which is in fact quite general[13]. Let point m reside in processor $P(m)$ and let this processor be labelled by a binary number with $\sigma_\alpha(P(m)) = \sigma_\alpha(m)$ being the α'th bit of this processor label. If we have $N_{\text{proc}} = 2^d$ processors, then we associate d neural variables $\sigma_\alpha(m)$ with each point m. As described[36], it is straightforward to express C_i in terms of $\sigma_\alpha(m)$ and write down the Hopfield and Tank minimization formulae for Equation (3.19)[39]. Using multiscale techniques, one can show that it takes time of order $M \log M \log N_{\text{proc}}$ to minimize Equation (3.19) with a neural network method on a system of size M $(M n N_{\text{proc}})$. This is typically much faster than simulation time which is at least of order a large constant times M for a system that has an essentially constant computational graph for a large time interval—the definition of

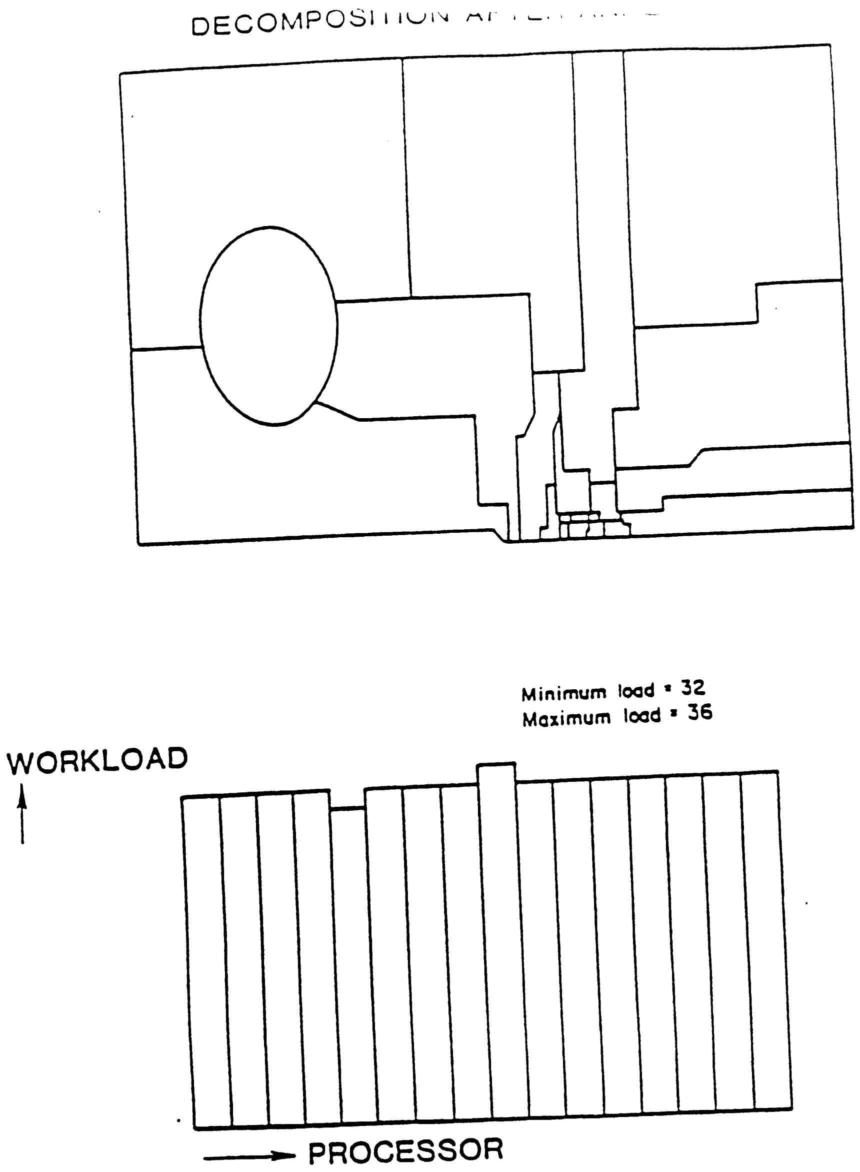


Figure 14. (a) The "optimal" decomposition found in Flower, et al.[38] using simulated annealing and the formalism described in Section 3; (b) The approximately equal workload for the decomposition of Figure 14(a).

an adiabatic system. Both simulation and decomposition performance can be speeded up by a factor of order N_{proc} on the parallel computer. Koller has implemented a load balancer for the iPSC/1 hypercube where simulation and balancer run as separate tasks on each node of the hypercube[40].

Originally, we had expected decomposition to be difficult but we now believe it to be quite straightforward to obtain adequate although non-optimal decompositions. As well as simulated annealing and neural networks, there are a variety of simple ad-hoc but satisfactory heuristic methods[41–48]. We have used these methods routinely for both finite element and particle dynamics problems[38,49–52].

Simulated annealing directly realizes the physical analogy introduced in Section 3.3; neural networks can be viewed[13] as the mean field approximation applied to the underlying Hamiltonian of the physical system. Genetic methods seem to give similar results to annealing and here we don't directly exploit the particle analogy but rather view load balancing "just" as an optimization problem[53,54].

3.5 Dynamic Load Balancing

The problem in Section 3.4 was posed "statically": we needed to find the equilibrium state of the underlying physical analogy and this decomposition of nodal points onto processors suffices for the full calculation. In such problems, adaptive meshes are often needed where the center of attention, such as the crack in Figure 12, and then one needs to change the distribution to keep the physical problem in its adiabatic equilibrium. The example in Figure 11 would also need changing distributions.

We will illustrate some of the issues that can arise with the irregular finite element mesh shown in Figure 15, and we will use this to describe the simple scattered decomposition which is one of the effective heuristics discussed at the end of Section 3.4. 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 Figure 16. This is all well and good, but it must be admitted that simulated annealing is a non-trivial undertaking: if we could find a simple method which gave decompositions almost as good, we would be happy. The "scattered decomposition" accomplishes this[6,55].

The scattered 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 Figure 17. Call these smaller rectangles "templates". The fundamental idea is to decompose each of the templates onto the parallel computer by the usual square equal-area decomposition. This is illustrated for template "A" in Figure 18. Where a processor region doesn'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 Figure 19—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 re-synchronization during the update of each template[55], it will load balance quite accurately for arbitrary problems!

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 Figure 16. Often, however, communication is relatively cheap and

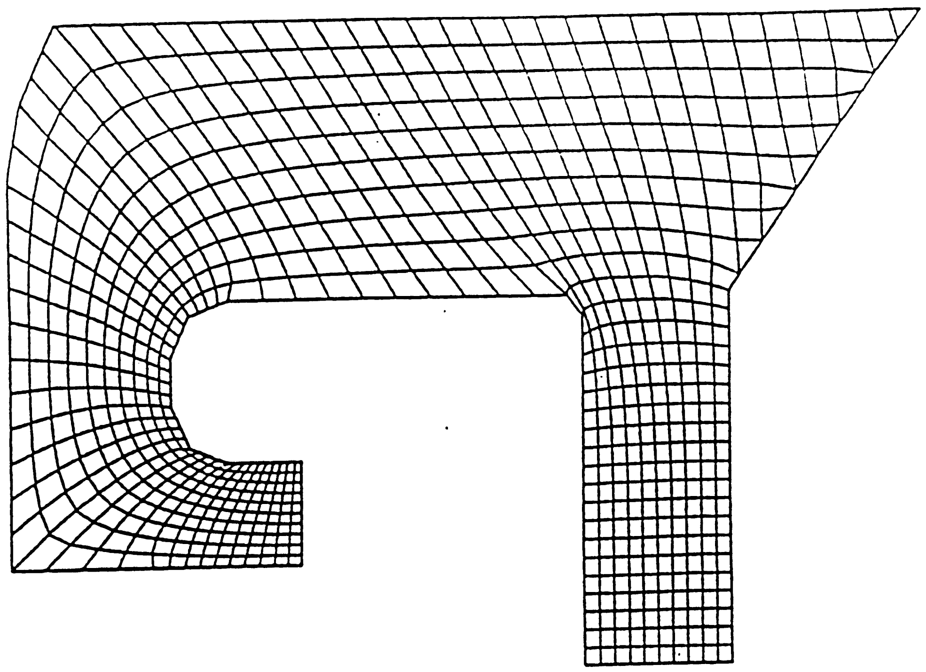


Figure 15. The Example Finite Element Problem

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 grid communication (NEWS) systems on the CM-2 which is much faster than the general router.

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 as a deeper understanding would possibly result.

One of the outstanding features of the scattered decomposition is it's "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 Figure 12. Suppose now that the crack moves and new clumping of finite element nodal points occurs somewhere else in the plate. 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 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

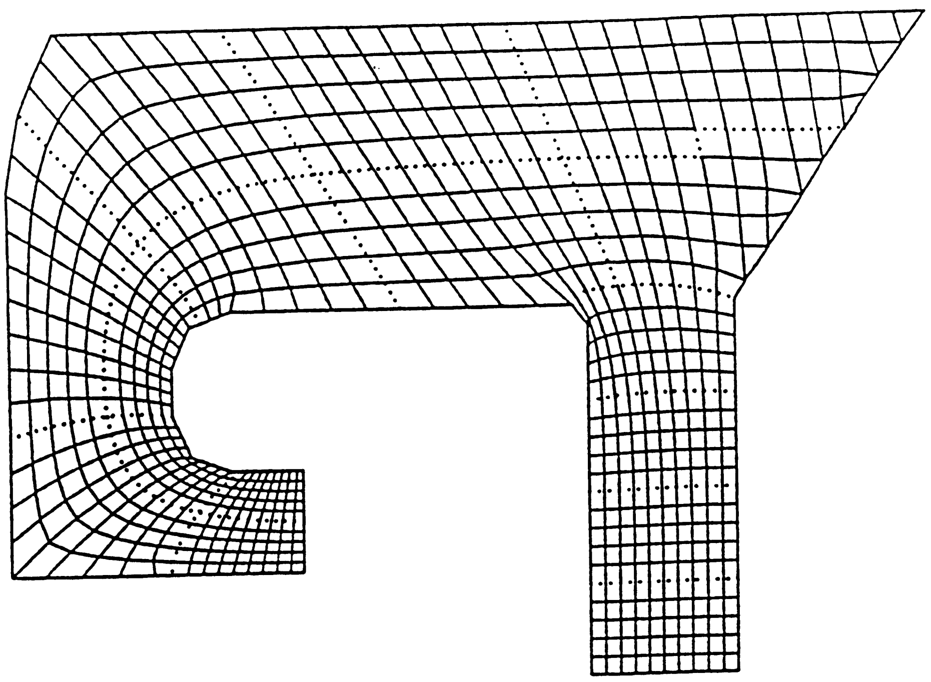


Figure 16. A decomposition of Figure 15 onto a 16 processor parallel computer which is close to optimal. The dotted lines show the areas of responsibility of each processor.

even when the problem changes. We term this property "stability".

Stability can be understood in an abstract way in terms of the Hamiltonian. Figure 20 shows a schematic picture of the shape of the Hamiltonian function at some particular stage in the time evolution of the crack in Figure 12. 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 crack 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 Figure 21 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 operating system will not be able to "keep up" perfectly with the computation. This means that the Hamiltonian which actually matters is not the instantaneous version plotted in Figure 20 and Figure 21, but a time averaged Hamiltonian, $\overline{H}$:

$$\overline{H}\,(t,\,t_{\mathrm{av}}) = \int_t^{t+t_{\mathrm{av}}} H(u)\,du \tag{3.22}$$

where the averaging time, t_{av} is some natural time scale of the operating system. An interesting point is that, in terms of $\overline{H}$, the better decomposition may actually be

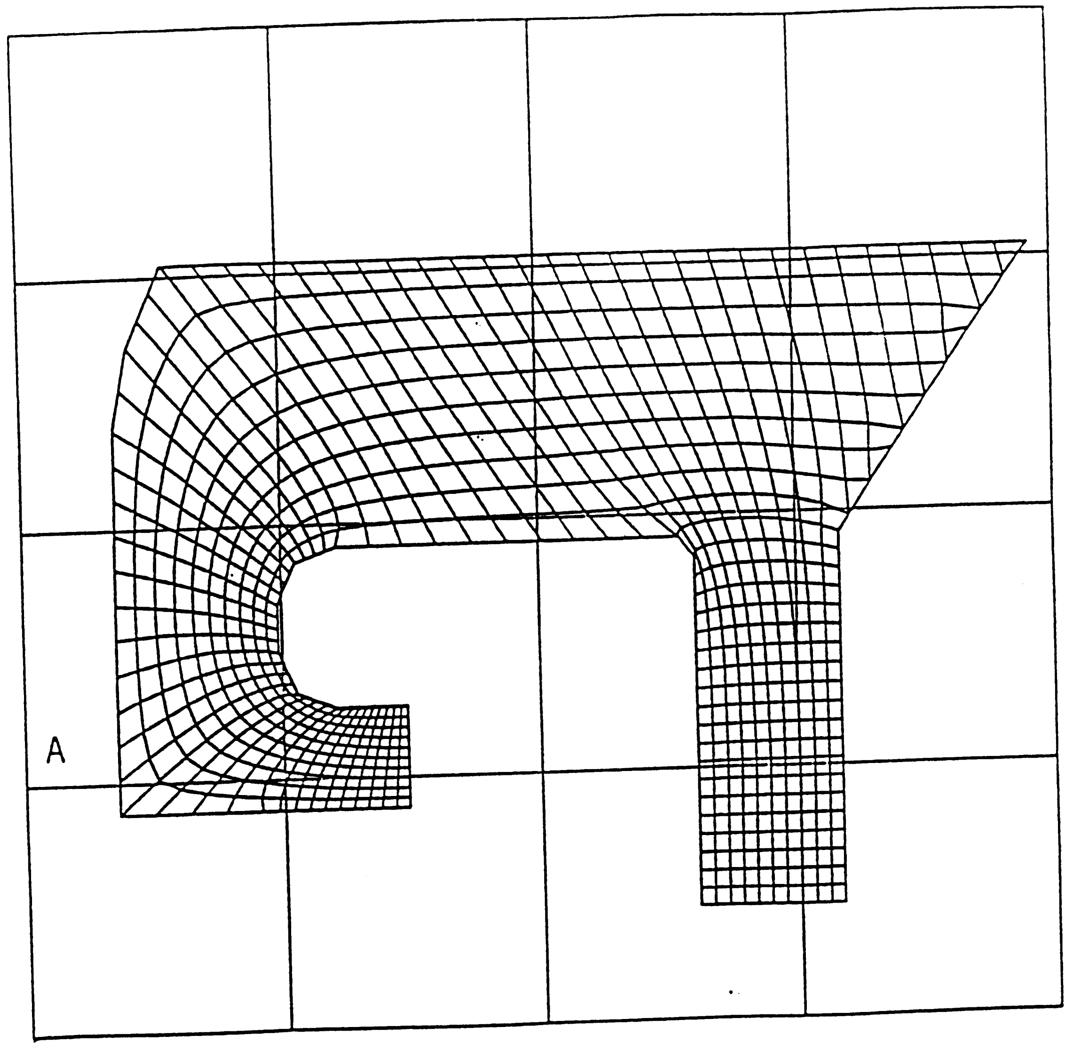


Figure 17. The Template overlay. The large squares are "templates", each of which will be decomposed onto the parallel computer.

the scattered one. Because of the rapid shifting of the optimal decomposition as a function of time, the minimum of $\overline{H}$ corresponding to this will be raised upwards, while the scattered minimum will remain approximately the same. As illustrated in Figure 22, 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 parallel machine, a "phase transition" may occur whereby the scattered decomposition actually becomes the better decomposition for $\overline{H}$.

In particular, the relative importance of the terms in Equation (3.21a) and Equation (3.21b) are governed by the ratio $t_{\text{comm}}/t_{\text{calc}}$ introduced in Section 3.1. This plays a role of a coupling constant

$$g^2 = \frac{t_{\text{comm}}}{t_{\text{calc}}} \tag{3.33}$$

which increases in size as communication performance of hardware decreases. The

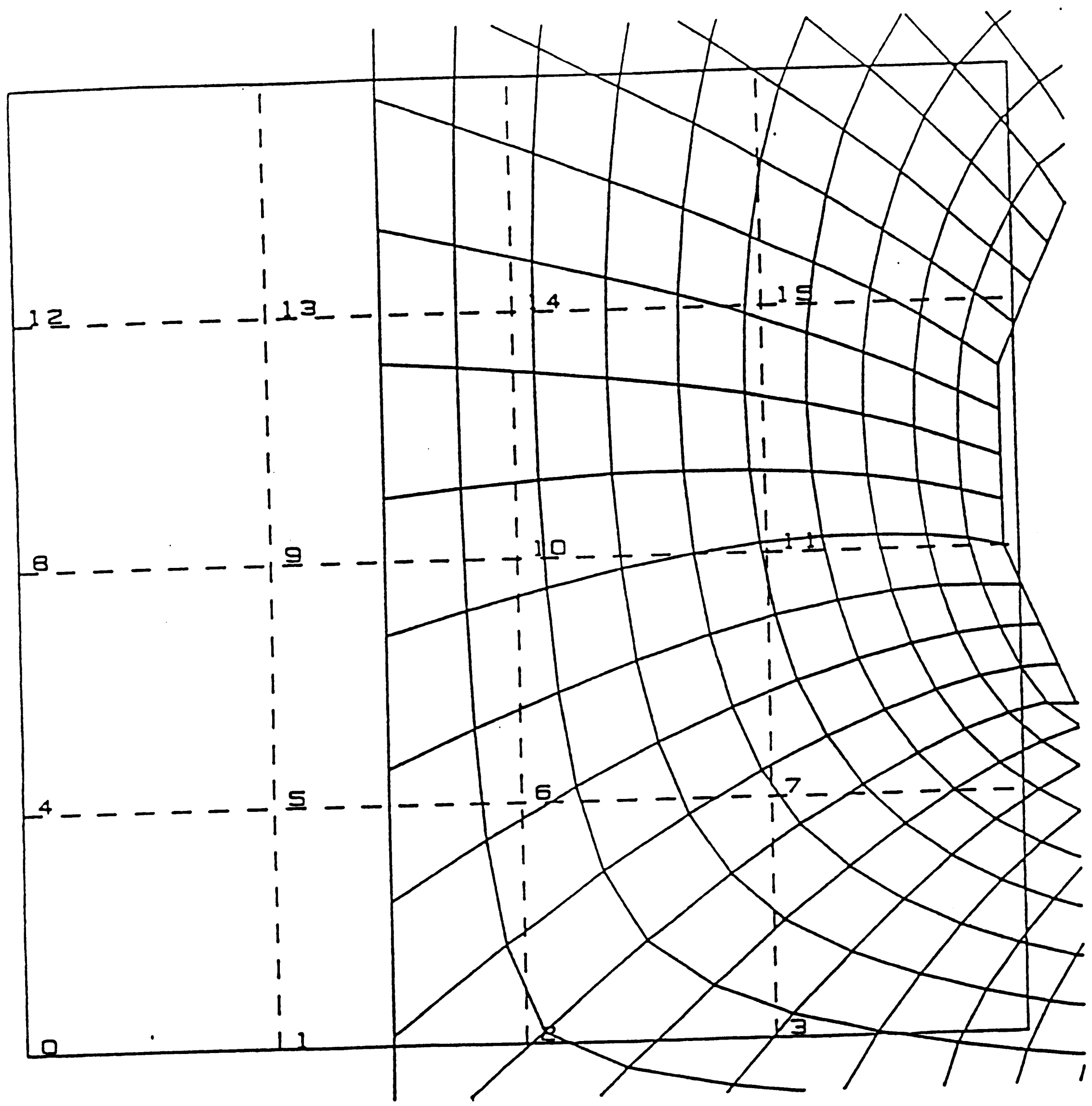


Figure 18. A Magnified view of the Template marked "A" in Figure 17. Each of the smaller, dotted squares is a processor of the parallel computer: the template has been decomposed onto a 4×4 mesh of processors. The processors are responsible for the finite elements landing within their regions.

scattered decomposition is favored as either the averaging time t_{av} increases or as the coupling g^2 decreases. Large t_{av} corresponds to rapidly varying problems which the operating system finds hard to equilibriate. In the earlier terminology, large t_{av} are high "temperature" complex systems. Thus, as we increase g^2 or decrease problem temperature, we transition from the scattered high temperature phase to the domains of Figure 16. This picture is of course quite analogous to the behavior of spin systems. We can view the scattered decomposition as a spin-wave. We obtain this analogy by associating with each member of the physical domain (nodal point in Figure 15) a

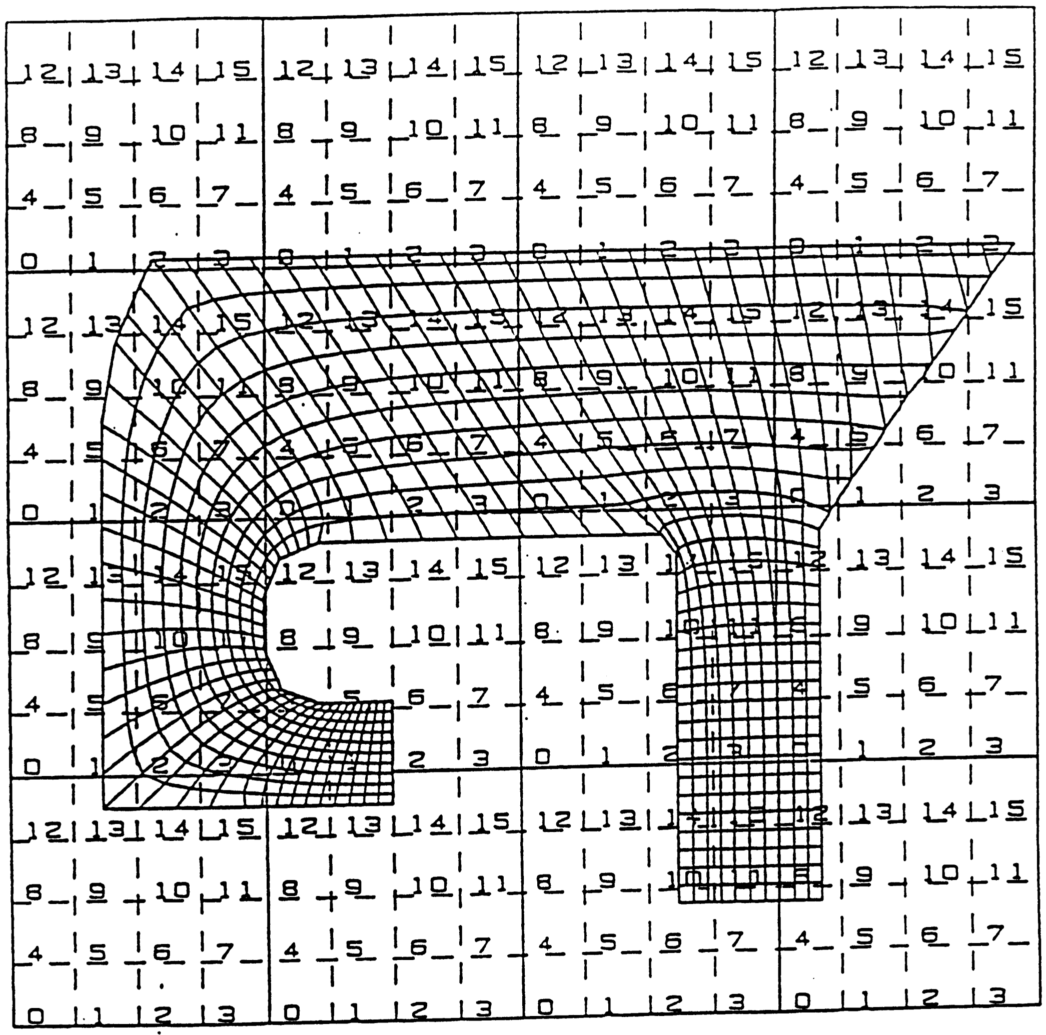


Figure 19. The entire scattered decomposition, with processor numbers shown.

spin whose value is the processor number $(1 \ldots N_{\text{proc}})$ where this member is stored. We then find spin waves at high temperature and domains at low temperature as one would expect.

4. The Dynamics of Temporal Structure

4.1 The String Formalism

In Section 3, we considered problems like those shown in Figure 2(a) where the temporal structure was either static or slowly varying. Here we consider cases such as that in Figure 2(b), where there is either complicated or rapidly varying time

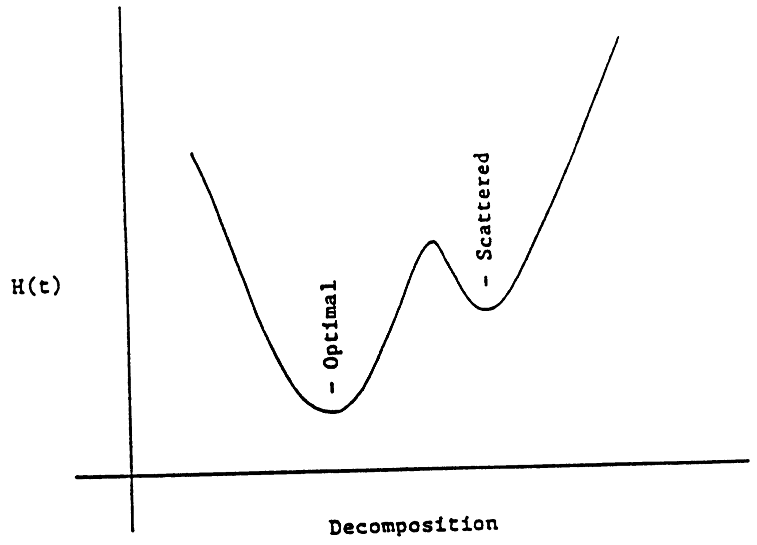


Figure 20. A sketch of the Hamiltonian versus all possible decompositions.

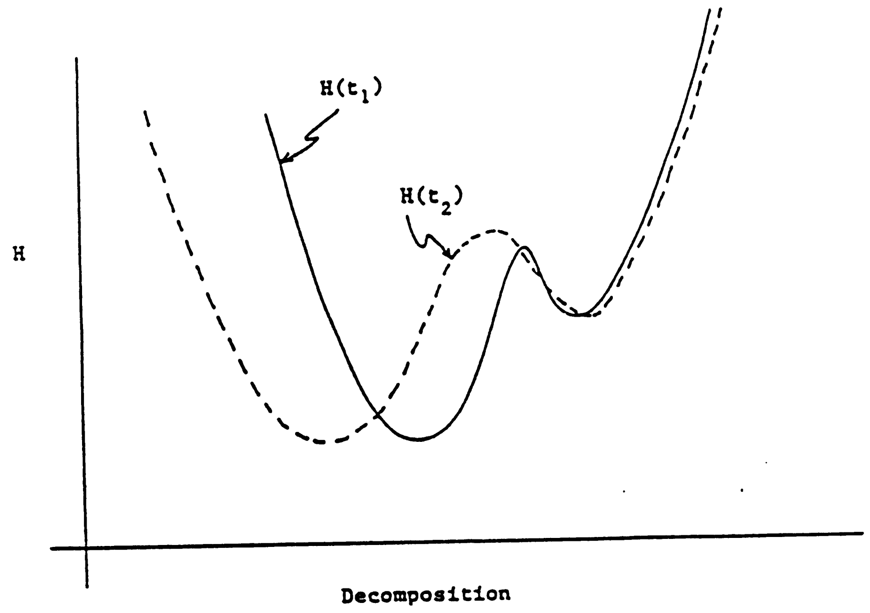


Figure 21. The Hamiltonian at two different times. The scattered decomposition is a relatively stable minimum.

structure. First, we describe how the dynamics changes from the particular picture of Section 3.3 to one of strings.

Consider a complex system with basic entities labelled by p. Then it is specified by the set of worldlines $\{\underline{x}_p(t)\}$ where $(\underline{x}, t)$ is a point in the generalized "space"-

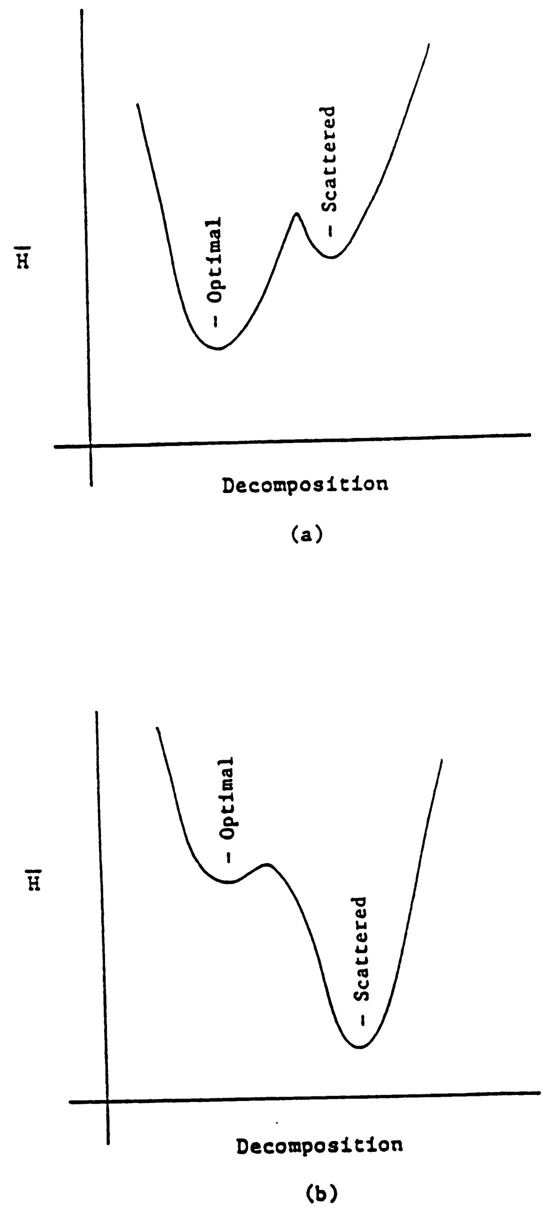


Figure 22. The Time Averaged Hamiltonian. Two scenarios are possible: the "optimal" decomposition remains the true minimum (a), or the scattered wins (b).

"time" (data-domain, evolution label) associated with this system. This set of strings or paths $\{\underline{x}_p(t)\}$ are the basic degrees of freedom. In the parallel computer decomposition problem, p labels processes and $\underline{x}_p$ the processor (node) number where p is located at clock cycle t. Clearly, the execution time T_{par} of the problem represented by this collection of processes is a functional of these paths

$$T_{\text{par}} \equiv T_{\text{par}}\left(\{\underline{x}_0(t)\},\ \{\underline{x}_1(t)\}\ldots\{\underline{x}_p(t)\}\ldots\right) \tag{4.1}$$

The minimization of T_{par} is another example of an optimization problem of the type discussed in Section 3.4 and Section 3.5.

The most straightforward approach views T_{par} as a function of the string parameters and a typical minimization would again use Monte Carlo or simulated annealing. As shown in Figure 23, one considers local changes in paths

$$\{\underline{x}_p(t)\} \rightarrow \{\underline{x}_p(t)\}' \tag{4.2}$$

and recovers a formalism very similar to that used in lattice gauge theories and quantum chemistry.

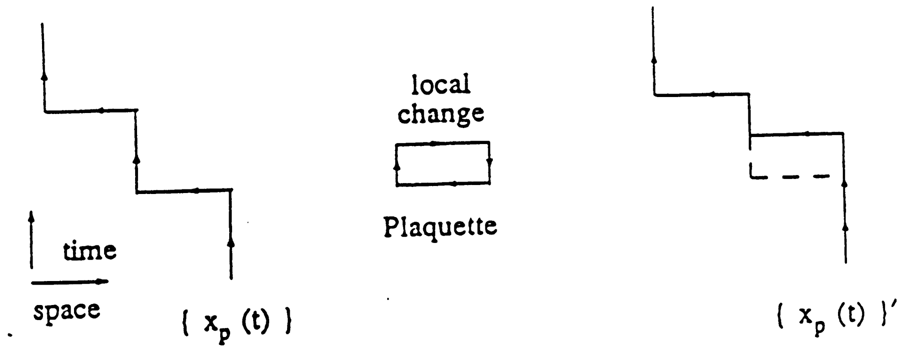


Figure 23. A typical local trajectory (string) change used in a Monte Carlo approach to the string formalism. The original string $\{\underline{x}_p(t)\}$ is changed to $\{\underline{x}_p(t)\}'$.

The alternative formalism uses the trick, due to Hopfield and Tank, where one introduces redundant variables $\eta_p(\underline{x}, t)$ describing the string p. The binary variables η_p are defined so that

$$\begin{aligned} \eta_p(\underline{x}, t) &= 1 \text{ if string } p \text{ is at } (\underline{x}, t) \\ &= 0 \text{ if string } p \text{ does not pass through } (\underline{x}, t) \end{aligned} \tag{4.3}$$

For each t value, the string only passes through one $(\underline{x}, t)$ value and this is enforced by adding a syntax term such as

$$T_{\mathrm{par}} \rightarrow T_{\mathrm{par}} + \text{const.} \sum_t \left(\sum_{\underline{x}} \eta_p(\underline{x}, t) - 1 \right)^2 \tag{4.4}$$

which is zero when the constraint is satisfied and otherwise positive.

Often we will extend this technique of using penalty functions for redundant variables by replacing quantities like T_{par} by approximations $\widetilde{T}_{\mathrm{par}}$ that becomes equal to T_{par} when constraints are satisfied. Exact constraints—such as the C.P.U. for the neural compiler can only execute one instruction at a time are replaced by penalty functions that "encourage" this. The combination of the trick (Equation (4.3)) and penalty functions, allows one to express complex optimization problems in a simple—often local—form with seemingly difficult constraints expressed easily if redundantly.

We will find, in several cases, two variants of the space-time minimization problem. In the most straightforward but computationally intense formulation, one solves the full optimization problem over the full space time region. Alternatively, one can use a window approach where if t is the current time, the state of the system at time t_0+1 is found by using a window $t_0 \leq t \leq t_0+\Delta t$ (here $\Delta t > 1$ corresponds to several steps in t) with the full string dynamics. The "future" $t > t_0+\Delta t$ is represented as an average over possibilities. In the physics analogy where objective functions like T_{par} are thought of as Hamiltonians, H, then this future average leads to external fields in H. This is the neural controller approach which is computationally less intense and further is insensitive to uncertainties in future data. Of course, the reliability of this approach depends on the accuracy of the "future" average.

4.2 Message Routing

There is an interesting class of very regular problems which exhibit a rhythmic or cyclical computational graph for which one needs the string rather than particle picture. Two examples are shown in Figure 24 and Figure 25 for the message routing[10] and combining switch problems. The latter is found in matrix-vector multiplication when both matrix M and vector x are distributed[3,36,56]. Consider

$$y_j = \sum_i M_{ji}\, x_i \tag{4.5}$$

If one sums over all values $M_{ji}\, x_i$ corresponding to each fixed j and all x_i stored in a particular node I, then Equation (4.5) becomes

$$Y_j = \sum_I y_j^{(I)}$$

or the accumulation of many (= number of components of y) global sums. The analytic solution of this is known as the algorithm *fold*[3,56,57] and illustrated in Figure 25.

In the physics analogy of Section 3.3, one might consider the separate $y_j^{(I)}$ as particles. However, these must move in a correlated fashion through the nodes of the computer in a way such that $y_j^{(I_1)}$ and $y_j^{(I_2)}$ are combined (added) when they "collide" at a common node "on the way" from (I_1) and (I_2) to the destination node J containing y_j. The physics analogy is incomplete as we have an instantaneous energy function but no equations of motion. Rather, we use the worldline formalism introduced in Section 4.1 and consider as degrees of freedom the complete time dependent strings which terminate at one end on y_j in node J and at the other end on $y_j^{(I)}$ in node I. We must drape these strings on the computer nodes so as to minimize the total time. In a traditional physics problem, one can use either a path integral or equations of motion formulation. Here only the former seems possible and one must regard the paths and not the instantaneous particles as the basic degrees of freedom.

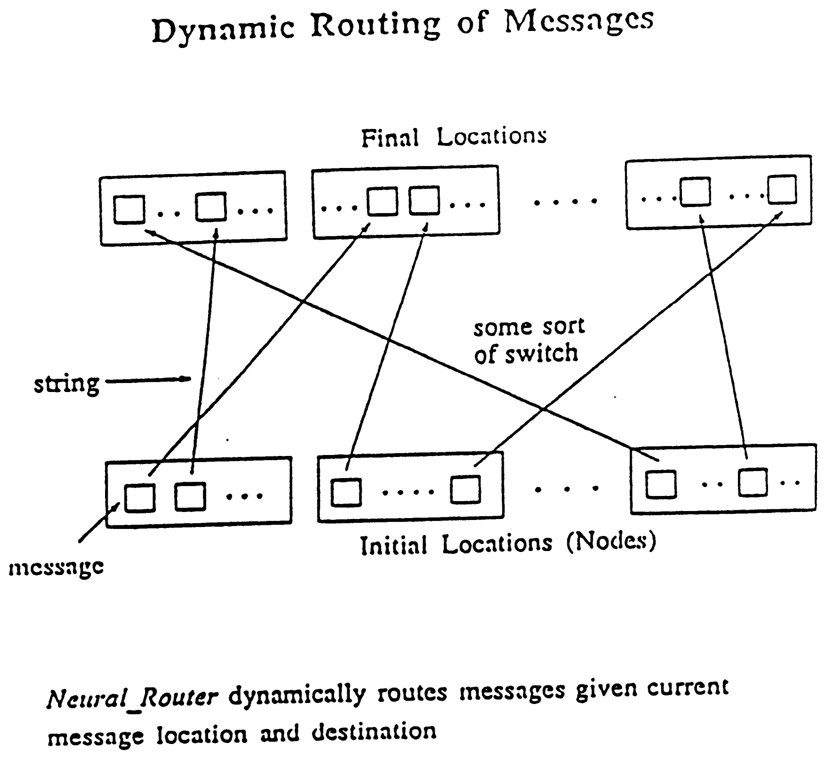

Dynamic Routing of Messages
Final Locations
some sort of switch
string
message
Initial Locations (Nodes)
Neural_Router dynamically routes messages given current message location and destination

Figure 24. A Message Routing Schematic of the Problem Addressed by the *neural_router*.

Initial results[9,10] are presented for this combining switch using neural networks for the controller or "window" formulation mentioned at the end of Section 4.1. The neural networks include terms that

- Attract strings corresponding to the same sum y_j.
- Repel strings corresponding to different sums y_{j_1} and y_{j_2}.
- Attract strings to destination node J containing y_j.

Typical results are shown in Figure 26 for our implementation called the neural accumulator. We considered 16 sums accumulated on 16 nodes for a sparse matrix M such that only a fraction $f (0 \leq f \leq 1)$ of nodes contain a contribution $M_{ji} x_i$ for each y_j. The case $f = 1$ is solved exactly by the analytic *fold* algorithm but the case $f < 1$ only has an approximate deterministic solution called the crystal accumulator[3,58].

Hopfield and Tank originally illustrated the neural network approach to the simple travelling salesman (TSP) problem. One minimizes the total travel time of a single salesman who must visit each of M cities once. Our path formalism can be viewed as a multiple travelling salesman method. In the original TSP, one has a single path or string to be routed (draped) over all the cities. In the new formalism, we have several interacting salesmen.

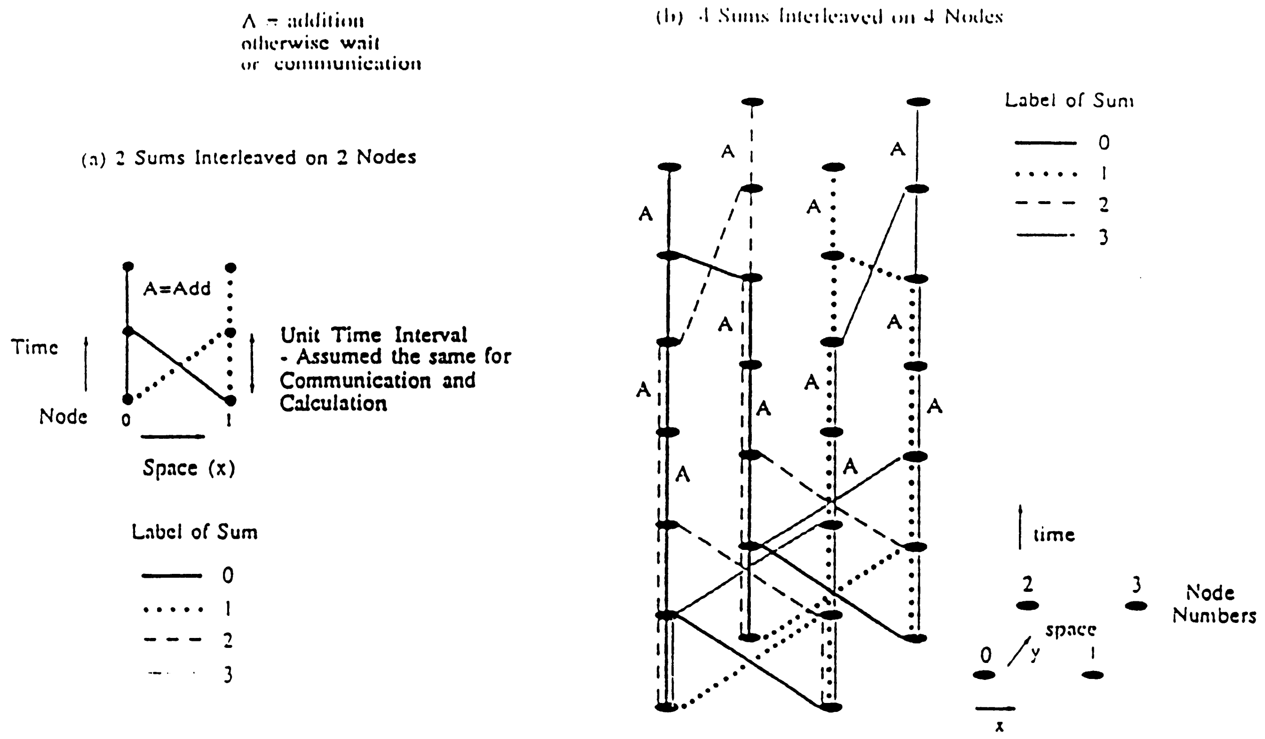


Figure 25. The combining switch illustrated for two and four nodes. The *fold* algorithm is known analytically in this case and shown in the figures.

4.3 Optimizing Compilers

In the previous section, we considered salesmen which were messages or processes moving between computers. Here we focus on a single node and consider salesmen as variables moving between memories (registers, cache, main memory, paged out memory...) and C.P.U. of a single computer. This leads us to consider our formalism for optimizing compilers. Let us illustrate our ideas with the problem of producing code for the simple C program:

$$z = z * (x + y) - y \tag{4.6}$$

As shown in Figure 27, this is represented by a directed acyclic graph (dag) where we will label nodes and leaves of the dag by an index i. We will consider the evaluation of Equation (4.6) on a very idealized computer with a single register on which all arithmetic operations are performed. Of course, the solution of this code generation problem is "obvious" by inspection but it is sufficient to illustrate our neural network approach.

We let m label the registers and memory locations in the machine, t label the clock cycle, and introduce neural variables η (m, i, t) to indicate whether quantity i is in location m at cycle t. The code generation problem is quite similar to the routing problem of the previous section: we construct an energy function containing syntax terms to ensure that

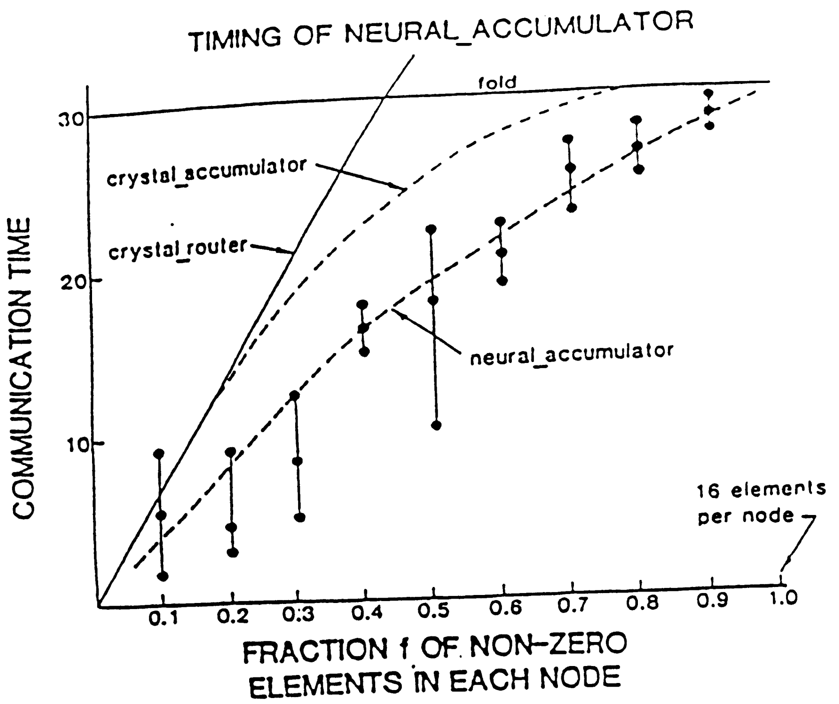


Figure 26. Results from the neural accumulator described in Section 4.2 which addresses the dynamic irregular version of the problem shown in Figure 25.

1) code correctly represents the program (e.g., t1 and t2 in Figure 27(b) are created before being used),
2) fixed-t configurations represent possible machine states (i.e., one quantity per storage location), and
3) states at consecutive times are related by a machine operation.

We also know the values of the initial machine state $\eta(m, i, 0)$ and the desired final machine $\eta(m, i, T)$ for some appropriate T. We add terms to the energy to minimize the number of intermediate machine states, and use Hopfield Tank style neural network evolution equations to find a minimum of this energy.

We have found a convenient way to build syntactic terms for even complicated constraints of the type (1,2,3) above. Since we ultimately interpret the neural variable η as logical variables with 1 = TRUE and 0 = FALSE, any constraint is some logical statement involving the $\eta(m, i, t)$, say $P(\eta_1, \eta_2, \ldots)$ = TRUE. Extend the logical operations to fraction values of η via $a \wedge b \rightarrow ab$, $a \vee b \rightarrow a + b - ab$, $\overline{a} \rightarrow 1 - a$. Then we penalize violations of this constraint by adding a multiple of $\overline{P}$ to the energy. In the simple study reported[59], an appropriate coefficient for the penalty function $\overline{P}$ was found by trial and error.

A typical example corresponds to constraint 2) above. Suppose $\eta(m, i, t) = 1$ indicates that memory location m contains quantity i. Then we cannot store any other variables in this location, i.e., $\eta(m, j, t)$ must be zero for all $j \neq i$. This

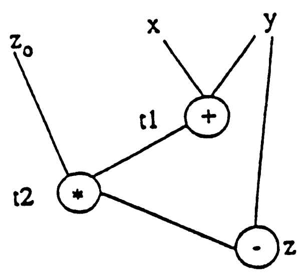
(a) Very Simple Expression

e.g. C: $z = z * (x + y) - y$

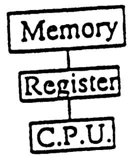
Intermediate C:

```
t1 = x + y
t2 = z * t1
 z = t2 - y
```

Directed Acyclic Graph (dag):



(b) Very Simple Model Computer



One - register machine, with operations

LOAD	M	R ← M
STORE	M	M ← R
OP	M	R ← OP M
OP	R, M	R ← R OP M

Figure 27. (a) A simple line of code and its representation by a directed acyclic graph. (b)The simple model computer discussed in Section 4.3.

constraint corresponds to a term $\overline{P}$ in the energy function E of form

$$E = \overline{P} = \eta(m, i, t)\left(\sum_{i \neq j} \eta(m, j, t)\right) \tag{4.7}$$

i.e., if $\eta(m, j, t) = 1$, we require all $\eta(m, j \neq i, t) = 0$. More generally, we have constraints of the form

$$A(\eta_1, \eta_2 \ldots) \text{ requires } B(\eta_1, \eta_2 \ldots) \tag{4.8}$$

where A and B are logical expressions, i.e., functions of the $\eta(m, i, t)$. The constraint Equation (4.8) corresponds to

$$P = B \vee \overline{A} \tag{4.9}$$

and a term $\overline{P} = A \wedge \overline{B} = A\overline{B}$ in the energy function.

We showed[36] that adding a noise term to the Hopfield Tank equations improved the performance of the resultant network for the problems of Section 3.4. We have used the same "bold network" for code generation and the results are encouraging[60]. However, there is an important issue we need to address before a practical system could be produced. We do not need an exact minimum of the term in E corresponding to the execution time of the code. However, many of the syntax constraints must be exactly satisfied or else the code will lead to incorrect results. One strategy is to allow small syntax violations, and use a postprocessor to repair them. This is a promising alternative to the all-or-nothing conventional approach to neural network optimization. It also provides a natural way to normalize the penalty terms in the energy: the penalty for a syntax violation should equal the extra execution time the postprocessor would have to add to the program to fix up the error. Note that the postprocessor could also be a neural network. This idea applies equally to the original travelling salesman neural network formulation of Hopfield and Tank.

The neural network approach to optimizing compilers has several attractive features:

1) As our approach explicitly minimizes an analytic function, it is possible to systematically improve any solution—perhaps by using simulated annealing. This would allow one to adjust the compile time and optimality of code according to ones needs. A long extensive optimization would be appropriate before a 6000 hour CM-5 run on a "grand challenge"; a quick non-optimal option would be appropriate when debugging.
2) This method naturally incorporates the "exact" architecture of the computer in the detailed form of E. In particular, it should in principle be able to handle complex memory hierarchies, which are present in high performance computers (such as the CRAY-2) but hard to handle with conventional techniques.
3) One should be able to build rather portable compilers with this technique. The manufacturer of a new RISC architecture multi-function superchip need only specify the particular form of E to allow a portable neural network based compiler to be used.

4.4 The Neural Navigator

The application of the methods of Section 4.1 to Section 4.3 to navigation are quite general but for definiteness, consider an optimal path problem where we have a collection of objects, which we call vehicles, in a two-dimensional space. We wish to navigate the vehicles from initial starting positions to final destinations so as to minimize the travel time. In the following, we consider just two vehicles labelled by $i = 1, 2$. The essential idea is to again view the paths as the degrees of freedom and use the redundant neural variables $\eta_i(\underline{x}, t)$ to parameterize these paths. We again need to form an energy functional $E(\{\eta_1\}, \{\eta_2\})$ which incorporates both the goal (minimal travel time) and the various constraints. Perhaps the problem is best illustrated by typical results shown in Figure 28 and Figure 29. We have the two vehicles starting at the bottom of the figure and reaching destinations at the top. They must navigate so as to avoid each other and respect the terrain constraints. In this case, the latter corresponds to a collection of hills (rocks) with sharp boundaries, i.e., the shaded areas are to be avoided. These hills consist of several randomly placed rocks and a major "range" with only a narrow passable region. Our energy function E has several terms

$$E = \sum_{j=0}^{5} A_j E_j \tag{4.10}$$

with variable coefficients A_j reflecting the "importance" of the constraint. Let us describe these terms qualitatively.

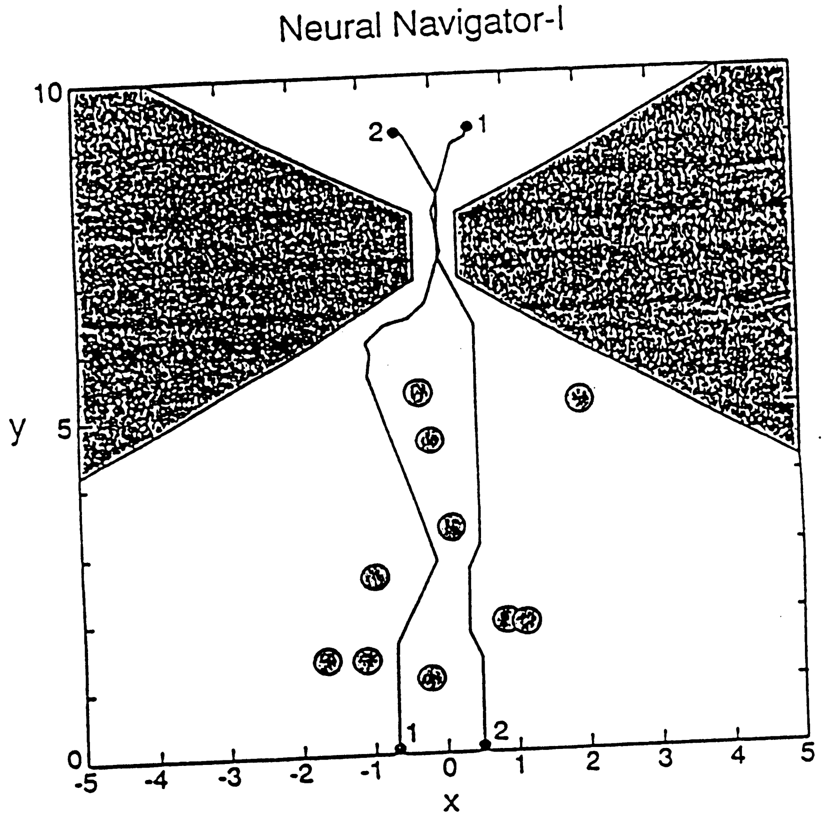


Figure 28. An example of the neural navigator for the motion of two vehicles (1,2) from the bottom to the top of the figure.

We will do this in the context of the window philosophy explained in Section 4.1. We have a starting time t_0 and only use the neural variables $\eta_i(\underline{x}, t)$ in the window $t_0 \leq t \leq t_0 + \Delta t$ as the dynamical variables. The future $t > t_0 + \Delta t$ is, in principle represented as an average over paths but in practice is approximated intuitively.

1) The first term represents the goal of minimal travel time. Let us just note that we

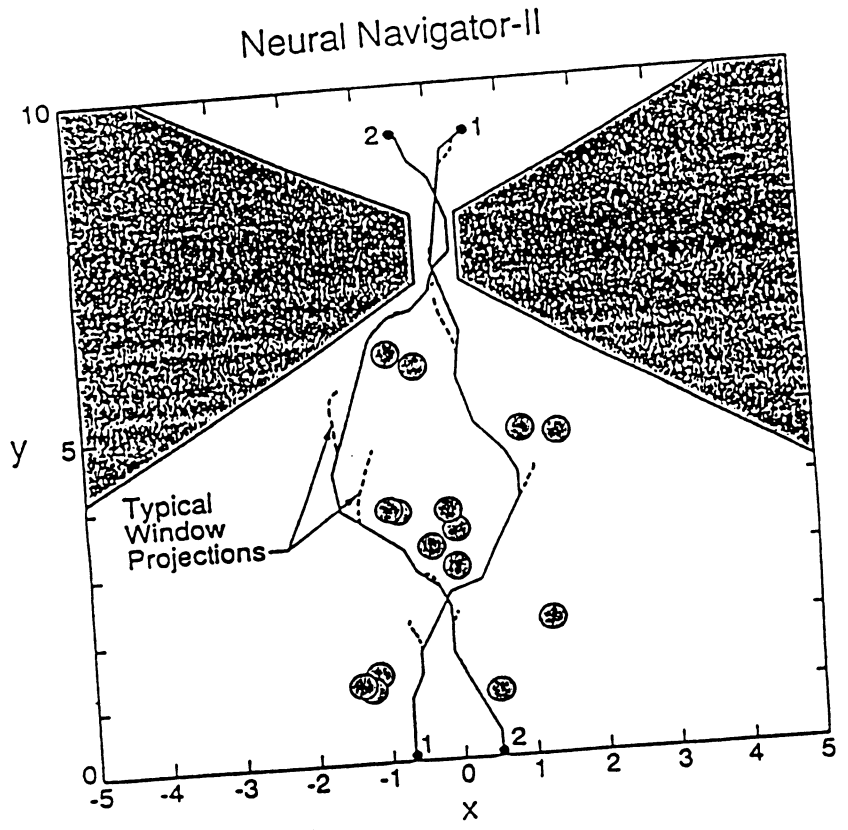


Figure 29. A second example of the neural navigator with rather more complex terrain than in Figure 28. Shaded areas (rocks, hills) are to be avoided. shown are typical window projections discussed in Section 4.4.

can go back and forth between a neural and conventional space time description with the simple equations.

$$\text{Positions } \underline{x}_i(t) = \sum_{\underline{y}} \underline{y}\,\eta_i(\underline{y}, t) \tag{4.11}$$

$$\begin{aligned} \underline{v}_i(t) &= \text{Velocity } \frac{d\underline{x}_i(t)}{dt} \\ &= \sum_{\underline{y}} \underline{y}\,\left(\eta_i(\underline{y}, t+\delta t) - \eta_i(\underline{y}, t)\right)/\delta t \end{aligned} \tag{4.12}$$

Let $\underline{d}_i$ be the final destination of the i'th vehicle. Then we express the goal of reaching the destination by

$$E_0 = A_0 \sum_{\substack{i \\ \text{over} \\ \text{vehicles}}} \sum_{\substack{t \\ \text{over} \\ \text{window}}} \left[|\underline{x}_i(t+1) - \underline{x}_i(t)| + |\underline{x}_i(t+1) - \underline{d}_i| - |\underline{x}_i(t) - \underline{d}_i| \right] \tag{4.13}$$

In the examples of Figures 28 and 29, this simple form is an adequate "average of the future"; it would not be sufficient if, for instance, the narrow pass was displaced (in x) from the destinations. We are considering a general multiscale method in space and time to provide a generally accurate estimate of E_0. The whole problem is first solved with a coarse space and time grid and this crude solution is used to estimate the goal constraint (Equation (4.13)). We have good experience with multiscale methods in space[61,62] but need to extend them to allow variable temporal scales.

1) The second term in Equation (4.10) expresses the smoothness of the trajectories and is taken as

$$E_1 = A_1 \sum_{\substack{i \\ \text{over} \\ \text{vehicles}}} \sum_{\substack{t \\ \text{window}}} \left(\frac{d^2 \underline{x}_i}{dt^2} \right)^2 \tag{4.14}$$

2) The third term ensures that the vehicles keep a reasonable distance apart and in the case of two vehicles we use

$$E_2 = A_2 \sum_{\underline{x}_1 \underline{x}_2} \sum_{\substack{t_1 t_2 \\ \text{in} \\ \text{window}}} \exp\left(-|t_1 - t_2|\right). \tag{4.15}$$
$$\exp\left(-|\underline{x}_1 - \underline{x}_2|^2/\sigma^2\right) \eta_1(\underline{x}_1, t_1)\, \eta_2(\underline{x}_2, t_2)$$

where σ is a suitable distance scale and we actually cut off sums and keep only those $\underline{x}_i$, t_i where vehicles are close.

3) In the current model calculations, the terrain constraint comes in two terms. In the first we represent hills and rocks by a function $H(\underline{x})$ which is zero on the level and unity in forbidden regions occupied by the hills. Then we constrain the vehicles to passable regions by the constraint

$$E_3 = A_3 \sum_{\substack{i \\ \text{over} \\ \text{vehicles}}} \sum_{\substack{t \\ \text{over} \\ \text{window}}} \sum_{\underline{x}} H(\underline{x})\, \eta_i(\underline{x}, t) \tag{4.16}$$

4) We also incorporate a maximum velocity $\underline{v}_i^{\max}(\underline{x})$ by the constraint

$$E_4 = A_4 \sum_{\substack{i \\ \text{over} \\ \text{vehicles}}} \sum_{\substack{t \\ \text{over} \\ \text{window}}} \sum_{\underline{x}} \eta_i(\underline{x}_i, t)\ \Theta\ \left(\underline{v}_i^{\max}(\underline{x})^2 - \underline{v}_i^2(t)\right) \tag{4.17}$$

where $\underline{v}_i$ is calculated as in Equation (4.12); actually this is not accurate and we remove "jitter" by averaging not over two but many time intervals in Equation (4.12). This is appropriate as we choose the grid so that a vehicle moves about two space grid positions in a single time step. In Equation (4.17), Θ is any reasonable function, such as the Heaviside function, that is zero when its argument is negative.

We introduce neural variables $\eta(x, t)$ as before, and write $\dot{x}$ in terms of η using Equation (4.12).

We note that the neural network path integral formalism is much more computationally complex than direct integration. However, it can be implemented efficiently on parallel machines; in particular, the SIMD Connection Machine CM-2 or special purpose neural network hardware. Thus, in a future world dominated by parallel machines, such path integral formalisms could be attractive compared to the direct sequential method of Equation (4.22).

In Section 4.4, we used a separate neural variable $\eta_i(x, t)$ for each vehicle i. If the vehicles were of identical type, it would be natural to use a single neural field $\eta(x, t)$ representing the vehicle density. At each time instance t, one requires a given number N of the $\eta(x, t)$ to be one and the rest zero. One would replace Equation (4.18) by

$$E_5 = A_5 \sum_t \left(\sum_x \eta(x, t) - N \right)^2 \tag{4.25}$$

In this way one could, for instance, solve several (N) pendula problems (with different initial conditions) simultaneously.

4.6 Event Driven Simulation

The string formalism is the natural description of an event driven simulation. As shown in Figure 30, we have several world lines i_1, i_2, i_3, ... interacting by events at discrete variable times $t_j^{i_k}$. Nature is time stepped and a time driven (synchronized) simulation is the obvious formulation of a physical simulation. However, this can be very inefficient if the world is modelled as many macroscopic objects interacting at variable irregular times. We note that such an event driven description of nature is usually inherently inexact as we ignore the possibility of other events at intermediate times. For instance, suppose we model a game of billiards where the world lines are trajectories of balls and events are collisions. In a time stepped approach, we are guaranteed "exact" results as long as the time step is small enough; we cannot miss a collision. In an event driven approach, we can take long, ball dependent, time steps between collisions; this is much faster as long as we catch all collisions.

The time stepped approach is exact, easily parallelized (the problem is loosely synchronous[3,63]) but computationally complex. The event driven approach is faster on a sequential machine but hard to parallelize (asynchronous[63] in the language of Section 5). As the strings interact irregularly, they cannot easily be evolved in parallel[64]. However, these problems in parallel simulation are only present if one insists on exactly reproducing the sequential simulation. As the original event driven formulation is intrinsically inexact, it seems natural to use an approximate simulation method. This could be either the neural network method, as described in Section 4.4, or simulated annealing. In the latter case, by varying the annealing temperature one can control the precision of the parallel simulation.

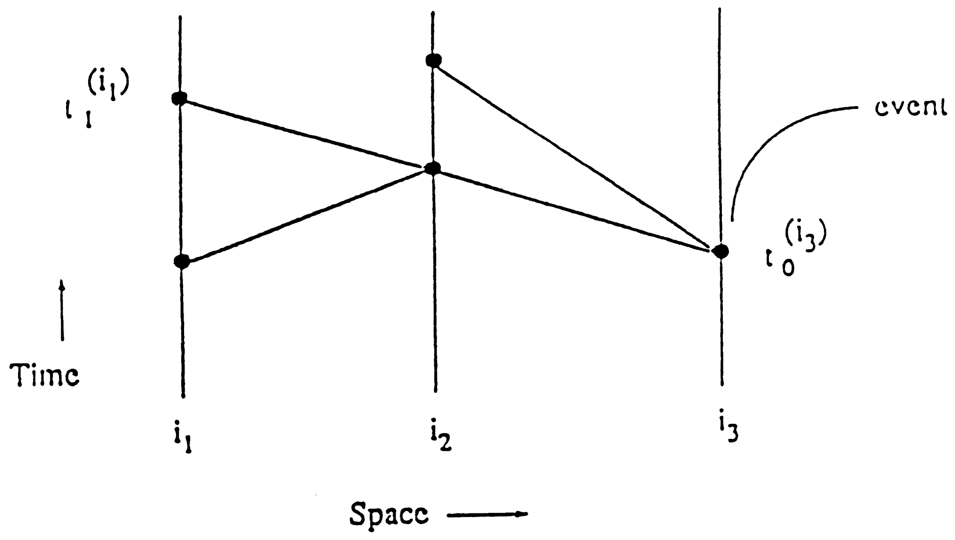


Figure 30. Three world lines of an event driven simulation showing typical events at times $t_j^{(i_k)}$.

This approach to event driven simulation was suggested to us by Dan Wiener and we are currently looking for some good examples to test our ideas on. We need a case where we understand both the event and time driven approaches so that we can quantify the precision of the event driven approach.

4.7 The Elastic Network

The above examples have extended Hopfield and Tank's neural network approach to the TSP, Travelling Salesman Problem. However, it is now clear[59] that this method is not as effective as other heuristics for the TSP. In particular, the elastic network[65–68] of Durbin and Wilshaw is clearly superior. In a set of papers[69,70], Petar Simic has shown that one can consider neural networks and elastic networks as similar mean field approximations to physical analogies for the TSP for different choices of degrees of freedom. In fact, returning to Section 4.1, the elastic network for the TSP is gotten by using the path as the degree of freedom rather than the neural variables of Equation (4.3). The string formalism performs better because it does not have the syntax enforcing term as in Equation (4.4).

This result does not imply neural approaches are "always" wrong. In fact, we can understand the great success of our neural methods in Section 3.4 because there we found a formalism where neural variables required no syntax enforcing term as they had no redundancy.

We have looked at the navigation problem of Section 4.4 again[13,71] and found much better results using strings to represent the vehicle paths—these are elastic bands stretched between the source and destination of the vehicles. We need to do more study of the string approach, such as the neural compiler of Section 4.3, both in this case and other areas where the neural networks led to great redundancy.

5. Problem Architectures

Here we describe a theory of problem structure or problem architecture which is analogous to the classification of computers by their control mechanism into SIMD and MIMD architectures[14,15,72,73].

This classification[4,63,74] was deduced from our experience at Caltech combined with a literature survey which was reasonably complete up to the middle of 1989. At Caltech, we developed some 50 applications on parallel machines of which 25 led to publications in the scientific literature describing the results of simulations performed on our parallel computers[1,2,3,75]. Our work was mainly on the hypercube, but the total of 300 references cover work on the Butterfly, Transputers and the SIMD Connection Machine and DAP. We were interested in applications and algorithms where we could evaluate the scaling to very large parallel machines. Table 3b illustrates what we mean by an application—"modelling the acoustic signature of a submarine using direct simulation of turbulence" would be another example, and in Table 3a we divide 84 application areas into eight disciplines.

The three general temporal structures are called synchronous, loosely synchronous, and asynchronous; we sometimes shorten these here to Classes I, II, and III, respectively. The temporal structure of a problem is analogous to the hardware classification into SIMD and MIMD. Further detail is contained in the spatial structure or computational graph of Figure 2 describing the problem at a given instant of simulation time[9] which is important in determining the performance as shown in Section 3.1 and Section 3.2 of an implementation[3] but it does not affect the broad issues discussed here. In Table 3c, we only single out one special spatial structure, "embarrassingly parallel", where there is little or no connection between the individual parallel program components, i.e., the spatial (computational) graph of Figure 2 is disconnected. For embarrassingly parallel problems, the synchronization (both software and hardware) issues are greatly simplified. As shown in Table 3c, asynchronous problems do not clearly scale to massively parallel systems unless they are embarrassingly parallel.

Synchronous problems are data parallel in the language of Hillis[76] with the restriction that the time dependence of each data point is governed by the same algorithm. Both algorithmically and in the natural SIMD implementation, the problem is synchronized microscopically at each computer clock cycle. Such problems are particularly common in academia as they naturally arise in any description of some world in terms of identical fundamental units. This is illustrated by quantum chromodynamics (QCD) simulations of the fundamental elementary particles which involve a set

Table 3: Summary of Problem Architectures

A. Data Sample from 300 Papers[4,62]

84	Total Applications
9	Biology
4	Chemistry and Chemical Engineering
14	Engineering
10	Geology and Earth Science
13	Physics
5	Astronomy and Astrophysics
11	Computer Science
18	Numerical Algorithms

B. Typical Applications

Analyze Voyager Data from Neptune	Image Processing
Calculate Proton Mass	Multiple Target Tracking
Computer Chess	Optimization of Oil Well Placement
Dynamics of H + HO	Seismic Modelling
Evolution of the Universe	Access Database

C. Conclusions of Survey of Applications

About 50% of applications clearly run well on SIMD machines.
About 90% of applications scale to large SIMD/MIMD machines.

Category	Number	Fraction		Natural Support Hardware
I: Synchronous	34	0.4	Total Class I and II	SIMD
II: Loosely Synchronous (not Synchronous)	30	0.36	Spatially Connected 0.76	MIMD Distributed Memory
I: Embarrassingly Parallel	6	0.07	Spatially Disconnected	SIMD
II: or III: Embarassingly Parallel but Asynchronous and needs MIMD	6	0.07		MIMD Distributed Memory
III: Truly Asynchronous (Spatially connected)	8	0.1	Unclear Scaling	Unclear Maybe MIMD Maybe Shared Memory

of gluon and quark fields on a regular four-dimensional lattice. These computations form the largest use of supercomputer time in academia[77,78].

Loosely synchronous problems are also typically data parallel, but now we allow different data points to be evolved with distinct algorithms. Such problems appear whenever one describes the world macroscopically in terms of the interactions between irregular inhomogeneous objects evolved in a time synchronized fashion. Typical examples are computer or biological circuit simulations where different components or neurons are linked irregularly and modelled differently. Figure 11 shows a loosely synchronous algorithm as the inhomogeneity makes the algorithm very different for each particle. Time driven simulations and iterative procedures are not synchronized at each microscopic computer clock cycle, but rather only macroscopically "every now and then" at the end of an iteration or a simulation time step.

Loosely synchronous problems are spatially irregular but temporally regular. The final *asynchronous* class is irregular in space and time, as in Figure 2b. A good example is an event driven simulation as in Section 4.6, which can be used to describe the irregular circuits we discussed above, but now the event paradigm replaces the regular time stepped simulation. Other examples include computer chess[79] and transaction analysis. Asynchronous problems are hard to parallelize and some may not run well on massively parallel machines. They require sophisticated software and hardware support to properly synchronize the nodes of the parallel machine as is illustrated by time warp mechanism[80].

Both synchronous or loosely synchronous problems parallelize on systems with many nodes. The algorithm naturally synchronizes the parallel components of the problem without any of the complex software or hardware synchronization mentioned above for event driven simulations. As shown in Table 3c, 90% of the surveyed applications fell into the classes which parallelize well. This also includes the embarrassingly parallel I, II, III-EP classes. It is interesting that massively parallel distributed memory MIMD machines which have an asynchronous hardware architecture are perhaps most relevant for loosely synchronous scientific problems.

In Table 4, we give details behind some of the applications in Table 3c by listing a few of the recent (end of 1989) Caltech applications with their problem architectures and an estimate of the appropriateness of SIMD or MIMD hardware.

We have looked at many more applications since the detailed survey[63] and the general picture described above remains valid! We have recently recognized that many complicated problems are mixtures of the basic classifications. An important case is illustrated by a battle management simulation implemented by my collaborators at JPL[82]. This is formally asynchronous with temporally and spatially irregular interconnections between various modules, such as sensors for control platforms and input/output tasks. However, each module uses a loosely synchronous algorithm such as the multi-target Kalman filter[83] or the target-weapon pairing system. Thus, we had a few ($\sim$ 10–50) large grain asynchronous (Class III) objects, each of which was a data parallel Class I or II algorithm. This type of asynchronous problem can be implemented in a scaling fashion on massively parallel machines. We can denote this IIICG–IIFG to indicate the $\underline{C}$oarse $\underline{G}$rain asynchronous controlling of

Table 4. Problem Architecture of 14 Selected Caltech Parallel Applications[81]

Application	Problem Architecture	Does SIMD Perform Well
QCD	I – Regular	Yes
Continuous Spin (High T_c)	I – Regular	Yes
Ising/Potts Models	I – Regular	Yes
Strings	III – Embarrassingly Parallel (forall)	No
Particle Dynamics		
$O(N \log N)$	II – Irregular	Maybe
$O(N * N)$	I – Regular	Yes
Astronomical Data Analysis	IIICG–IIFG (several different loosely synchronous modules)	Unknown
Chemical Reactions		
$H + H_2$ Scattering	I – Regular + forall	Probably
$e^- + CO$ Scattering	I – Regular + forall	Probably
Grain Dynamics	I – Regular	Yes
Plasma Physics	II – Can Be Irregular	Probably
Neural Networks	II – Typically Irregular	Sometimes
Computer Chess	III – Asynchronous	No
Multi-target Tracking	II – Irregular	Maybe

Fine Grain loosely synchronous subproblems. A similar example of this problem class is machine vision and signal processing, where one finds an asynchronous collection of data parallel modules to perform various image processing tasks, such as stereo matching and edge detection. Figure 31 illustrates another example where we outline an approach to designing a new airframe which involves aerodynamics, structures, radar signature and the optimization discussed above for the oil reservoir case. This Figure 31 also points to the interesting analogy between heterogeneous problems of class IIICG–IIFG and a heterogeneous computer network.

In the above cases, the asynchronous components of the problems were large grain modules with modest parallelism. This can be contrasted with Otto and Felten's

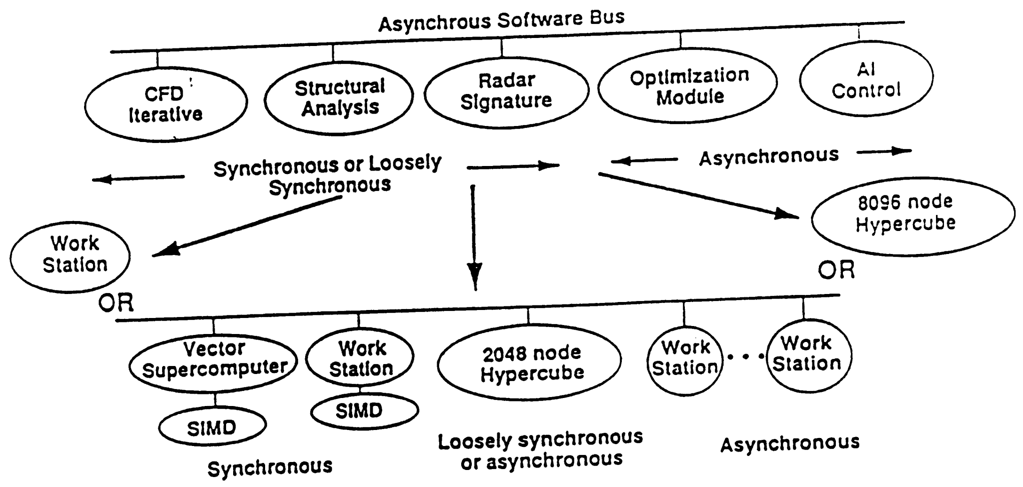


Figure 31. The Mapping of Heterogeneous Problems onto Heterogeneous Computer Systems

MIMD computer chess algorithm, where the asynchronous evaluation of the pruned tree is "massively parallel"[79]. Here, one can break the problem up into many loosely coupled but asynchronous parallel components which give excellent and scalable parallel performance. Each asynchronous task is now a Class I or Class II modestly parallel evaluation of a given chess position.

6. Matching the Space Time Structure of Problems and Computers

6.1 The Duality Between Memory Structure of the Computer and Space Time Structure of the Problem

Here, we review the analysis concerning memory hierarchy[8,17]. We first note that this is very directly related to an analysis of shared memory architectures because high performance machines of this shared memory class need a fast cache or local memory to buffer data from the slow shared memory. We will use the term "cache" to refer interchangeably to a true hardware controlled cache or a user or software controlled fast local memory.

We have already decomposed problems into parts designed to minimize communication between them. This was the subject of Section 3 and is essentially all that is necessary to obtain good performance from machines like the NCUBE or Transputer arrays. We will use the same idea for hierarchical memories and divide the problem into parts (grains) so that each grain fits into the lowest level of memory hierarchy. This is illustrated in Figure 32 and Figure 33 for shared memory machines, hierarchical multicomputers and the simpler homogeneous multi-computer. In Figure 32(a), we see a simple special case where the total problem will fit into the "caches" when summed over the nodes. Then the global shared memory can just be used as a communication path and one can easily see that the overheads take the same form as Equation (3.10) with t_{mem} replacing t_{comm}. Here t_{mem} was already illustrated in Figure 4 and is defined as the time taken to read and write a word between the two levels of the memory hierarchy. However in the general case, shown in Figure 32(b), one may fill the "caches" with grains but there are still other (virtual) grains waiting in the shared memory to be executed. Figure 33(b) shows how this looks for a hierarchical hypercube where we note the grain size is defined by the "cache" size and not by the total node memory. I have given a detailed analysis[8] of the extra overhead needed for the cases of Figure 32(b) and Figure 33(a) to swap the grains in between the two memory levels. The essential idea is contained in Figure 34 which illustrates that the overhead f_{H} is proportional to $t_{\mathrm{mem}}/t_{\mathrm{calc}}$ divided by the average temporal size of the grain. On general principles, communication overhead proportional to t_{comm} is associated with the spatial structure and that proportional to t_{mem} is associated with the temporal properties of the problem.

These results leads to a universal decomposition methodology which we call the method of space-time blocking. For homogeneous multicomputers one only needs to divide the problem into local spatial blocks. This is a special case of a more general and difficult technique which divides the full space-time structure of the problem into blocks. This idea is illustrated in Figure 35 for a very simple one-dimensional partial differential equation. One can also illustrate this idea with the BLAS-3 project introduced by Dongarra and collaborators[84–86]. Their well thought out strategy of using matrix-matrix and not matrix-vector suboperations is precisely the implementation of space-time blocking for this problem. Although initially introduced for shared and hierarchical memory machines, the BLAS-3 idea is the correct basis for a universal library of full matrix operations across all the architectures of Section 1.

We see that all high-performance computers appear to need locality to achieve their performance. This is spatial locality for homogeneous hypercubes but more general and indeed higher performance architectures exploit locality in space and time.

6.2 Parallel Computer Software

Figure 3 illustrated our view of software as implementing a map of problem onto computer. Traditional languages such as Fortran map the space-time of the original problem onto a purely temporal structure corresponding the execution of a serial program on a sequential computer. The spatial (data) parallelism becomes a temporal or control parallelism expressed as a DO loop. A parallelizing compiler tries to convert the temporal structure of Fortran back into a space-time structure which can execute well on a parallel machine. Usually this fails as the original map of the problem into sequential code has thrown away information necessary to reverse this map. The first (and some ongoing) efforts in parallelizing compilers tried to directly "parallelize the

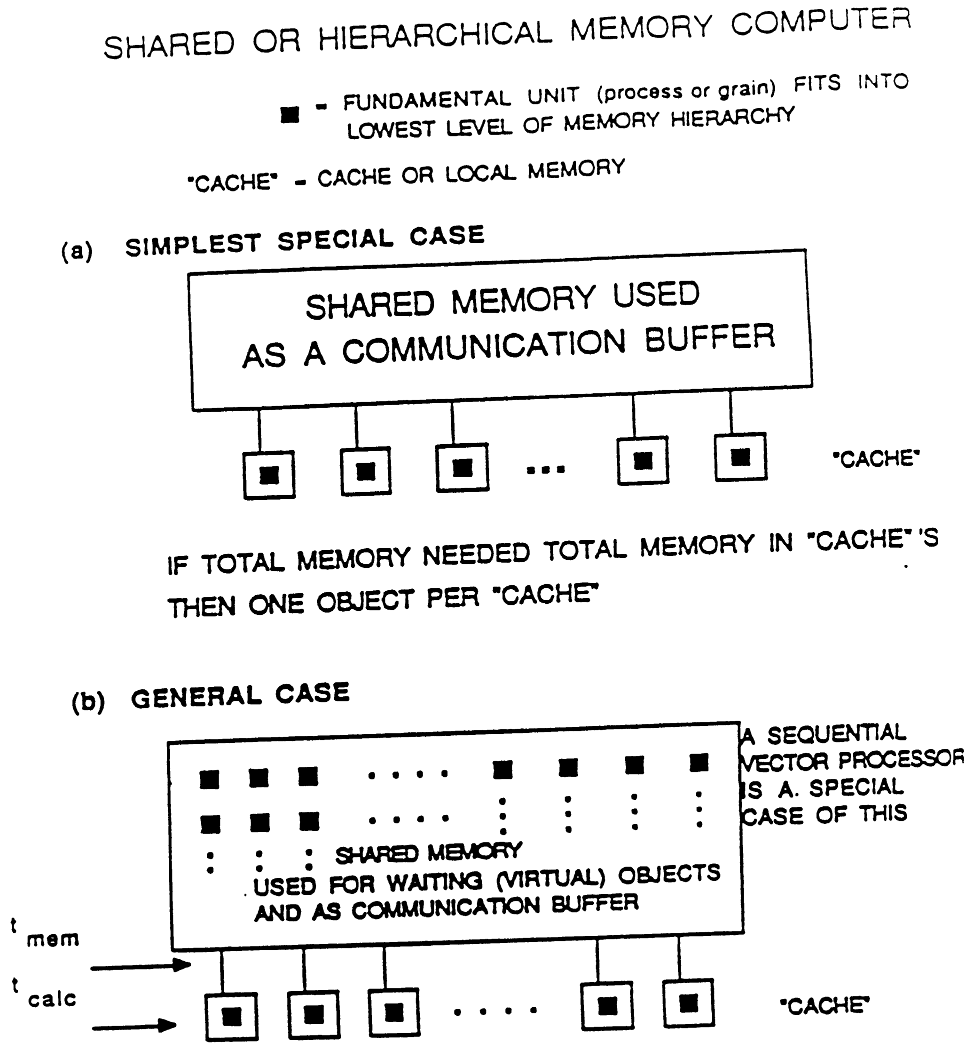

SHARED OR HIERARCHICAL MEMORY COMPUTER
■ - FUNDAMENTAL UNIT (process or grain) FITS INTO LOWEST LEVEL OF MEMORY HIERARCHY
"CACHE" - CACHE OR LOCAL MEMORY
(a) SIMPLEST SPECIAL CASE
SHARED MEMORY USED AS A COMMUNICATION BUFFER
"CACHE"
IF TOTAL MEMORY NEEDED TOTAL MEMORY IN "CACHE"'S
THEN ONE OBJECT PER "CACHE"
(b) GENERAL CASE
A SEQUENTIAL VECTOR PROCESSOR IS A SPECIAL CASE OF THIS
SHARED MEMORY
USED FOR WAITING (VIRTUAL) OBJECTS AND AS COMMUNICATION BUFFER
t mem
t calc
"CACHE"

Figure 32. Shared or hierarchical memory computers showing the total problem divided into parts or processes that individually fit into the "caches". In (a) the total problem fits into the "caches" but (b) shows the general case where the processes are held in the (slow) main memory and need to be cycled through the "caches".

DO loops". This seems doomed to failure in general as it does not recognize that in nearly all cases the parallelism comes from spatial and not control (time) structure. Thus, we are working with Kennedy at Rice and others on a parallelizing compiler FortranD where the user adds additional information to tell the compiler about the spatial structure. We are optimistic that the resultant FortranD project[14,15,72,87,88] will be successful for the synchronous and loosely synchronous problem classes defined in Section 5.

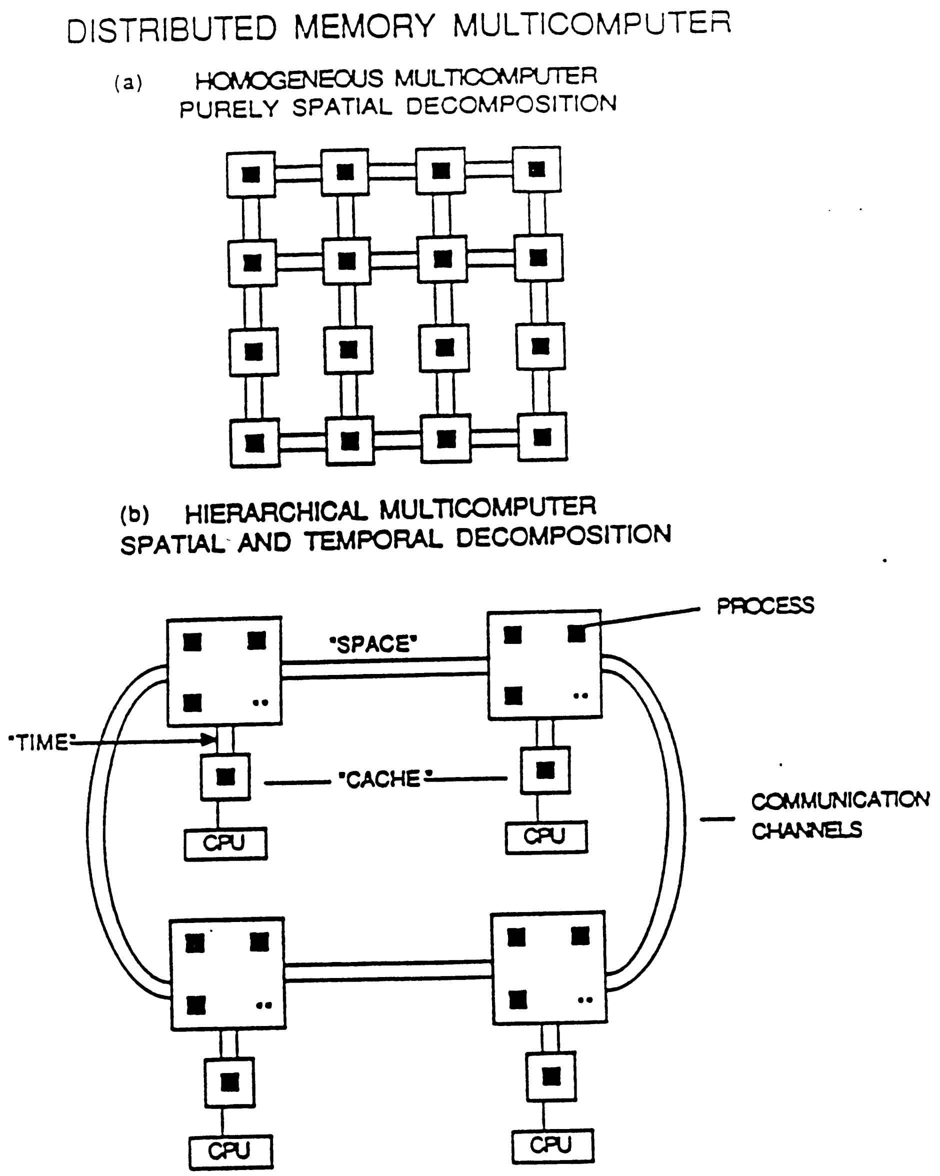
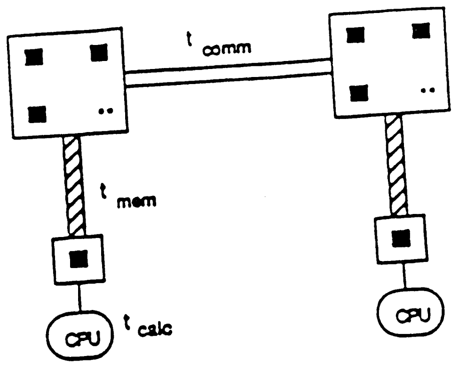

DISTRIBUTED MEMORY MULTICOMPUTER
(a) HOMOGENEOUS MULTICOMPUTER
PURELY SPATIAL DECOMPOSITION
(b) HIERARCHICAL MULTICOMPUTER
SPATIAL AND TEMPORAL DECOMPOSITION
PROCESS
"SPACE"
"TIME"
"CACHE"
COMMUNICATION CHANNELS
CPU
CPU
CPU
CPU

Figure 33. An illustration of (a) homogeneous and (b) hierarchical memory multicomputers.

Most languages do not express and preserve space time structure. Array languages such as APL and Fortran 90 are examples of data parallel languages which at least partially preserve the space time structure of the problem in the language. Appropriate class libraries can also be used in C++ to achieve this goal. We expect that development of languages which better express problem structure will be essential to get good performance with an attractive user environment on large scale parallel computers. The results in Section 6.1 show that data locality is critical in sequential high performance (hierarchical memory) machines as well. Thus, we would expect that the use of languages which properly preserve problem structure will lead to better performance on all computers.

ANALYSIS OF HIERARCHICAL MEMORY OVERHEAD



CACHE LOADING TIME = t_{mem} × object spatial size

VERSUS

TIME SPEND IN CACHE = t_{calc} × temporal extent(computational extent) of object × spatial size

MEMORY OVERHEAD IS GENERALLY

$$1/(\textit{OBJECT OR GRAIN SIZE}\,)^{1/d} \times \frac{t_{mem}}{t_{calc}} \ \textit{hierarchical}$$

$$\frac{t_{comm}}{t_{calc}} \ \textit{distributed}$$

Figure 34. A summary of the overheads associated with hierarchical memory. d is the system dimension introduced in Section 2(b).

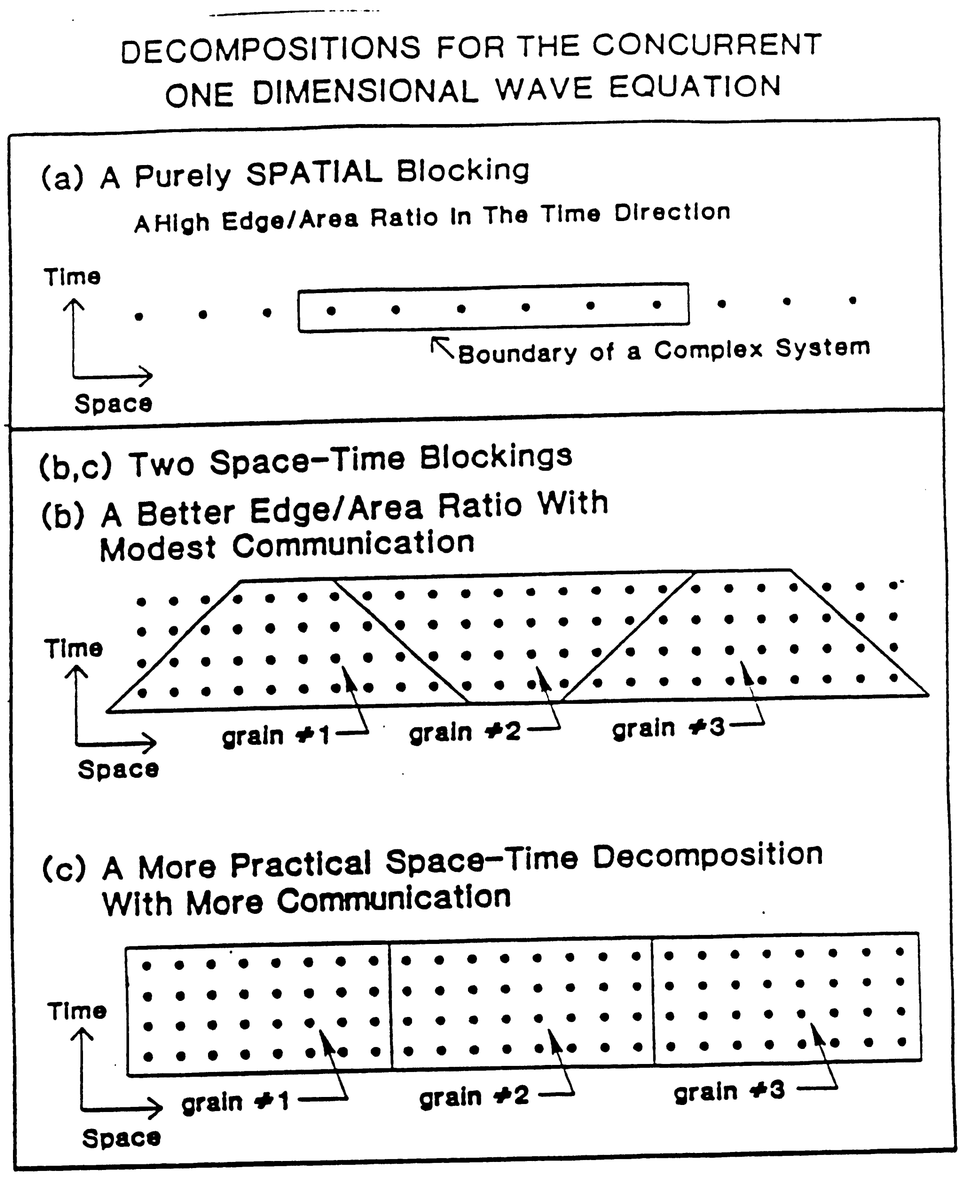

DECOMPOSITIONS FOR THE CONCURRENT ONE DIMENSIONAL WAVE EQUATION
(a) A Purely SPATIAL Blocking
A High Edge/Area Ratio In The Time Direction
Time
Space
Boundary of a Complex System
(b,c) Two Space-Time Blockings
(b) A Better Edge/Area Ratio With Modest Communication
Time
Space
grain #1
grain #2
grain #3
(c) A More Practical Space-Time Decomposition With More Communication
Time
Space
grain #1
grain #2
grain #3

Figure 35. Space-time blocking (b,c) constrasted with conventional spatial (a) decompositions used for the one-dimensional wave equations.

Acknowledgements

This work has been collaborative with many people, especially W. Furmanski and S. Otto and I would like to thank them and my colleagues in the Caltech Concurrent Computation Program for their insight and help.

References

1. G. C. Fox, 'Questions and Unexpected Answers in Concurrent Computation' in *Experimental Parallel Computing Architectures*, ed. J. J. Dongarra (Elsevier Science Publishers B.V., North-Holland, 1987) p. 97–121. Caltech Technical Report C^3P-288 [Fox:87d].

2. G. C. Fox, 'The Hypercube and the Caltech Concurrent Computation Program: A Microcosm of Parallel Computing' in *Special Purpose Computers*, ed. B. J. Alder (Academic Press, Inc., 1988) p. 1–40. Caltech Technical Report C^3P-422 [Fox:88oo].

3. G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, *Solving Problems on Concurrent Processors, Volume 1* (Prentice-Hall, Inc., 1988), [Fox:88a].

4. I. G. Angus, G. C. Fox, J. S. Kim, and D. W. Walker, *Solving Problems on Concurrent Processors, Volume 2* (Prentice-Hall, Inc., 1990) [Angus:90a].

5. G. C. Fox, P. C. Messina, and R. D. Williams, *Parallel Computing Works* (Morgan Kaufmann Publishers, 1992) [Fox:92a].

6. G. C. Fox, S. W. Otto, and E. A. Umland, 'Monte Carlo Physics on a Concurrent Processor', *Journal of Statistical Physics*, **43**, 5/6 (1986). Caltech Technical Report C^3P-214 [Fox:85a].

7. G. C. Fox and S. W. Otto, 'Concurrent Computation and the Theory of Complex Systems' in *Hypercube Multiprocessors*, ed. M. T. Heath (SIAM, 1986), p. 244–268. Caltech Technical Report C^3P-255 [Fox:86a].

8. G. C. Fox, 'Domain Decomposition in Distributed and Shared Memory Environments — I: A Uniform Decomposition and Performance Analysis for the NCUBE and JPL Mark IIIfp Hypercubes', in *Supercomputing*, ed. E. N. Houstis, T. S. Paptheodorou, and C. D. Polychronopoulos (Springer-Verlag, 1987), Volume 297, p. 1042–1073. Caltech Technical Report C^3P-392 [Fox:87b].

9. G. C. Fox and W. Furmanski, 'The Physical Structure of Concurrent Problems and Concurrent Computers', *Phil. Trans. R. Soc. Lond. A.*, **326**, (1988) p. 411–444. Caltech Technical Report C^3P-493 [Fox:88tt].

10. G. C. Fox and W. Furmanski, 'A String Theory for Time Dependent Complex Systems and its Application to Automatic Decomposition' in *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 1*, ed. G. C. Fox (ACM Press, 1988), p. 285–305. Caltech Technical Report C^3P-521 [Fox:88f].

11. G. C. Fox, W. Furmanski, and J. Koller, 'The Use of Neural Networks in Parallel Software Systems', *Mathematics and Computers in Simulation*, **31**, No. 6, p. 485–495 (1989). Caltech Technical Report C^3P-642b [Fox:89q].

12. G. C. Fox, W. Furmanski, A. Ho, J. Koller, P. Simic, and Y. F. Wong, 'Neural Networks and Dynamic Complex Systems' in *Proceedings of 1989 SCS Eastern Conference*, (1988). Caltech Technical Report C^3P-695 [Fox:88kk].

13. G. C. Fox, 'Physical Computation', *Concurrency: Practice and Experience*, **3**, No. 6, p. 627–653 (1991). Syracuse Technical Report SCCS-2b [Fox:90i] (C^3P-928b).

14. G. C. Fox, 'Hardware and Software Architectures for Irregular Problem Architectures' in *ICASE Workshop on Unstructured Scientific Computation on Scalable Microprocessors*, 1990 Syracuse Technical Report SCCS-111 [Fox:90p] (CRPC-TR91164), to be published by MIT Press.

15. G. C. Fox, 'The Architecture of Problems and Portable Parallel Software Systems', Syracuse Technical Report SCCS-134 (1991) [Fox:91g].

16. G. C. Fox, 'Parallel Computing', California Institute of Technology Technical Report C3P-830 (1989); published in Encyclopedia of Physical Science and Technology 1991 Yearbook (Academic Press, Inc.). Caltech Technical Report C^3P-830 [Fox:89y].

17. G. C. Fox, 'Parallel Supercomputers' in *Computer Engineering*, ed. C. H. Chen (McGraw-Hill Publishing Company, New York, 1992), Chapter 17. Caltech Technical Report C^3P-451d [Fox:92b].

18. G. C. Fox, 'Parallel Computing and Education', *Daedalus Journal of the American Academy of Arts and Sciences*, **121**, No. 1, p. 111–118 (1992). Caltech Technical Report C^3P-958 [Fox:92d] (SCCS-83, CRPC-TR91123).

19. G. C. Fox, 'Achievements and Prospects for Parallel Computing', *Concurrency: Practice and Experience*, **3**, No. 6, p. 725–739 (1991). Caltech Technical Report C^3P-927b [Fox:91f] (SCCS-29b, CRPC-TR90083).

20. P. Messina, 'Parallel Computing in the 1980s—One Person's View', *Concurrency: Practice and Experience*, **3**, No. 6, p. 501–524 (1991). Caltech Technical Report CCSF-4-91 [Messina:91a].

21. G. C. Fox and S. Otto, 'Algorithms for Concurrent Processors', *Physics Today*, **37**, No. 5, p. 50 (1984). Caltech Technical Report C^3P-071 [Fox:84a].

22. G. C. Fox, 'The Performance of the Caltech Hypercube in Scientific Calculations: A Preliminary Analysis', in *SuperComputers—Algorithms, Architectures, and Scientific Computation*, ed. F. A. Matsen and T. Tajima (University of Texas Press, 1985). Caltech Technical Report C^3P-161 [Fox:85c].

23. H.-Q. Ding, 'The 600 Megaflops Performance of the QCD Code on the Mark IIIfp Hypercube', in *The Fifth Distributed Memory Computing Conference*, Volume 2, ed. D. W. Walker and Q. F. Stout (IEEE Computer Society Press, California, 1990), p. 1295–1301. Caltech Technical Report C^3P-799b [Ding:90c].

24. H.-Q. Ding, 'Heavy Quark Potential in Lattice QCD: A Review of Recent Progress at Caltech', *International Journal of Modern Physics C*, **2**, No. 2, p. 637–658. Caltech Technical Report C^3P-963b [Ding:91b].

25. J. L. Gustafson, G. R. Montry, and R. E. Benner, 'Development of Parallel Methods for a 1024-Processor Hypercube', *SIAM J. Sci. Stat. Comput.*, **9**, No. 4, p. 609–638 (1988) [Gustafson:88a].

26. B. Mandelbrot, 'Fractals: Form, Chance, and Dimension' (Freeman, San Francisco 1979) [Mandelbrot:79a].

27. W. Heller, private communication (1985).

28. B. S. Landman and R. L. Russo, 'On a Pin versus Block Relationship for Partitions of Logic Graphs', *IEEE Trans. Comp.*, C20, p. 1469 (1971) [Landman:71a].

29. W. E. Donath, 'Placement and Average Interconnection Lengths of Computer Logic, *IEEE Trans. Circuits and Systems*, **16**, p. 272 (1979) [Donath:79a].

30. G. Fox, A. J. G. Hey, and S. Otto, 'Matrix Algorithms on the Hypercube I: Matrix Multiplication', *Parallel Computing*, **4**, p. 17, 1987. Caltech Technical Report C^3P-206 [Fox:85b].

31. M. S. Warren, W. H. Zurek, P. J. Quinn, and J. K. Salmon, 'The Shape of the Invisible Halo: N-Body Simulations on Parallel Supercomputers' to appear in the *Proceedings of After the First Three Minutes*, ed. S. Holt, V. Trimble, and C. Bennetti (AIP, 1991). Caltech Technical Report C^3P-961 [Warren:91a].

32. J. Salmon, 'Parallel Hierarchical N-Body Methods', California Institute of Technology PhD Thesis, December 1990. Caltech Technical Report C^3P-966 (SCCS-52, CRPC-TR90115).

33. J. Barnes and P. Hut, 'A Hierarchical $O(N \log N)$ Force Calculation Algorithm', *Nature*, **324**, p. 446 (1986).

34. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, 'Optimization by Simulated Annealing', *Science*, **220**, No. 4598, p. 671–680 (1983) [Kirkpatrick:83a].

35. G. C. Fox, 'A Review of Automatic Load Balancing and Decomposition Methods for the Hypercube' in *Numerical Algorithms for Modern Parallel Computer Architectures*, ed. M. Schultz (Springer-Verlag, 1988), p. 63–76. Caltech Technical Report C^3P-385b [Fox:88mm].

36. G. C. Fox and W. Furmanski, 'Load Balancing Loosely Synchronous Problems with a Neural Network' in *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 1*, ed. G. C. Fox (ACM Press, New York, 1988), p. 241–278. Caltech Technical Report C^3P-363b [Fox:88e].

37. R. D. Williams, 'Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations', *Concurrency: Practice and Experience*, **3**, No., 5, p. 457–481 (John Wiley and Sons, Ltd., England, 1991). Caltech Technical Report C^3P-913b [Williams:91a].

38. J. Flower, S. Otto, and M. Salama, 'Optimal Mapping of Irregular Finite Element Domains to Parallel Processors' in *Proceedings, Symposium on Parallel Computations and their Impact on Mechanics*, (ASME, Massachusetts, 1987). Caltech Technical Report C^3P-292b [Flower:87a].

39. J. J. Hopfield and D. W. Tank, 'Computing with Neural Circuits: A Model', *Science*, **233**, p. 625 (1986) [Hopfield:86a].

40. J. Koller, 'A Dynamic Load Balancer on the Intel Hypercube' in *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 1*, ed. G. C. Fox (ACM Press, New York 1988), p. 279–284. Caltech Technical Report C^3P-497 [Koller:88a].

41. J. Barhen, S. Gulati, and S. S. Iyengar, 'The Pebble Crunching Model for Load Balancing in Concurrent Hypercube Ensembles', in *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 1*, ed. G. C. Fox (ACM Press, New York 1988), p. 189–199. Caltech Technical Report C^3P-610 [Barhen:88b].

42. W. K. Chen and E. F. Gehringer, 'A Graph-Oriented Mapping Strategy for a Hypercube', in *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 1*, ed. G. C. Fox (ACM Press, New York 1988), p. 200–209 [Chen:88a].

43. F. Ercal, J. Ramanujam, and P. Sadayappan, 'Task Allocation onto a Hypercube by Recursive Mincut Bipartitioning', in *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 1*, ed. G. C. Fox (ACM Press, New York 1988), p. 210–221 [Ercal:88a].

44. G. C. Fox, 'A Graphical Approach to Load Balancing and Sparse Matrix Vector Multiplication on the Hypercube', in *Numerical Algorithms for Modern Paral-*

lel Computer Architectures, ed. M. Schultz (Springer-Verlag, 1988) p. 37–62. Caltech Technical Report C^{3}P-327b [Fox:88nn].

45. M. Livingston and Q. F. Stout, 'Distributing Resources in Hypercube Computers', in *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 1*, ed. G. C. Fox (ACM Press, New York 1988), p. 222–231 [Livingston:88a].

46. F. Ercal, 'Heuristic Approaches to Task Allocation for Parallel Computing', Ohio State University PhD Thesis, 1988.

47. A. Pothen, H. Simon, and K.-P. Liou, 'Partitioning Sprase Matrices with Eigenvectors of Graphs', *SIAM J. Matrix Anal. Appl.*, **11**, No. 3, p. 430–452 (July 1990).

48. H. Simon, 'Partitioning of Unstructured Mesh Problems for Parallel Processing' in *Proc. Conf. Parallel Methods on Large Scale Structural Analysis and Physics Applications*, (Permagon Press, 1991).

49. G. A. Lyzenga, A. Raefsky, and B. Nour-Omid, 'Implementing Finite Element Software on Hypercube Machines' in *The Third Conference on Hypercube Concurrent Computers, Volume 2* , ed. G. C. Fox (ACM Press, New York 1988), p. 1755–1761. Caltech Technical Report C^{3}P-594 [Lyzenga:88a].

50. R. D. Williams, 'DIME: A Programming Environment for Unstructured Triangular Meshes on a Distributed-Memory Parallel Processor', in *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 1*, ed. G. C. Fox (ACM Press, New York 1988), p. 1770–1787. Caltech Technical Report C^{3}P-502 [Williams:88a].

51. R. Williams, 'Free-Lagrange Hydrodynamics with a Distributed-Memory Parallel Processor', *Parallel Computing*, **7**, p. 439–443 (1988). Caltech Technical Report C^{3}P-424b [Williams:88d].

52. D. W. Walker, 'Characterizing the Parallel Performance of a Large-Scale, Particle-In-Cell Plasma Simulation Code', *Concurrency: Practice and Experience*, **2**, No. 4, p. 257–288 (John Wiley and Sons, Ltd., England, 1990). Caltech Technical Report C^{3}P-912 [Walker:90b].

53. N. Mansour and G. C. Fox, 'Allocating Data to Multicomputer Nodes by Physical Optimization Algorithms for Loosely Synchronous Computations', *Concurrency: Practice and Experience*, (John Wiley and Sons, Ltd., England, 1992), to be published.

54. N. Mansour and G. C. Fox, 'A Hybrid Genetic Algorithm for Task Allocation in Multicomputers' in *The International Conference on Genetic Algorithms and Applications*, p. 466-473 (July 1991).

55. For more details of the scattered decomposition applied to Finite Element Problems, see R. Morison and S. Otto, 'The Scattered Decomposition for Finite Element Problems', *Journal of Scientific Computing*, 2, No. 1, p. 59–76 (1986). Caltech Technical Report C^{3}P-286. Similar ideas are used in some types of matrix algorithms (see reference 3 above).

56. G. C. Fox and W. Furmanski, 'Optimal Communication Algorithms for Regular Decompositions on the Hypercube' in *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 1*, ed. G. C. Fox (ACM Press, New York, 1988), p. 648–713. Caltech Technical Report C^{3}P-314b [Fox:88h].

57. G. Fox, 'Iterative Full Matrix-Vector Multiplication on the Hypercube', Caltech Technical Report C^{3}P-336 (1986) [Fox:86e].

58. G. C. Fox and W. Furmanski, 'Hypercube Algorithms for Neural Network Simulation the Crystal Accumulator and the Crystal Router' in *The Third conference on Hypercube Concurrent Computers and Applications, Volume 1*, ed. G. C. Fox (ACM Press, 1988), p. 714–724. Caltech Technical Report C^{3}P-405b [Fox:88g].

59. G. V. Wilson and G. C. Pawley, 'On the Stability of the Travelling Salesman Problem Algorithm of Hopfield and Tank', *Biol. Cybern.*, 58 p. 63–70 (1988) [Wilson:88a].

60. G. C. Fox and J. G. Koller, 'Code Generation by a Generalized Neural Network: General Principles and Elementary Examples', *Journal of Parallel and Distributed Computing*, 6, No. 2, p. 388–410 (1989). Caltech Technical Report C^{3}P-650 [Fox:88cc].

61. R. Battiti, 'Surface Reconstruction and Discontinuity Detection: A Fast Hierarchical Approach on a Two-Dimensional Mesh' in *The Fifth Distributed Memory Computing Conference, Volume 1*, ed. D. W. Walker and Q. F. Stout (IEEE Computer Society Press, California, 1990), p. 184–193. Caltech Technical Report C^{3}P-900 [Battiti:90a].

62. W. Furmanski and G. C. Fox, 'Integrated Vision Project on the Computer Network' in *Biological and Artificial Intelligence Systems*, p. 509–527 (ESCOM Science Publishers B. V., The Netherlands, 1988). Caltech Technical Report C^{3}P-623 [Furmanski:88c].

63. G. C. Fox, 'What Have We Learned from Using Real Parallel Machines to Solve Real Problems?' in *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 2*, ed. G. C. Fox, p. 897–955 (ACM Press, New York, 1988). Caltech Technical Report C^{3}P-522 [Fox:88b].

64. D. Jefferson, B. Beckman, L. Blume, M. DiLoreto, P. Hontalas, P. Reiher, K. Sturdevant, J. Tupman, J. Wedel, F. Wieland, and H. Younger, 'The Status of the

Time Warp Operating System' in *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 1*, ed. G. C. Fox, p. 738–744 (ACM Press, New York, 1988). Caltech Technical Report C^{3}P-627 [Jefferson:88a].

65. R. Durbin and D. Wilshaw, 'An Analogue Approach to the Traveling Salesman Problem using an Elastic Net Method', *Nature*, **326**, p. 689–691 (1987) [Durbin:87a].

66. C. Peterson and B. Söderberg, 'A New Method for Mapping Optimization Problems Onto Neural Networks', *Int. J. Neural Syst.*, **1**, p. 3-22 (1989).

67. A. Durbin, R. Szeliski, and A. Yuille, 'An Analysis of the Elastic Net Approach to the Travelling Salesman Problem', *Neural Computation*, **1**, p. 348–358 (1989).

68. A. L. Yuille, 'Generalized Deformable Templates, Statistical Physics and Matching Problems', *Neural Computation*, **2**, p. 1–24 (1990).

69. P. Simic, 'Statistical Mechanics as the Underlying Theory of 'Elastic' and 'Neural' Optimizations', *Network*, **1**, p. 89–103 (IOP Publishing, Ltd., United Kingdom, 1990). Caltech Technical Report C^{3}P-787 [Simic:90a].

70. P. Simic, 'Constrained Nets for Graph Matching and Other Quadratic Assignment Problems', *Neural Computation*, **3**, p. 268–281 (1991). Caltech Technical Report C^{3}P-973 [Simic:91a].

71. G. C. Fox, 'Applications of the Generalized Elastic Net to Navigation', Caltech Technical Report C^{3}P-930 (1990) [Fox:90k].

72. G. C. Fox, 'FortranD as a Portable Software System for Parallel Computers' in *The Proceedings of Supercomputing USA/Pacific 91*, (1991). Syracuse Technical Report SCCS-91 [Fox:91d] (CRPC-TR91128).

73. G. C. Fox, 'Parallel Problem Architectures and Their Implications for Portable Parallel Software Systems', presentation at DARPA Workshop, Rhode Island (February 28, 1991). Caltech Technical Report C^{3}P-967 [Fox:91c] (SCCS-78, CRPC-TR91120).

74. P. J. Denning and W. F. Tichy, 'Highly Parallel Computation', *Science*, **250**, p. 1217–1222 (1990) [Denning:90a].

75. G. C. Fox, 'Parallel Computing Comes of Age: Supercomputer Level Parallel Computations at Caltech', *Concurrency: Practice and Experience*, **1**, No. 1, p. 63–103 (John Wiley and Sons, Ltd., England, 1989). Caltech Technical Report C^{3}P-795 [Fox:89n].

76. W. D. Hillis, 'The Connection Machine', *Scientific American*, **256**, p. 108, 1987 [Hillis:87a].

77. C. F. Baillie, R. D. Brickner, R. Gupta, and L. Johnsson. 'QCD with Dynamical Fermions on the Connection Machine', in *Proceedings of Supercomputing '89* (ACM Press, 1989), p. 2–9. Caltech Technical Report C^3P-786 [Baillie:89e].

78. C. F. Baillie, 'Lattice QCD: Commercial vs. Home-grown Parallel Computers' in *The Fifth Distributed Memory Computing Conference*, Volume 1, ed. D. W. Walker and Q. F. Stout (IEEE Computer Society Press, California, 1990), p. 397–405. Caltech Technical Report C^3P-878 [Baillie:90f].

79. E. W. Felten and S. W. Otto, 'A Highly Parallel Chess Program', in *Proceedings of International Conference on Fifth Generation Computer Systems 1988*, p. 1001–1009 (ICOT, Japan, 1988). Caltech Technical Report C^3P-579c [Felten:88i].

80. F. Wieland, L. Hawley, A. Feinberg, M. DiLoreto, L. Blume, J. Ruffles, P. Reiher, B. Beckman, P. Hontalas, S. Bellenot, and D. Jefferson, 'The Performance of a Distributed Combat Simulation with the Time Warp Operating System', *Concurrency: Practice and Experience*, **1**, No. 1, p. 35–50 (John Wiley and Sons, Ltd. England, 1989). Caltech Technical Report C^3P-798 [Wieland:89a].

81. G. C. Fox, P. Hipes, and J. Salmon, 'Practical Parallel Supercomputing: Examples from Chemistry and Physics', in *Proceedings of Supercomputing '89*, p. 58–70 (ACM Press, 1989). Caltech Technical Report C^3P-818 [Fox:89t].

82. D. L. Meier, K. C. Cloud, J. C. Horvath, L. D. Allan, W. H. Hammond, and H. A. Maxfield, 'A General Framework for Complex Time-Driven Simulations on Hypercubes', Caltech Technical Report C^3P-761, 1989 [Meier:89a].

83. T. D. Gottschalk, 'Concurrent Multi-Target Tracking' in *The Fifth Distributed Memory Computing Conference*, Volume I, ed. D. W. Walker, Q. F. Stout (IEEE Computer Society Press, California, 1990), p. 85–88. Caltech Technical Report C^3P-908 [Gottschalk:90b].

84. J. J. Dongarra, J. Du Croz, S. Hammaling, and R. J. Hanson, 'An Update Notice on the Extended BLAS, *ACM Signum Newsletter*, **21**, No. 4, p. 1, 1987 [Dongarra:87c].

85. J. J. Dongarra in *Proceedings of ICS87, International Conference on Supercomputing*, ed. C. Polychronoupolos (Springer-Verlag, New York, 1988) [Dongarra:88c].

86. J. Demmel, 'LAPACK: A Portable Linear Algebra Library for High-performance Computers' *Concurrency: Practice and Experience*, **3**, No. 6, p. 655-666 (John Wiley and Sons, Ltd., England, 1991) [Demmel:91a].

87. G. C. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu, 'FortranD Language Specification', Syracuse Technical Report SCCS-42c, 1991 [Fox:91i] (CRPC-TR90079).

88. M.-Y. Wu and G. C. Fox, 'Fortran 90D Compiler for Distributed Memory MIMD Parallel Computers', Caltech Technical Report C^{3}P-948c [Wu:91b] (SCCS-88b), 1991.