Parallel Computers and Complex Systems

G.C. Fox and P.D. Coddington

Northeast Parallel Architectures Center, Syracuse University,
Syracuse, NY 13244, USA

June 4, 1994

Abstract

We present an overview of the state of the art and future trends in high performance parallel and distributed computing, and discuss techniques for using such computers in the simulation of complex problems in computational science. The use of high performance parallel computers can help improve our understanding of complex systems, and the converse is also true — we can apply techniques used for the study of complex systems to improve our understanding of parallel computing. We consider parallel computing as the mapping of one complex system — typically a model of the world — into another complex system — the parallel computer. We study static, dynamic, spatial and temporal properties of both the complex systems and the map between them. The result is a better understanding of which computer architectures are good for which problems, and of software structure, automatic partitioning of data, and the performance of parallel machines.

1 Introduction

The power of high performance computing is being used in an increasingly wide variety of applications in the physical sciences, and in particular in the study of complex systems. The performance of supercomputers has increased by roughly a factor of two every 18 months since electronic computers were first developed. A number of supercomputer manufacturers are aiming to deliver Teraflop (10^{12} floating point operations per second) performance well before the end of the decade.

Hardware trends imply that all computers, from PCs to supercomputers, will use some kind of parallel architecture by the end of the century. Until recently parallel computers were only marketed by small start-up companies (apart from Intel Supercomputer Systems Division), however recently Cray, Hewlett-Packard and Convex, IBM, and Digital have all begun marketing commercial parallel computers. Software for these systems is a major challenge, and could prevent or delay this hardware trend toward parallelism. Reliable and efficient systems software, high level standardized parallel languages and compilers, parallel algorithms, and applications software all need to be available for the promise of parallel computing to be fully realized.

A characteristic feature of the research on parallel computing at the Caltech Concurrent Computation Program (C3P), and more recently the Northeast Parallel Architectures Center (NPAC) at Syracuse University, is that many of the people who have worked in these groups (including ourselves) have a background in physics, so much of this research has made use of ideas from both physics and computer science. The goal of this work has been to make parallel computers more effective and easier to use for a wider variety of applications [12, 30, 14, 1, 32].

Parallel computers are complex entities used to simulate complex problems. While the physical sciences have developed several qualitative and quantitative methods to understand complex systems, other fields, in particular computer science, have not. Thus, it is not surprising that physics concepts, especially those related to complex systems, are helpful in developing a theory of computation and indeed may become more important as the computers and the problems they simulate get larger and more complicated. Here we present a review of these concepts. Several references contain more detailed discussions [34, 33, 25, 26, 27, 30, 28, 17, 21, 22, 24, 32].

In Section 2 we give an overview of the state of the art and future trends in parallel computing, concentrating on the use of parallel computers for simulation, particularly of complex systems. We describe recent progress in defining a standardized, portable, high level parallel language called High Performance Fortran, an extension of Fortran 90 designed for efficient implementation of data parallel applications on parallel, vector and sequential high performance computers. An outline of the language is presented, and we discuss its ability to handle different applications in computational science, particularly the difficulties of implementing irregular problems. We also discuss other problems such as software integration and the use of concepts such as visualization, virtual reality and metacomputing to enhance the usability of high performance computers. Further discussions of issues concerning parallel computing can be found in [30, 15, 20, 47, 23, 32].

We have found that when trying to understanding the use of parallel computers it is often very helpful to view the application, the software and the computer as complex systems. We have used these concepts to develop a theory of computation for parallel computers. In Section 3 we lay the foundations for this theory by presenting the view of computation as a set of maps from one of these complex systems to another, and introducing the concepts of space and time for these complex systems. Section 4 describes spatial properties — size, topology, dimension, and a physical analogy for data partitioning of slowly-varying problems leading to concepts of temperature and phase transitions.

In Section 5, we discuss temporal properties — a string model for very adaptive problems and a duality between the temporal structure of problems and the memory hierarchy of computers. 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 readily available for the arithmetic unit. Delays increase as the data is placed in memory which is further away from the processor. Locality underlies the design and use by compilers of caches in "ordinary" sequential 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.

In Section 6, we briefly discuss the concept of problem architecture and its relation to the better understood computer architecture, in order to understand which problems are suitable for which computers. We also apply these ideas to compilers, which are viewed as mapping one space-time system into another.

Finally, in Section 7, we discuss the idea of *physical computation*, or adapting techniques from the physical sciences to create useful computational algorithms for such general problems as optimization. Such techniques are also applied to problems in parallel computing, such as balancing the computational load between processors.

2 Parallel Computers and Simulation

2.1 Parallel Computing

Carver Mead of Caltech in an intriguing public lecture once surveyed the impact of a number of new technologies, and introduced the idea of "headroom" — how much better a new technology needs to be for it to replace an older, more entrenched technology. Once the new technology has enough headroom, there will be a fairly rapid crossover from the old technology, in a kind of phase transition. For parallel computing the headroom needs to be large, perhaps a factor of 10 to 100, to outweigh the substantial new software investment required. The headroom will be larger for commercial applications where programs are generally much larger, and have a longer lifetime, than programs for academic research. Machines such as the nCUBE and Thinking Machines CM-2 were comparable in price/performance to conventional supercomputers, which was enough to show that "parallel computing works" [30, 32], but not enough to take over from conventional machines. It will be interesting to see whether the new batch of parallel computers, such as the CM-5, Intel Paragon, IBM SP-2, Maspar (DECmpp) MP-2, Cray T3D, etc., have enough headroom.

Parallel computers have two different models for accessing data:

- Shared Memory — processors access a common memory space,
- Distributed Memory — data is distributed over processors and accessed via message passing between processors,

and two different models for accessing instructions:

- SIMD (Single Instruction Multiple Data) — processors perform the same instruction synchronously on different data,
- MIMD (Multiple Instruction Multiple Data) — processors may perform different instructions on different data.

Different problems will generally run most efficiently on different computer architectures, so a range of different architectures will be available for the some time to come, including vector supercomputers, SIMD and MIMD parallel computers, and networks of RISC workstations. The user would prefer not to have to deal with the details of the different hardware, software, languages and programming models for the different classes of machines. So the aim of supercomputer centers is transparent distributed computing, sometimes called "metacomputing" — to provide simple, transparent access to a group of machines of different architectures, connected by a high speed network to each other and the outside world, and to data storage and visualization facilities. Users should be presented with a single system image, so they do not need to deal with different systems software, languages, software tools and libraries on each different machine. They should also be able to run an application across different machines on the network.

Parallel computing implies not only different computer architectures, but different languages, new software, new libraries, and will open up computation to new fields and new applications. It also offers a different way of viewing problems. Virtually all complex real-world problems are inherently parallel, in that many different elements of the problem domain interact with one another at any given time. In a sequential language, the structure of the problem must be artificially broken up to fit within the confines of the sequential computer, for which only one computation can occur at a time. Using parallel computers and parallel languages allows the programmer to better preserve the problem structure in the software, and perhaps also in the algorithm.

Over the last 10 years we have learned that parallel computing works — the majority of computationally intensive applications perform well on parallel computers, by taking advantage of the simple idea of "data parallelism", which means obtaining concurrency by applying the particular algorithm to different sections of the data set concurrently [38, 39, 30]. Data parallel applications are scalable to larger numbers of processors for larger amounts of data.

Another type of parallelism is "functional parallelism", where different processors (or even different computers) perform different functions, or different parts of the algorithm. Here the speed-ups obtained are usually more modest and this method is often not scalable, however it is important, particularly in multidisciplinary applications.

Surveys of problems in computational science [1, 8, 23] have shown that the vast majority (over 90%) of applications can be run effectively on MIMD parallel computers, and approximately 50% on SIMD machines (probably less for commercial, rather than academic, problems). Currently there are many different parallel architectures, but only one — a distributed memory MIMD multicomputer — is a general, high performance architecture that is known to scale from one to very many processors.

2.2 Parallel Languages

Using a parallel machine requires rewriting code written in standard sequential languages. We would like this rewrite to be as simple as possible, without sacrificing too much in performance. Parallelizing large codes involves substantial effort, and in many cases rewriting code more than once would be impractical. A good parallel language therefore needs to be portable and maintainable, that is, the code should run effectively on all current *and* future machines (at least those we can anticipate today). This means the language should be scalable, so that it can work effectively on machines using one or millions of processors. Portability also means that programs can be run in parallel over different machines across a network (distributed computing).

There are some completely new languages specifically designed to deal with parallelism, for example `occam`, however none are so compelling that they warrant adoption in precedence to adapting existing languages such as Fortran, `C`, `C++`, Ada, Lisp, Prolog, etc. This is because users have experience with existing languages, good sequential compilers exist and can be incorporated into parallel compilers, and migrating existing code to parallel machines is much easier. In any case, to be generally usable, especially for scientific computing, any new language would need to implement the standard features and libraries of `C` and Fortran [19, 21].

The purpose of software, and in particular computer languages, is to map a problem onto a machine, as described in Section 3.1. A drawback of current software and languages is that they are often designed around the *machine* architecture, rather than the *problem* architecture. This can make it very difficult to port the code from one machine to another, and in particular from a sequential computer to a parallel computer. It is possible for compilers to extract parallelism from a dependency analysis of sequential code (such as Fortran 77), however this is not usually very effective. In many cases the parallelism inherent in the problem will be obscured by the use of a sequential language or even a sequential algorithm. A particular application can be parallelized efficiently if, and only if, the details of the problem architecture are known. Users know the structure of their problems much better than compilers do, and can create their algorithms and programs accordingly. If the data structures are explicit, as in Fortran 90, then the parallelism becomes much clearer.

Each class of problem architectures requires different general constructs from the software, and a study of problem architectures is helpful in formulating the requirements for parallel languages and software (this is described in more detail in Section 6). Currently there are two language paradigms for distributed memory parallel computers: message passing and data parallel languages. Both of these have been implemented as extensions to Fortran and `C`. Here we will concentrate on Fortran.

2.2.1 Message Passing Fortran

Message passing is a natural model of programming distributed memory MIMD computers, and is currently used in the vast majority of successful applications using MIMD machines. The basic idea is that each *node* (processor plus local memory) has a program that controls, and performs calculations on, its own data (the "owner-computes" rule). Non-local data may need to be obtained from other nodes, which is done by communication of messages.

In its simplest form, there is one program per node of the computer. The programs can be different, although they are usually the same. However they will generally follow different threads of control, for example different branches of an `IF` statement. Communication can be asynchronous, but in most cases the algorithms are *loosely synchronous* [30], meaning that they are usually controlled by a time or iteration parameter and there is synchronization after every iteration, even though the communications during the iteration process may not be synchronous.

If parallelism is obtained from standard domain decomposition, then the parallel program for each node can look very similar to the sequential program, except that it computes only on local data, and has a call to a message passing routine to obtain non-local data. Schematically, a program might look something like the following:

CALL COMMUNICATE (*required non-local data*)
DO i *running over local data*
    CALL CALCULATE (*with* i*'s data*)
END DO

Note that it is more efficient to pass all the non-local data required in the loop as a single block before processing the local data, rather than pass each element of non-local data as it is needed within the loop. The advantages of this style of programming are:

- It is portable to both distributed and shared memory machines.
- It should scale to future machines, although to achieve good efficiencies schemes to overlap communication with itself and with calculation may be required.
- Languages are available now and are portable to many different MIMD machines. Current message passing language extensions include Express, PICL, PVM, and Linda.
- There will soon be an industry standard Message Passing Interface [48].
- All problems can be expressed using this method.

The disadvantages are:

- The user has complete control over transfer of data, which helps in creating efficient programs, but explicitly inserting all the communication calls is difficult, tedious, and error prone.
- Optimizations are not portable.
- It is only applicable to MIMD machines.

2.2.2 Data Parallel Fortran

The goal of the Fortran 90 standard is to "modernize Fortran, so that it may continue its long history as a scientific and engineering programming language". Although Fortran 90 is a sequential language, some of its major new features are the array operations to facilitate vector and data parallel programming.

Data parallel languages have distributed data just as for the message passing languages, however the data is explicitly written as a globally addressed array. As in the Fortran 90 array syntax, the expression

```
DIMENSION A(100,100), B(100,100), C(100,100)
A = B + C
```

is equivalent to

```
DO i = 1, 100
DO j = 1, 100
   A(i,j) = B(i,j) + C(i,j)
END DO
END DO
```

The first expression clearly allows easier exploitation of parallelism, especially as a `DO` loop of Fortran 77 can often be "accidentally" obscured, so a compiler can no longer see the equivalence to Fortran 90 array notation. Migration of data is also much simpler in a data parallel language. If the data required to do a calculation is on another processor, it will be automatically passed between nodes, without requiring explicit message passing calls set up by the user. For example, a program fragment might look something like the following, using either an array syntax with shifting operations to move data (as in Fortran 90)

A = B + SHIFT (C, *in* i *direction*)

or explicit parallel loops in a FORALL statement using standard array indices to indicate where the data is to be found (FORALL is not in the Fortran 90 standard, but is present in many dialects of data parallel Fortran)

```
FORALL i,j
A(i,j) = B(i,j) + C(i-1,j)
```

The advantages of this style of programming are:

- Relatively easy to use, since message passing is implicit rather than explicit, and parallelism can be based on simple Fortran 90 array extensions.
- Scalable and portable to both MIMD and SIMD machines.
- Should be able to handle all synchronous and loosely synchronous problems, including ones that only run well on MIMD.
- Data parallel languages such as CM Fortran and MasPar Fortran are available now that are based on Fortran 90 array syntax.
- An industry standard, High Performance Fortran (HPF), has been adopted, which is an extension of Fortran 90 that builds on existing data parallel languages [37, 42].

The disadvantages are:

- Need to wait for good HPF compilers.
- Not all problems can be expressed in this way.

2.2.3 High Performance Fortran

A major hindrance to the development of parallel computing has been the lack of portable, industry standard parallel languages. Currently, almost all parallel computer vendors provide their own proprietary parallel languages which are not portable even to machines of the same architecture, let alone between SIMD and MIMD, distributed or shared memory, parallel or vector architectures. This problem is now being addressed by the High Performance Fortran Forum (HPFF), a group of over 40 organizations including universities, national laboratories, computer and software vendors, and major industrial supercomputer users. HPFF was created to discuss and define a set of extensions to Fortran called High Performance Fortran. The goal was to address the problems of writing portable code that would run efficiently on any high performance computer, including parallel computers of any architecture (SIMD or MIMD, shared or distributed memory), vector computers, and RISC workstations. Here "efficiently" means "comparable to a program hand-coded by an expert in the native language of a particular machine".

The HPF standard was finalized in May 1993. HPF is designed to support data parallel programming. It is an extension of Fortran 90, which provides for array calculations and is therefore a natural starting point for a data parallel language. HPF attempts to deviate minimally from the Fortran 90 standard, while providing extensions that will enable compilers to provide good performance on a variety of parallel and vector architectures. While HPF was motivated by data parallel languages for SIMD machines, it was developed to enable such languages to be portable to any computer architecture, including MIMD, vector and sequential machines [19, 6, 4].

HPF has a number of new language features, including:

- New directives that suggest implementation and data distribution strategies to the compiler. They are structured so that a standard Fortran compiler will see them as comments and thus ignore them.
- New language syntax extending Fortran 90 to better express parallelism.
- Standard interfaces to a library of efficient parallel implementations of useful routines, such as sorting and matrix calculations.
- Access to extrinsic procedures which can be defined outside the language, for example by using Fortran with message passing, in order to handle certain operations that cannot be expressed very well (or at all) in HPF.

The strategy behind HPF is that the user writes in an SPMD (Single Program Multiple Data) data parallel style, with conceptually a single thread of control and globally accessible data. The program is annotated with assertions (compiler directives) giving information about desired data locality and distribution. The compiler then generates code implementing data and work distribution.

In the HPF model, the allocation of data to processors is done using a two-level mapping of data objects to processor memories, referred to as *abstract processors*. This is shown in Figure 1. First the data objects (typically array elements) are *aligned* relative to one another, using an abstract indexing space called a *template*. A template is then *distributed* onto a rectilinear arrangement of abstract processors. The final mapping of abstract processors to the same or a smaller number of physical processors is not specified by HPF, and is implementation dependent.

2.2.4 HPF Compilers and Fortran 90D

HPF is defined to be portable between computers of different architectures. As its name suggests, a major goal of High Performance Fortran is to have efficient compilers for all these machines. Vectorizing compilers work by analyzing data dependencies within loops, and identifying independent data sets that can be processed simultaneously. However, obtaining parallelism solely through

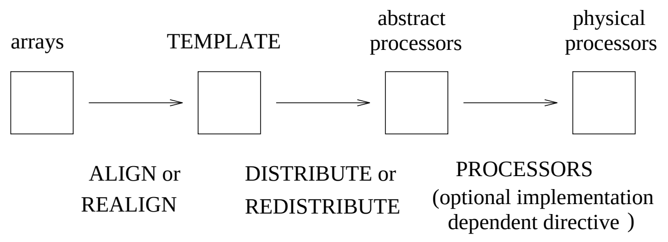

Figure 1: Mapping data onto processors in High Performance Fortran

dependency analysis has not proven to be effective in general, so for all commands in HPF the dependencies are directly implied, enabling the compiler to generate more efficient code.

Compilers will implement HPF differently on different computer architectures, for example:

SIMD computers — parallel code with communication optimized by compiler placement of data.

MIMD computers — a multi-threaded message passing code with local data and optimized send/receive communications.

vector computers — vectorized code optimized for the vector units.

RISC computers — pipelined superscalar code with compiler generated cache management.

A subset of HPF has been defined to enable early availability of compilers. The first implementation of HPF is the Fortran 90D compiler being produced by NPAC [4]. The alpha version of the compiler was demonstrated at Supercomputing 92, and the beta release is now available. The compiler currently runs on MIMD parallel computers: the nCUBE/2, Intel iPSC/860, and a network of Sun workstations. The next target architecture is heterogeneous networks, and in the future the compiler will be optimized for specific architectures and released as a commercial product by the Portland Group. An example of the performance of the current Fortran 90D compiler compared to a hand-coded message passing program is shown in Figure 2 for a Gaussian elimination problem. The HPF code for the main routine is shown in Figure 3.

Fortran 90D will continue to be developed as a superset of HPF, in order to research new functionality that may be added to the HPF standard in the future. For example, language facilities for handling parallel I/O are being investigated [3], which is a major area of concern that was not addressed by the initial HPF standard.

2.3 Systems Integration and Visualization

Recent advances in parallel programming languages such as High Performance Fortran are expected to improve the usability of parallel processing for the simulation of complex problems, especially in industry. However, complex "real world" computationally intensive applications in areas such as fluid dynamics, product design, or concurrent engineering require even more powerful and versatile tools. Such applications typically contain several modules with varying degrees of inter-modular interaction. Some modules such as digital signal processors, 3D renderers or partial differential equation (PDE) solvers map naturally onto the HPF programming model, while some others are inherently sequential. Also, a realistic application contains typically several data parallel modules, some of them interacting in the data parallel mode as well, for example different layers of a multigrid PDE algorithm, or subsequent filters in the machine vision systems. Finally, the process of integrating individual components into the full application is a complex task itself, and so is the process of

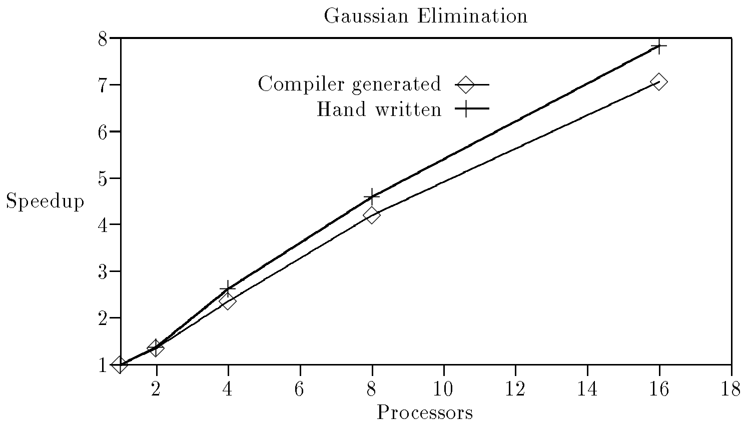

Figure 2: Performance of the Fortran 90D compiler versus hand-written message passing code on the Intel iPSC/860 for Gaussian elimination.

```
      PROGRAM gaussian

      PARAMETER (N = 100)
      PARAMETER (NN = 100)
      INTEGER index(N), iTmp(1)
      INTEGER indexRow, i, j, k
      REAL a(N,NN), row(NN), fac(N)
      REAL maxNum

!HPF$ PROCESSORS p(4)
!HPF$ TEMPLATE templ(100)
!HPF$ DISTRIBUTE templ (BLOCK) ONTO p
!HPF$ ALIGN a(*,i) WITH templ(i)
!HPF$ ALIGN row(i) WITH templ(i)

      index = -1
      DO k = 1, N
         iTmp = MAXLOC(a(:,k), MASK = index .EQ. -1)
         indexRow = iTmp(1)
         maxNum = a(indexRow,k)
         index(indexRow) = k
         fac = a(:,k) / maxNum
         row = a(indexRow,:)
         FORALL (i=1:N, j=k:NN, index(i) .EQ. -1)
     &        a(i,j) = a(i,j) - fac(i) * row(j)
      END DO
      END
```

Figure 3: Gaussian elimination programmed in High Performance Fortran

module synchronization, interactive debugging and fine-tuning the parameters of a prototype. New generation High Performance Distributed Computing (HPDC) software integration tools, required to handle this type of computational complexity, are currently being constructed. We summarize here recent activities in this area at NPAC.

Currently a popular approach is based on dataflow visualization systems such as AVS (Advanced Visualization System). This model supports a network of computational modules, implemented as individual UNIX processes, and interacting via the RPC (Remote Procedure Call) protocol under control of the AVS kernel. Individual modules can be placed on different machines and hence the model provides support for heterogeneous distributed computing. Some of these modules can also be installed on parallel platforms, thereby extending the paradigm to the HPDC level. Visual editing tools for such a network are also offered by the system which facilitiate application prototyping, integration, monitoring and fine-tuning. Finally, several default visualization modules come with the system and allow for sophisticated data visualization and rendering tasks.

AVS is only adequate for relatively static scientific visualization tasks. There is no support for system wide synchronization nor for real-time interactive services required for advanced simulation tasks such as virtual reality (VR). Also, while parallel (e.g. HPF) modules can be included in the AVS network, there is no support for parallel I/O or parallel dataflow between individual HPF tasks. All communication must be mediated by the corresponding host programs, which causes substantial bottlenecks.

We are currently developing a set of tools that will allow us to extend AVS functionality in these areas and to provide support for virtual reality simulations as well as for televirtuality services providing remote VR user interfaces. The underlying software model is provided by the MOVIE (Multitasking Object-oriented Visual Interactive Environment) model [35, 36]. A MOVIE system is a network of MOVIE servers — interpreters of a high-level object-oriented programming language, MovieScript. MovieScript extends PostScript in areas such as graphical user interface (GUI) prototyping, Fortran 90 style array syntax and operating support for real-time multi-threading.

A specific design of a MOVIE network can be adapted to a particular computational domain. In particular, all dynamic features of the AVS model in the heterogenous distributed mode can be reproduced in terms of MOVIE tasks or threads, but the model also offers support for multitasking data parallel processing and interactive real-time programming. In the early development stage is the next level tool, which will allow for concurrent execution of and parallel dataflow between several HPF modules. An AVS-like visual network editor will also be provided to facilitate application editing tasks. Finally, we also plan to provide support for World Wide Web (WWW) services in terms of CGI (Common Gateway Interface) scripts. Such scripts, passed from the Web browser to the Web server to the MOVIE host server will allow for interactive control of simulations running on remote high performance computers. This architecture will also enable the development of prototype televirtuality (TVR) services. Current hypermedia browsers such as Mosaic are not adequate to support two-way interactivity but there are ongoing VR-oriented activities in the WWW community, and several new consumer level VR products will soon offer remote support as well.

3 Complex Systems and a Theory of Parallel Computing

3.1 Mapping Problems onto Computers

For this article, we shall consider a *complex system* as 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 probabilistic or deterministic set of rules that relate the complex system at a later "time" to its state at an earlier "time". Complex systems studied in chemistry and physics, such as a protein or the universe, obey rules that we believe we understand more or less accurately. The military play war games, which is the complex system formed by a military engagement. This and more general complex systems found in society, obey less clear rules.

One particular important class of complex systems is that of the *complex computer*. In the case of a hypercube such as the nCUBE, or other multicomputers such as the Intel Paragon or Thinking Machines CM-5, the basic entity in the complex system 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.

In many situations, one is concerned with mapping one complex system into another. 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 modeling or simulating a particular complex problem involves mapping it onto the complex computer.

Simulation or modeling begins with a map

$$\text{Nature (or system to be modelled)} \quad \overset{\text{map}}{\underset{\text{theory}}{\longrightarrow}} \quad \text{Idealization or Model} \tag{1}$$

This map would often be followed by a computer simulation, which consists of mapping the model onto the computer. This whole process can be broken up into several maps, as shown in Figure 4. We illustrate the procedure using the example of a computational fluid dynamics study of airflow around an airplane, where the complex systems used are:

S_0 is nature — the actual flow of air around the airplane.

S_1 is a (finite) collection of molecules interacting with long-range Van der Waals and other forces. This interaction defines a complete interconnect between all members of the complex system S_1.

S_2 is the infinite degree of freedom continuum with the fundamental entities as infinitesimal volumes of air connected locally by the partial differential operator of the Navier Stokes equation.

$S_3 = S_{\text{num}}$ could depend on the particular numerical formulation used. Multigrid, conjugate gradient, direct matrix inversion and alternating gradient would have very different structures in the direct numerical solution of the Navier Stokes equations. The more radical cellular automata approach would be quite different again.

$S_4 = S_{\text{HLSoft}}$ would depend on the final computer being used and division between high and low level in software. The label HLSoft denotes "High Level Software".

$S_5 = S_{\text{comp}}$ would be S_{HLSoft} embroidered by the details of the hardware communication (circuit or packet switching, wormhole or other routing). Further, we would often need to look at this complex system in greater resolution and expose the details of the processor node architecture.

Nature, the model, the numerical formulation, the software, and the computer are all complex systems, and they can be quite different. We are interested in the structure of all these complex systems and the maps between them. Note that each of the successive maps in Figure 4 results in a loss of information. As reviewed in Section 6, we can discuss key problems in the design of software systems in terms of minimizing information loss.

Typically, one is interested in constructing the maps in Figure 4 to satisfy certain goals, such as minimizing the execution time of the computer simulation (the main focus of the high performance computing community), minimizing the time required to write the computer program (the main focus of the computer science and software engineering community), and obtaining the best agreement of the model with the effects seen in nature (the main focus of the scientific community). We therefore get a class of optimization problems associated with the different complex systems and the mappings between them. Parallel computing can therefore be looked at as "just" an optimization problem, even if we can't agree on exactly what to optimize — there are obvious tradeoffs between fidelity of the model and the amount of computation required to solve it, the speed of the program and the ease of implementation (for example using assembler versus a high level language), and so on.

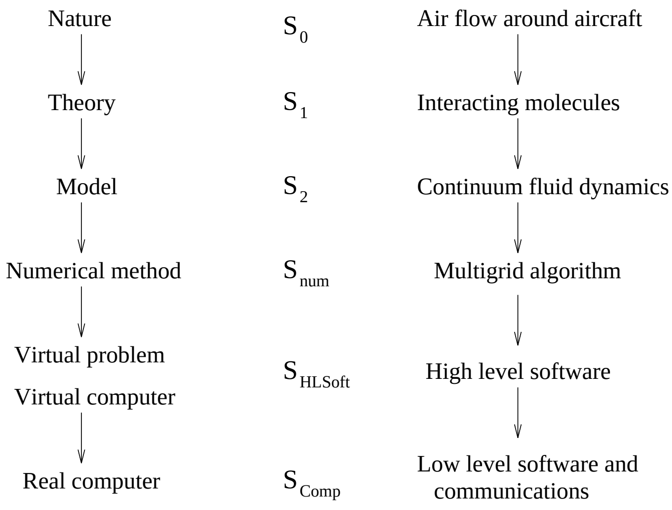

Figure 4: Computation and simulation as a series of maps

One approach to solving these optimization problems is the use of methods developed from the study of complex physical systems, such as simulated annealing, genetic algorithms, or neural networks. These are used to minimize a cost function that expresses the goals described above. Typically, in studying performance, the cost function would be the execution time of the problem on a computer. For software engineering, the cost function would also reflect user productivity. These physical optimization methods were originally developed as ways of minimizing the energy of a physical system. In the rest of this section, and in sections 4 and 5, we will show that computational problems can be looked at using a space-time analogy to a physical system, so that the cost function for optimizing the map from complex problem to complex computer does in fact resemble the energy function of a physical system. This motivates the use of these physical optimization techniques for solving these problems, an approach we refer to as *physical computation*, which is discussed in Section 7.

In this chapter we will concentrate on the mappings in Figure 4 that take us from the model of the world (the complex problem) to the simulation of that model on a parallel computer (the complex computer). Mapping a complex problem onto a complex computer involves *decomposition*. We can consider the simulation of the complex problem as an *algorithm* applied to a *data domain*. We divide the data domain into pieces that we call *grains* and place one grain in each node of the concurrent computer.

If we consider a typical matrix algorithm such as multiplication

$$a_{ij} = \sum_k b_{ik}\, c_{kj} \tag{2}$$

we have a data domain formed by the matrix elements, which we generally call *members*. The algorithm (2) defines a graph connecting these members and these connected members form a complex system. The standard *decomposition* involves submatrices stored in each node. Edges of the graph connect members in different submatrices (i.e., members of the complex system stored in separate nodes of the complex computer). To be precise, in the map

Complex Problem	$\longrightarrow$	**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 4, we will be considering topological properties of complex systems which correspond to the map

Complex Problem	$\longrightarrow$	**Topological Structure**
Members	map into	points in a space geometric
Connections	map into	(nearest neighbor) structure

In the optimal decomposition studies in Section 4 and Section 5, we will be considering dynamic properties of complex systems for which it will be useful to consider the map

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 systems realize their members and interconnections in different ways. We find it very useful to map general systems into classes having 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 multicomputer implementation. On the other hand, in the simulated annealing approach to load balancing, we map all these interconnects to forces.

3.2 The Space-Time Picture of Computation

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 discrete objects labelled by an index i and is described as a function of the positions $\underline{x}_i(t)$ of the objects at each time t.

For example, seismic exploration for oil fields involves measuring echoes of sound waves that are reflected off various underground strata. Using these measurements to reconstruct the strata formation involves solving the wave equation, a standard second order differential equation that describes the propagation of the sound waves. The equation is discretized in space (a three-dimensional grid representing some part of the earth's crust) and time (the time-step) to give a finite difference equation that can be solved on a computer. Only local data (nearest-neighbor points in the grid) is required to solve the difference equations at each time-step.

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 3.1), "space" is the set of nodes of the parallel computer, or at a finer resolution, the set of memory locations.

The data domain will have certain dependencies contingent on the model, for example a dependence on nearest-neighbor grid points for problems with local interactions, or a dependence on all other data points for N-body simulations where forces are long-ranged. The data domain can thus be viewed as a set of interconnected nodes (the data elements) connected by edges (the data dependencies), which form what we call the *computational graph*. This is defined by a time slice of the full complex system.

Note that we can examine the data domain of the complex computer hardware in terms of a computational graph, just as we can for the computer software or algorithms. The computational graph of a multicomputer is formed by the individual computer nodes with the edges of the graph determined by the interconnection topology (or architecture) of the multicomputer.

In a physical simulation, the complex system evolves with time and is specified by the nature of the computational graph at each time-step. 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. For 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. Complex computer: the time dependence is just the evolution given by either the cycle time of the nodes or the executed instructions. SIMD machines give an essentially static or synchronous time dependence, whereas MIMD machines can be very dynamic.

We expand the discussion of temporal properties in Section 5. We will also discuss in Section 6 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.

Domain decomposition for data parallel computing is just the mapping of the spatial domain (data) of the complex problem onto the spatial domain (nodes) of the complex computer. This differs from the computational model for a sequential computer, where all aspects of the problem are mapped to the time domain of the computer. In contrast, another type of data processor, a seismograph, maps the time dependence of an earthquake onto the spatial extent of a recording chart. The general problem of computation is to map the space-time domain of the problem onto the space-time domain of the computer in an effective way.

4 Spatial Properties of Complex Problems and Complex Computers

4.1 System Size and Geometry

The *size* N of the complex system is an obviously important property. Note that we think of a complex system as a set of *members* with their spatial structure evolving with time. Sometimes the time domain has a definite "size", but often one can evolve the system indefinitely in time. However, most complex systems have a natural spatial size with the spatial domain consisting of N members.

In the matrix example, Gaussian elimination has n^2 spatial members (matrix elements) evolving for a fixed number n of "time" steps. As usual, the value of the spatial size N will depend on the granularity or detail with which one looks at the complex system. One could consider a parallel computer at the level of transistors with very large value of N, but usually we look at the processor node as the fundamental entity and define the spatial size of a parallel computer viewed as a complex system, by the number N_{proc} of processing nodes.

Consider mapping a finite difference simulation with N_{num} grid points, such as solving the wave equation for a seismic exploration simulation as described in Section 3.2, onto a parallel machine with N_{proc} processors. An important parameter is the *grain size* n of the resultant decomposition. We can introduce the *problem* grain size $n_{\mathrm{num}} = N_{\mathrm{num}}/N_{\mathrm{proc}}$, and the *computer* grain size n_{mem} as the memory contained in each node of the parallel computer. Clearly we must have $n_{\mathrm{num}} < n_{\mathrm{mem}}$ if we measure memory size in units of seismic grid points. More interestingly, we will later relate the performance of the parallel implementation of the seismic simulation to n_{num} and other problem and computer characteristics. We find that in many cases, the parallel performance only depends on N_{num} and N_{proc} in the combination $N_{\mathrm{num}}/N_{\mathrm{proc}}$ and so grain size is a critical parameter in determining the effectiveness of parallel computers for a particular application.

Another set of parameters describe the topology or structure of the spatial domain associated with the complex system. The simplest parameter of this type is the geometric dimension d^{geom} of the space. Our early parallel computing used the binary hypercube of dimension d, which has $d_{\mathrm{comp}} = d$ as its geometric dimension. This was an effective architecture because it was richer than the topologies of most problems. For many physical grid-based simulations such as seismic exploration, the geometric dimension of the problem d_{num} is just the dimension of the physical space being simulated (3 for this example). The performance of the simulation also depends on whether the software system preserves the spatial structure of the problem, in which case $d_{\mathrm{HLSoft}} = d_{\mathrm{num}}$.

4.2 Performance Model for a Multicomputer

The performance of a multicomputer is usually defined in terms of parallel *speedup* S and *efficiency* ε. Speedup is just how much faster a multicomputer executes the parallel program on N nodes compared to the sequential program on one node. Efficiency measures what fraction of the maximum speedup N is actually achieved, so that

$$S = \varepsilon N \tag{3}$$

Efficiency will usually be less than 1 since there are overheads involved in parallel computing, such as the cost of communicating data between processors. Let us try to quantify these costs by defining the following parameters for a multicomputer:

- t_{calc} — the typical time required to perform a generic calculation. For scientific problems, this can be taken as a floating point operation.
- t_{comm} — the typical time taken to communicate a single word between two nodes connected in the hardware topology.

The definitions of t_{comm} and t_{calc} are imprecise above. In particular, t_{calc} depends on the nature of node and can take on very different values depending on the details of the implementation; floating point operations are much faster from registers than from slower parts of the memory hierarchy. On systems built from processors like the Intel i860 chip, these effects can be large — t_{calc} could be $.0125\mu$sec from registers (80 Megaflops) and a factor of ten larger when the variables a, b are fetched from dynamic RAM. Communication speed t_{comm} depends on internode message size (a software characteristic) and the latency (startup time) and bandwidth of the computer communication subsystem. It will also generally be slower for communications between nodes that are not directly connected in the multicomputer topology, so that messages have to be routed between intermediary nodes. This effect will depend on the problem being solved — it can be negligible for grid-based problems with only local data dependencies (such as the seismic simulation), or a factor of 2 or more for problems with a lot of non-local communication (such as a parallel Fast Fourier Transform).

The overhead f_C due to communication can be expressed as

$$f_C = \frac{\text{total time for communication}}{\text{total time for calculation}} \tag{4}$$

It is easy to see that if the parallel overhead is due solely to communication, then

$$S = \frac{N}{1 + f_C} \tag{5}$$

Let us examine the communication overhead for a simple grid-based problem, such as our seismic simulation example. We use standard domain decomposition to map the problem domain (a $d_{\rm num} = 3$ grid of points) onto the computer (a hypercube, for example) so that every processor has a cubic section of the grid. The grain size $n_{\rm num}$ will be $L^{d_{\rm num}}$, where L is the length of a side of the grain (the cube of grid points on each processor).

The total amount of computation on each node of the computer will be proportional to $t_{\rm calc}$ times the grain size, which is the volume $L^{d_{\rm num}}$ of the grain. For this problem, the data dependencies are all local (nearest-neighbor), so only data at the edge of the grain needs to be communicated. The total amount of communication is thus proportional to $t_{\rm comm}$ times the surface area $L^{d_{\rm num}-1}$ of the grain. So from Equation 4 we have that

$$\begin{aligned} f_C &\propto \frac{1}{L}\,\frac{t_{\rm comm}}{t_{\rm calc}} \\ &\propto \frac{1}{n_{\rm num}^{1/d_{\rm num}}}\,\frac{t_{\rm comm}}{t_{\rm calc}} \end{aligned} \tag{6}$$

It can be shown [24] that in general the overhead due to internode communication can be written in the form

$$f_C \propto \frac{N_{\rm proc}^{\alpha}}{n_{\rm num}^{\beta}}\,\frac{t_{\rm comm}}{t_{\rm calc}} \tag{7}$$

The term $t_{\rm comm}/t_{\rm calc}$ indicates that communication overhead depends on the relative performance of the internode communication system and node (floating point) processing unit. A real study of parallel computer performance would require a deeper discussion of the exact values of $t_{\rm comm}$ and $t_{\rm calc}$. More interesting here is the dependence on the number of processors $N_{\rm proc}$ and the problem grain size $n_{\rm num}$. As described above, grain size $n_{\rm num} = N_{\rm num}/N_{\rm proc}$ depends on both the problem and the computer. The value of β is given by

$$\beta = \frac{1}{d_{\rm info}} \tag{8}$$

where the *information dimension* $d_{\rm info}$ is a generalization of the geometric dimension for problems whose structure is not geometrically based. This will be described in the next subsection. It is independent of the parameters of the computer. α is given by

$$\begin{aligned} &\text{if} \quad d_{\rm num} < d_{\rm comp} \quad , \quad \alpha = 0 \\ &\text{if} \quad d_{\rm num} > d_{\rm comp} \quad , \quad \alpha = \left(\frac{1}{d_{\rm comp}} - \frac{1}{d_{\rm num}}\right) \end{aligned} \tag{9}$$

which quantifies the penalty, in terms of a value of f_C that increases with $N_{\rm proc}$, for a computer architecture that is less rich than the problem architecture. An attractive feature of the hypercube architecture is that $d_{\rm comp}$ is large and one is essentially always in the regime governed by $\alpha = 0$ in

Equation 9. Recently, there has been a trend away from rich topologies like the hypercube towards the view that the node interconnect should be considered as a routing network or switch to be implemented in the very best technology. The original MIMD machines from Intel, nCUBE and Ametek all used hypercube topologies as did the SIMD Connection Machine CM-1 and CM-2. The nCUBE-2 introduced in 1990 still uses a hypercube topology, but both it and the second generation Intel iPSC/2 used more sophisticated routing. The latest Intel Touchstone Delta and Paragon use a two-dimensional mesh with wormhole routing. It is not clear how to incorporate these new node interconnects into the above picture, and further research is needed here. Presumably, we would need to add new complex system properties and perhaps generalize the definition of dimension d_{comp}, as we will now do for d_{num} in order for Equation 7 to be valid for problems whose structure is not geometrically based.

4.3 Information Dimension

Returning to equations 5, 7, 8, and 9 we note that we have not properly defined the correct dimension d_{num} or d_{comp} to use. We have implicitly equated this to the natural *geometric dimension* but this is not always correct. This is illustrated by the complex system S_{num} consisting of a set of particles in three dimensions interacting with a long-range force such as gravity or electrostatic charge. The geometric structure is local with $d_{\text{num}}^{\text{geom}} = 3$ but the complex system structure is quite different; all particles are connected to all others.

We define the *information dimension* d^{info} for a general complex system to reflect the system connectivity [30, 32]. This is analogous to the fractal dimension introduced in [44], in that it may not be equal to the geometric dimension, and need not be an integer. Consider Figure 5 which shows a general domain D in a complex system. We define the volume V_D of this domain by the information in it. Mathematically, V_D is the computational complexity needed to simulate D in isolation. In a geometric system

$$V_D \propto L^{d^{\text{geom}}} \tag{10}$$

where L is a geometric length scale. The domain D is not in general isolated and is connected to the rest of the complex system. Information I_D flows into D, and again in a geometric system I_D is a surface effect with

$$I_D \propto L^{d^{\text{geom}}-1} \tag{11}$$

If we view the complex system as a graph (i.e., the computational graph), V_D is related to the number of edges of the graph with at least one of the nodes in D, and I_D is related to the number of edges cut by the surface of D. Equation 10 and Equation 11 are altered in cases like the long-range force problem where the complex system connectivity is no longer geometric. We define the information dimension to preserve the surface versus volume interpretation of Equation 11 compared to Equation 10. Thus, generally we define

$$I_D \propto V_D^{1-1/d^{\text{info}}} \tag{12}$$

With this definition of information dimension d^{info}, we find that Equations 5, 7, 8, and 9 essentially hold in general. For simple problems, the information dimension will be approximately equal to the geometric dimension. However the information dimension will in general be larger for systems with complex structure, which have non-geometric (or "hidden") dimensions of complexity.

An interesting example of nontrivial information dimension comes from the simulation of electronic circuits. Rent's Rule [43, 9] is a phenomenological rule that is used in the packaging of circuits. It relates the number of output lines (pinouts) to a power ($\approx 0.5 \rightarrow 0.7$) of the number of internal components. This implies a non-integer information dimension $d^{\text{info}} \approx 3$, which is greater than the geometric dimension $d^{\text{geom}} = 2$ for an electronic circuit. Rent's Rule is approximately independent of the size of the circuit, which is analogous to the self-similarity and scaling properties of systems with non-trivial fractal dimension.

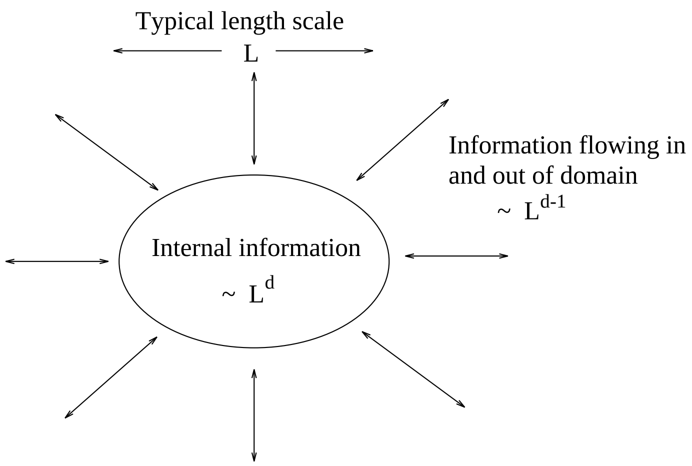

Figure 5: The information density and flow in a general complex system of length L

For the long-range force problem, it can be shown that $d^{\rm info} = 1$ independent of $d^{\rm geom}$ [24, 32]. One might naively expect that the information dimension of such a problem would be infinite, rather than 1, since all objects interact with all other objects. However infinite information dimension applies to systems such as the telephone network, for which everyone is connected to everyone else, but *different* information is communicated to every different person. In contrast, the Voice of America radio broadcast has an information dimension of 1, since the *same* information is communicated to everyone. In the long-range force problem, the same information is broadcast by an object to every other object (e.g., the mass and position of the object for an N-body gravitational interaction problem), so the information dimension is 1.

4.4 A Physical Analogy for Domain Decomposition

In the previous three subsections, we described static spatial properties of complex systems that are 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 where the use of a parallel computer requires that the universe be divided into domains that, due to the gravitational interactions, will change as the simulation evolves.

The performance of a computation executing on a parallel machine is crucially dependent on *load balance.* This refers 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 load balanced. This balance is often not easy to achieve, however.

As described in the previous section, a key to parallel computing is to split the underlying spatial domain into grains, each of which 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 the astrophysical simulation. 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

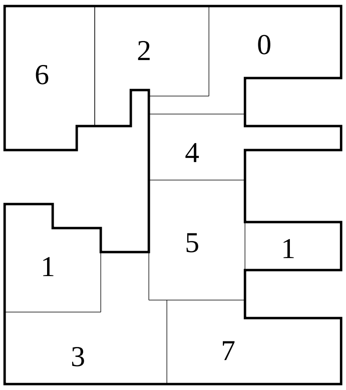

Figure 6: A mapping of an irregular data domain onto processors in a hypercube

— 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 realize a reasonable fraction (say 80%) of the potential computing power of the parallel machine for a wide variety of computations. An example of a non-trivial domain decomposition of an irregular data domain onto processors configured in a hypercube is given in Figure 6.

One can usefully think of a parallel computation in terms of a physical analogy. Treat the processes (or the data elements) as "particles" free to move about in the "space" of the parallel machine. Minimizing the total execution time of the parallel computation formally requires that one minimize:

$$\max_{\text{nodes}\, i} C_i \tag{13}$$

where C_i is the total computation time for calculation and communication. We choose to replace this mini-max problem by a least squares minimization [13] of

$$E = \sum_i C_i^2 \tag{14}$$

Let m label the nodal points and (m, m') the edges of the computational graph. Then

$$C_i = \sum_{m \in i} \left[\sum_{(m,m')} Comm\,(m,\, m') + Calc\,(m) \right] \tag{15}$$

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

$$C_i^2 \sim \sum_{\substack{(\mathrm{m,m'}) \\ \mathrm{m} \in \mathrm{i}}} Comm\,(m,\, m')$$

$$+ \sum_{m,\, m' \in i} Calc(m)\, Calc(m') \tag{16}$$

In this physical analogy, the above equation describes a "Hamiltonian" (or "energy function") for parallel computation, that the operating system must try to minimize, and if possible find the "ground state" (the lowest energy state), which corresponds to the most efficient decomposition of data onto nodes of the parallel computer.

The last term in the Hamiltonian (Equation 16) is zero unless particles m and m' are at the same place, i.e., in the same node. In the physical analogy, this is like a short-range "potential", where range is measured by distance between nodes in the space of the complex computer. This provides a short-range, repulsive "force", causing the particles, and thereby the computation, to spread throughout the parallel computer in an evenhanded, balanced manner, corresponding to the requirement of load balancing.

A conflicting requirement to that of load balancing is shown in the first term of the Hamiltonian as 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. This represents a long-range, attractive force between those pairs of particles which need to communicate with one another. This force is 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.

Exact minimization of the function in Equation 16 is not necessary — we have already "wasted" some computational power using convenient high level languages, and we can surely afford to lose another 10% to load imbalance. The problem of distributing a computation onto a parallel machine in an efficient manner can therefore be fruitfully attacked using simulated annealing [41] and other "physical" optimization methods such as neural networks and genetic algorithms [11, 25, 27, 13, 55, 22, 45]. The physical analogy described above makes this plausible, since these methods are highly appropriate minimization techniques for Equation 14. For example, simulated annealing is a standard Monte Carlo technique that was originally devised to find the ground states of spin models of magnetism that have competing interactions, such as spin glasses. In this case, the competing interactions are the attractive "communication" force and the repulsive "load balance" force. We have used these methods routinely for load balancing a variety of simulations including finite element and particle dynamics simulations. Physical optimization methods are described in more detail in Section 7.

4.5 System Temperature and Dynamic Load Balancing

Equation 16 holds for the case of static load balancing, that is, where the data structures are static, so that the domain decomposition is done only once, at the beginning of the computation. However in general, problems and the data structures and computational graphs that describe them will be changing. In this case the data will have to be redistributed throughout the computation in order to keep the load balanced. For dynamic load balancing there will be an extra attractive "force" in Equation 16, corresponding to the penalty for moving data or processes to different nodes.

High Performance Fortran provides a mechanism for the redistribution of data at runtime, which is particularly important for dynamic and irregular problems. The user can either specify the distribution, or specify the computational graph, in which case the compiler will find a good distribution using various optimization techniques such as those described above [5, 51].

Using the physical analogy we introduced in Section 4.4, we can think of the operating system as a "heat bath" that keeps the computation "cool" and therefore near its "ground state" (optimal solution). 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 complex system *temperature* T_{problem} which reflects the ease of finding a reasonable minimum. T_{problem} will be larger for problems that change more rapidly and where the operating system does not have "time" to find as good an equilibrium [34, 33, 24]. Redistribution of the data takes time, and for some problems this time may be significant, perhaps even longer than the simulation time between data redistribution. However there is a simple static data distribution called *scattered decomposition* that can work quite well for very irregular and dynamic problems.

Standard static domain decomposition works by splitting the data space into large connected partitions and mapping the partitions onto nodes in the computer space in a way that preserves data locality and balances the computational load. However this static decomposition will produce substantial load imbalance for problems with dynamic data structures that produce "hot spots", or data intensive regions, that change position during the simulation, for example in N-body simulations of galactic collisions, where particles clump together due to gravitational attraction.

Scattered decomposition works by going to the opposite extreme, that is, breaking up the data into very small partitions and then "scattering" the partitions among the nodes of the computer, so that each node receives some data from all regions of the problem space [49, 30, 24]. In this case any data hot spots will also get distributed fairly evenly among the processors. As the partitions are made smaller, the load balancing will improve. The price paid, of course, is increased communication overhead. The scattered decomposition will require much more communication traffic than the standard domain decomposition. Often, however, communication between nodes is relatively cheap compared to the computation required, and so the scattered decomposition becomes an attractive possibility. This corresponds to ignoring the first term in Equation 16 (the communications cost) and trying to minimize only the second term, which we have seen requires spreading data connected in the computational graph to different nodes, which is exactly what scattered decomposition attempts to achieve.

One of the outstanding features of the scattered decomposition is its *stability*, meaning that as the computation changes with time (e.g., particles move, clumping occurs, etc.), the scattered decomposition is quite insensitive to these changes and will continue to load balance rather well. So it is possible to get good load balance without having to use time-consuming optimization techniques such as simulated annealing to obtain good data distribution. Scattered decomposition has proven to be very effective for problems such as adaptive mesh finite element simulations, where the grid is much finer in regions that are changing more rapidly [49, 11]; growing a cluster in spin models of magnetism [2]; and certain matrix problems [30].

For a dynamic problem, the Hamiltonian of Equation 16 will vary with (computer) time, and so will its minimum value (the optimal domain decomposition). The operating system will need to redistribute the data periodically to try to keep the system close to the global minimum value. In contrast, the static scattered decomposition presumably corresponds to a stable local minimum of the Hamiltonian that does not change much with time. For highly dynamic problems, the operating system may not be able to "keep up" perfectly with the computation. In this case the Hamiltonian that actually matters is not the instantaneous version in Equation 16, but a time-averaged Hamiltonian, $\overline{H}$:

$$\overline{H}(t,\, t_{\text{av}}) = \int_t^{t+t_{\text{av}}} H(u)\, du \tag{17}$$

where the averaging time, t_{av} is the time scale for the operating system to find a good domain decomposition. In the earlier terminology, t_{av} is related to the "temperature" T_{problem} of the complex system. Note that t_{av} and the temperature are in fact characteristics of the problem, not the computer. t_{av} will be smaller for a faster computer, however what is important here is the *relative* time scale of the operating system — the time taken to compute a new domain decomposition relative to the time to do the computations between redistributing the data. Increasing the speed of the computer will decrease the time for both these tasks in roughly equal proportion, so the relative time scale will remain about the same.

An interesting point is that, in terms of $\overline{H}$, the better decomposition may actually be the scattered one. Because of the rapid shifting of the optimal decomposition as a function of time, the minimum of $\overline{H}$ corresponding to this will be raised upwards, while the scattered minimum will remain approximately the same. 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 $\overline{H}$.

The relative importance of the two terms in Equation 16 is governed by the ratio $t_{\mathrm{comm}}/t_{\mathrm{calc}}$ introduced in Section 4.2. This plays the role of a "coupling constant" or "interaction strength" J, such as occurs in Hamiltonians for spin glasses and other spin models of magnetism. J increases in size as the communication performance of the hardware decreases. The scattered decomposition is favored as either the coupling J decreases, which means communications are relatively fast so the non-locality of the data is not a problem, or as the averaging time t_{av} increases. Large t_{av} corresponds to rapidly varying problems which the operating system finds hard to equilibrate. In the earlier terminology, large t_{av} means high "temperature" complex systems.

Thus, as we increase J or decrease problem temperature, we transition from a high temperature phase, where scattered decomposition is optimal, to a low temperature phase where standard domain decomposition is optimal. This is of course analogous to a statistical mechanics system having a phase transition separating a high temperature disordered state and a low temperature ordered state. Which phase or data decomposition is relevant depends on the properties of both the computer architecture ($J \sim t_{\mathrm{comm}}/t_{\mathrm{calc}}$) and the problem architecture ($t_{\mathrm{av}} \sim T_{\mathrm{problem}}$).

5 Temporal Properties of Complex Problems and Complex Computers

5.1 The String Formalism for Dynamic Problems

In the previous section, we thought of a problem (the complex system S_{num} or S_{HLSoft}) as a graph (the computational graph) with vertices labelled by the system member m and edges corresponding to the linkage between members established by the algorithm. This is a good picture for what we called "adiabatic" problems that change slowly with time. In this case, it makes sense to think of slicing the "space-time" cylinder formed by the complex system and just consider the computational graph — the spatial structure at fixed time. However, this is not appropriate for asynchronous problems or for loosely synchronous problems that are rapidly varying or *dynamic* — those with high temperature T_{problem} in the language of Section 4.4. For such problems, the operating system cannot "keep up" with the variation of the computational graph — the graph changes significantly over the time period that the operating system takes to partition the computational graph.

In adiabatic problems, our physical analogy was that of members mapped to particles interacting by forces given by the member interconnect. One might imagine that a reasonable analogy for *dynamic* problems would be to add a "kinetic energy" term to give time dependence to the member positions, however it is not clear how to do this. Rather, we change the analogy so that members are mapped to "strings" representing their *world lines*, that is, their path through the space-time of the complex computer. At computer time t, the complex system member m is located at position $\underline{x}_m(t)$. $\underline{x}$ is a position in the complex computer space. At its simplest $\underline{x}$ is just a node number, but we can look at a finer resolution and consider $\underline{x}$ as a position in the global computer memory. This allows one, in principle at least, to set up a formalism to study the full memory hierarchy of the system including caches and register use. Each member now corresponds not just to a position $\underline{x}_m$ but to a world line $\{\underline{x}_m(t)\}$. The execution time T_{par} on a parallel machine is a functional of the world lines

$$T_{\mathrm{par}} \equiv T_{\mathrm{par}}\left(\{\underline{x}_0(t)\}\ldots\ldots\{\underline{x}_m(t)\}\ldots\right) \tag{18}$$

The structure of the original dynamic complex system leads to an expression for Equation 18 that is similar to the simpler Equation 16. There is a repulsive force between world lines corresponding to load balancing. There is an attractive force corresponding to the dynamic interconnection between the members m. The details of this depend on the relation between clock time t and the simulation time t_m of each member m.

The most straightforward approach to minimize T_{par} would be simulating annealing with the basic "move" being a change $\{\underline{x}_m(t)\} \rightarrow \{\underline{x}_m(t)\}'$ which is typically local in both $\underline{x}$ and t. This gives a formalism similar to quantum chemistry or lattice gauge theories. One can also use an optimization method based on neural networks. These points are described in greater detail in Section 7.

We have applied these ideas to message routing in a network [26], and more generally to combining networks which implement global reduction formulae such as forming a set of sums

$$y_j = \sum_i M_{ji}\, x_i \tag{19}$$

where y_j, M_{ji}, and x_i are all distributed over the nodes of a parallel computer.

A very preliminary examination was given in [31] of the application of these ideas to register allocation for compilers. We have explored more deeply the application of these methods to multi-vehicle navigation [27, 16]. In that case $\{\underline{x}_m(t)\}$ is the path of vehicle m in a two or three dimensional space with m at position $\underline{x}_m$ at time t.

5.2 Memory Hierarchy

Modern workstations have heirarchical memory, formed by the cache and main memory. Obtaining good performance from these computers requires minimizing cache misses, so that data is referenced from cache and not main memory, which can be an order of magnitude slower. This is often referred to as the need for "data locality". This makes clear the analogy with distributed memory parallel computers, where data locality is needed to minimize communication between processors.

There is one essential difference between cache and distributed memory. Both need data locality, but in the parallel case the basic data is static and fetches additional information as necessary. This gives the familiar surface-over-volume communications overheads of Equation 7. However, in the case of a cache, all data must stream through it, not just the data needed to provide additional information. We can use our space-time picture of computation to view the data streaming through heirarchical memory as a distribution of data in the temporal direction.

Let us introduce a new time constant, t_{mem}, which is the time it takes to load a word into cache. This is illustrated in Figure 7. The cache overhead has exactly the same form as the communication overhead in Equation 7, if we simply replace t_{comm} by t_{mem} and n_{num} for n_{time}, where n_{time} is the temporal blocking factor, or the number of iterations in the problem between cache flushes. The overhead is a surface-over-volume effect just as for a distributed memory machine, but now the surface is in the temporal direction, and the volume is that of a domain in both space and time.

It is remarkable that t_{mem}, time, and memory hierarchy are completely analogous to t_{comm}, space, and distributed memory. In particular, the well-known methods for improving the performance of caches and registers correspond to blocking (clumping) the problem in its time direction. This is analogous to blocking the problem in space to improve performance on a distributed memory parallel machine.

High performance computer architectures exploit data locality with a memory hierarchy implemented either as a multilevel cache and/or with distributed memory on a parallel machine. Good use of cache requires blocking in time; good use of distributed memory requires blocking in space. In general, full space-time blocking is required to give a universal implementation of data locality that will lead to good performance on both distributed and heirarchical memory machines. This strategy is used in the implementation of the BLAS-3 primitives in LAPACK [7].

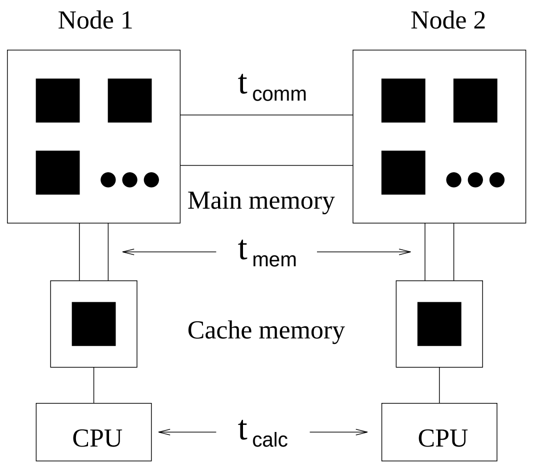

Figure 7: The fundamental time constants of a heirarchical memory parallel computer

The directives in High Performance Fortran essentially specify data locality, so we believe that an HPF compiler can use the concepts of this section to optimize cache use on heirachical memory machines. Thus, HPF should provide good performance on all high-performance computers, not just parallel machines.

6 Problem Architectures and Parallel Software

In a series of papers [1, 19, 21], we have developed a qualitative theory of the architectures of problems, analogous to the well-known classification of parallel computer architectures into SIMD and MIMD. This is summarized in Table 1, which introduces five general problem classes. Let us return to the concept of Figure 4 — namely, computation is map between problem and computer, and software is an expression of this map. We have explored in depth this concept of problem architecture and its use for clarifying which problems run well on SIMD machines and which on MIMD. One can also understand which problem classes parallelize naturally on massively parallel machines. Here, we just describe the consequences for software, which are summarized in Table 1.

We believe that successful software models will be built around problem and not machine architecture. We see that some of the current languages — both old and new — are flawed because they do not use this principle in their design. The language often reflects artifacts of a particular machine architecture and this naturally leads to nonportable codes that can only be run on the machine whose architecture is expressed by the language. On the other hand, if the language expresses properly the problem structure, then a good compiler should be able to map it into a range of computer architectures.

We can illustrate this with Fortran 77, which we can view as embodying the architecture of a sequential machine. Thus, software written in Fortran 77 maps the space-time structure of the original complex system into a purely temporal or control structure. The spatial (data) parallelism of the problem becomes purely temporal in the software, which implements this as a sequential loop over the data (a `DO` loop in Fortran 77). Somewhat perversely, a parallelizing compiler tries to convert the temporal structure of a `DO` loop back into spatial structure to allow concurrent execution on a spatial array of processors. Often parallelizing compilers produce poor results as the original map of the problem into sequential Fortran 77 has "thrown away" information necessary to reverse

Synchronous: Data Parallel

Tightly coupled. Software needs to exploit features of problem structure to get good performance. Comparatively easy, as different data elements are essentially identical.
Candidate software paradigms: High Performance Fortran, Parallel Fortran 77D, Fortran 90D, CMFortran, Crystal, APL, C++.

Loosely Synchronous: Data Parallel

As above but data elements are not identical. Still parallelizes due to macroscopic time synchronization.
Candidate software paradigms: may be extensions of the above, however C or Fortran with message passing is currently the only guaranteed method.

Asynchronous

Functional (or data) parallelism that is irregular in space and time. Often loosely coupled and so need not worry about optimal decompositions to minimize communication. Hard to parallelize, not usually scalable to large numbers of processors.
Candidate software paradigms: PCN, Linda, object-oriented approaches.

Embarrassingly Parallel

Independent execution of disconnected components.
Candidate software paradigms: Many approaches work – PVM, PCN, Linda, Network Express, ISIS, etc.

Metaproblems

Asynchronous collection of loosely synchronous components where the component program modules can be parallelized.
Candidate software paradigms: PCN, Linda, ADA, controlling modules written in synchronous or loosely synchronous fashion.

Table 1: Architectures for Five Problem Classifications

this map and recover unambiguously the spatial structure. The first (and some ongoing) efforts in parallelizing compilers tried to directly "parallelize the 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, as described in Section 2.2.4, we are working on the development of a parallelizing compiler for Fortran D and High Performance Fortran, where the user adds additional information to tell the compiler about the spatial structure [17, 19, 21, 29, 57, 4]. We are optimistic that this project will be successful for the synchronous and loosely synchronous problem classes defined in Table 1.

Most languages do not express and preserve space time structure. Array languages such as APL and Fortran 90 are examples of data parallel languages that 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 5.2 show that data locality is critical in sequential high performance (hierarchical memory) machines as well. Thus, we would expect that the use of languages that properly preserve problem structure will lead to better performance on all computers.

7 Physical Computation and Optimization

Physical computation can be loosely classified as the use of physical analogies or methods from the physical sciences in computational studies of general complex systems [22]. One example is the use of simulated annealing (an idea from physics) for optimization problems such as chip routing and placement [41]. Another is the use of neural networks (an idea from biology) in learning and pattern recognition for problems in computer vision and robotics.

Optimization is a particularly important application of physical computation. Simic has introduced the term *physical optimization* to describe the many different optimization methods of this kind [53, 18]. It is not surprising that techniques based in the physical sciences are useful for solving optimization problems, since most laws of physics can be formulated variationally as optimization problems, many physical systems act so as to minimize energy or free energy, and evolution in nature is also involved in optimization.

As mentioned in Section 4.4, physical optimization techniques such as simulated annealing, neural networks and genetic algorithms can be usefully applied to domain decomposition and load balancing, an important optimization problem in parallel computation. However these methods can be used to tackle general optimization problems, and indeed have successfully been applied to a wide variety of problems. Physical optimization methods can be contrasted with other methods for optimization: heuristics can be considered as an approach motivated by the problem; maximum entropy or information theory as approaches from electrical engineering; combinatorial optimization methods from mathematics; and linear programming and rule-based expert systems from computer science.

There is no universally good approach to optimization. Each method has different tradeoffs in robustness, accuracy, speed, suitability for parallelization, and problem size dependence. For instance, neural networks do simple things on large data sets and parallelize easily, whereas expert systems do complex things on smaller data sets and are difficult to parallelize. Parallel algorithms for physical optimization methods are not usually trivial, and are not always similar to the sequential algorithms.

The nature of the problem is very important in terms of which method is most suitable. For instance, what is the shape of the cost function? Are the local minima deep or shallow, wide or narrow, relatively few or numerous? Are the minima correlated or uncorrelated? Does the problem require the exact global minimum, or is a good approximate minimum sufficient? Physical optimization methods try to find good approximate solutions, not necessarily the best solution.

They work well for many complex real-world problems, for which approximate solutions are all that is required, and indeed all that is warranted by the usually imprecise data or models. Also, many of these problems are NP-complete, so that only approximate solutions are feasible given a limited computational resource.

Here we briefly describe some physical optimization methods. More detailed reviews can be found in [18, 32].

7.1 Simulated Annealing

Simulated annealing is a very general optimization method that stochastically simulates the slow cooling of a physical system to its ground state [41, 50, 54]. The cost function for the problem is viewed as an energy function, and a parameter T analogous to temperature is introduced. The algorithm works by using an iterative Monte Carlo technique, that is, by proposing changes to the state of the system, and either accepting or rejecting a change using the Metropolis criterion — if the cost (energy) is decreased, the change is accepted; if the energy is increased by δE, it is accepted with a probability $\exp(-\delta E/T)$. The process is started at a high temperature where almost all proposed changes are accepted, and the temperature is gradually reduce to zero, where changes are only accepted if they decrease the energy. The zero temperature algorithm is just the *greedy* or *hill-climbing* algorithm, which works poorly in most cases since it can get trapped in local minima. Simulated annealing at non-zero T allows the system to probabilistically increase the energy and thus escape from local minima.

The rate of cooling is crucial to the performance of the algorithm. It can be shown that if the temperature is decreased slowly enough (logarithmically), then the global optimum will be found (with enough trials, since this is a probabilistic method). The basic idea is the same as real annealing (for example, of steel) — if the temperature changes are small enough, the system can maintain thermal equilibrium throughout the procedure, so it will finish up in the zero temperature equilibrium state, i.e. the ground state or global optimum. However a logarithmic cooling schedule is much too slow for most problems, so usually a faster (exponential) cooling schedule is used. For many problems this will still keep the system close enough to equilibrium that it will be very near the ground state energy at zero temperature, and thus find a near-optimal solution. One of the main problems with simulated annealing is that finding a good cooling schedule is generally a trial-and-error procedure. Some advances have been made in finding *adaptive* annealing schedules, where the temperature is reduced depending on the measured values of the energy for the particular problem. Theoretically this promises a great improvement in performance, however in practice it is often difficult to find the optimal cooling schedule given the limited number of measurements available.

Another critical part of the algorithm is the choice of the method for updating the system state. If large changes are made to the system variables, then the energy change δE will generally be large, which will result in most of the proposed changes being rejected. If the changes at each iteration are chosen to be small, so that most of them are accepted, it will take many iterations to reach a very different, uncorrelated state of the system. A tradeoff is required, since in both these cases, moving through the search space will be very slow. The method can be greatly improved if a way can be found to make substantial changes to the system without changing the energy too much.

Simulated annealing is popular because it is simple and it can be easily applied to any optimization problem. However it may not be effective unless a good update method and cooling schedule are used.

7.2 Deterministic Annealing

This approach is similar to simulated annealing, however instead of using a stochastic (Monte Carlo) approach, a simple heuristic is used to minimize the free energy $F = E - TS$ at each temperature T, where E is the cost function (or energy) and S is analogous to the entropy of a physical system.

Notice that at zero temperature the energy and free energy are equivalent. The free energy is formally defined as $F = -T \log Z$, where

$$Z = \sum_{states\, C} \exp\left[-E(C)/T\right] \tag{20}$$

is known as the partition function in statistical mechanics. This approach is similar to methods used in quantum chemistry to find the ground state of a complex molecule.

Deterministic annealing has been used very effectively in data clustering problems [52]. For the simple case of grouping points in space into clusters based on distance, it is possible to construct an energy function for which the minimum of the free energy can be computed deterministically by iteratively solving an implicit equation. This particular example has an interesting temperature dependence. At high temperature the points will all be in a single cluster. As the temperature is decreased, the points will split into 2 clusters, then 3, and so on, at various critical temperatures. The temperature is related to the size of the clusters, or the distance scale at which the system is observed. For a given problem we will need to specify a particular minimum distance scale or temperature.

7.3 Neural Networks

The use of neural networks for optimization was introduced for the traveling salesman problem in [40], and although the method is not very effective for this application, the method and basic ideas are important for a range of problems. The traveling salesman problem (TSP) is the classic NP-complete discrete optimization problem, for which the salesman has to find a tour that minimizes the distance traveled in visiting a given set of cities. We introduce neural variables η_p^i that are 1 if the i^{th} step of the tour passes through city p, and zero otherwise. The cost function can be written very simply in terms of these neural variables, however an extra penalty term needs to be added to implement the constraint that only one of the η_p^i can be non-zero for a given i or p. The cost function then looks like a statistical physics problem, with "spins" η_p^i governed by an "energy function". This formulation of the TSP can in fact be solved using simulated annealing, however the neural network approach uses a faster approximate method, similar to a mean-field approximation in statistical physics. Unfortunately this approximate method does not work well for even a modest number of cities [56].

This general approach has however been very effective for a number of other problems, including load balancing a parallel computer [25]. In this case, we can introduce neural variables just as for the TSP, except that η_p^i is now 1 if data element i is assigned to processor p. However this will give the same problems as found for the TSP. Instead, we define the neural variables by the binary decomposition of the processor number

$$p = \sum_{k=0}^{d-1} 2^k \eta_k^i \tag{21}$$

where there are M data elements and $N = 2^d$ processors. The $Md = M \log_2 N$ neural variables provide a non-redundant specification of the data decomposition, compared to the MN redundant variables in the TSP-like formulation. This approach was obviously motivated by parallel computers with a hypercube topology, however it can be used for an arbitrary topology.

Using non-redundant variables allows us to construct an energy (cost) function for data decomposition along the lines of Equation 16

$$\begin{aligned} E &= E_{\text{calc}} + E_{\text{comm}} \qquad \text{where} \\ E_{\text{calc}} &= \frac{1}{N} \sum_{\text{m, m}'} Calc(m)\; Calc(m') \prod_{k=0}^{d-1} \left[1 + s_k(m)\, s_k(m')\right] \end{aligned}$$

$$E_{\mathrm{comm}} = \frac{1}{4}\sum_{\mathrm{m,\,m'}} Comm\,(m,\,m') \sum_{k=0}^{d-1} [1 - s_k(m)\, s_k(m')] \tag{22}$$

with "spins" $s_k(m) = 2\eta_k^m - 1$ taking the values ± 1. In this case, the energy function has no constraint (penalty) terms, and the Hopfield-Tank method works extremely well, being comparable in quality to simulated annealing results but much faster, since a deterministic mean-field approximation is used [55]. This indicates that the problems found in using neural networks for optimization lie not with the method, but with the choice of variables and the necessity of introducing penalty terms.

We have also used neural network optimization successfully for optimizing compilers [31, 28], and robotic vehicle navigation [27, 16].

7.4 Elastic Net

The elastic net was introduced as a physically based approach that outperformed the neural network method for TSP [10]. The basic idea behind the elastic net is to "invent" a physical system whose equilibrium state is the desired optimum. In the case of the TSP, we consider an elastic string with beads for each city. The beads are attracted to each other with a simple elastic force that will try to shrink the length of the path to zero, and thus drive the system towards the minimum path. There is also an attractive force between the beads and the cities that drives the system towards enforcing the constraint that the tour must pass through each city. The comparative strength of these two competing forces is a parameter similar to temperature in annealing. We start with the elastic force being dominant, and then slowly change the forces until finally the bead-city force is dominant so that we end up with a valid (hopefully near-optimal) tour.

Simic has shown an interesting relation between neural networks and elastic nets for the TSP [53]. Both correspond to deterministic annealing using similar mean field approximations, but with different choices of degrees of freedom and thus different constraints.

7.5 Genetic Algorithms

Genetic algorithms are based on evolutionary processes in biology. The basic idea is to encode the system parameters as "genes" which make up a set of "chromosomes", each describing a different state of the system. For example, in load balancing a multicomputer, each gene would specify the processor number for a specific node in the computational graph. We start with a base population of chromosomes that undergo changes due to the application of genetic operators such as crossover and mutation. Crossover is like mating, in that the genes of two individual chromosomes are randomly combined to form a new individual. Mutation occurs by randomly changing a gene in the chromosome. Once a new population is formed by these operations, it is compared with the old population, with each chromosome being assigned a "fitness" (the cost for the optimization problem). Only the fittest individuals are retained for the next generation ("survival of the fittest").

Although genetic algorithms provide an interesting and often very effective optimization method, obtaining good performance usually requires very careful mapping of the problem variables onto the genes, a good choice of genetic operators, and a lot of tuning of parameters in the algorithm. This is analogous to the necessity for choosing good problem-specific update moves and cooling schedules for simulated annealing. Another problem with genetic algorithms is that there is no natural way to decide when a good solution has been reached and the process should be stopped, so it is usually followed by some postprocessing using a hill-climbing technique or other fast heuristic.

7.6 Simulated Tempering

A variation of simulated annealing, known as *simulated tempering*, has recently been introduced and used to study certain types of lattice spin models [46]. Tempering differs from annealing in two ways:

it allows both heating and cooling of the system; and it keeps the system in equilibrium when the temperature is changed. Both of these changes are beneficial for optimization. The first allows for "reheating" when the annealing gets stuck in local minima at low temperatures. This is currently often used as an ad-hoc addition to standard simulated annealing. The second takes care of one of the main difficulties in using simulated annealing, which is coming up with a cooling schedule that is not too slow but keeps the system close to equilibrium as the temperature goes to zero. We are currently working on applying this new method to general optimization problems.

8 Conclusion

We have found that using ideas and techniques from the realms of complex systems and the physical sciences can provide useful and powerful insights into parallel computing and computer science, and in particular the efficient use of parallel computers for problems in computational science. However many of the ideas presented here are well out of the mainstream of research in computer science, and have attracted little attention. This is perhaps due to the general unfamiliarity of the computer science community with many of these methods and concepts from the physical sciences.

There are interesting synergies between computer science, computational science, the physical sciences, and the theory of complex systems. The development of models of computation and general optimization techniques seem particularly suited to an interdisciplinary approach drawing from all these areas.

Acknowledgments

This work has been collaborative with many people, especially W. Furmanski and S. Otto. We would like to thank them and our colleagues in the Caltech Concurrent Computation Program (C3P) and the Northeast Parallel Architectures Center (NPAC) for their insight and help.

This work was supported by the Center for Research on Parallel Computation (CRPC) with National Science Foundation Cooperation Agreement No. CCR-9120008. The US Government has certain rights in this material.

References

[1] Angus, I. G., Fox, G. C., Kim, J. S. & Walker, D. W. (1990). *Solving Problems on Concurrent Processors, Volume 2: Software for Concurrent Processors.* Prentice Hall, Englewood Cliffs, NJ.

[2] Bae, S., Coddington, P. D. & Ko, S.-H. (1994). Parallel Wolff Cluster Algorithms. NPAC Technical Report SCCS-619.

[3] Bordewaker, R. & Choudhary, A. (1994). HPFio: HPF with Parallel I/O Extensions. NPAC Technical Report SCCS-613.

[4] Bozkus, Z., Choudhary, A., Fox, G., Haupt, T. & Ranka, S. (1993). Fortran 90D/HPF Compiler for Distributed Memory MIMD Computers: Design, Implementation, and Performance Results. NPAC Technical Report SCCS-498.

[5] Choudhary, A., Fox, G., Ranka, S., Hiranandani, S., Kennedy, K., Koelbel, C. & Saltz, J. (1992). Software Support for Irregular and Loosely Synchronous Problems. In *Computing Systems in Engineering* Vol. 3, pp. 43–52. Pergamon Press, Oxford.

[6] Choudhary, A., Fox, G., Ranka, S. & Haupt, T. (1992). Which Applications Can Use High Performance Fortran and FortranD — Industry Standard Parallel Languages? In *Proc. Fifth Australian Supercomputer Conference.*

[7] Demmel, J. (1991). LAPACK: A Portable Linear Algebra Library for High-Performance Computers. *Concurrency: Practice and Experience* **3**, 655-666.

[8] Denning, P.J. & Tichy, W.F. (1990). Highly Parallel Computation. *Science* **250**, 1217.

[9] Donath, W. E. (1979). Placement and Average Interconnection Lengths of Computer Logic. *IEEE Trans. Circuits and Systems* **16**, 272.

[10] Durbin, R. & Willshaw, D. (1987). An analogue approach to the traveling salesman problem using an elastic net method. *Nature* **326**, 689–691.

[11] Flower, J., Otto, S. & Salama, M. (1987). Optimal mapping of irregular finite element domains to parallel processors. In *Proc. Symposium on Parallel Computations and Their Impact on Mechanics*, ASME Winter Meeting, Boston, Mass.

[12] Fox, G. C. (1987). Questions and unexpected answers in concurrent computation. In *Experimental Parallel Computing Architectures*, J. J. Dongarra, editor, pp. 97–121. Elsevier Science, North-Holland, Amsterdam.

[13] Fox, G. C. (1988a). A review of automatic load balancing and decomposition methods for the hypercube. In *Numerical Algorithms for Modern Parallel Computer Architectures*, M. Schultz, ed., pp. 63–76. Springer-Verlag, Berlin.

[14] Fox, G. C. (1988b). The hypercube and the Caltech Concurrent Computation Program: A microcosm of parallel computing. In *Special Purpose Computers*, B. J. Alder, ed., pp. 1–40. Academic Press, New York.

[15] Fox, G. C. (1989). Parallel Computing. In *The Encyclopedia of Physical Science and Technology 1991 Yearbook*. Academic Press, New York.

[16] Fox, G. C. (1990a). Applications of the generalized elastic net to navigation. Caltech Technical Report C3P-930. Unpublished.

[17] Fox, G. C. (1990b). Hardware and software architectures for irregular problem architectures. In *Unstructured Scientific Computation on Scalable Microprocessors*, P. Mehrotra, J. Saltz & R. Voigt, editors, pp. 125–160. Scientific and Engineering Computation Series. MIT Press, Cambridge, Mass.

[18] Fox, G. C. (1991a). Approaches to physical optimization. In *Proc. 5th SIAM Conference on Parallel Proccesses for Scientific Computation*, J. Dongarra *et al.* editors, pp. 153–162. SIAM, Philadelphia.

[19] Fox, G. C. (1991b). FortranD as a portable software system for parallel computers. In *Proc. Supercomputing USA/Pacific 91.*

[20] Fox, G. C. (1991c). Achievements and Prospects for Parallel Computing. *Concurrency: Practice and Experience* **3**, 725–739.

[21] Fox, G. C. (1991d). The architecture of problems and portable parallel software systems. NPAC Technical Report SCCS-134. Unpublished.

[22] Fox, G. C. (1991e). Physical computation. *Concurrency: Practice and Experience* **3**, 627–653.

[23] Fox, G. C. (1992a). Lessons from Massively Parallel Applications on Message Passing Computers. In *Proc. 37th IEEE International Computer Conference.* NPAC Technical Report SCCS-214.

[24] Fox, G. C. (1992b). The use of physics concepts in computation. In *Computation: The Micro and the Macro View*, B. A. Huberman, ed., chapter 3. World Scientific Publishing Co., River Edge, NJ.

[25] Fox, G. C. & Furmanski, W. (1988a). Load balancing loosely synchronous problems with a neural network. In *Proc. Third Conference on Hypercube Concurrent Computers and Applications, Volume 1*, G. C. Fox, ed., pp. 241–278. ACM Press, New York.

[26] Fox, G. C. & Furmanski, W. (1988b). The physical structure of concurrent problems and concurrent computers. In *Scientific Applications of Multiprocessors*, R. J. Elliott & C. A. R. Hoare, editors, pp. 55–88. Prentice Hall, Englewood Cliffs, NJ.

[27] Fox, G. C., Furmanski, W., Ho, A., Koller, J., Simic, P. & Wong, Y. F. (1988). Neural networks and dynamic complex systems. In *Proc. 1989 SCS Eastern Conference.*

[28] Fox, G. C., Furmanski, W. & Koller, J. (1989). The Use of Neural Networks in Parallel Software Systems. *Mathematics and Computers in Simulation* **31**, 485–495.

[29] Fox, G. C., Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C.-W. & Wu, M.-Y. (1991). Fortran D language specification. NPAC Technical Report SCCS-42c, CRPC-TR90079. Unpublished.

[30] Fox, G. C., Johnson, M. A., Lyzenga, G. A., Otto, S. W., Salmon, J. K. & Walker, D. W. (1988). *Solving Problems on Concurrent Processors, Volume 1.* Prentice Hall, Englewood Cliffs, NJ.

[31] Fox, G. C. & Koller, J. G. (1989). Code generation by a generalized neural network: General principles and elementary examples. *J. Parallel and Distributed Computing* **6(2)**, 388–410.

[32] Fox, G. C., Messina, P. C. & Williams, R. D., editors (1994). *Parallel Computing Works.* Morgan Kaufmann Publishers, San Francisco (in press).

[33] Fox, G. & Otto, S. (1986). Concurrent computation and the theory of complex systems. In M. T. Heath, editor, *Hypercube Multiprocessors*, pp. 244–268. SIAM, Philadelphia.

[34] Fox, G., Otto, S. W. & Umland, E. A. (1985). Monte Carlo physics on a concurrent processor. *J. Stat. Phys.* **43**, 1209.

[35] Furmanski, W. (1992). MOVIE - Multitasking Object-oriented Visual Interactive Environment. NPAC Technical Report SCCS-353.

[36] Furmanski, W., Faigle, C., Haupt, T., Niemiec, J., Podgorny, M., & Simoni, D.A. (1993). MOVIE model for Open Systems based High Performance Distributed Computing. *Concurrency: Practice and Experience* **5**, 287-308.

[37] High Performance Fortran Forum (1993). High Performance Fortran Language Specification. Center for Research in Parallel Computing Technical Report CRPC-TR92225. Available via anonymous ftp from titan.cs.rice.edu in the directory public/HPFF/draft.

[38] Hillis, W.D. (1985). *The Connection Machine.* MIT Press, Cambridge, Mass.

[39] Hillis, W.D. & Steele, G. (1986). Data Parallel Algorithms. *Comm. ACM* **29**, 1170.

[40] Hopfield, J. J. & Tank, D. W. (1986). Computing with neural circuits: a model. *Science* **233**, 625.

[41] Kirkpatrick, S., Gelatt, C. D. & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science* **220**, 671–680.

[42] Koelbel, C. H., Loveman, D. B., Schreiber, R. S., Steele, G. L. Jr. & Zosel, M. E. (1994). *The High Performance Fortran Handbook.* MIT Press, Cambridge, Mass.

[43] Landman, B. S. & Russo, R. L. (1971). On a Pin versus Block Relationship for Partitions of Logic Graphs. *IEEE Trans. Comp.* **C20**, 1469.

[44] Mandelbrot, B. (1979). *Fractals: Form, Chance, and Dimension.* Freeman, San Francisco.

[45] Mansour, N. & Fox, G. C. (1993). Allocating Data to Multicomputer Nodes by Physical Optimization Algorithms for Loosely Synchronous Computations. *Concurrency: Practice and Experience*, (in press). NPAC Technical Report SCCS-350.

[46] Marinari, E. & Parisi, G. (1992). Simulated Tempering: A New Monte Carlo Scheme. *Europhys. Lett.* **19**, 451.

[47] Messina, P. (1991). Parallel Computing in the 1980s — One Person's View. *Concurrency: Practice and Experience* **3**, 501–524.

[48] Message Passing Interface Forum (1994). MPI: A Message Passing Interface Standard. Available via anonymous ftp from ftp.mcs.anl.gov in the directory pub/mpi.

[49] Morrison, R. & Otto, S. (1986). The Scattered Decomposition for Finite Element Problems. *Journal of Scientific Computing* **2**, 59–76.

[50] Otten, R. H. J. M. & van Ginneken, L. P. P. P. (1989). *The Annealing Algorithm.* Kluwer Academic Publishers, Dordrecht, The Netherlands.

[51] Ponnusamy, R., Saltz, J. & Choudhary, A. (1993). Runtime Compilation Techniques for Data Partitioning and Communication Schedule Reuse. University of Maryland Technical Report UMIACS-TR-93-32.

[52] Rose, K., Gurewitz, E. & Fox. G. C. (1990). A deterministic annealing approach to clustering. *Pattern Recognition Letters* **11**, 589–594.

[53] Simic, P. (1990). Statistical Mechanics as the Underlying Theory of 'Elastic' and 'Neural' Optimizations. *Network* **1**, 89–103.

[54] van Laarhoven, P. J. M. & Aarts, E. H. L. (1987). *Simulated Annealing: Theory and Applications.* Kluwer Academic Publishers, Dordrecht, The Netherlands.

[55] Williams, R. D. (1991). Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency: Practice and Experience* **3**, 457–481.

[56] Wilson, G. V. & Pawley, G. C. (1988). On the Stability of the Travelling Salesman Problem Algorithm of Hopfield and Tank. *Biol. Cybern.* **58**, 63–70.

[57] Wu, M. & Fox, G. C. (1991). Fortran 90D compiler for distributed memory MIMD parallel computers. NPAC Technical Report SCCS-88b, CRPC-TR91126. Unpublished.