Fortran D as a Portable Software System for Parallel Computers

Geoffrey Fox

CRPC-TR91128
June, 1991

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

SCCS - 91
CRPC - TR91128

FortranD as a Portable Software System for Parallel Computers

Geoffrey C. Fox
Northeast Parallel Architectures Center
111 College Place
Syracuse, NY 13244-4100
315-443-1723
gcf@nova.npac.syr.edu

Presentation at Supercomputing USA/Pacific 91 Conference, Santa Clara, CA June 1991.

Abstract

We discuss how extensions of Fortran in a distributed computing environment may be the basis of a portable software system for a heterogeneous computing network consisting of SIMD and MIMD parallel machines connected with conventional (super) computers.

*This work was partially supported by ASAS Program Office Techbase Program and the National Science Foundation under Cooperative Agreement No. CCR-8809165--the Government has certain rights in this material.

I: Introduction

Where do we stand in parallel computing? Current commercial parallel computers are immature, but they offer better peak performance and better cost performance than conventional machines for many problems. A large number of parallel applications have been developed. There are only a tiny fraction of the computations running on sequential machines, but the initial parallel experimental applications cover a wide range of algorithms and perhaps the "essence" (computational kernels) of the majority of large scale scientific and engineering calculations. Clearly, the main limitation to the rapid spread of parallel computing is not hardware or algorithms but rather software.

Here we discuss Fortran, with especially the array extensions of Fortran90, as a portable software environment. We use a classification of problem modules into three main classes with general applications built up as a mixture of these classes. We show how a combination of Fortran and a distributed object-oriented environment may be used for problems of this structure. The testing of these ideas is certainly incomplete; we only have extensive experience with small academic codes - typically up to 10,000 lines in length. The extrapolation to the much larger and complex industrial and government applications is uncertain. For instance, we have little understanding of the parallelization of problems requiring real-time and extensive input/output support.

In Figure 1, we follow Kennedy and divide software into five layers, where we are most interested in the upper two layers which should be portable to a variety of hardware architectures, including both SIMD and MIMD machines. As shown in Figure 2, we view software as mapping problems onto machines. Both problems and machines have an architecture and the software system is dependent on both. However, we expect that the high level software systems discussed here will be built around the problem and not the machine architecture. The latter is reflected in low level machine dependent support software which should be hidden from most users. We can probably persuade many users to produce parallel versions of their code; however, they will be loath to do this more than once. It is not reasonable that each new parallel architecture require a significantly different software implementation.

In Section II we review an architectural classification for problems, and in Section III we analyze current parallel software experience from this point of view. In Section IV, we discuss Fortran for synchronous problems and in Section V, the harder and still uncertain irregular loosely synchronous case is treated briefly. In Section VI, we combine these ideas for program modules to suggest a general portable parallel software strategy.

II: Problem Architectures

We have introduced three broad classes of problem [Fox:88b, Denning:90, Angus:90a, Fox:91d]. These were deduced from our experience at Caltech combined with a literature survey which was reasonably complete up to the middle of 1989. At Caltech, we developed some fifty applications on parallel machines of which twenty-five led to publications in the scientific literature describing the results of simulations performed on our parallel computers [Fox:88a, Fox:89n, Fox:87d, Fox:88oo]. This analysis led us to introduce three broad classes of problem architectures which technically describe the temporal (time or synchronization) structure of the problem [Fox:88b]. Further detail is contained in the spatial structure or computational graph describing the problem at a given instant of simulation time [Fox:88tt]. Here we only need to consider "embarrassingly parallel" problems, where there is little or no spatial connection between the individual parallel program components. For embarrassingly parallel problems, the synchronization (both software and hardware) issues are greatly simplified.

The three general temporal structures are called synchronous, loosely synchronous, and asynchronous, which we sometimes shorten these to Classes I, II, and III, respectively. The temporal structure of a problem is analogous to the hardware classification into SIMD and MIMD. The spatial structure of a problem is analogous to the interconnect or topology of the hardware. The detailed spatial structure is important in determining the performance of an implementation [Fox:88a] but it does not affect the broad programming issues discussed here.

Synchronous problems are data parallel in the language of Hillis [Hillis:87a] with the restriction that each data point is evolved in time with the same procedure. The problem is synchronized microscopically at each computer clock cycle. Such problems are particularly common in academia as they naturally arise in any description of some world in terms of identical fundamental units. This is illustrated by quantum chromodynamics (QCD) simulations of the fundamental elementary particles which involve a set of gluon and quark fields on a regular four dimensional lattice. These computations form the largest use of supercomputer time in academia.

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

Loosely synchronous problems are spatially irregular but temporally regular. The final *asynchronous* class is irregular in space and time. A good example is an event driven simulation which can be used to describe the irregular circuits we discussed above, but now the event paradigm replaces the regular time stepped simulation. Other examples include computer chess [Felten:88i] and transaction analysis. Asynchronous problems are hard to parallelize unless they are "embarrassingly parallel" (termed Class III-EP). More general asynchronous applications require sophisticated software and hardware support to properly synchronize the nodes of the parallel machine as is illustrated by time warp mechanism for event driven simulations [Wieland:89a].

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

The above classification is really only applicable to individual program modules, and many important problems consist of several modules with different classifications. An interesting illustration is the battle management simulation implemented by my collaborators at JPL [Meier:89a]. This is formally asynchronous with temporally and spatially irregular interconnections between various modules, such as sensors for control platforms and input/output tasks. However, each module uses a loosely synchronous algorithm such as the multi-target Kalman filter [Gottschalk:90b] or the target-weapon pairing system. Thus, we had a few (~ 10-50) large grain asynchronous (Class III) objects, each of which was a data parallel Class I or II algorithm. This type of asynchronous problem can be implemented in a scaling fashion on massively parallel machines. We will denote this IIICG-IIFG to indicate the <u>C</u>oarse <u>G</u>rain asynchronous control of <u>F</u>ine <u>G</u>rain loosely synchronous subproblems. A similar example of this mixed problem class is machine vision and signal processing, where one finds an asynchronous collection of data parallel modules to perform various image processing tasks, such as stereo matching and edge detection. A somewhat different example is a project of Dennis from Rice in the NSF center CRPC to study optional well placement in an oil reservoir. Here, the reservoir simulation for a given placement is loosely synchronous, whereas the overall optimization is a naturally asynchronous Class III algorithm. In the above cases, the asynchronous components of the problems were large grain modules with modest parallelism. This can be contrasted with Otto and Felten's MIMD computer chess algorithm, where the asynchronous evaluation of the pruned tree is "massively parallel" [Felten:88i]. Here, one can break the problem up into many loosely coupled but asynchronous parallel components which give excellent and scalable parallel performance. Each asynchronous task is now a Class I or II modestly parallel evaluation of a given chess position.

III: Current Software Scenario and Lessons

The dominant software environment on SIMD machines has been Fortran 90, which on the CM-2 has replaced *LISP and C* as the primary language for scientific codes because of the quality of the compiler and the familiarity of scientific users with Fortran [TMC:89a].

On MIMD machines, the major environment has been Fortran or C plus explicit message passing. This has been adequate for synchronous and loosely synchronous problems, which cover about 90% of scientific and engineering computations. OCCAM has been used extensively on transputer systems but this has not gained general acceptance, and it is also a system with explicit message passing. The success of the early applications on parallel machines is exciting -- it certainly shows that "parallel computing works." But what do we understand by this? It means that nearly all large scale problems parallelize, but not that we have the best software methodology. Further, most of the current implementations are small academic or research codes; for instance, the fifty Caltech codes were nearly all between 1000 and 10,000 lines long. Longer industrial codes will require better software approaches. These need to address several issues. Explicit message passing, which we have used up to now, is formally portable among MIMD machines, as one can parameterize the number of nodes so that a given program will run on any size machine. However, this is deceptive as performance optimization does make the message passing approach machine dependent. One must consider issues such as the overlap of communication and calculation, decomposition choices, and message location trade-offs for latency and bandwidth. These introduce machine dependence, especially for the irregular Class II problems. To be truly portable, the user must implement an arbitrary decomposition, and this is impractical. The problem is clear; Fortran (C) plus message passing or OCCAM are software models built around the machine and not the problem architecture. Thus, this approach is not truly portable.

We now discuss how for program modules of our three temporal classes, we can suggest portable software pradigms.

Class I Synchronous Problems

These problems are tightly coupled synchronous problems which are regular in space and time. Their data or geometric parallelism can be naturally expressed in Fortran 90D (appropriately extended Fortran 90) or similar languages such as CM Fortran, Crystal [Chen:88b], C* [Quinn:90a, 90b], or even APL. This allows the user to specify the problem structure in a natural high level fashion using the vector and matrix constructs of Fortran 90. The compiler can take care of mapping this onto different machines including those of SIMD and MIMD architecture [Wu:90c, Fox:91b]. As described in Section IV, we have defined extensions to Fortran77, called Fortran77D, which will allow Fortran77, as well as Fortran90, to be used in this problem class.

Class III Asynchronous Problems

This class is irregular in space and time and often exhibits functional or process parallelism. Considering the battle management problem discussed in Section II, there is a natural class of parallel components formed by the different sensors and control platforms, and these objects communicate with messages even in the real world! Thus, in this architecture, we see a natural break-up into processes and message passing at the problem level, and software engineering approaches, such as object oriented programming, ADA, C++, Strand, [Foster:90] PCN, ISIS or Linda [Gelernter:89a] are possibilities. In many cases we do not need to use carefully optimized decompositions but rather, use statistical load balancing and decomposition methods. This problem class includes distributed computing and the software such as ISIS designed to support it. As well as this loosely coupled category, we also see the event driven simulations with their specialized software, which we mentioned in Section II.

Class II Loosely Synchronous Problems

These problems are irregular in space but regular in time. Often their spatial structure changes dynamically, and adaptive algorithms are needed. This class is hard because the tightly coupled spatial structure demands the same kind of detailed optimizations provided by the Fortran90 compiler for Class I. However, the irregularities make this hard to implement.

We know that Fortran plus message passing works for this problem class, but we need a more portable user friendly approach. This can involve new data structures to extend languages like Fortran 90. It needs sophisticated run time support, such as that provided by the PARTI system from ICASE. In particular, we need dynamic load balancing modules for which the basic research has been done, but no general implementations are yet available [Fox:88mm]. We will expand this brief discussion in Section V.

IV: Fortran D as a Portable Parallel Software Environment for Synchronous Problems

Although direct parallelization of Fortran77 has proven to be very difficult, we believe that one can build an excellent portable Fortran environment for synchronous (Class I) problems. This is the goal of a collaboration with Kennedy's group at Rice and the Parasoft corporation and Figure 3 illustrates our strategy. One needs to "help" the compiler disentangle the problem architecture by, for instance, specifying how the Fortran arrays are distributed over the distributed memory parallel machines. These extensions to Fortran 77 or Fortran90 are called FortranD [Fox:91c].

The success of CM Fortran as the programming environment for the CM-2 suggests that it is a good approach for our synchronous Class I applications. As discussed earlier, we view Fortran 90D (CM Fortran, C*) as programming systems for "SIMD" (synchronous) problems and not as languages for SIMD machines. Compilers can map Fortran 90D effectively into all parallel architectures suitable for this problem class including MIMD, SIMD parallel machines, systolic arrays and heterogeneous networks. Fortran 90 was not originally designed as a massively parallel programming system but it has one key attribute that makes it effective. It uses high level data structures explicitly (as vectors and matrices) and so the problem architecture is clear and not hidden in values of pointers and DO loop indices. It is portable, as high level constructs such as A=B*C with A, B, and C matrices, can be optimized by the compiler for each new machine. Our experience has been that in many cases, users prefer Fortran 90 to Fortran 77, even for sequential applications, as it expresses applications naturally with much shorter code. Often one finds a factor of 2 to 3 reduction for Fortran 90 compared to Fortran 77.

Table 1 displays one example that helped us evaluate Fortran 90D as a portable environment [Keppenne:89a, 90a]. We isolated a 1500 line computational kernel from climate code using spectral methods. Extensive use of pointers made this code perform poorly on vector machines, such as the Cray YMP and made it essentially impossible for either a compiler or an outside person to improve or parallelize code. However, the code was rewritten by the original developer in Fortran 90, reducing the code size to 600 lines. This new code had an order of magnitude better performance on the Cray YMP while an outside "computer scientist" was able to convert it into Fortran 77 and Fortran 77 plus message passing without difficulty. We believe that this last step can be performed by a compiler using lessons from this and other manual conversions. In this sense, the new Fortran 90D code is portable and scalable to new machines.

The use of Fortran77 is, of course, critical due to the large amount of existing code and experience in this language. However, we can discuss Fortran90D more straightforwardly as its array extensions are easier for the compiler to parallelize efficiently. In Figure 3, we can view Fortran 90D as a "permanent annotation language" for user assisted parallelization of Fortran 77. It will require more experimentation with real application codes to compare the relative merits of parallelizing Fortran 77D versus Fortran 90D.

V: Loosely Synchronous Extensions of Fortran 90D

In Table 2, we illustrate how increasingly complex problem architectures require extensions to a Fortran 90D environment. We see a progression of extensions to Fortran 90 including:

a) Decomposition directives

b) *forall* commands to control aspects of problems involving asynchronous but uncoupled calculations

c) Run-time support for decomposition of irregular scientific computations such as those found in molecular dynamics and unstructured finite element calculations. This area has been pioneered by Saltz with the PARTI system [Saltz:90a, Saltz:87a].

d) The above extensions of Fortran 90 handle problems in which the data structure is an array — including arrays of pointers needed in c). However, there are important cases where more general data structures are needed to naturally capture the architecture of the problem. This area has received little attention in the computer science community. I see it as a critical motivation for new parallel computing environments and languages. Thus, Fortran 90 handles simple array data structures quite well; one may prefer comparable array extensions of C (i.e., C*), ADA, or functional languages such as Crystal [Chen:88b]. I believe our study of Fortran 90D will naturally extend to comparable parallel versions of other languages. However, the key uncertainties are in the support of the difficult Class II and III problems. One data structure of importance is that of a tree which occurs in any recursive algorithm, such as sorting [Fox:88a] or most importantly, in scientific simulations using one of the various multiscale approaches. These are of growing interest in vision, partial differential equations, and particle dynamics.

One particularly good example of a loosely synchronous problem which needs Fortran90D extensions, is the Barnes-Hut [Barnes:86a] clustering algorithm, which Salmon and Warren have implemented on the hypercube [Fox:89n, Salmon:90a]. Consider the evolution of a collection of N stars where the long range force between stars gives a complexity of $O(N^2)$ for the direct calculation. As illustrated in Figure 4, this can be reduced by noting that for widely separated systems, one can approximate the effect of the M stars by their centroid, or generally, their multipole expansion [Greengard:88a]. Applied recursively, this approach reduces the complexity to $O(N)$ or $O(N \log N)$, depending on the details of the implementation. This gives the tree like data structure exemplified in Figure 5, where we form (in two dimensions) successive quad-trees until there is, at most, one star in each final "leaf" of the tree. This method parallelizes well with efficiencies of 80% on large realistic three dimensional problems on the 512 node NCUBE-1. However, this careful user decomposition and parallelism is not easy to capture in current languages. The "natural" (in C) data structure is a linked list to represent the dynamic tree. Presumably, no static compiler analysis can decode the pointer values to uncover this data structure. Subtleties used by Salmon -- including replication of the top of the tree among all nodes to avoid a hot spot there -- are hard to automate with current approaches.

We are currently investigating this and other difficult examples, such as high level image analysis and other multiscale algorithms, to see if they can be supported by additional data structures (e.g., a tree) and a new run-time library to manipulate these structures and relate them to existing Fortran (C*) constructs.

VI: A Strategy for Portable Parallel Programming

We have discussed how we can express synchronous and loosely synchronous program modules in terms of extensions of Fortran. Consider the more general circumstance illustrated in Figure 6 where we have a heterogeneous collection of program modules to be mapped onto a heterogeneous computer network. This picture illustrates that there are two distinct ways of extending existing languages, such as Fortran, for parallelism. The first extension is exemplified by the discussion in Sections IV and V of FortranD with new language extensions and run time support. The second type of extensions are illustrated by Chandy's PCN, C++ and Birman's ISIS, which essentially allow one to build and manipulate tasks written in sequential ADA, C, or Fortran 77. This is a reasonable software environment for Class III problems. We emphasize that these two types of extensions are complementary and not competing approaches. Indeed, we have shown in Section II the importance of the mixed Class IIICG-IIFG, which is naturally supported by, for example PCN, where each module could be a data parallel Fortran 90D or Fortran 77D code. Future software system development should be coordinated so that such mixed systems are possible by integrating the development of groups concentrating on these different extensions.

The combination of FortranD with an (object oriented) distributed system, such as PCN or ISIS, will give us a hybrid software system, which appears suitable for most problems. There is an overall software environment oriented towards asynchronous applications with full functionality for creating and controlling objects communicating with a sophisticated message passing environment. Often at this level, we will only see modest parallelism. Each of the "asynchronous objects" is potentially massively data parallel synchronous or loosely synchronous module supported by languages optimized for these classes. Clearly, we now have a sophisticated software environment which has to map mixed problem architectures onto heterogeneous distributed computer systems with networks of SIMD and MIMD parallel machines. I believe that we know "in principle" how to tackle these issues but we have a lot of technology development for the separate components of the system before we can hope to implement the full sophisticated mixed environment.

Acknowledgements

I would like to thank Ken Kennedy, Adam Kolawa, Joel Saltz and Min-You Wu for helping me understand these issues.

References

[Angus:90a] Angus, I. G. Fox, G. C., Kim, J. S., and Walker, D. W. *Solving Problems on Concurrent Processors: Software for Concurrent Processors,* volume 2. Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1989.

[Barnes:86a] Barnes, J., and Hut, P. "A hierarchical $O(N\log N)$ force calculation algorithm," *Nature,* 324:446, 1986.

[Chen:88b] Chen, M., Li, J., and Choo, Y. "Compiling parallel programs by optimizing performance," *Journal of Supercomputing,* 2:171-207, 1988.

[Denning:90] Denning P. J. and Tichy W. F. "Highly Parallel Computation," *Science* 250, 1217-1222 (1990).

[Felten:88i] Felten, E. W., and Otto, S. W. "A highly parallel chess program," in *Proceedings of International Conference on Fifth Generation Computer Systems 1988,* pages 1001-1009. ICOT, November 1988. Tokyo, Japan, November 28-December 2. Caltech Report C3P-579c.

[Foster:90] Foster, I., Taylor, S. "Strand™: New Concepts in Parallel Programming," Prentice-Hall, Englewood Cliffs, NJ 07632, 1990.

[Fox:87d] Fox, G. C. "Questions and unexpected answers in concurrent computation," in J. J. Dongarra, editor, *Experimental Parallel Computing Architectures,* pages 97-121. Elsevier Science Publishers B. V., North-Holland, 1987. Caltech Report C3P-288.

[Fox:88a] Fox, G. C., Johnson, M. A., Lyzenga, G. A., Otto, S. W., Salmon, J. K., and Walker, D. W. *Solving Problems on Concurrent Processors,* Volume 1. Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1988.

[Fox:88b] Fox, G. C. "What have we learned from using real parallel machines to solve real problems?," in G. C. Fox, editor, *The Third Conference on*

Hypercube Concurrent Computes and Applications, Volume 2, pages 897-955. ACM Press, 11 West 42nd Street, New York, NY 10036, January 1988. Caltech Report C3P-522.

[Fox:88mm] Fox, G. C. "A review of automatic load balancing and decomposition methods for the hypercube," in M. Schultz, editor, *Numerical Algorithms for Modern Parallel Computer Architectures*, pages 63-76. Springer-Verlag, 1988. Caltech Report C3P-385.

[Fox:88oo] Fox, G. C. "The hypercube and the Caltech Concurrent Computation Program: A microcosm of parallel computing," in B. J. Alder, editor, *Special Purpose Computers*, pages 1-40. Academic Press, Inc., 1988. Caltech Report C3P-422.

[Fox:88tt] Fox, G. C., and Furmanski, W. "The physical structure of concurrent problems and concurrent computers," *Phil. Trans. R. Soc. Lond. A*, 326:411-444, 1988. Caltech Report C3P-493.

[Fox:89n] Fox, G. C. "Parallel computing comes of age: Supercomputer level parallel computations at Caltech," *Concurrency: Practice and Experience*, 1(1):63-103, September 1989. Caltech Report C3P-795.

[Fox:91b] Fox, G. C. "Achievements and Prospects for Parallel Computing", Invited Talk at *International Conference on Parallel Computing: Achievements, Problems and Prospects*, Anacapri, Italy, June 3-9, 1990. C3P-927.

[Fox:91c] Fox, G. C., Hiranadani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C.-W., Wu, M.-Y. "Fortran D Language Specifications," December 1990. Rice COMP TR90-141.

[Fox:91d] Fox, Geoffrey C. "Parallel Problem Architectures and Their Implications for Parallel Software Systems," Presentation at DARPA Workshop, Providence, Rhode Island, Feb 28, 1991, White paper, April 1991.

[Gelernter:89a] Gelernter, D. "Multiple tuple spaces in Linda," in *Proceedings of Parallel Architectures and Languages Europes*, volume 2, page 366. Springer-Verlag, LNCS, June 1989.

[Gottschalk:90b] Gottschalk, T. D. "Concurrent multi-target tracking," Technical Report C3P-908, California Institute of Technology, April 1990. Published in *Proceedings of the Fifth Distributed Memory Computing Conference*, April 9-12, Charleston, South Carolina.

[Greengard:88a] Greengard, L. "The rapid evaluation of potential fields in particle systems," in *ACM Distinguished Dissertation Series, Vol. IV*. MIT Press, Cambridge, Mass., 1988. Yale research report YALEU/DCS/RR-533, April 1987.

[Hillis:87a] Hillis, W. D. "The Connection Machine," *Scientific American*, page 108, June 1987.

[Keppenne:89a] Keppenne, C. L. *Bifurcations, Strange Attractors and Low-Frequency Atomospheric Dynamics*. PhD thesis, Universite Catholique de Louvain, 1989.

[Keppenne:90a] L., K. G., Ghil, M., Fox, G. C., Flower, J. W., Kolawa, A., Papaccio, P. N., Rosati, J. J., Shepanski, J. F., Spadaro, F. G., and Dickey, J. O. "Parallel processing applied to climate modeling." Technical Report SCCS-22, Syracuse University, November 1990.

[Meier:89a] Meier, D. L., Cloud, K. C., Horvath, J. C., Allan, L. D., Hammond, W. H., and Maxfield, H. A. "A general framework for complex time-driven simulations on hypercubes," Technical Report C3P-761, California Institute of Technology, March 1989. Published in the *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers and Applications*.

[Quinn:90a] Quinn, M. J., and Hatcher, P. J. "Data-parallel programming on multicomputers," *IEEE Software*, pages 69-76, September 1990.

[Quinn:90b] Quinn, M. J. "Compiling SIMD Programs for MIMD Architectures," *Proceedings of the IEEE Computer Society 1990 International Conference on Computer Languages*, March 1990.

[Salmon:90a] Salmon, J. *Parallel Hierarchical N-Body Methods*. PhD thesis, California Institute of Technology, December 1990.

[Saltz:87a] Saltz, J., Mirchandaney, R., Smith, R., Nicol, D., and Crowley, K. "The PARTY parallel runtime system," in *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*. Society for Industrial and Applied Mathematics, 1987. held in Los Angeles, CA.

[Saltz:90a] Saltz, J., Berryman, H., and Wu, J. "Multiprocessor and runtime compilation," *Concurrency: Practice and Experience*, 1991. To be published.

[TMC:89a] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, version 5.2-0.6 edition, September 1989.

[Wieland:89a] Wieland, F., Hawley, L., Feinberg, A., Diloreto, M., Blume, L., Ruffles, J., Reiher, P., Beckman, B., Hontalas, P., Bellenot, S., and Jefferson,

D. "The performance of a distributed combat simulation with the time warp operating system," *Concurrency: Practice and Experience,* 1(1):35-50, 1989. Caltech Report C3P-798.

[Wu:90c] Wu, M.-Y., and Fox, G. C. "An outline of Fortran90 compiler for distributed memory systems.," Technical Report SCCS-41, Syracuse Center for Computational Science, 1990.

Table 1: Performance of a Climate Modelling Computational Kernel

Code	**Machine**	**Performance Mflops**	
Original C	CRAY Y-MP (1 head)	1.5	Old Code
Fortran 90 (CM Fortran)	8K CM-2	66 (problem too small)	New Portable code
Fortran 77 Generated from Fortran 90	CRAY Y-MP	20	
Fortran 77 + Message Passing Generated from Fortran 90	NCUBE-1 (16 node) hypercube	3.3	
	NCUBE-2 (16 node) hypercube	20	
	Intel i860 (16 node) hypercube	80	

In each case only minor [obviously needed] optimizations were performed.

Table 2: **Fortran 90D for Synchronous (SIMD) and Loosely Synchronous (MIMD) Data Parallel Programming (about 90% of Scientific and Engineering Computations)**

Program Class	Language and Environment Features
a) I - Regular Geometry eg., full matrix eg. ,finite difference eg., Monte Carlo	"pure" Fortran 90 with arrays of values Need decomposition directives in Fortran D
b) I - Regular + III - EP eg., chemical potential and dynamics problems: Calculate matrix elements (needs *forall*) full matrix algerbra (Class I) for energies. and cross sections	Add *forall* to Fortran 90
c) I/II - Regular Topology but irregular geometry eg., finite element	Add arrays of pointers to arrays of values. Need new run-time library as in PARTI.
d) "True" Loosely Synchronous (II) Irregular Problems eg., High level image processing eg., Multiscale simulations Problem architectures are more general than that of array.	New data structures in Fortran 90D
e) IIICG - I, IIFG Complex System Simulations (See Sec. II)	Fortran 90D modules controlled by object oriented systems.

Domain Specific	e.g. Ellpack, Lapack
Ø	
Highish level system	e.g. Parallel Fortran, PCN, Linda, C ++, C*
Ø	
Lowish level system	e.g. Fortran or C plus message passing
Ø	
Message Passing	Portable Syntax but high performance
Ø	
Virtual machine	Machine specific implementation

Figure 1: Five layers for a parallel software system

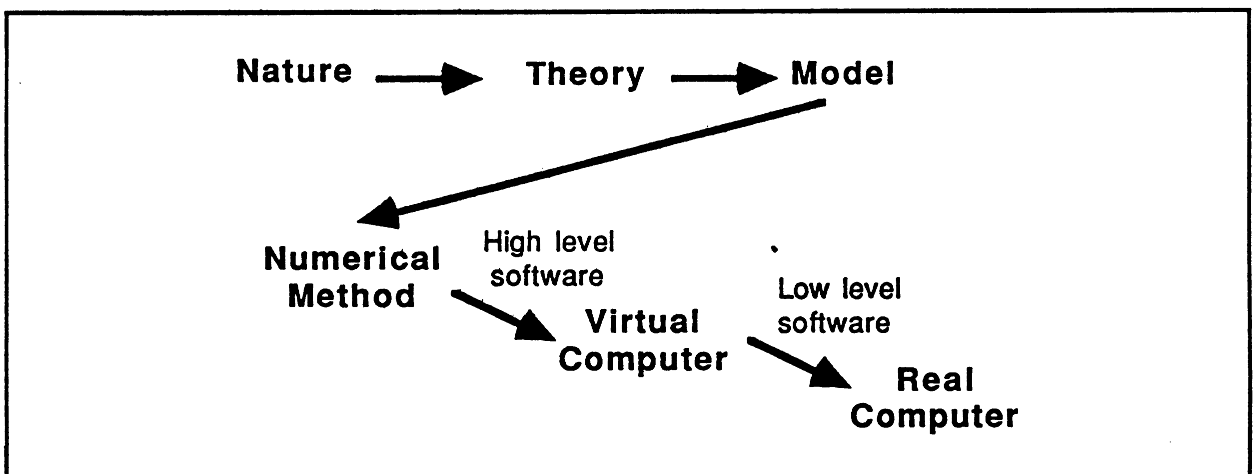

Figure 2: Theory and simulation as mappings

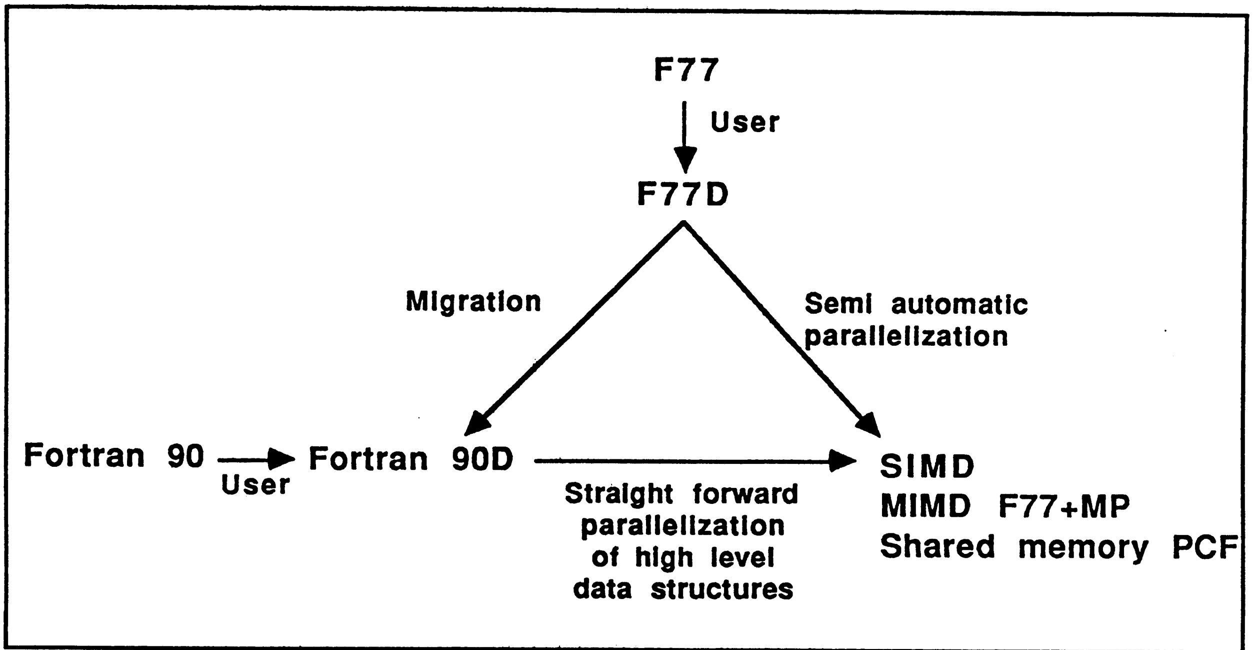

Figure 3: An integrated Fortran environment

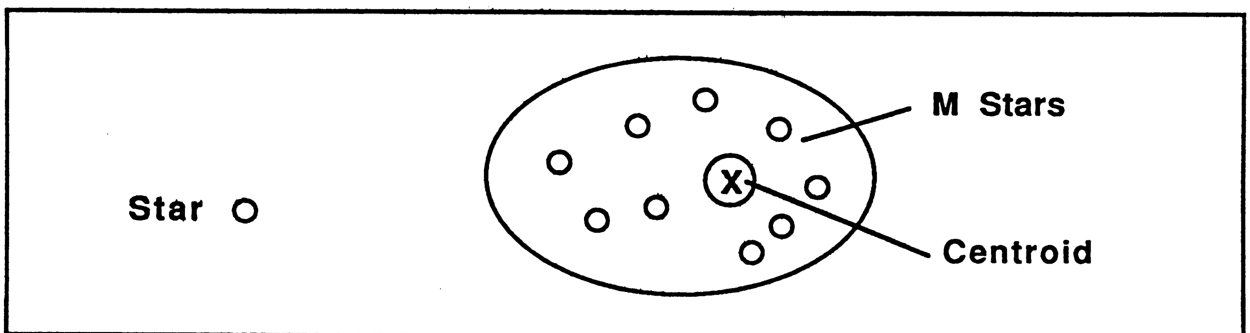

Figure 4: A cluster of stars replaced by their centroid

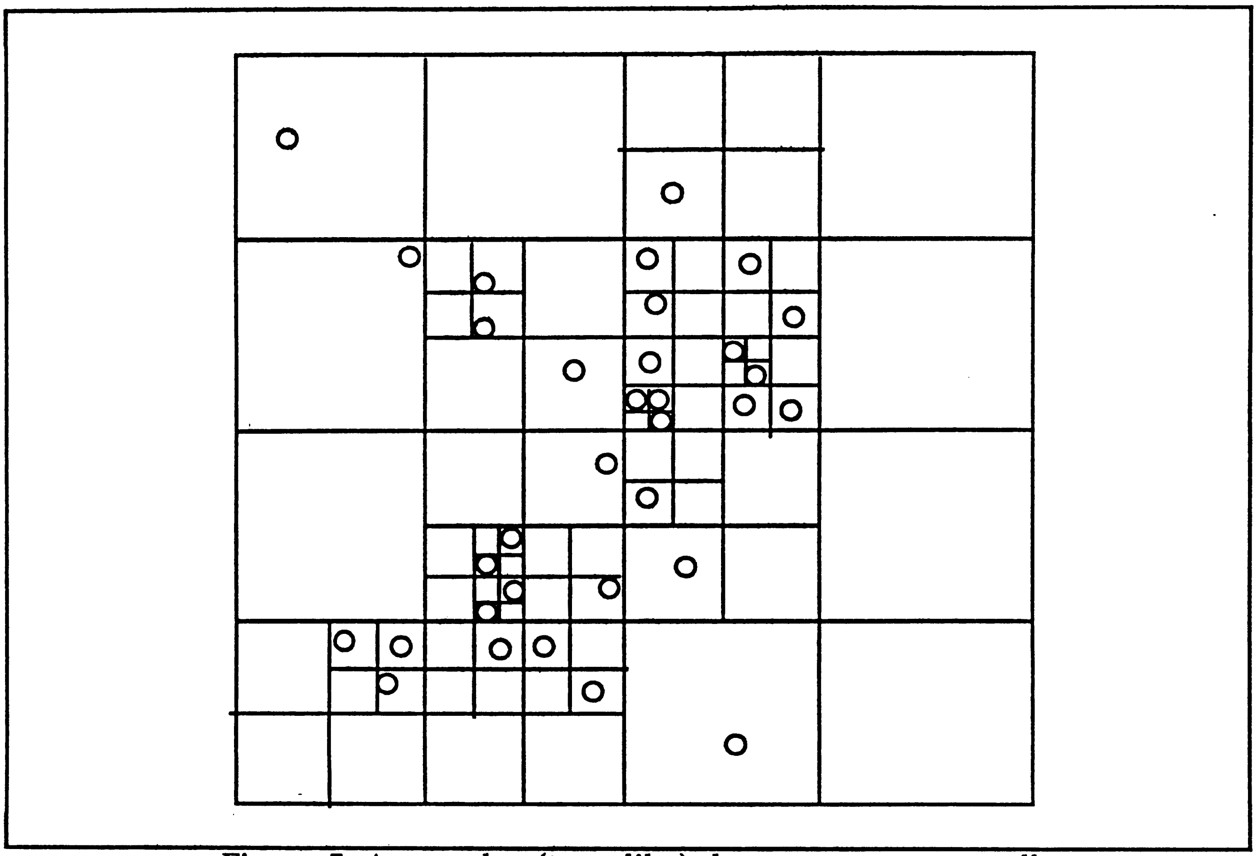

Figure 5: A complex (tree--like) data structure not well expressed in sequential or parallel Fortran. "O" represents a star.

Fortran D as a Portable Software System for Parllel Computers

Geoffrey Fox

CRPC-TR91128
June, 1991

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

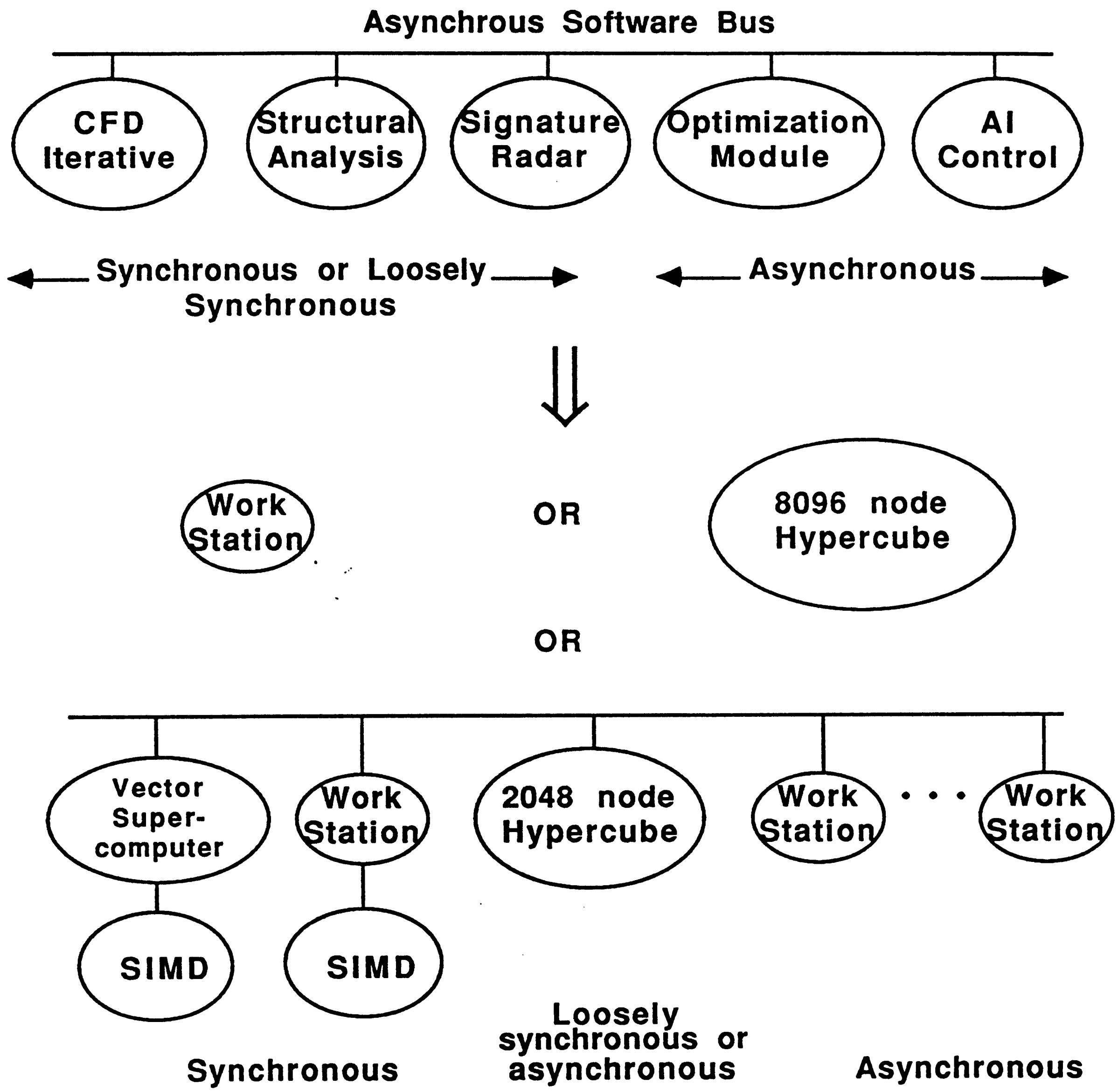

FIGURE 6: THE MAPPING OF HETEROGENEOUS PROBLEMS ONTO HETEROGENEOUS COMPUTERS.