Lessons from Massively Parallel Applications on Message Passing Computers

Geoffrey C. Fox
Syracuse University
Northeast Parallel Architectures Center
111 College Place
Syracuse, New York 13244-4100

Abstract

We review a decade's work on message passing MIMD parallel computers in the areas of hardware, software and applications. We conclude that distributed memory parallel computing works, and describe the implications of this for future portable software systems.

1 Introduction

We start with a nostalgic note. The 1984 COMPCON conference was my first opportunity to discuss our early hypercube results from Caltech [1] based on the software and science applications we built for C. Seitz's 64-node Cosmic Cube which started "production" runs on Quantum Chromodynamics (QCD) in October, 1983. That first MIMD machine was only two megaflops performance — ten times better than the VAX11/780 we were using at the time. However, the basic parallelization issues remain similar in the 1991 six gigaflop QCD implementation on the full size 64 K CM-2. What have we and others learned in the succeeding eight years while the parallel hardware has evolved impressively with in particular a factor of 3000 improvement in performance? There is certainly plenty of information! In 1989, I surveyed some four hundred papers describing parallel applications [2], [3] — now the total must be over one thousand. A new complete survey is too daunting for me. However, my personal experience, and I believe the lesson of the widespread international research on message passing parallel computers, has a clear message.

The message passing computational model is very powerful and allows one to express essentially all large scale computations and execute them efficiently on distributed memory SIMD and MIMD parallel machines.

Less formally one can say that parallel computing works, or more controversially but accurately in my opinion that "distributed memory parallel computing works". In the rest of this paper, we will dissect this assertion and suggest that it has different implications for hardware, software and applications. Formally, we relate these as shown in Figure 1 by viewing computation as a series of maps. Software is an expression of the map of the problem onto the machine. In Section 2, we review a classification of problems described in more detail in [3], [4], [5], [6], [7], [8]. In the following three sections, we draw lessons for applications, hardware, and software and quantify our assertion above about message passing parallel systems.

2 Problem Architecture

Problems like computers have architectures. Both are large complex collections of objects. A problem will perform well when mapped onto a computer if their architectures match well. This loose statement will be made more precise in the following, but not completely in this brief paper. At a coarse level, we like to introduce five broad problem classes which are briefly described in Table 1. These can and should be refined, but this is not necessary here. Thus, as described in Table 1, we do need to differentiate the application equivalent of the control structure — SIMD and MIMD — for computers. However, details such as the topology (hypercube, mesh, tree, etc.) are important for detailed performance estimates but not for the general conclusions of this paper. Note that the above implies that problems and computers both have a topology.

We will use the classification of Table 1 in the following sections which will also expand and exemplify the brief definitions of Table 1.

3 Applications

Let us give some examples of the five problem architectures.

Synchronous: These are regular computations on regular data domains and can be exemplified by full matrix algorithms such as LU decomposition; finite difference algorithms and convolutions such as the fast Fourier transform.

- *Synchronous: Data Parallel* Tightly coupled and software needs to exploit features of problem structure to get good performance. Comparatively easy as different data elements are essentially identical.

- *Loosely Synchronous:* As above but data elements are not identical. Still parallelizes due to macroscopic time synchronization.

- *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 (massively) unless ...

- *Embarrassingly Parallel:* Independent execution of disconnected components.

- *A=LS: (Loosely Synchronous Complex)* Asynchronous collection of (loosely) synchronous components where these programs themselves can be parallelized.

Table 1: Five Problem Architectures

Loosely Synchronous: These are typified by iterative calculations (or time evolutions) on geometrically irregular and perhaps heterogeneous data domains. Examples are irregular mesh finite element problems, and inhomogeneous particle dynamics.

Asynchronous: These are characterized by a temporal irregularity which makes parallelization hard. An important example is even driven simulation where events, as in a battlefield simulation, occur in spatially distributed fashion but irregularly in time. Branch and bound and other pruned tree algorithms common in artificial intelligence such as computer chess [9] fall in this category.

Synchronous and Loosely synchronous problems parallelize naturally in a fashion that scales to large computers with many nodes. One only requires that the application be "large enough" which can be quantified by a detailed performance analysis [10] which was discussed quite accurately in my original COMPCON paper [1]. The speed up

$$S = \frac{N}{(1 + f_c)} \tag{1}$$

on a computer with N nodes where the overhead f_c has a term due to communication which has the form

$$f_c \propto \frac{1}{n^{1-1/d}} \; \frac{t_{\mathrm{comm}}}{t_{\mathrm{calc}}} \tag{2}$$

where t_{comm} and t_{calc} are respectively typical node to node communication and node (floating point) calculation time. n is the application grain size and d its dimension which is defined precisely in [10]; in physical simulations d is usually the geometric dimension. Good performance requires $\frac{1}{n^{1-1/d}}$ be "small" with a value that depends on the critical machine parameter $t_{\mathrm{comm}}/t_{\mathrm{calc}}$. The grain size n would be the number of grid points stored on each node in a finite difference problem so that the complete problem had Nn grid points. Implicit in the above discussion is that these problems are "data parallel" in the language of Hillis [11], [12]. This terminology is sometimes only associated with problems run on SIMD machines but in fact, data parallelism is the general origin of massive parallelism on either SIMD or MIMD architectures. MIMD machines are needed for loosely synchronous data parallel problems where we do not have a homogeneous algorithm with the same update operation on each data element.

The above analysis does not apply to asynchronous problems as this case has additional synchronization overhead. One can, in fact, use mes-

$$
\begin{array}{ccccc}
\text{Numerical} & \rightarrow & \text{Virtual} & \rightarrow & \text{Real} \\
\text{Formulation} & \text{“compiler”} & \text{Machine} & \text{Machine} & \text{(Parallel)} \\
\text{of Problem} & & \text{(Virtual} & \text{Specific} & \text{Computer} \\
 & & \text{Problem)} & \text{“assembler”} &
\end{array}
\qquad (3)
$$

sage passing to naturally synchronize synchronous or loosely synchronization problems. These typically divide into communication and calculation phases as given by individual iterations or time steps in a simulation. These phases define an algorithmic synchronization common to the entire application. This is lacking in asynchronous problems which require sophisticated parallel software support such as that given by the time warp system [13].

However, there is a very important class of asynchronous applications for which large scale parallelization is possible. These we call *loosely synchronous complex* as they consist of an asynchronous collection of loosely synchronous (or synchronous) modules. A good example, shown in Figure 2, is the simulation of a satellite based defense system. Viewed at the level of the satellites, we see an asynchronous application. However, the modules are not elemental events but rather large scale data parallel subsystems. In a simulation developed by JPL, these modules included sophisticated Kalman filters and target weapon association [14]. This problem class shows a functional parallelism at the module level and a conventional data parallelism within the modules. The latter ensures that large problems of this class will parallelize on large machines. Image analysis, vision and other large information processing or command and control problems fall in the loosely synchronous complex class.

A final problem class of practical importance is termed “embarrassingly parallel”. These consist of a set of independent calculations. This is seen in parts of many chemistry calculations where one can independently compute the separate matrix elements of the Hamiltonian. Another example is seen in the operation of the New York stock exchange where the trading of 2000 stocks can be independently controlled by separate computers — in practice the SIAC corporation distributes the stocks over a few hundred personal computers or workstations with each handling the independent trading of a few stocks.

Problem Class	*Machine Architecture*
Synchronous	SIMD, MIMD
Loosely Synchronous	MIMD
Asynchronous	unclear
Loosely Synchronous Complex (A=LS)	Heterogeneous network of SIMD and MIMD machines
Embarrassingly Parallel	Network of workstations SIMD, MIMD

Table 2: Parallel Computer Architectures Suitable for each Problem Class

4 Hardware

Table 2 shows that the five different problem architectures are naturally suited (i.e., will run with good performance) to different parallel machine architectures.

As described in the previous section, all problems except those in the pure asynchronous class, naturally parallelize on large scale machines as long as the application is large enough. In my 1989 analysis [2], [3] of 84 applications in 400 papers, I estimated that synchronous and loosely synchronous problems dominated scientific and engineering computations, and these two classes were rightly equal in number. This argues that both SIMD and MIMD machines are valuable. Around 50% of the surveyed problems could effectively use a SIMD architecture whereas a comparable number can exploit the additional flexibility of MIMD machines. Note that all distributed memory machines — whether MIMD or SIMD — are message passing and so subject to similar analysis. One views the 64 K CM-2 not as a bunch of virtual processors controlled by data parallel CMFortran, but rather as a set of 2048 WEITEK based nodes exchanging messages over a hypercube network.

We found 14% embarrassingly parallel applications and 10% asynchronous problems in [2], [3]. The latter contain some loosely synchronous complex problems, but we had not identified this separate class at the time. As parallel computing matures, we expect to see more examples of this complex heterogeneous class — especially in commercial and government applications.

5 Software

In our picture shown in Equation 3, software maps problems onto the hardware in one or more stages.

We can understand many of the different software approaches in terms of choices for the virtual machine which is the user's view of the target computer. Essentially all our Caltech work on the hypercube and other MIMD machines used a C (Fortran) plus explicit message passing software model. This corresponds to choosing a virtual machine model that was either a hypercube or more generally a collection of nodes able to exchange messages independent of a particular topology. The latter was called VMLSCS in [10] for Virtual Machine Loosely Synchronous Communication System. This software model was very successful in that as shown in Figure 3, one is able to use it to map essentially all problems onto a MIMD distributed memory multicomputer. Its strengths and weaknesses are a consequence of using a virtual machine model close to a real machine. This allows great generality in problems but produces non-portable code that is specific to one machine class. Further, it is hard work as the user must map the problem a "long way" from the original application onto the virtual machine.

Over the last few years, another approach has become popular which corresponds to using a virtual machine model which is close to the problem and not the machine architecture. We view the use of CMFortran in this fashion corresponding to a virtual machine representing data parallel synchronous problems. The two approaches are contrasted in Figure 4. This analysis suggests that data parallel Fortran can be mapped onto both SIMD and MIMD machines. We view CMFortran as supporting a SIMD virtual machine (SIMD problem architecture) and not as the language for just SIMD hardware. For this reason, we prefer to term the "compiler" target in Equation 3 as the virtual problem and not the more common description as a virtual machine. This terminology makes it more natural to consider languages like CMFortran as the languages for "SIMD problems" (synchronous problems) rather than the languages for SIMD machines.

The Rice and Syracuse groups [15], [16], [17] have proposed FortranD

Table 3: Extensions of FortranD for Different Problem Classes

as a data parallel Fortran suitable for distributed memory machines. This generalizes the concepts behind CMFortran in several ways. As shown in Figure 5, FortranD includes Fortran 77D and Fortran 90D with implicit and explicit parallelism respectively; the compiler for Fortran 77D uses dependency analysis to uncover data parallel constructs which are explicit in the array operations and run-time library of Fortran 90D. FortranD targets both SIMD and MIMD machines. Although the initial design for FortranD was largely aimed at synchronous problems, it is flexible enough to include loosely synchronous problems. In fact, we expect that with suitable extensions, FortranD and similar languages should be suitable for the majority of synchronous and loosely synchronous problems. Thinking Machines has pioneered many of these ideas with their adoption of CMFortran for the SIMD CM-2 and MIMD CM-5.

The loosely synchronous extensions to FortranD are designed to handle irregular problems which we already understand how to implement with explicit message passing. However, higher level software models as defined by Figure 6, such as FortranD are I believe essential if parallel processing is to become generally accepted. We have used the ideas behind Parti [18], [19] in the loosely synchronous implementation of FortranD. Table 3 summarizes work in progress with Saltz. We need to divide the loosely synchronous class into subclasses which each have rather different needs in language extensions. We have examined initially some partial differential equation and particle dynamics problems. We see four major subclasses. The simplest case is typified by an unstructured mesh which has a single static irregular data structure. The hardest case is typified by the fast multipole method for particle dynamics [20], [21] where one has an irregular dynamic data structure which is implicitly defined. As we consider further examples such as vision and signal proceedings, we may discover new issues or in our problem architecture language, new loosely synchronous problem architecture characteristics which need to be explicitly recognized in FortranD.

6 Conclusions

We have claimed that the message passing model was and will continue to be very successful. The vendors will build better and better hardware with lower communication latency and reasonable $t_{\mathrm{comm}}/t_{\mathrm{calc}} \lesssim 10$. We view the message passing software model as "assembly-language" which in many cases we can and should hide from the user with a software model "nearer" that of the problem. Optimizing compilers will translate from a problem oriented software model convenient for users to the message passing level supported by the machine. This latter level will continue to be used directly for applications for difficult cases which are not efficiently supported in the high level software.

Acknowledgements

This work was supported by the ASAS Program Office Techbase Program, and by DARPA under contract #DABT63-91-C-0028. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred. This work was supported in part with equipment provided by the Center for Research on Parallel Computation with National Science Foundation Cooperation Agreement No. CCR-8809165—the Government has certain rights in this material.

References

[1] Fox, G. C. "Concurrent processing for scientific calculations," in *Proceedings of the IEEE COMPCON*. IEEE Computer Society Press, February 1984. Conference held in San Francisco. Caltech Report C3P-048.

[2] 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, 1990.

[3] Fox, G. C. "What have we learnt from using real parallel machines to solve real problems?," in G. C. Fox, editor, *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 2*, pages 897–955. ACM Press, 11 West 42nd Street, New York, NY 10036, January 1988. Caltech Report C3P-522.

[4] 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.

[5] Fox, G. C., and Furmanski, W. "The physical structure of concurrent problems and concurrent computers," in R. J. Elliott and C. A. R.

Hoare, editors, *Scientific Applications of Multiprocessors*, pages 55–88. Prentice Hall, 1988. Caltech Report C3P-493.

[6] Fox, G. C. "Achievements and problems for parallel computing." Technical Report SCCS-29b, California Institute of Technology, June 1990. Proceedings of the International Conference on Parallel Computing: Achievements, Problems and Prospects; held in Anacapri, Italy, June 3–9, 1990; to be published in *Concurrency: Practice and Experience*; CRPC-TR90083.

[7] Fox, G. C. "Hardware and software architectures for irregular problem architectures," in *ICASE Workshop on Unstructured Scientific Computation on Scalable Microprocessors*, October 1990. Held in Nags Head, North Carolina. SCCS-111; CRPC-TR91164.

[8] Fox, G. C. "The architecture of problems and portable parallel software systems." Technical Report SCCS-134, Syracuse University, July 1991. Revised SCCS-78b.

[9] 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.

[10] 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.

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

[12] Hillis, D., and Steele, G. "Data parallel algorithms," *Comm. ACM*, 29:1170, 1986.

[13] 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.

[14] Meier, D. L., Cloud, K. L., Horvath, J. C., Allan, L. D., Hammond, W. H., and Maxfield, H. A. "A general framework for complex time-driven simulations on hypercubes," in D. W. Walker and Q. F. Stout,

editors, *The Fifth Distributed Memory Computing Conference, Volume I*, pages 117–121, 10662 Los Vaqueros Circle, P. O. Box 3014, Los Alamitos, California 90720-1264, 1990. IEEE Computer Society Press. Held April 9–12, Charleston, South Carolina. Caltech Report C3P-960.

[15] Fox, G. C. "FortranD as a portable software system for parallel computers." Technical Report SCCS-91, Syracuse University, June 1991. Published in the Proceedings of Supercomputing USA/Pacific 91, held in Santa Clara, California. CRPC–TR91128.

[16] Fox, G. C., Hiranadani, S., Kennedy, K., Hoelbel, C., Kremer, U., Chau-Wen, T., and Min-you, W. "FortranD language specifications." Technical Report SCCS-42c, Syracuse University, April 1991. Rice Center for Research in Parallel Computation; CRPC-TR90079.

[17] Wu, M., and Fox, Geoffrey, C. "Fortran 90D compiler for distributed memory MIMD parallel computers." Technical Report SCCS-88b, Syracuse Center for Computational Science, September 1991. CRPC-TR91126.

[18] Saltz, J., Crowley, K., Mirchandaney, R., and Berryman, H. "Runtime scheduling and execution of loops on message passing machines," *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[19] Saltz, J., Berryman, H., and Wu, J. "Multiprocessor and runtime compilation," *Concurrency: Practice and Experience*, 3(5), 1991. Special Issue from International Conference on Parallel Computing, held in Anacapri, Italy June 3–9, 1990.

[20] Salmon, J. *Parallel Hierarchical N-Body Methods.* PhD thesis, California Institute of Technology, December 1990. Caltech Report C3P-966.

[21] 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).