C^3P-392

Domain Decomposition in Distributed and Shared Memory Environments

I: A Uniform Decomposition and Performance Analysis for the NCUBE and JPL Mark IIIfp Hypercubes

Geoffrey C. Fox †
Caltech Concurrent Computation Program
Mail Code 158-79
California Institute of Technology
Pasadena, CA 91125
June 8, 1987

Invited Paper at ICS 87, International Conference on Supercomputing, June 8-12, 87, Athens, Greece. To be published as a *Lecture Note in Computer Science* by Springer-Verlag, and edited by Constantine Polychronopoulos.

Abstract:

We describe how explicit domain decomposition can lead to implementations of large scale scientific applications which run with near optimal performance on concurrent supercomputers with a variety of architectures. In particular, we show how one can discuss from a uniform point of view two architectural characteristics; distributed memory and hierarchical memory where a large relatively slow memory is buffered by a faster cache or local memory. We consider two hypercubes in particular; the commercial NCUBE and JPL's Mark IIIfp with hierarchical memory at each node of a hypercube. We remark on the application of these ideas to other architectures and other concurrent computers. We present a performance analysis in terms of basic parameters describing the hardware and the applications.

I: Introduction

This paper is the first of a series from Caltech that will discuss some of the issues in developing decomposition and software tools for concurrent supercomputers. In particular we need to develop implementations of major scientific problems that run well on concurrent supercomputers with a variety of different architectures. At Caltech, there is widespread support among many of the computational scientists and engineers for the use of concurrent supercomputers [1]. However, there is clearly no agreement among either research groups or commercial vendors as to the "right" architectures either at the present or in the future. It is important that we can find techniques that will allow (Caltech) users to develop code which will be useable not only on today's prototypes but will not need major revision for future machines.

Presently available concurrent supercomputers suitable for scientific computations can be divided into three classes [2-4]

- Small grain size, SIMD, Distributed Memory: Such as the ICL DAP, Goodyear MPP, and Connection Machine.
- Large grain size, MIMD Distributed Memory: Such as the various hypercubes and the transputer based systems like the MEIKO Computing Surface.

† Work supported in part by DOE grant DE-FG03-85ER25009 and DE-AC03-85ER40050, the Program Manager of the Joint Tactical Fusion Office, and the ESD division of the USAF. Also, grants from IBM, SANDIA and the Parsons and the System Development Foundations.

- Large grain size, MIMD Shared Memory: Such as the CRAY, ETA, ELXSI, ALLIANT, ENCORE, SEQUENT, CEDAR, BUTTERFLY, and RP3.

Here we will concentrate on a version of domain decomposition that is appropriate for the large grain size machines and not consider the SIMD architectures. This is not due to lack of interest - the SIMD machines have been very successful in many scientific problems - rather I am largely constrained by ignorance. Further, we will not consider dataflow which could in the future be an interesting approach to supercomputers. In order to focus the paper further, we will not discuss shared memory machines in general but rather concentrate on one aspect of some of these machines - namely a hierarchical memory where good performance requires data to lie in a cache or local memory and not fetched from a relatively slow shared memory each time. We note that a memory hierarchy is difficult to avoid in high performance machines with pipelined floating point units. We will not discuss important issues such as the contention in a shared memory access network nor will we consider alternative programming methodologies such as automatic (user or computer generated) parallelization of *do* and *for* loops even though these techniques have had important successes [28, 29, 32].

I have chosen domain decomposition because it is the natural and at present required method for programming the hypercube. Further, we understand it reasonably well and shared memory machines typically support it elegantly [5] whereas the favored shared memory methodologies are not easy to port to the simpler hardware of the distributed memory machines. I will sharpen the hypercube discussion further by considering two particular machines.

- The commercial NCUBE hypercube with 1/2 megabyte of memory on each node and up to 1024 nodes. Our initial 512 node machine has 0.25 gigabytes of memory and 0.05 gigaflops peak performance.
- The so-called Mark IIIfp hypercube under construction at Caltech's Jet Propulsion Laboratory (JPL). This will have 128 nodes each with 4 megabytes of slow memory and a high performance WEITEK XL chip based floating point unit buffered by a 64 - 128K bytes data cache i.e., each node has a hierarchical memory. The total machine has 0.5 gigabytes of memory and up to 1 gigaflop performance.

We expect these two machines to be the initial hardware facility around which we will build our new "Concurrent Supercomputing Initiative at Caltech" or CSIC [4,6].

In Sec. II, we describe the target machines in greater detail and introduce and quantify the necessary hardware parameters to measure performance. In the third section, we describe domain decomposition, its application to distributed and hierarchical memories and relation to the theory of complex systems [2, 7].

The heart of the paper lies in Sec. IV where we analyze several specific scientific algorithms including particle dynamics, iterative methods for partial different equations (PDE), fast fourier transform (FFT), matrix inversion and multiplication and finally neural network simulation. We conclude in Sec. V.

II: Characterization of Concurrent Machines

The hardware architectures considered here are shown in Fig. 1. In 1(a) we show a generic sequential machine with a cache as a buffer to a main memory. It should be noted that this conventional machine is not the focus of this article; however, the techniques described here could allow coding such machines in a way that makes excellent use of the cache. In Figs. 1(b) and (c), we generalize this architecture in two ways. Firstly in 1(b), we show a hierarchical shared memory machine with a group of C.P.U.'s connected by an unspecified network to a large global shared memory. This model is a reasonably accurate description of such machines as the CRAY-2, ETA-10, ELXSI, BUTTERFLY, and RP3 among others. This design, or variants of it, is more or less required if one either has a large number of nodes (as in the BUTTERFLY or RP3) or very fast C.P.U.'s (as in the other cases). We note that the memory denoted as "cache" in Fig. 1 should often be more properly called local memory. The functions of cache and local memory are similar; both provide faster access to variables than is possible from the main memory. Our discussion will apply to both memory constructs but is perhaps more natural for local memory which is user (software) controlled rather than the hardware controlled cache. We will hereafter use the term "cache" to describe the dual concept. The necessity of a multistage network connecting nodes to the global memory implies the importance of the "cache" for machines like the RP3 and BUTTERFLY with very many nodes. The use of a fast C.P.U., even with a small number of processors, also typically needs a "cache" to get good performance. Presumably the future will see shared memory machines with many very high-performance nodes; these may depend on the "cache" to an even greater extent. We note that there is no reason for the memory hierarchy to only have two levels; many machines have more. For instance, the RP3 has global memory, local memory, and a true cache. The analysis presented here can be extended to a multilevel hierarchy or just applied to one part of it. For instance on the RP3, the high-speed network gives good access to the shared memory and it would probably be most natural to apply the analysis here with the "cache" being the hardware cache on each node and the local and global memory being lumped together at a single level.

In Fig. 1(c), we show a distributed memory architecture. The initial hypercubes had rather simple nodes

but newer machines have gone to hierarchical node designs to obtain high performance. Examples of the latter include the INTEL iPSC-VX and the new JPL Mark IIIfp design. We will concentrate on the latter here as we understand it better and INTEL's current tools do not, we believe, allow the VX hypercube to be programmed in the fashion we will suggest. The Mark IIIfp uses complex nodes with large memory and potentially high performance on each node. These NCUBE and Mark IIIfp nodes are contrasted in Fig. 2 and pictured in Figs. 3 and 4.

The NCUBE does actually have a small on chip cache but for our purposes it should be viewed as a simple uni-level memory at each node. We need a reasonably large "cache" - at a minimum the 64 - 128K bytes of the Mark IIIfp - for our considerations to be relevant. This will become clearer later when in Secs. III and IV we quantify the role of the "cache" size. Similarly, in the transputer based system, the on chip memory of 2K bytes is probably usually too small to be used as a "cache" in the sense advocated in this paper.

In Fig. 5 and Table 1, we introduce three basic parameters t_{calc}, t_{mem} and t_{comm} which we use for our performance analysis. We will restrict ourselves to considering 32-bit arithmetic as this is sufficient for many calculations and the WEITEK XL chip set used on the Mark IIIfp currently only supports 32-bit arithmetic. 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 word between nodes of a hypercube and t_{mem} the time taken to send a word back and forth between "cache" and main memory. We now need to make several comments and caveats on these parameters.

Firstly, we note that these three parameters are a very incomplete description of the hardware. For this paper, we will need the "cache" and node memory sizes as well. We will describe these in application dependent fashion as n_{cache}, n_{node}, and n_{total} for the number of basic entities (e.g., matrix elements in a matrix problem or particles in a dynamics problem) that can be held in the "cache", node of hypercube or total memory respectively. 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. For the Mark IIIfp, node to node communication (governed by t_{comm}) is concurrent with calculation while "cache" to node memory transfers can *not* be overlapped with calculation. For the NCUBE, communication is presently not concurrent with calculation.

The NCUBE has 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 Mark IIIfp, ETA-10, and CRAY-2 can expect very different values of t_{calc} on different applications. We have listed the approximately smallest possible value of t_{calc} as this will be the pacing value for the performance analysis. The techniques described here 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 paper.

The parameter t_{mem} has been defined in a somewhat unnatural fashion in Fig. 5 as the time to read *and* write a word between "cache" and main memory. This makes the definition symmetric with t_{comm} and indeed if one uses a shared memory machine to emulate a distributed memory environment, then with our definition t_{mem} is the appropriate value of t_{comm} to use to describe node to node communication in the emulation. This assumes that in the emulation, a hypercube node process is fully contained in the "cache". We will describe in Sec. III how one takes care of the case when the problem is too large for all the processes to fit into "cache". In Sec. IV, we will use $1/2\ t_{mem}$ as the transfer time when we are just reading or writing. We again emphasize that the shared memory value and indeed our later discussion of t_{mem} completely ignores any contention when accessing the global memory.

The Mark IIIfp can be seen to have interesting multi-level hierarchy with "cache" $\rightarrow$ node memory $\rightarrow$ other nodes' memories. The relevant communication speeds t_{comm}, t_{mem} between the three levels have comparable values although as we will see they do enter the performance analysis in related but rather different ways. The Mark IIIfp illustrates another tactically important issue controlling performance. Namely, in the default node, variables written to "cache" are also stored in main memory. As we will find that this can degrade performance, the Mark IIIfp has the capability to temporarily disable the "write through" nature of the cache and so allows the communication between "cache" and main memory to be under user control. This control structure is implied for machines where the "cache" is designed as a local memory. Referring back to our previous discussion of concurrency, we note that on the Mark IIIfp direct loads of either the cache or local memory cannot be overlapped with calculation. The storing of variables into main memory is similarly non overlapped with one exception; automatic write through from cache is transparently overlapped with other operations as long as a small "pending write" buffer of three requests is not filled i.e., if the writes are sparse, they are overlapped. We will not use this overlap feature in our discussion although it would be possi-

ble to improve performance in many cases if full overlap had been allowed in the Mark IIIfp design.

In discussing the hypercubes, we have listed t_{comm} as the node to node transmission time; messages between nodes that are not directly connected in the hypercube topology will be characterized by a larger value of t_{comm}. We will accommodate this by reflecting this as an application dependent effect which will be seen in Sec. IV as a $1/2 \log N_{proc}$ factor in the communication overhead for the FFT. We will use N_{proc} as the number of physical processors throughout this paper.

We will use the term *hypercube* broadly in this paper to include simple nodes like the NCUBE or more complex designs such as the Mark IIIfp. When we wish to single out machines like the NCUBE, we will use the term *homogeneous hypercube* while terming machines like the Mark IIIfp as a *hierarchical hypercube.*

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 [9].

III: Domain Decomposition and Complex Systems

One can formulate concurrent computation as a mapping of a problem onto a computer [2, 7, 8]. We consider both the problem and the computer as *complex systems*; for our purposes, these can be considered as a set of, in general, dynamically interconnected entities. The performance of a particular implementation can be related to the structure of the systems describing the problem and computer [2, 3, 9]. In this paper, we are only considering large grain size concurrent computers i.e., each node has substantial memory. In this case, it is natural to consider the problem, or more precisely its underlying complex system, as being divided up into subdomains which we will call *grains.* At any one time, each node of the concurrent computer is responsible for a single grain. In this context, *domain decomposition* is the division of the problem, or typically its defining data domain, into appropriate grains. We can make this clearer by considering the map of the problem onto the computer in more detail. We can isolate four stages. The use of a vectoring or parallelizing computer corresponds to the map:

$$Problem \rightarrow Algorithm \rightarrow Code \rightarrow Compiler\ Generated\ Decomposition \rightarrow Concurrent\ Computer \quad (1)$$

i.e., performing the map onto the concurrent computer between the last two stages.

We will consider domain decomposition corresponding to the map being generated by:

$$Problem \rightarrow Algorithm \rightarrow Domain\ Decomposition \rightarrow Code \rightarrow Concurrent\ Computer \quad (2)$$

where one forms the basic grains not from the code as in (1) but from the basic algorithm or problem in (2). The advantage of (1) is that one directly uses existing software on concurrent machines whereas in (2) it is implied that one must generate code that describes individual grains. (1) is presently only practical on shared memory machines and indeed the ability to use this programming methodology is a key advantage of this architecture over distributed memory systems.

The hypercube and other distributed memory machines are traditionally programmed by domain decomposition as described above. This methodology has advantages of generality (it is potentially useable over a broad range of architectures) but it does require significantly more user involvement in the concurrency and software. In particular, there is no easy way to make use of existing sequential software and at Caltech, we have typically recoded our hypercube applications from scratch. As will become clear, we are interested in domain decomposition for the shared memory architecture, not only because it allows the porting of hypercube targeted programs but also because it may give near optimal performance for some problems on shared memory machines.

In a series of papers, we have developed a set of optimization methods, in particular, neural networks and simulated annealing, for choosing the optimal definition of the grains for a particular problem [7, 8, 10, 11, 12]. This work was in the context of a simple hypercube as the target machine. We will not discuss this research here but note that the discussion of the current paper lays the groundwork for extending the earlier work to give optimal domain decompositions for complicated hypercubes like the Mark IIIfp and some shared memory machines.

The optimal decomposition for a simple hypercube divides the problem into grains satisfying two criteria:

- The amount of communication between grains is minimized.
- Each node of the hypercube does the same amount of work; measured in terms of calculational complexity, each grain is the same "size".

We can now see qualitatively the possible importance of domain decomposition for hierarchical memory systems. The latter perform well as long as references to memory are largely local to the "cache" and do not require access to the slower main memory. This is clearly analogous to minimizing communication in a distributed memory machine. We will see that there are important quantitative differences in the "locality" constraints for hierarchical and distributed memories. However, the purpose of the current paper is to explore and explain their similarities and derive the ground rules for a uniform user interface which can be implemented well on either architecture.

The second constraint of the hypercube decomposition given above corresponds to load balancing the work of the nodes. We will not stress this here as it is not central to our discussion. We will choose as our examples *regular* problems for which load balancing is not a difficulty [2, 7]. In the future explicit realizations of the ideas presented here the constraints of load balancing may, in fact, be very important. In particular, an elegant dynamic load balancer for the hypercube appears to require a full multitasking environment with each node responsible for several (~ 10) processes or grains [13-15]. We will adopt a simpler point of view here with, in the case of the NCUBE and other hypercubes with simple nodes, only one process per node. This is sufficient for the problems discussed here and will be pedagogically clearer. In fact, the multitasking environment already introduces hierarchy into a simple hypercube environment as now one has different process to process communication speeds depending on whether or not the communicating processes lie in the same or different nodes. Thus the multitasking environment will tend to unify the ideas across architectures and we will include it in future discussions. In this paper, we are not attempting to discuss the detailed user environment and explicit software implementation but rather the functional structure of the software environment as "seen" by the machine. We will address the (very important) implementation issues elsewhere.

Let us focus our ideas by considering a specific problem of particle dynamics with a short-range interaction. An example is shown as a complex system in Fig. 6. Typically, the computation involves particles linked by a force and the number of such links measures the calculational complexity. Consider the underlying complex system as a graph whose nodes are the particles and nodes are linked if and only if the corresponding particles interact with each other. Then the calculation complexity is measured by the number of edges in the graph. Domain decomposition will divide the graph into grains (subgraphs) such that each grain has approximately equal numbers of edges and we minimize the number of edges that cross grain boundaries. When implemented on a homogeneous hypercube, the ratio f_C of communication to calculation is measured by the ratio of the number of edges crossing grain boundaries to the total inside a given grain. In many previous papers, we have shown that this leads to the result [2,3,7,9]

$$f_C = \frac{constant}{n_{grain}^{1/d}} \frac{t_{comm}}{t_{calc}} \tag{3}$$

Where n_{grain} is the number of entities (number of nodes of graph i.e., particles in our example) in each grain and the parameters t_{comm} and t_{calc} were introduced in Sec. II.

The application dependent *constant* in Eq. (3) is typically of order unity. The *connection dimension d* has been defined generally in Refs. [2] and [7] and coincides with the geometric dimension for short-range or nearest-neighbor problems.

In architectures where communication is serial with calculation such as all the initial (Caltech's Cosmic Cube and Mark II, INTEL iPSC, NCUBE) hypercubes, one aims for $f_C \leq 0.25$ which was obtained for the initial Caltech applications as long as

$$\frac{t_{comm}}{t_{calc}} \leq 3 \; ; \; non \; overlapped \tag{4}$$

The goal $f_C \leq 0.25$, corresponds to a *speed-up* $S = \epsilon N_{proc}$ with efficiency ϵ given as

$$\epsilon = \frac{1}{1+f_C} \; ; \; non \; overlapped \tag{5}$$

$$\geq 0.8 \text{ for } f_C \leq 0.25$$

This is a phenomenological result averaged over the values of n_{grain}, d, and *constant* seen in typical applications. In table 2, we give a current list of Caltech hypercube applications to indicate the problem areas from which our results have been obtained. We do not wish to discuss here the many issues underlying the validity of (3) but rather refer the reader to the detailed discussions in Refs. [2, 3, 7, 9]

In the case where communication is overlapped with calculation, one can afford an order of magnitude more message traffic with a typical goal of $f_C \leq 1.25$ or the less stringent constraint:

$$\frac{t_{comm}}{t_{calc}} \leq 15 \; ; \; overlapped \tag{6}$$

with efficiency

$$\epsilon = \min(1, 1/f_C)\ ;\ overlapped \qquad (7)$$
$$\geq 0.8 \text{ for } f_C \leq 1.25$$

We see that the difference between (6) and (4) partly explains why the Mark IIIfp and NCUBE can both give good performance even though the former has an order of magnitude larger value for t_{comm}/t_{calc}. As we see from (3) and is explored in detail in Ref. [16], the Mark IIIfp will also have the value of f_C lowered by the larger value of n_{grain} allowed by the larger memory of the Mark IIIfp. Explicitly for systems of dimension d=3, we see that $1/n_{grain}^{1/4}$ is decreased by a factor $(4/0.5)^{1/3} = 2$ in comparing the Mark IIIfp to the NCUBE.

Now return to the original particle dynamics example and consider its implementation on a variety of architectures. In the case of a *homogeneous hypercube* we have already described how we divide the particles into groupings and assign each grouping to an individual hypercube node. We have a number N_{grain} of grains equal to the number of processors N_{proc} in the concurrent machine. Using the notation of Sect. II, we also have the number of particles in each grain, $n_{grain} = n_{node}$ and a total number n_{total} of particles given by

$$n_{total} = N_{proc}\ n_{grain} \qquad (8)$$

Now consider a hierarchical shared memory machine such as that pictured in Fig. 1(b). As explained earlier, we would naturally expect best performance if memory references were largely local to each "cache" and this will be achieved if one can assume that each "cache" contains a complete grain. However, there is a crucial complication that in general, one can not expect that the full problem can be held in the "caches". Some fraction of it can be in "cache" with the remainder "waiting" in the large shared memory. In contrast, it is reasonable to assume that the homogeneous hypercube will be able to contain the full problem in the aggregate of it's nodes' memories. In general, we will define a *grain* so that it can be contained in the "cache" or lowest level of the memory hierarchy under consideration; this *grain* can contain n_{grain} members (nodes of graph). We can now **define** an effective or virtual number N_{grain} of processors so that the total system can be contained in N_{grain} nodes. Thus

$$n_{total} = N_{grain} n_{grain} \qquad (9)$$

In general, N_{grain} is larger than the real number N_{proc} of nodes and we can assume that the grains have been defined so that the ratio $r_{virtual} = N_{grain}/N_{proc}$ is an integer.

We can now summarize three relevant circumstances:

Homogeneous Hypercube

$$n_{node} = n_{grain} \text{ entities fit in a single node} \qquad (10a)$$
$$N_{grain} = N_{proc}$$
$$r_{virtual} = 1$$

Hierarchical Shared Memory Machine

$$n_{grain} \text{ entities fit in a single "cache"} \qquad (10b)$$
$$N_{grain} = n_{total}/n_{grain}$$
$$r_{virtual} \geq 1$$
$$(r_{virtual} - 1)N_{grain} \quad \textit{processes "waiting" for loading into "cache"}$$

Hierarchical Hypercube

$$n_{grain} \text{ entities fit in a single "cache"} \qquad (10c)$$
$$N_{grain} = n_{total}/n_{grain}$$
$$r_{virtual} \geq 1 \text{ grains in each node}$$
$$n_{node} = r_{virtual}\ n_{grain} \text{ entities fit in a single node}$$

The computer system is to be viewed as having N_{grain} virtual nodes which, except in the case of the homogeneous hypercube, is typically larger than the number of physical processors N_{proc}. For small problems, one could find that $r_{virtual}$ was unity even for the hierarchical designs. In this case, hierarchical shared memory or distributed memory architectures can be considered by the same performance analysis given ear-

lier in Eqs. (3 to 7) for homogeneous hypercubes. In this case, one can substitute t_{mem} for t_{comm} on these equations when considering the shared memory case. However, it is the purpose of this paper to consider the more general and interesting case $r_{virtual} > 1$.

Consider an example of the consequence of these ideas comparing the 1024 node NCUBE, each node having 0.5 megabytes of memory with the 128 node Mark IIIfp, each node having 4 megabytes of memory. Normally one considers the latter machine as having larger grain size but consider the use of these machines to solve a problem requiring the full 512 megabytes available on each machine. The NCUBE has 1024 grains each of 0.5 megabyte size but the Mark IIIfp with $r_{virtual} = 32$ has 4096 grains each of 0.125 megabytes! The addition of the floating point unit has, in fact, given the Mark IIIfp a smaller grain size and required one to decompose the problem into more and not less (than for the NCUBE) grains. Note that if the total problem size had been 128 megabytes, then both machines would be used with 1024 grains, each of 128 kilobytes.

Let us consider some of the issues from a more fundamental, or perhaps philosophical, point of view. In the particle dynamics example, we have a spatially distributed system which nature evolves simultaneously in time. On a homogeneous hypercube, the spatial complex system corresponding to the problem is directly mapped onto a spatially distributed memory of the hypercube. Then as in nature, each node evolves different parts of the spatial system simultaneously. In this case, we have a rather clean association.

$$\begin{aligned} &\textit{Space in Problem} \rightarrow \textit{Space in Homogeneous Hypercube} \\ &\textit{Time in Problem} \rightarrow \textit{Elapsed Time in Computer Execution} \end{aligned} \tag{11a}$$

This association is particularly precise in the case when each node of the computer holds a single particle. In the normal case, where each grain has several particles, then we have an intermediate situation when the spatial system within each node is evolved sequentially in time by the node; i.e., we have partially mapped the spatial extent of the system into a temporal extent in the computer implementation.

To extend the above picture, we generalize the concept of a complex system to include both the spatial and temporal aspects of a problem. This is illustrated in Figs. 7 and 8, and in the particle dynamics case, we would consider the extended complex system as the physical system or a graph generated by, for instance, a space-time region defined by some condition such as:

$$|\underline{x} - \underline{x}_0| \leq r \tag{11b}$$

In Fig. 7, we show this for one spatial dimension and a regular lattice. In our previous work [2,7], we have only needed to consider the spatial aspects of complex systems because of the rather clean correspondence of space and time, expressed in Eq. (11a), between the computer and problem present for the homogeneous hypercube.

Returning to Fig. 1, we now see that the sequential computer shown in Fig. 1(a) corresponds to:

$$\begin{matrix} \textit{Space and Time} \\ \textit{in Problem} \end{matrix} \rightarrow \begin{matrix} \textit{Elapsed Time in} \\ \textit{Sequential Computer} \end{matrix} \tag{11c}$$

Typically one cycles through individual particles and processes them sequentially. If we now consider the hierarchical memory systems 1(b) and 1(c), we find the intermediate situation:

$$\begin{aligned} &\textit{Space in Problem} \rightarrow \begin{matrix} \textit{Space and Time} \\ \textit{in Heirarchical Memory Machine} \end{matrix} \\ &\textit{Time in Problem} \rightarrow \textit{Time in Computer} \end{aligned} \tag{11d}$$

(11d) would reduce to (11a) in the degenerate case where the problem can be contained in the "caches" - summed over nodes.

Above we have pointed out that varying degrees of parallelism correspond to mapping spatial aspects of the problem into different mixes of space and time on the computer system. Fig. 8 illustrates an important technical reason to introduce a complex system extended in space and time. For a homogeneous hypercube, communication costs are related to graph edges crossing spatial boundaries of the system. For the "cache" based architectures in Fig. 1(b) and 1(c) we will need to load the initial value of the system at the time t_0 and store back in main memory after evolution to time t_1. We see that this load and store correspond to the temporal boundaries of the system. t_{mem} and t_{comm} correspond respectively to the costs associated with temporal and spatial limitation of the system.

Figure 8 makes it clear why we may need to disable "write through" on the Mark IIIfp and machines with comparable architectures. We will find cases where we need significant (t_1 many time steps larger than t_0) time extent to minimize "edge" effects corresponding to the boundaries formed by cache load and store. "Write through" typically implies that the system is stored to main memory after every time step.

We will now quantify this general picture with several examples in the next section. We will find similarities and differences between the temporal (t_{mem}) and spatial (t_{comm}) aspects of the problems. We will only summarize the results with the intention of providing a detailed analysis elsewhere.

IV: Examples

IVA: The Long Range Force or Complete Interconnect

This case is interesting because we will find low overheads from both the spatial and temporal boundaries in Fig. 8 with these two overheads having the same dependence on grain size n_{grain}. The generic problem, that we will consider, is the time evolution of a set of particles interacting with a long-range force; we use the brute force algorithm and not the faster FFT [2] or clustering method [17]. We calculate the force on each particle by summing the contribution of all others [2]. We now consider first homogeneous hypercubes and then hierarchical memory computers.

a) Homogeneous Hypercube

One decomposes the problem with an equal number n_{grain} of particles in each node [2]. Another set (identical copy) of the particles travels completely around the cube (which can be mapped into a ring) updating the mutual interaction as the travelers pass through the node containing the fixed particles. There is some care needed to ensure Newton's law of action and reaction is exploited and each interaction is only calculated once [2]. However, the performance analysis is straightforward and if at each step one transports M particles

$$Calculation\ Time\ T_{calc} \sim \frac{M\ n_{grain}}{2}\ t_{calc} \tag{12a}$$

$$Communication\ Time\ T_{comm} \sim M\ t_{comm} \tag{12b}$$

and the overhead f_C introduced in Sec. III, Eq. (3), is given by:

$$f_C \sim \frac{1}{n_{grain}}\ \frac{t_{comm}}{t_{calc}} \tag{12c}$$

independent of M.

b) Hierarchical Machines

Now M is the total number of particles cycled from main memory through the "cache". Each grain takes time

$$T^{(1)}_{cache} = n_{grain}\ t_{mem} \tag{13a}$$

to load and store. The travelling particles must be read from main memory and stored back with their force updated. This takes time:

$$T^{(2)}_{cache} = M\ t_{mem} \tag{13b}$$

These numbers should be compared with the identical values (12a) and (12b) for calculation and communication in the case of the hierarchical hypercube. We see that good performance requires that M be chosen large ($\geq n_{grain}$) but this is algorithmically possible and natural. Hence for this case, the calculation time (12a) is much larger than the "cache" overhead and one finds for $M \geq n_{grain}$ that:

$$f_H \sim \frac{1}{n_{grain}}\ \frac{t_{mem}}{t_{calc}} \tag{13c}$$

is the hierarchical memory access overhead to be compared with the communication overhead f_C in (12c). The latter needs to be added to f_H for the hierarchical hypercube which has both forms of overhead.

For the hierarchical shared memory machine, the natural value of M is n_{total} and for the hierarchical hypercube the smaller value $M = n_{node} = r_{virtual}\ n_{grain}$; in each case Eq. (13c) is valid.

We do find that care is needed to reduce f_H. One must cycle all particles through the "cache", i.e., choose a large value of M, in between reloading the "cache". Further, one needs to update each particle in the cache in a fashion that one only writes out the results after all M particles are considered. In the case of the Mark IIIfp, this implies that one disables the cache "write-through" feature until all M particles have travelled through.

Comparing Eqs. (12c) and (13c) with Eq. (3), we see that both exhibit our standard form with *connection dimension* $d = 1$ independent of the underlying topological structure of the space.

IVB: Two Dimensional Finite Difference

We will consider a finite difference solution to Poisson's equation, $\nabla^2 \phi = 4\pi\rho$, solved by an iterative (relaxation) method on a regular two dimensional mesh. This problem is simple but the discussion generalizes to other short-range problems including wave equations, image processing, particle dynamics, and Monte Carlo. The choice of two dimensions is not essential and at the end we will generalize to higher dimensions. We will use the simplest stencil where the value at the next iteration only depends on the original value and its four neighbors at the current step. This is shown in Fig. 9(a); as discussed in Ref. [2] and later in this paper, more complicated stencils such as that in 9(b), do **not** alter one's conclusions. We are not underestimating the difficult implementation issues in realistic problems, but we believe that our simple case embodies the essential issues for the performance estimate.

(a) Homogeneous Hypercube

One performs a simple two dimensional decomposition with each node containing a $\sqrt{n_{grain}} \times \sqrt{n_{grain}}$ submesh. A typical iteration takes a calculation time for each iteration of

$$T_{calc} = 5\, n_{grain}\, t_{calc} \tag{14a}$$

and the communication is associated with the edges of the region in each processor and takes time:

$$T_{comm} = 4\sqrt{n_{grain}}\, t_{comm} \tag{14b}$$

and one finds a communication overhead

$$f_C \sim \frac{1}{n_{grain}^{1/2}} \frac{t_{comm}}{t_{calc}} \tag{14c}$$

with the form of Eq. (3) and the connection dimension d equal to the topological dimension.

(b) Hierarchical Machines

As in the previous example, Eq. (13a), one takes time for load and store of:

$$T_{cache} = n_{grain}\, t_{mem} \tag{14d}$$

The grain is stored in the "cache" and one also needs to load the neighboring points and in analogy to (14b), one finds a communication cost.

$$T_{comm} = 4\sqrt{n_{grain}}\, (1/2\, t_{mem} \text{ or } t_{comm}) \tag{14e}$$

where the communication may be from a process in the same or neighboring node; the corresponding communication time is either $1/2\, t_{mem}$ or t_{comm}. In fact, in terms of the ratio $r_{virtual}$ introduced in Sec. III, one can rewrite Eq. (14e) as

$$T_{comm} = 4\sqrt{n_{grain}}\, \frac{(1/2 t_{mem}\, (\sqrt{r_{virtual}} - 1) + t_{comm})}{\sqrt{r_{virtual}}} \tag{14e^I}$$

$$\sim 2\sqrt{n_{grain}}\, t_{mem} \tag{14e^{II}}$$

in the interesting case with large values of $r_{virtual}$.

If we compare (14a) with the overheads (14d) and (14e^{II}), we see that cache loading dominates (for large n_{grain}) and

$$f_H \sim \frac{1}{5} \frac{t_{mem}}{t_{calc}} \tag{14f}$$

and unlike our previous results summarized in Eq. (3), f_H does not decrease as one increases n_{grain}. The reason for this is clear from Fig. 8 and the discussion at the end of Sec. IV. We have a space-time complex system stored in our cache, with as shown in Fig. 10(a), a single iteration (time) count as its temporal extent. We need to reduce the edge/area ratio in the temporal direction. This can be achieved by updating the region stored in "cache" by more than one time step. This is a nontrivial issue because it now implies a different decomposition than that usually used for the homogeneous hypercube. In Sec. IVA, we saw a somewhat similar situation where the hierarchical implementation required M large whereas $M \simeq 1$ was sufficient for the homogeneous case. Now the hierarchical case has required that each grain update for several, say M, iterations or time cycles.

Two possibilities are shown in Figs. (10b) and (10c) for the case of one spatial dimension. If the wave

equation was the underlying problem, Fig. (10b) separates regions by the characteristics of the equation. We see that Fig. (10b) still corresponds to short range (spatial) communication. T_{cache} is unchanged. T_{calc} and T_{comm} in Eqs. (14a, $14e^{II}$) are increased by a factor equal to the number M, of iterations. Optimal is the choice of equal extents in the space and time dimensions, $M = {n_{grain}}^{1/d}$, when we get back the result

$$f_{C,H} \sim \frac{1}{n_{grain}^{1/d}} (t_{mem} \text{ or } t_{comm})/t_{calc} \tag{14g}$$

for a system of dimension d.

A simpler alternative, shown in Fig. (10c), is easier to implement. We chose cylinders with sides parallel to the time axis for our space-time complex systems. These have an interesting property illustrated precisely in Fig. 7 of cutting more edges with the temporal planes. In fact the number of edges cut is proportional to M^2 where M is the number of temporal steps. This leads to communication costs, appearing as additional calculations, proportional to M^2. We find in with the dimension $d = 2$,

$$\begin{aligned} T_{calc} &\propto n_{grain}\, M \\ T_{comm} &\propto {n_{grain}}^{1-1/d}\, M^2 \\ T_{cache} &\propto n_{grain} \end{aligned} \tag{14h}$$

The ration $f_C + f_H$ is proportional to $(\frac{1}{M} + \frac{M}{n_{grain}^{1/d}})$ and is minimized for $M \sim n_{grain}^{1/2d}$

and we find the amusing total overhead

$$f_C + f_H \sim \frac{1}{n_{grain}^{1/2d}} \frac{(t_{mem} \text{ or } t_{comm})}{t_{calc}} \tag{14i}$$

Although the dependence on the grain size is slower than for Eq. (14g), this overhead does decrease as the grain size increases.

IVC: Matrix Multiplication

We now consider a simple matrix algorithm which is not the most important but exhibits interesting contrasts with the previous two cases. Take the multiplication of large full matrices

$$C = A \cdot B \tag{15}$$

where each matrix is $\sqrt{n_{total}} \times \sqrt{n_{total}}$. The basic ideas in this section are the same as in Refs. [2, 18, 23, 31].

(a) Homogeneous Hypercube

Here one considers a simple decomposition dividing each matrix into N_{proc} square subblocks each of size $\sqrt{n_{grain}} \times \sqrt{n_{grain}}$ where the dimensions are given by $\sqrt{n_{grain}} = \sqrt{n_{total}} / \sqrt{N_{proc}}$. As discussed in Ref. [18], there is an efficient algorithm in which the subblocks of A and B are respectively broadcast (along a subcube) and shifted along nearest neighbor links of the hypercube. The basic calculation at each stage involves subblock multiplication with

$$T_{calc} \propto n_{grain}^{3/2}\, t_{calc} \tag{16a}$$

and overhead

$$T_{comm} \propto n_{grain}\, t_{comm} \tag{16b}$$

One obtains a conventional overhead formula

$$f_C \sim \frac{1}{n_{grain}^{1/2}} \frac{t_{comm}}{t_{calc}} \tag{16c}$$

corresponding to connection dimension $d = 2$.

(b) Hierarchical Machines

The interesting feature now is that one does not need to modify the algorithm at all because the cache loading time is

$$T_{cache} \propto n_{grain}\, t_{mem} \tag{16d}$$

and so we obtain, with an unchanged algorithm

$$f_H \sim \frac{1}{n_{grain}^{1/2}} \frac{t_{mem}}{t_{calc}} \tag{16e}$$

The constant in (16e) can be affected by the exact algorithm used as it depends on the number of blocks read in between cache stores; this is analogous to the M dependence in Sec. IVA.

Let us compare this case with that of the finite difference algorithm in Sec. IVB. As shown in Eqs. (14c) and (16c), both have similar overheads in the case of the hypercube. However, the matrix multiplication naturally retains this result for hierarchical machines while for IVB, we needed to change the algorithm to reduce the memory overhead f_H. We can understand the issue by comparing Eqs. (14a) and (16a). In the finite difference case, we have a small communication overhead because we are transmitting a few ($\propto n_{grain}^{1/2}$) variables but doing little (0(1) per variable) work with them. In matrix multiplication, we communicate more variables ($\propto n_{grain}$) but do a lot of work ($\propto n_{grain}^{1/2}$) for each transmitted variable. The algorithms obtain the same result for f_C but for different reasons. In fact the matrix multiplication algorithm "succeeds" on the hierarchical memory machines for the same reason as the long range problem. Both do a lot of work per fundamental entity; in IVA, the work is proportional to n_{grain} and in IVC, proportional to $n_{grain}^{1/2}$. This is the origin of the different connection dimensions shown in Eqs. (12c) and (16c).

IVD: LU Decomposition

Now let us consider, a more important matrix algorithm; namely LU decomposition of full matrices or the related problem of matrix inversion. This has been studied in detail on the hypercube in Refs. [2, 19, 20-24] and we will find that the issues are more similar to those of Sec. IVB than those of IVC. We will not consider pivoting here; this is an interesting and tricky complication which leaves the results below qualitatively unchanged [2, 23]. We note that the issues discussed here and in Sec. IVC are closely related to the important work on the "Level-3 BLAS" discussed by Dongarra at this conference [25].

(a) Homogeneous Hypercube

We use the same square subblock decomposition introduced in IVC. At a typical stage of LU decomposition, one is subtracting the row containing the eliminated variable from all other rows. There is a separate multiplier for each row calculated from the column containing the eliminated variable. This is illustrated in Fig. 11 and one sees a situation rather similar to that of Sec. IVB.

Namely in a typical stage, one has a calculation

$$a_{ij} \rightarrow a_{ij} - c_i \, r_j \tag{17}$$

taking time

$$T_{calc} = 2 n_{grain} \, t_{calc} \tag{18a}$$

and a communication

$$T_{comm} = 2\sqrt{n_{grain}} \, t_{comm} \tag{18b}$$

with our standard form

$$f_C \sim \frac{1}{\sqrt{n_{grain}}} \frac{t_{comm}}{t_{calc}} \tag{18c}$$

Actually Eqs. (18a, b) ignore certain overheads connected with the calculation of the c_i and r_j as well as load imbalance overhead and the effect of the reduction in value of n_{grain} as one eliminates rows and columns. These effects alter the *constant* in Eq. (3) but leave the form (18c) unchanged.

(b) Hierarchical Memory

We are faced with the same problem as in Sec. IVB, with a "cache" load and store time which has the same n_{grain} dependence as the calculation in Eq. (18a). The solution is similar to that of IVB and involves eliminating several, say M, rows (and columns) at the same time. One can show that the M values of r_i and c_j can be calculated in a separate concurrent step which itself can be efficiently implemented. Given this, one will find a simple block algorithm in which the times in Eqs. (18a, b) are both increased by a factor M whereas the "cache" loading is still given by a time $t_{mem} \, n_{grain}$. Choosing $M \propto n_{grain}^{1/2}$, one will find overheads f_H of comparable form to f_C and a total overhead that is proportional to $n_{grain}^{-1/2}$ as in the original homogeneous hypercube case.

IVE: Fast Fourier Transform (FFT)

Originally we implemented the FFT in a natural fashion on the hypercube [2] using local communication in the hypercube topology where necessary. It is well known that the hypercube architecture exactly matches the pattern of the binary FFT. However, the discussion here will be easier using a formulation due to Furmanski [2, 33] which ingeniously lumps all the communication into a single stage and avoids any communication during the calculation steps.

(a) Homogeneous Hypercube

We will consider a one dimensional FFT and we can label the variables by a binary digit B illustrated in Fig. 12. This example shows the case $n_{total} = 2^{14}$ and $N_{proc} = 2^5$. Five digits g_{0-4} of B are used to label processor number and the remaining nine digits l_{0-8} the position within the local memory of the node. The FFT algorithms systematically alters the variable starting at the highest digit of B (l_0 in Fig. 12(a)) and ending at the lowest (g_4 in Fig. 12a). There are $\log n_{total}$ steps each of which takes calculation time n_{total}.

Furmanski's algorithm does the first nine steps, each altering one digit, as labelled in Fig. 12(a); then it transforms the data to the situation of Fig. 12(b) where the lower order bits are now stored locally. This is performed by a communication primitive called *index* in Ref. [33]. After the transformation the remaining five digits are processed locally in each node. The calculation is load balanced at each stage and takes total time

$$T_{calc} \sim \frac{1}{N_{proc}} n_{total} \log(n_{total})\, t_{calc} \tag{19a}$$

$$= n_{node} \log(n_{total})\, t_{calc}$$

while the communication primitive *index* takes time

$$T_{comm} \sim n_{node} \log(N_{proc})\, t_{comm} \tag{19b}$$

In this formula the $\log N_{proc}$ dependence just represents the typical distance ($1/2 \log N_{proc}$) between points in a hypercube topology. This communication time could be reduced on machines with automatic routing. One finds an overhead

$$f_C \sim \frac{\log(N_{proc})}{\log(n_{total})} \frac{t_{comm}}{t_{calc}} \tag{19c}$$

which corresponds to infinite connection dimension in Eq.(3).

(b) Hierarchical Memory

The discussion of hierarchical memory machines is now straightforward. We will use the same idea of calculating a certain number of bits in B at a time. Clearly we can perform a number of steps corresponding to at most $\log (n_{grain})$ bits at any one time and we find a set of calculations each taking time

$$T_{calc}^{elementary} \sim n_{grain} \log(n_{grain})\, t_{calc} \tag{19d}$$

and a cache load and store time of

$$T_{cache} \sim n_{grain}\, t_{mem} \tag{19e}$$

Thus, we perform $\log (n_{grain})$ steps for each cache loading and obtain an overhead

$$f_H \propto \frac{1}{\log(n_{grain})} \frac{t_{mem}}{t_{calc}} \tag{19f}$$

We have found a slightly different overhead formula from the original case, Eq. (19c), but fortunately we have been able to use the same decomposition to deal with distributed and hierarchical memories. This equivalence would not have been direct if one had used the traditional hypercube FFT approach.

IVF: Neural Networks

We have a growing interest at Caltech in using the hypercube to model both biological and applied (theoretical) neural networks. In general, distributed memory machines are well suited to modelling nature's own distributed memory "computers". It turns out that the optimal decomposition of the neural network is sensitive to the structure of the interconnection between the neurons [26]. Some cases have a rather full interconnect and the hypercube decomposition resembles the long range force algorithm described in Sec. IVA; other cases have a dominant short range structure and the analysis is then similar to that in Sec. IVB. The issues in neural network simulation are of course related to those in other circuit simulations.

We can discuss a recent explicit implementation on the hypercube of a model of the piriform cortex [27]. This corresponds to a nearly full interconnect but an interesting subtlety that changes the analysis from that in IVA. One is calculating the effect $G_{i\to j}$ of neuron i on j. A symmetry of the propagation Green's function ($G_{i\to j}$) means that it is optimal to associate the calculation of $G_{i\to j}$ with i and not the target neuron j. In the particle dynamics case in IVA we made the opposite choice of shipping the information to the target particle and then doing the calculation. The calculation of all the Green's functions ($G_{i\to j}$) is local to each grain and involves no communication. The latter is needed as one forms the global sums

$$\sum_i G_{i\to j} \text{ for each } j \tag{20}$$

Thus, the problem is reduced to the calculation of a large number of sums. The components of the sum (i) and the storage of the final result (j) are both uniformly distributed throughout the hypercube. This problem was solved in Ref. [33] in terms of a hypercube communication primitive called *fold*. It is straightforward to implement *fold* on a hierarchical memory; the issues are similar to those in Sec. IVA.

We see that the neural network decomposition can be unified over the various architectures as long as it is implemented in terms of the basic primitives *fold* which is then separately implemented in an optimal fashion on each architecture.

V: Conclusions

We have considered the issues involved in a unified approach to distributed and hierarchical memory machines. Our techniques also allow optimal implementations of message passing on shared memory parallel processors. We have discussed the general approach in Sec. III and the prettiest result is inclusion of both space and time in the description of complex systems. In Sec. IV, we worked through six problem classes and were able to show decompositions that effected our desired unifications. In all cases, the overheads were small as long as the "cache" size n_{grain} was big enough and that the communication performance of the computer, measured by t_{mem}/t_{calc}, t_{comm}/t_{calc} was good enough. Another way of summarizing our results is to say that problems are associated with data domains. Homogeneous machines just need the decomposition of this domain; this is often a spatial decomposition. Hierarchical computers also need one to consider the label (e.g., DO loop index) of the processing of elements in the domain. These machines require both spatial (data domain) decomposition and that associated with this new label; this label often corresponds to time i.e., hierarchical machines need "space-time" and not just "spatial" decompositions. Our results contain the essential information to make the necessary tradeoffs between memory/communication bandwidth and "cache" size. We note that technology improvements should permit "cache" sizes to increase in the future. This will increase the applicability of our techniques.

In future papers, we will present details of the ideas sketched in IV and describe the implementation issues in providing the desired portable high performance environments.

We would also point out that the basic ideas presented here are the message passing version of principles that have been used for some time in the field of vectorizing and decomposing compilers [28, 29, 30]. The manipulation of DO loops described by Gannon at this conference are similar in spirit to the physical transformations suggested here.

Acknowledgements

I would like to thank the many members of the Caltech Concurrent Computation Program (C^3P) on whose work many of the results discussed here are based. I hope they are content with implied reference in Table 2. The ideas presented here were first developed during a study of the use of the RP3 and the porting of hypercube applications to it. I would like to thank IBM for their support of this work and in particular, A. Frey, S. Harvey and A. Norton. The support of the Electronic Systems Division of the U. S. Air Force has allowed us to extend our basic hypercube research to a variety of different architectures. Our ELXSI implementations were funded by SANDIA. I would thank the organizers of ICS '87 for an excellent conference and the honor of the invitation to talk.

References

1. C^3P-394: "Caltech Supercomputer Initiative: A Commitment to Leadership and Excellence," A. H. Barr, R. W. Clayton, A. Kuppermann, L. G. Leal, A. Leonard, T. A. Prince, December 29, 1986.
2. G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, D. Walker, "Solving Problems on Concurrent Processors," April 1986. To be published by Prentice Hall, 1987.
3 C^3P-391: "The Hypercube as a Supercomputer," G. C. Fox, January 7, 1987. Published by the International Supercomputing Institute, Inc., St. Petersburg, Florida, May 1987.
4. C^3P-409: "Concurrent Supercomputer Initiative at Caltech," G. C. Fox, January 31, 1987. Published by the International Supercomputing Institute, Inc., St. Petersburg, Florida, May 1987.

5. "Portable Programs for Parallel Processors," J. Boyle, R. Butler, T. Disz, B. Glickfield, E. Lusk, R. Overbeek, J. Patterson, R. Stevens, published by Holt, Rinehart, and Winston, Inc., N. Y. 1987.
6. C^3P-435: "The Concurrent Supercomputing Initiative at Caltech," G. Fox and co-authors.
7. C^3P-255: "Concurrent Computation and the Theory of Complex Systems," G. C. Fox, S. W. Otto, March 3, 1986. Published in proceedings of 1985 Hypercube Conference at Knoxville, August 1985, edited by M. Heath and published by SIAM.
8. C^3P-214: "Monte Carlo Physics on a Concurrent Processor," G. C. Fox, S. W. Otto, E. A. Umland, Nov 6, 1985, invited talk by G. Fox at the "Frontiers of Quantum Monte Carlo" Conference at Los Alamos, September 6, 1985, and published in special issue of Journal of Statistical Physics, Vol. 43,1209, Plenum Press, 1986.
9 C^3P-161: "The Performance of the Caltech Hypercube in Scientific Calculations: A Preliminary Analysis:" G. Fox, April 1985, Invited Talk at Symposium in Austin, Texas, March 18-20, 1985, and published in "Supercomputers-Algorithms, Architectures and Scientific Computation," edited by F. A. Matsen and T. Tajima, University of Texas Press, Austin, 1985.
10. C^3P-292: "A Preprocessor for Irregular Finite Element Problems," CALT-68-1405, J. W. Flower, S. W. Otto, M. C. Salama, June 1986.
11. C^3P-363: "Load Balancing by a Neural Network," CALT-68-1408, G. C. Fox, W. Furmanski, September 1986.
12. C^3P-327B: "A Graphical Approach to Load Balancing and Sparse Matrix Vector Multiplication on the Hypercube," G. C. Fox, December 5, 1986. To be published in proceedings of IMA Workshop, Minnesota, November 1986.
13. C^3P-385: "A Review of Automatic Load Balancing and Decomposition Methods for the Hypercube," G. C. Fox, November 1986. To be published in proceedings of IMA Workshop, Minnesota, November 1986.
14. C^3P-328: "The Implementation of a Dynamic Load Balancer," G. Fox, A. Kolawa, R. Williams, November 1986, published in proceedings of 1986 Knoxville Hypercube Conference, edited by M. Heath and published by SIAM as "Hypercube Multiprocessors 1987."
15. C^3P-427: "A MOOSE Status Report," J. Salmon, S. Callahan, J. Flower, A. Kolawa, May 6, 1987.
16. C^3P-390: "An Evaluation of Mark III and NCUBE Supercomputers," G. C. Fox, December 9, 1986.
17. J. Barnes, P. Hut, "A Hierarchical O(NlogN) Force-Calculation Algorithm," Nature 324,446 (1986).
18. C^3P-206: "Matrix Algorithms on the Hypercube I: Matrix Multiplication," G. Fox, A. J. G. Hey, S. Otto, October 1985, published in Parallel Computing, 4, 17 (1987).
19. C^3P-97: "Square Matrix Decompositions: Symmetric , Local, Scattered," G. Fox, August 15, 1984, unpublished.
20. C^3P-99: "LU Decomposition for Banded Matrices," G. C. Fox, August 15, 1984, unpublished.
21. C^3P-347: "Gauss Jordan Matrix Inversion with Pivoting on the Hypercube," P. Hipes, A. Kuppermann, August 8, 1986.
22. C^3P-348: "A Banded Matrix LU Decomposition on the Hypercube," T. Aldcroft, A. Cisneros, G. Fox, W. Furmanski, D. Walker, paper in preparation.
23. C^3P-386: "Matrix," G. C. Fox and W. Furmanski, paper in preparation.
24. G. A. Geist, M. T. Heath "Matrix Factorization on a Hypercube Multiprocessor;" C. Moler "Matrix Computation on Distributed Memory Multiprocessor." Both these articles are contained in "Hypercube Multiprocessors, 1986", edited by M. T. Heath, SIAM, 1986.
25. J. Dongarra, Invited talk at 1987 International Conference on Supercomputing, Athens, June 8-12, 1987.
26. C^3P-405: "Hypercube Communication for Neural Network Algorithms," G. C. Fox, W. Furmanski, paper in preparation.
27. C^3P-404: "Piriform (Olfactory) Cortex Model on the Hypercube," J. M. Bower, M. E. Nelson, M. A. Wilson, G. C. Fox, W. Furmanski, February 1987.
28. D. Gannon, Invited talk at 1987 International Conference on Supercomputing, Athens, June 8-12, 1987.
29. K. Kennedy, Invited talk at 1987 International Conference on Supercomputing, Athens, June 8-12, 1987.
30. A. Sameh, "Numerical Algorithms on the Cedar System," Second SIAM Conference on Parallel Processing, Norfolk, Virginia, November 1985.
31. W. Jalby and U. Meier, "Optimizing Matrix Operations on a Parallel Multiprocessor with a Hierarchical Memory System," CSRD-555, University of Illinois report, 1986.
32. D. J. Kuck, E. S. Davidson, D. H. Lawrie, A. H. Sameh, "Parallel Supercomputing Today and the Cedar

Approach," *Science* **231**, 967 (1986).

33. C^3P-314: "Optimal Communication Algorithms on the Hypercube," G. C. Fox, W. Furmanski, July 8, 1986; "Communication Algorithms for Regular Convolutions on the Hypercube," G. C. Fox, W. Furmanski, September 1, 1986, published in proceedings of 1986 Knoxville Hypercube Conference, edited by M. Heath and published by SIAM as "Hypercube Multiprocessors, 1987."

Table 1: Hardware Parameters of Some Concurrent Supercomputers

	$\sim$ optimal $\frac{t_{comm}}{t_{calc}}$	t_{comm}	$\frac{t_{comm}}{t_{calc}}$	Main Memory Size M Byte	t_{mem}	$\frac{t_{mem}}{t_{calc}}$	"Cache" Size M Byte
NCUBE Hypercube	10 μs	13 μs	1.3	0.5	-	-	-
Mark IIIfp	0.1 μs	2.5 μs *	25	4	1.5	16	0.13

*Operation concurrent with calculation

Table 2: Some Hypercube Implementations at Caltech

General Field	Associated Scientists	Topic
Applied Math & Computer Science	A. Barr J. Goldsmith (JPL)	Computer Graphics
	B. Beckman (JPL) D. Jefferson (UCLA)	Time Warp Event Driven Simulation
	M. Buehler (JPL)	Computer Aided Design
	G. Fox	Matrix Algorithms Load Balancing Algorithms Optimization Computer Chess
	H. Keller P. Saffman	Parallel Shooting Multigrid Adaptive Meshes
	C. Seitz	Mathematics and Logic Computer Aided Design
Biology	J. Bower J. Hopfield C. Koch	Modelling of Cortex and Applied Neural Networks
Chemistry and Chemical Engineering	J. Brady	Flow of Porous Media
	W. Goddard	Protein Dynamics
	A. Kuppermann	Chemical Reaction Dynamics
Engineering	N. Corngold	Turbulence (strange attractors)
	R. Gould P. Liewer (JPL)	Plasma Physics (PIC)
	J. Hall	Finite Element Analysis of Earthquake Engineering
	W. Johnson	Condensed Matter Simulations for Material Science

	A. Leonard B. Sturtevant	Fluid Turbulence in Computational Aerodynamics
	R. McEliece E. Posner F. Pollara (JPL)	Convolution Decoding
	J. Solomon (JPL)	Image Processing
Geophysics	R. Clayton	Seismic Waves Tomography
	B. Hager	Geodynamics
	T. Tanimoto	Normal Modes of Earth
Physics	R. Blandford D. Meier (JPL)	Fluid Jets in Astrophysics
	M. Cross	Condensed Matter Two Dimensional Melting
	G. Fox	High Energy Physics Lattice Gauge Theory
	S. Koonin	Nuclear Matter
	A. Readhead T. Prince	Astronomical Data Analysis
	T. Tombrello	Granular Physics Molecular Dynamics

Figure Captions

Fig. 1. Block diagram of the three 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 CPU's of the shared memory architecture.

Fig. 2. Block Diagrams of the NCUBE and Mark IIIfp hypercube nodes. We do not show a 68020/68882 based applications controller in (b) as it is irrelevant for this paper. The WEITEK XL unit is a complete computer.

Fig. 3. 64 node NCUBE board and packaging into a 1024 node system

Fig. 4. The two printed circuit boards making up the Mark IIIfp node. The WEITEK board is functional but will be cleaned up for the production unit.

(a) Main board with 4 megabytes of memory and two 68020's.

Fig. 4. The two printed circuit boards making up the Mark IIIfp node. The WEITEK board is functional but will be cleaned up for the production unit.

(b) Floating Point coprocessor built around WEITEK XL chip set.

Fig. 5. Definitions of the three hardware parameters t_{calc}, t_{mem}, and t_{calc} discussed in Sec. II.

Fig 6. The complex system corresponding to a short range force particle dynamics problem with its decomposition into four grains.

Fig. 7. A complex system in space and time that corresponds to a regular nearest neighbor problem in one spatial dimension; the finite difference solution of $\frac{\partial^2 \psi}{\partial x^2} = c \frac{\partial^2 \psi}{\partial t^2}$ would lead to such a complex system.

Fig. 8. The boundaries of a space-time complex system of the type illustrated in Fig. 7. Edges of the graph crossed by spatial boundaries correspond to messages with neighboring systems. Those crossed by temporal boundaries corresponding to the loading and ejection of the grain to and from cache and main memory.

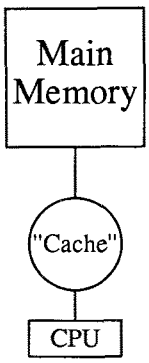
Fig. 9. Two stencils for the solution of Poisson's equation in two dimensions.

Fig. 10. Three decompositions in space and time of the one dimensional wave equation discussed in the text of Sec. IVB.

Fig. 11. A typical step in LU decomposition discussed in Sec. IVD.

Fig. 12. The *index* transformation used in the FFT discussion of Sec. IVE.

(a) Generic Sequential Computer
With Hierarchical Memory



(b) Shared Memory Computer
With Hierarchical Memory

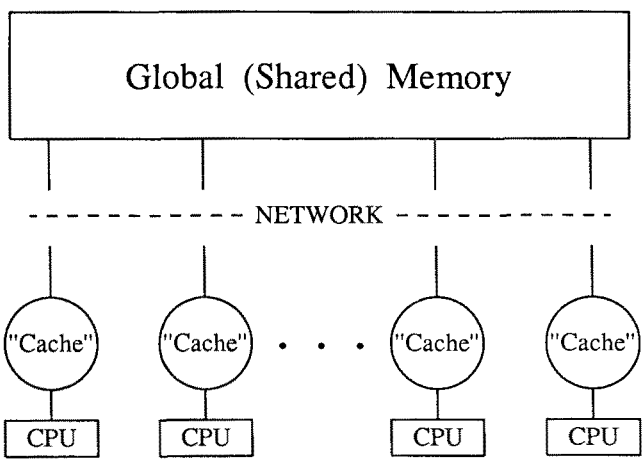
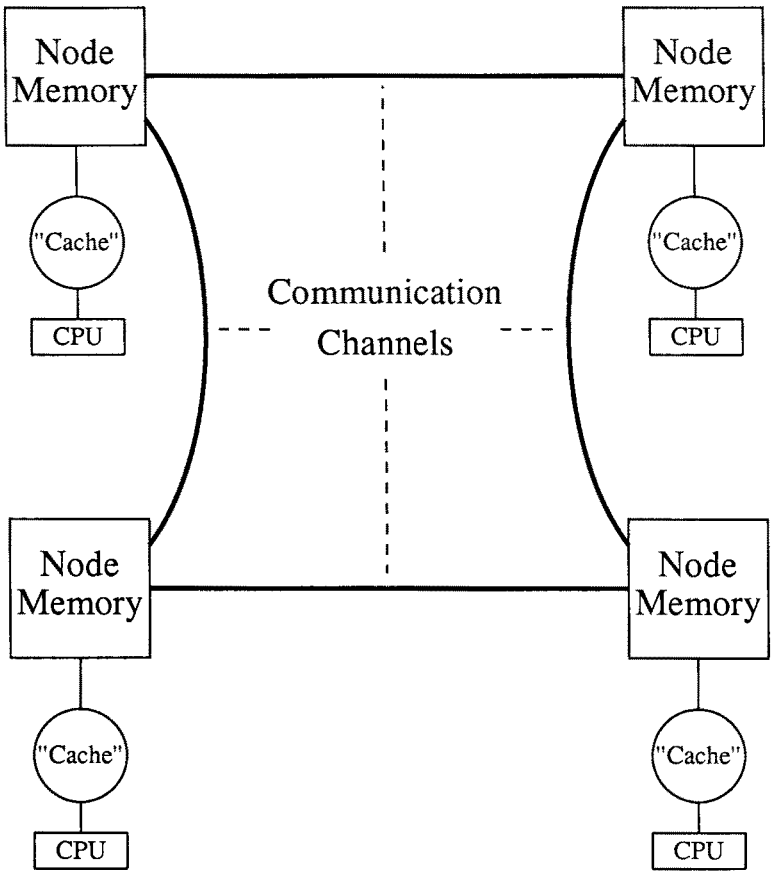
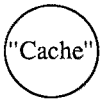


Fig. 1. Block diagram of the three machine architectures considered in this paper. We allow either a hardware controlled cache or user local memory to be the lowest level of the memory hierarchy. We ignore the important issues concerning the network connecting the global memory in (b) to the local "cache" and CPU's of the shared memory architecture.

(c) Hierarchical Memory Hypercube (d=2)





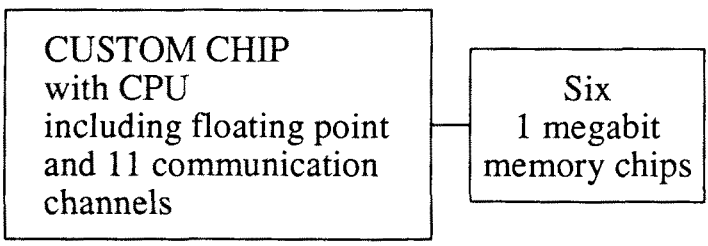
may be a true cache or user controlled memory

Fig. 1. Block diagram of the three 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 CPU's of the shared memory architecture.

Contrasting Hypercube Nodes

(a) Commercial NCUBE

Each Node is 7 Chips



(b) Mark IIIfp Constructed at JPL

Each Node is 2 Printed Circuit
Boards and a Total of 440 Chips

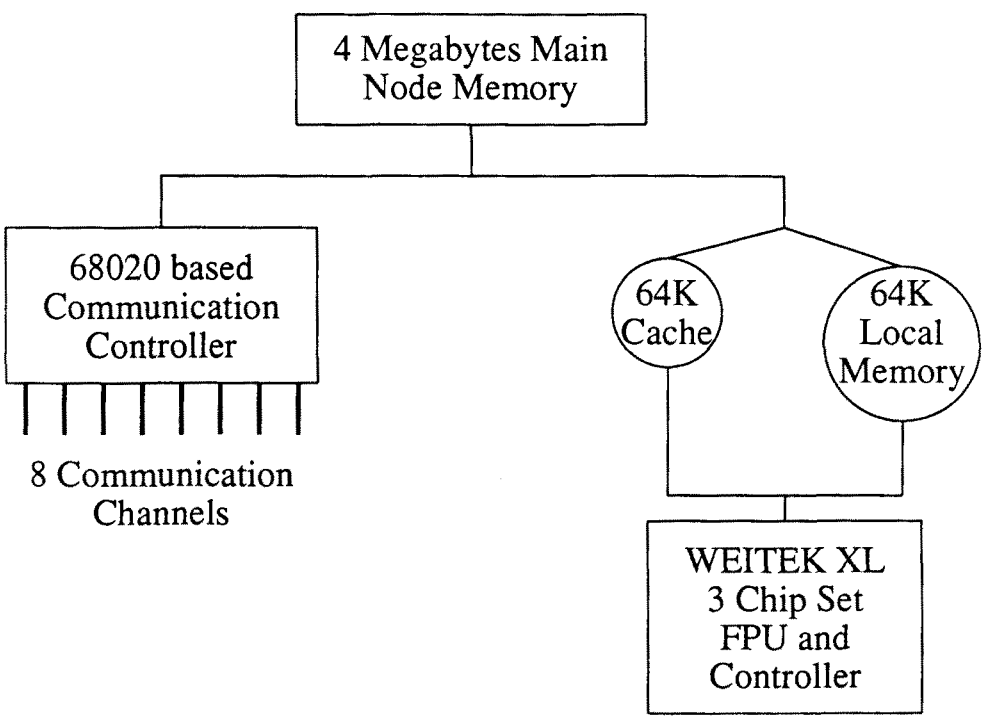


Fig. 2. Block Diagrams of the NCUBE and Mark IIIfp hypercube nodes. We do not show a 68020/68882 based applications controller in (b) as it is irrelevant for this paper. The WEITEK XL unit is a complete computer.

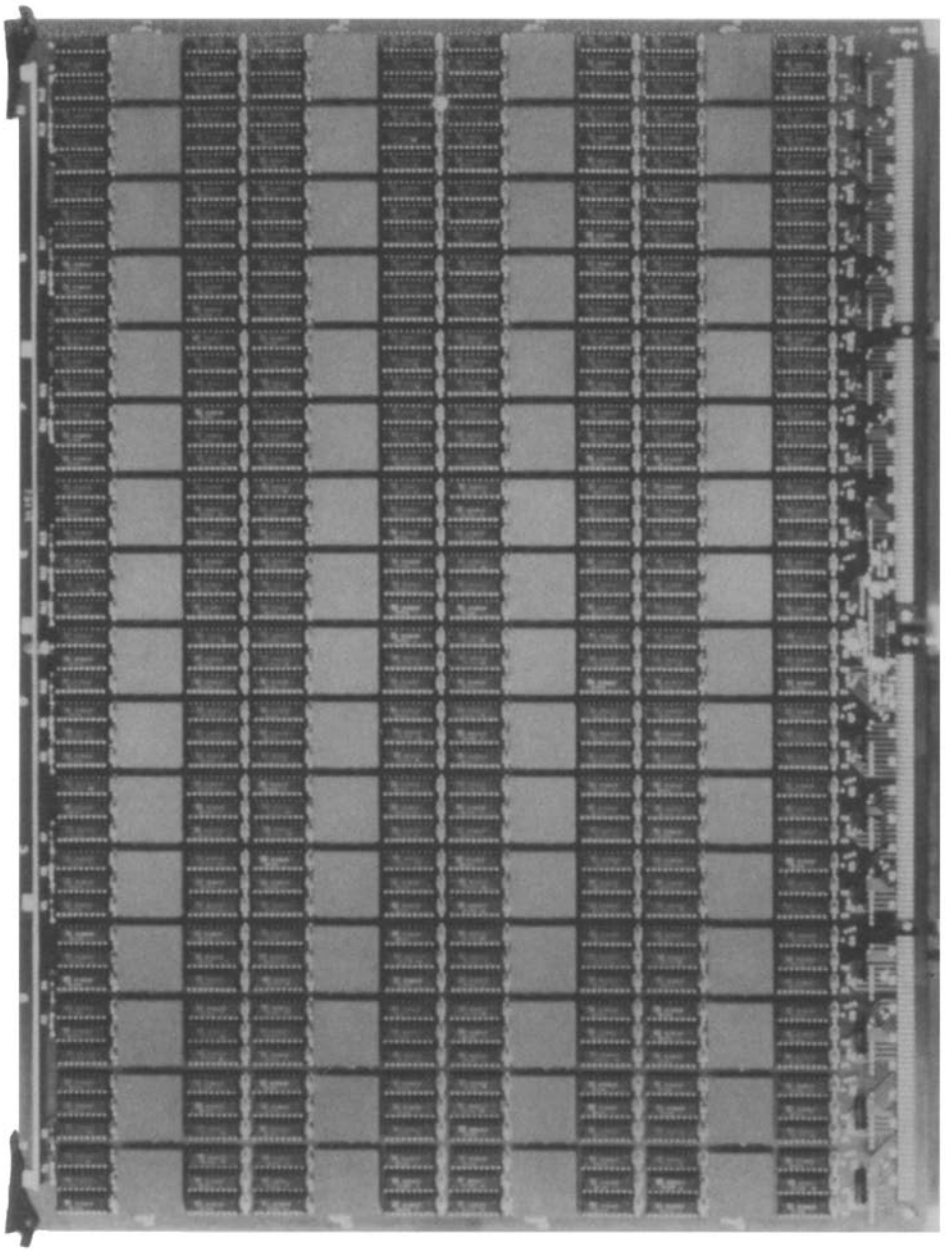


Fig. 3. 64 node NCUBE board and packaging into a 1024 node system

Fig. 3. 64 node NCUBE board and packaging into a 1024 node system

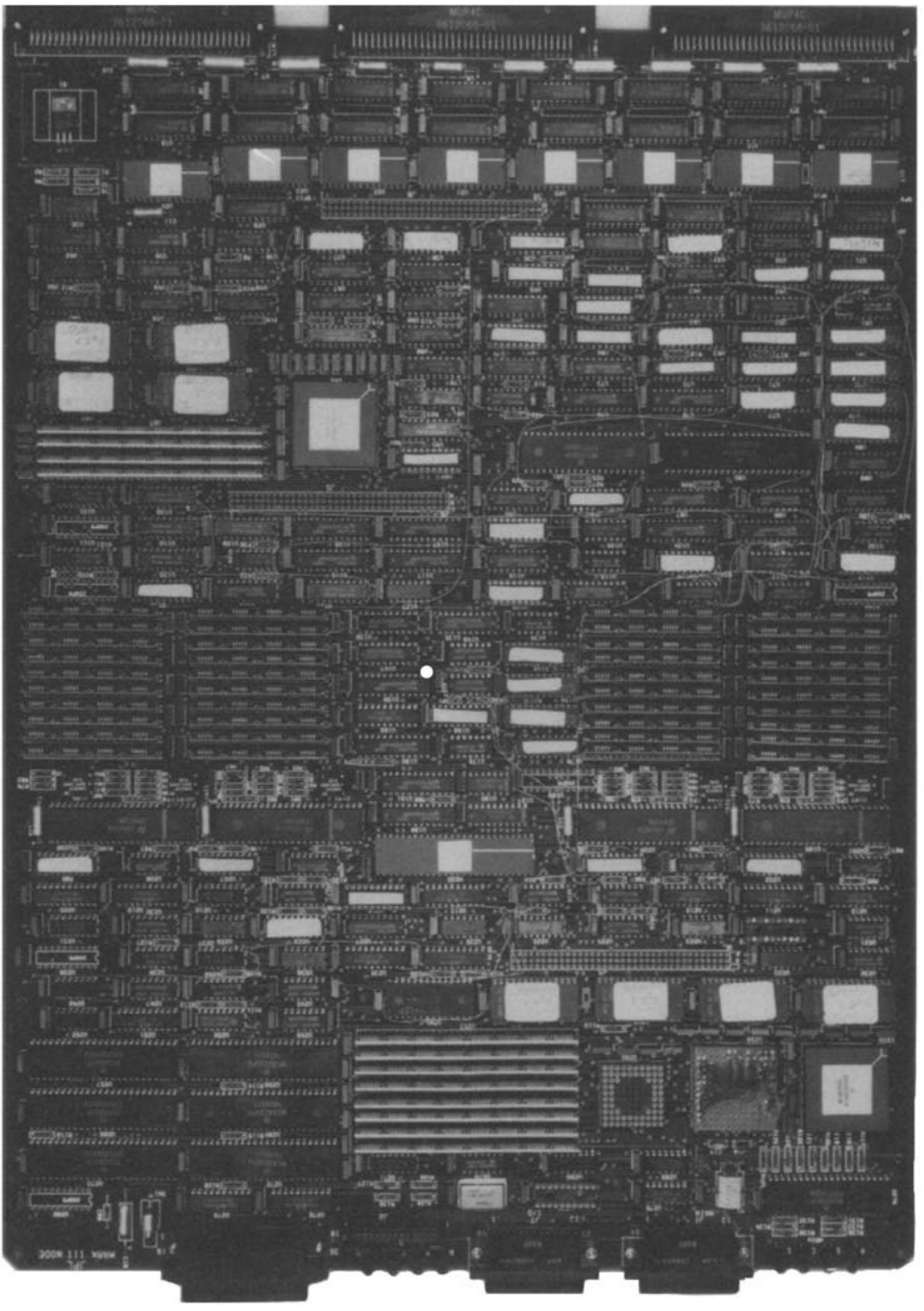


Fig. 4. The two printed circuit boards making up the Mark IIIfp node. The WEITEK board is functional but will be cleaned up for the production unit.
(a) Main board with 4 megabytes of memory and two 68020's.

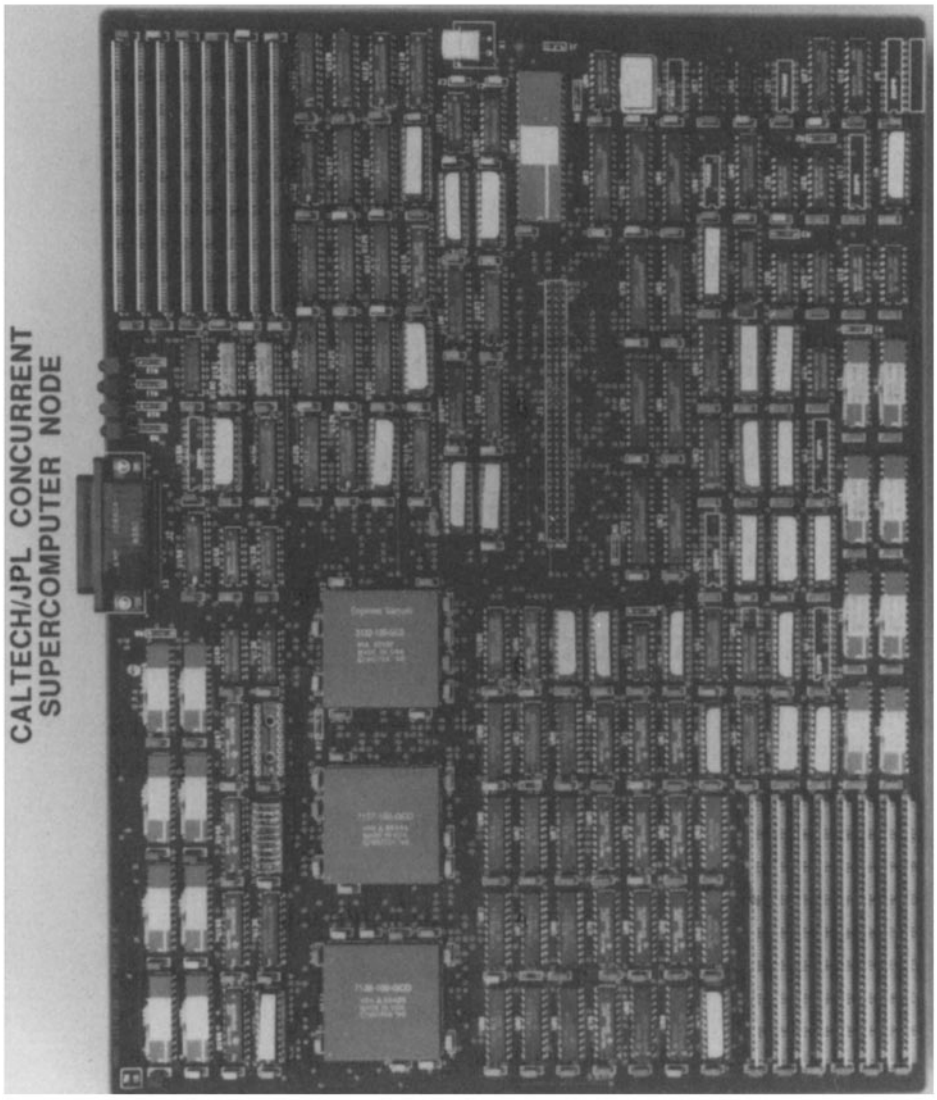


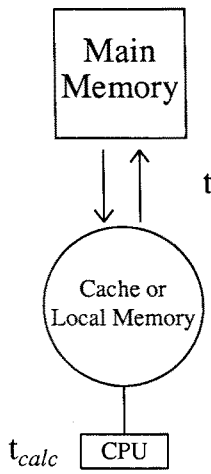
Fig. 4. The two printed circuit boards making up the Mark IIIfp node. The WEITEK board is functional but will be cleaned up for the production unit.
(b) Floating Point coprocessor built around WEITEK XL chip set.

Basic Hardware Parameters

Calculation Time

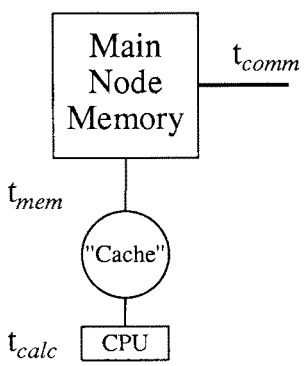
CPU t_{calc} = Time for basic floating point operation.

"Cache" - Main (Global) Memory Transfer Time



t_{mem} = Time spent to read AND write word in "cache" from and to main memory.

Node to Node Communication Time



t_{comm} = Time spent to communicate (read and write) a word between nodes.

Fig. 5. Definitions of the three hardware parameters t_{calc}, t_{mem}, and t_{calc} discussed in Sec. II.

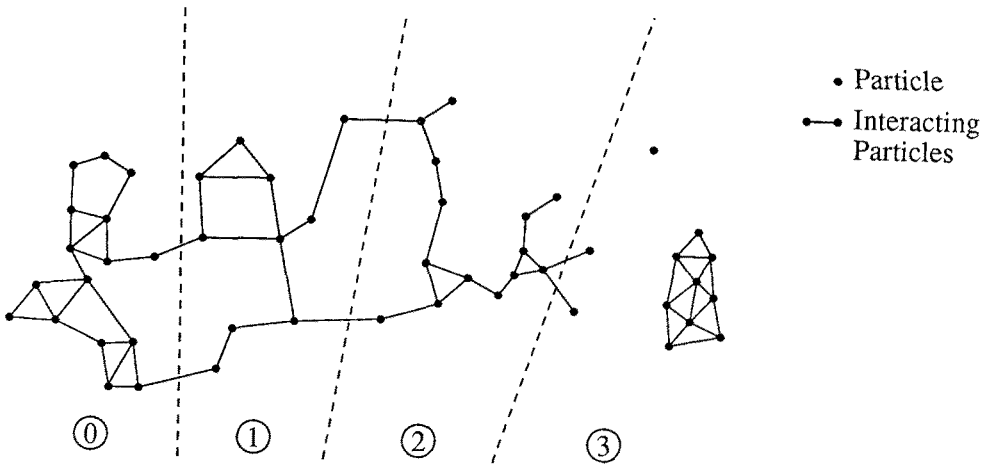


Fig 6. The complex system corresponding to a short range force particle dynamics problem with its decomposition into four grains.

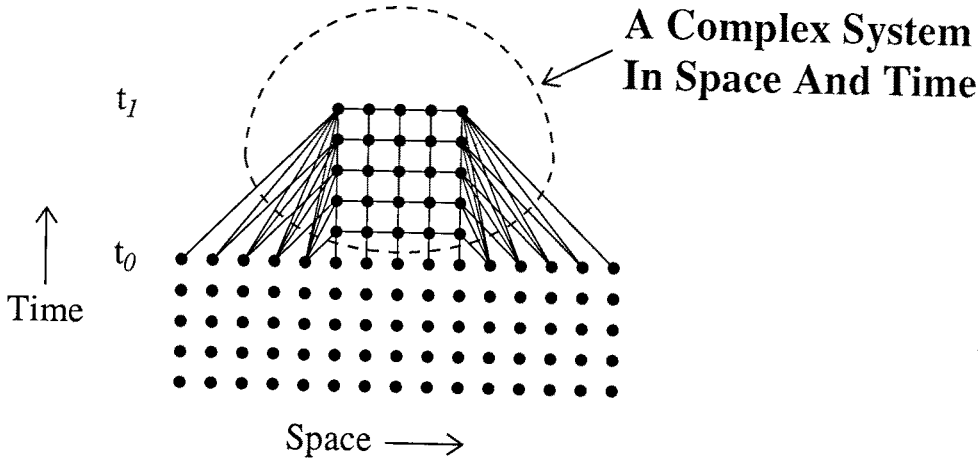


Fig. 7. A complex system in space and time that corresponds to a regular nearest neighbor problem in one spatial dimension; the finite difference solution of $\frac{\partial^2 \psi}{\partial x^2} = c \frac{\partial^2 \psi}{\partial t^2}$ would lead to such a complex system.

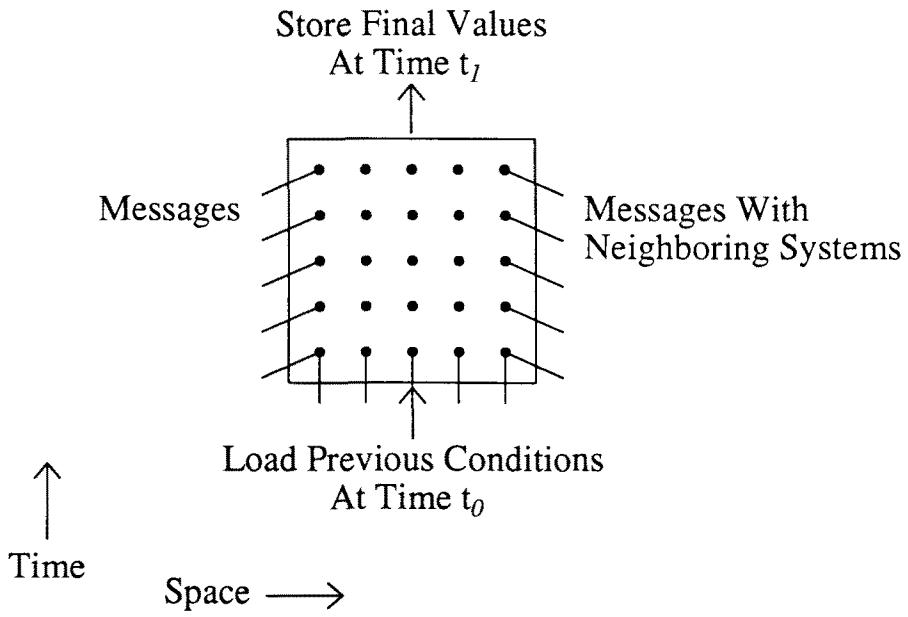
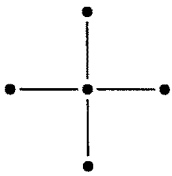


Fig. 8. The boundaries of a space-time complex system of the type illustrated in Fig. 7. Edges of the graph crossed by spatial boundaries correspond to messages with neighboring systems. Those crossed by temporal boundaries corresponding to the loading and ejection of the grain to and from cache and main memory.

(a) Simple Stencil



(b) Higher Order Stencil

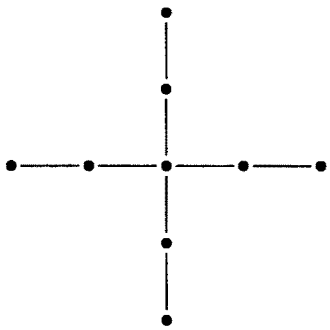
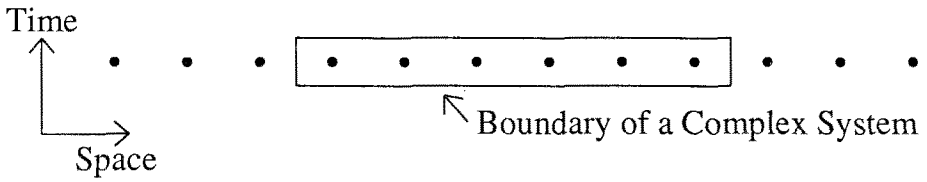
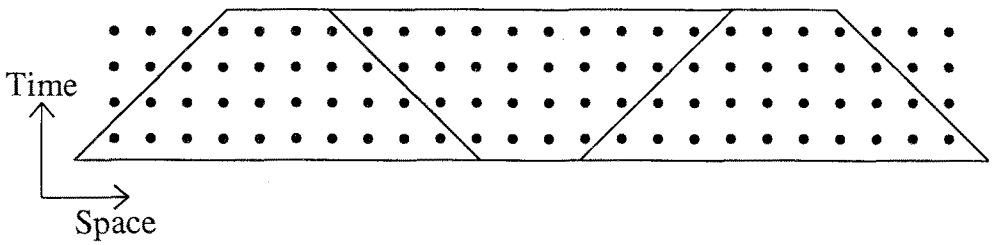


Fig. 9. Two stencils for the solution of Poisson's equation in two dimensions.

(a) A High Edge/Area Ratio In The Time Direction



(b) A Better Edge/Area Ratio With Modest Communication



(c) A More Practical Decomposition With More Communication

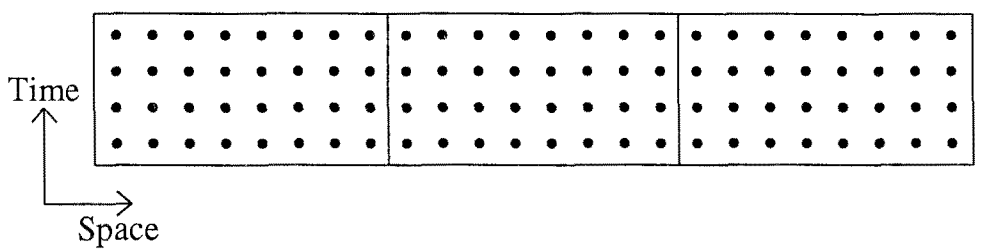


Fig. 10. Three decompositions in space and time of the one dimensional wave equation discussed in the text of Sec. IVB.

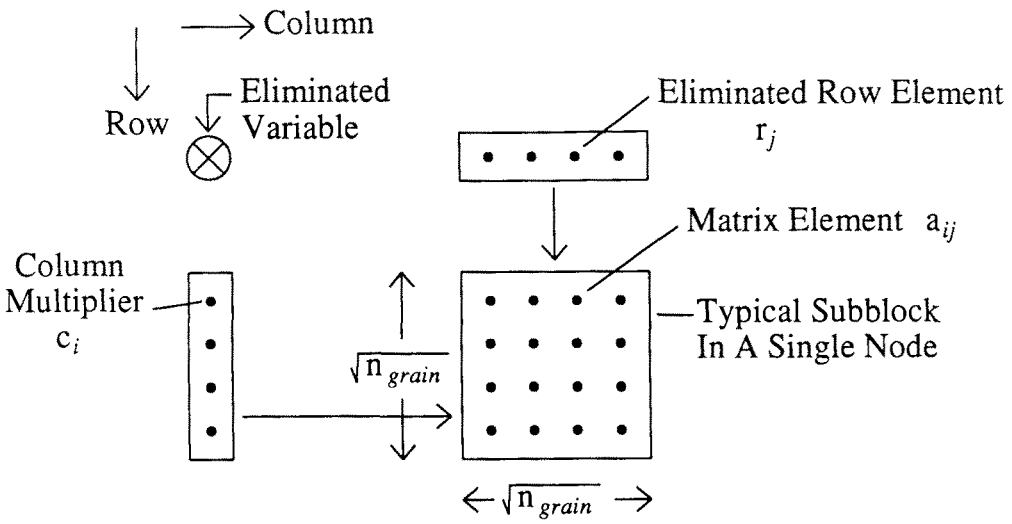


Fig. 11. A typical step in LU decomposition discussed in Sec. IVD.

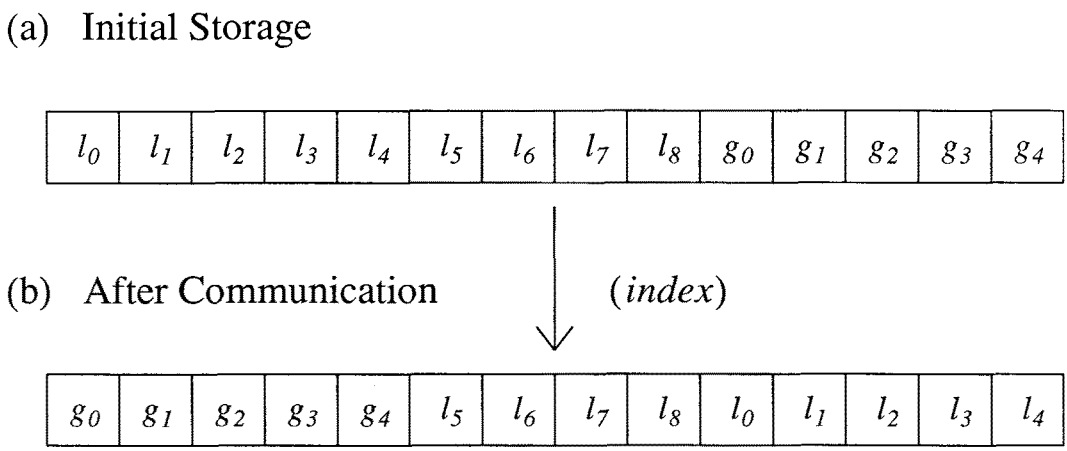


Fig. 12. The *index* transformation used in the FFT discussion of Sec. IVE.