SCCS-488

AN OVERVIEW OF HIGH PERFORMANCE COMPUTING FOR THE PHYSICAL SCIENCES

GEOFFREY C. FOX and PAUL D. CODDINGTON
Northeast Parallel Architectures Center, Syracuse University,
111 College Place, Syracuse NY 13244, U.S.A.

ABSTRACT

We present an overview of the state of the art and future trends in high performance computing, and in particular data parallel computing. 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, concentrating on the difficulties of implementing irregular problems.

1. High Performance Computing

In Fig. 1 we present a familiar plot showing the exponential increase in supercomputer performance as a function of time. A number of supercomputer manufacturers are aiming to deliver Teraflop (10^{12} floating point operations per second) performance by 1995. The power of High Performance Computing (HPC) has been used in an increasingly wide variety of applications in the physical sciences. [1] Fig. 2 shows the performance achieved as a function of time for a particular application which requires supercomputer performance – the simulation of the forces between quarks using quantum chromodynamics (QCD). [2, 3] It is interesting to note that some of these supercomputers were built primarily as dedicated machines for solving this particular problem.

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 announced massively parallel computing initiatives. Software for these systems is a major challenge, and could prevent or delay this hardware trend which suggests that parallelism will be a mainstream computer architecture. 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.

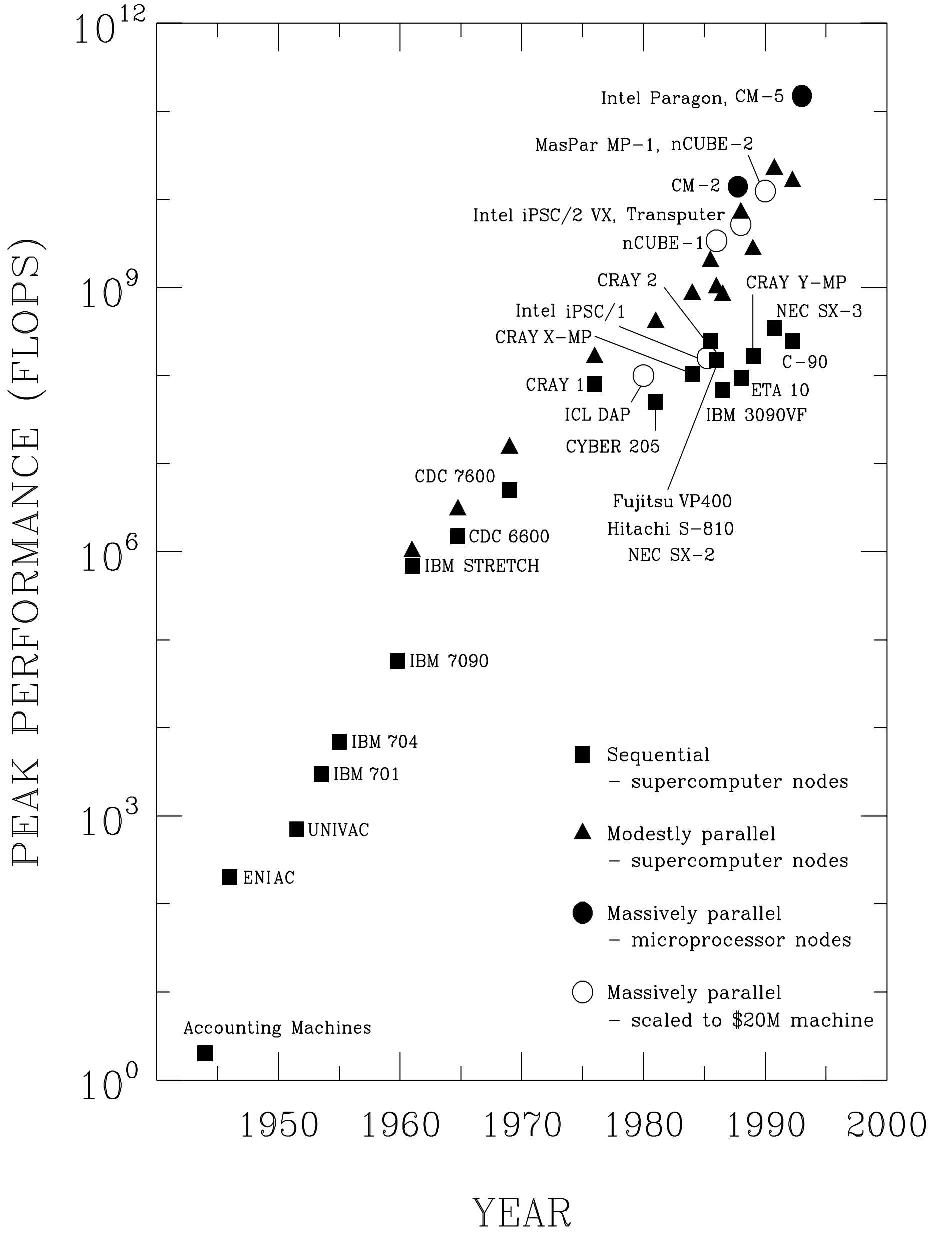

Figure 1: Peak performance in floating point operations per second (flops) as a function of time for various supercomputers.

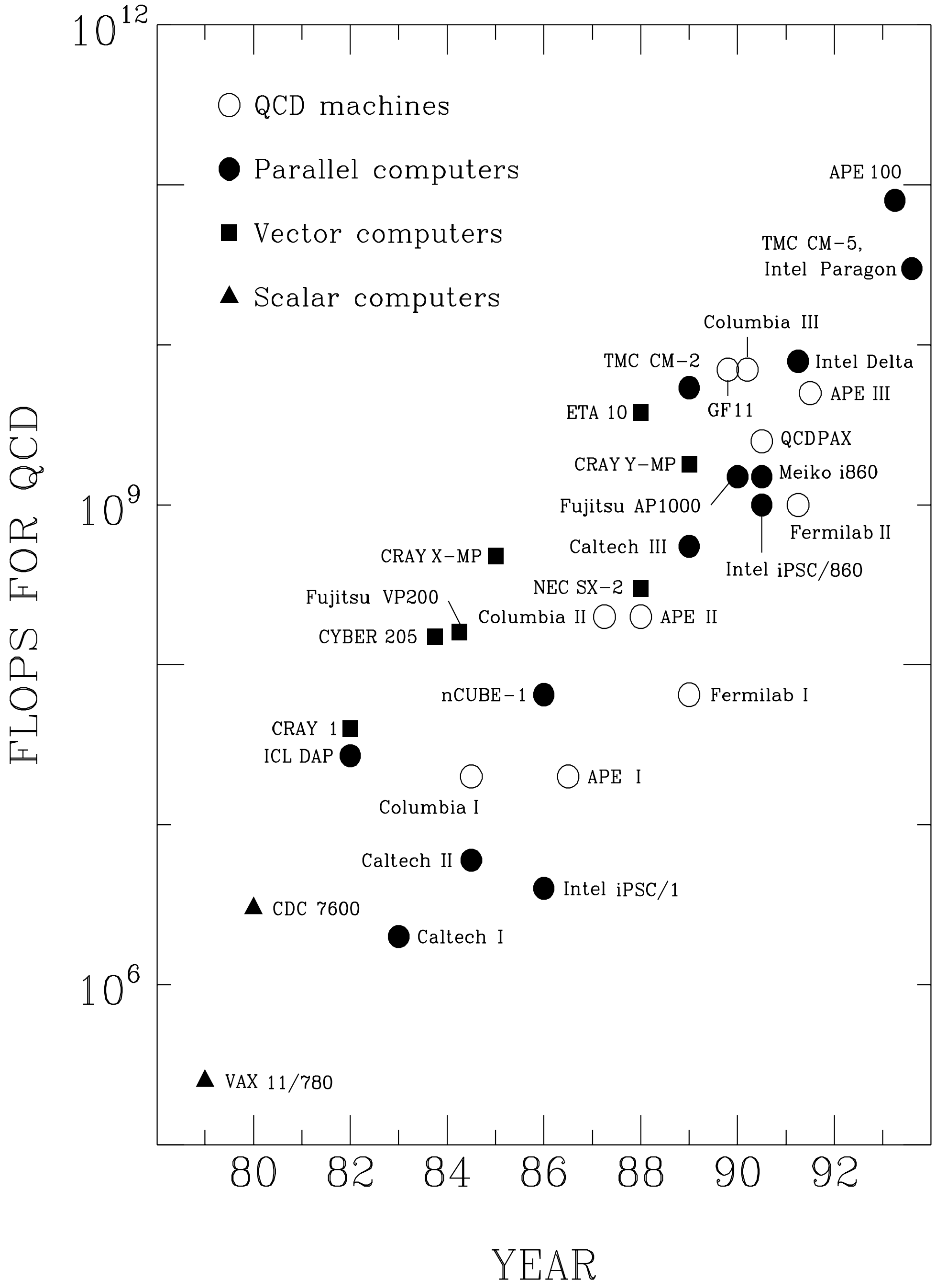

Figure 2: Performance (in flops) of various supercomputers for QCD code, as a function of the time when the code was first run. These are all measured preformance figures, except for the APE 100, CM-5, and Paragon, which are estimates.

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 codes are generally much larger, and have a longer lifetime, than codes for academic research. Machines such as the nCUBE and CM-2 were comparable in price/performance to conventional supercomputers, which was enough to show that "parallel computing works", [2, 4] 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-1, Maspar (DECmpp) MP-2, etc., have enough headroom.

Parallel computing implies not only different computer architectures, but different languages, new software, new libraries, and a different way of viewing problems. It will open up computation to new fields and new applications. To exploit this technology we need new educational initiatives in computer science and computational science. Education can act as the "nucleus" for the phase transition to the new technology of parallel computing, and accelerate the use of parallel computers in the real world.

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.

2. Parallel Computing

Parallel computers have two different models for accessing data, and two different models for accessing instructions: [2, 5]

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

- 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

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" (or domain decomposition), which means obtaining concurrency by applying the particular algorithm to different sections of the data set concurrently. [6] 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 [11] 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 general, high performance architecture which is known to scale from one to very many processors.

3. 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 on effectively all current *and* future machines (at least those we can anticipate today). This means that 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. [7, 8, 9]

The purpose of software, and in particular computer languages, is to map a problem onto a machine.[10, 4] An application in computational science starts out with some physical problem, goes via theory to a model of the physical process, then to an algorithm or numerical method for solving or simulating the model, which is then expressed in a high level language which is targeted to a virtual computer, and finally implemented by a compiler and systems software onto a real computer. Some information is lost in each of the above steps in translating the problem to the machine. The goal of good software should be to make this translation as simple as possible, and to minimize the loss of information.

A drawback of current software is that it is often designed around the machine architecture, rather than the problem architecture. Each class of problem architectures requires different general constructs from the software. It is possible for compilers to construct an approximate computational graph from a dependency analysis of sequential code (such as Fortran 77), and extract parallelism in this way, 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.

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.

3.1. Fortran With Message Passing

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 which 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 usually they are 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*,[2] 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. [16]
- 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.

3.1. 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". One of the major new features of Fortran 90 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 simple 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.

Schematically, a program might look something like the following, using either an array syntax with shifting operations to move data (as in Fortran 90), 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 other dialects of data parallel Fortran):

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

```
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.
- Languages such as CM Fortran and MasPar Fortran are available now and are similar to the Fortran 90 standard.
- An industry standard, High Performance Fortran (HPF), [13] will be adopted, which builds on Fortran 90 and other data parallel languages.

The disadvantages are:

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

4. 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 in early 1992 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 which 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'.

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 which will enable compilers to provide good performance on a variety of parallel and vector architectures. It is derived from a synthesis of ideas developed in many data parallel languages, such as DAP Fortran, Fortran 8X, Connection Machine Fortran, MasPar Fortran, Fortran 90, Fortran D (Rice University), Fortran 90D (Syracuse University), Vienna Fortran, Yale extensions, and Digital's HPF. While HPF is motivated by data parallel languages for SIMD machines, it should also be applicable to MIMD and vector machines. [7, 14]

HPF has a number of new language features:

1. *New Directives*
   These directives 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. The directives are consistent with Fortran 90, so that if HPF were to be adopted as part of a future Fortran standard, only the comment character and the directive prefix would have to be removed.

2. *New Language Syntax*
   These are extensions to Fortran 90 to better express parallelism in a program. They include the FORALL statement and some extra intrinsic functions.

3. *Library Routines*
   HPF will provide a standard interface to a library of efficient parallel implementations of useful routines, such as sorting and matrix calculations.

4. *Extrinsic Procedures*
   Since HPF is a high level, machine independent language, it cannot express certain operations very well (or at all). It therefore allows the use of extrinsic

procedures, which can be defined outside the language, for example by using Fortran with message passing.

5. *Language Restrictions*
   Some restrictions on the use of sequence and storage allocation in Fortran are defined in order to be compatible with the data distribution features of HPF.

6. *Parallel I/O*
   Language facilities for handling parallel I/O are being investigated, but have not been included in the initial HPF language standard.

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 giving information about desired data locality and/or distribution. The compiler then generates code implementing data and work distribution.

- For SIMD computers the implementation is parallel code with communication optimized by compiler placement of data.
- For MIMD computers the implementation is a multi-threaded message passing code with local data and optimized send/receive communications.
- For vector computers the implementation is vectorized code optimized for the vector units.
- For RISC computers the implementation is pipelined superscalar code with compiler generated cache management.

As its name suggests, a major goal of High Performance Fortran is to have efficient compilers for all these machines. Obtaining parallelism solely through dependency analysis has not proven to be effective in general, so for all commands in HPF the dependencies are implied directly, enabling the compiler to generate more efficient code.

4.1. Data Distribution Directives

TEMPLATE directive refers to an abstract, very fine-grain "virtual processor" space (in the language of the Connection Machine model), which labels independent objects (e.g. sites of a lattice) or data elements.

ALIGN directive specifies the lowest-level abstract alignment of array elements. These two directives are used for machine independent alignment of data structures.

```
DIMENSION A(100), B(0:99)
!HPF$ TEMPLATE X(100)
!HPF$ ALIGN A WITH X
!HPF$ ALIGN B(I) WITH X(I+1)
```

PROCESSORS directive specifies a coarse-grain processor grid, used for machine dependent partitioning of data structures.

DISTRIBUTE directive governs the data distribution (domain decomposition), and can be done in BLOCK or CYCLIC fashion.

```
!HPF$ TEMPLATE X(1024), Y(512,512)
!HPF$ PROCESSORS P(16)
!HPF$ DISTRIBUTE X(CYCLIC) ONTO P
!HPF$ DISTRIBUTE Y(BLOCK,BLOCK)
```

DYNAMIC and REDISTRIBUTE directives allow for dynamic data distribution.

4.2. Parallel Statements

INDEPENDENT directive asserts that the iterations in a loop may be executed in any order or concurrently. If the assertion is false, the compiler is free to take any action it deems necessary. There are no restrictions on the loop body, and no way to perform explicit synchronization.

```
!HPF$ INDEPENDENT
DO I = 1, N
   A(IX(I)) = B(I)
   C(I) = F(A(IX(I)), B(IX(I)))
END DO
```

FORALL statement performs elemental array assignment using explicit array indexing. Certain types of functions which do not have any complicating side-effects (so-called PURE procedures) may be called from within a FORALL statement.

```
FORALL (I=1:N, J=1:N, I < J)
   B(I,J) = B(J,I)
   C(I,J) = D(I,K) * B(K,J)
END FORALL
```

4.3. Intrinsic Functions and Standard Library

A number of new intrinsic functions have been added to the Fortran 90 intrinsics such as ALL, ANY, CSHIFT, SUM, SPREAD, etc. These are mainly inquiry functions which return information on the physical processors (e.g. NUMBER_OF_PROCESSORS) or the distribution of data (e.g. HPF_DISTRIBUTION).

HPF also has a standard library, HPF_LIB, with a number of new functions. These include prefix and suffix functions which perform a standard Fortran 90 reduction (e.g. SUM, ALL) on a subset of elements of an array, and sorting functions (GRADE_UP and GRADE_DOWN). Much of HPF's power is buried in its library functions. In many cases the HPF compiler need not know anything about parallelization except interfaces and data layout, since the parallelization is handled by a library routine.

4.4. Compilers and Fortran 90D

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

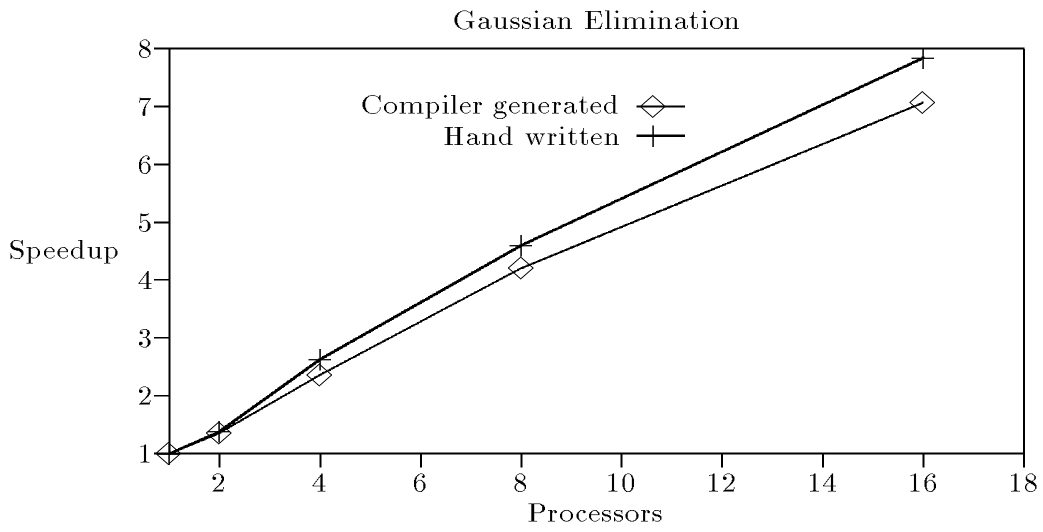

Figure 3: Performance of the Fortran 90D compiler versus hand-written message passing code on the Intel iPSC/860 for Gaussian elimination.

the Northeast Parallel Architectures Center (NPAC) at Syracuse University. The alpha version of the compiler was demonstrated at Supercomputing 92. It supports:

- One- and two-dimensional BLOCK, CYCLIC, and BLOCK-CYCLIC data distribution
- FORALL statement
- Full run-time support for regular communications
- PARTI (ICASE) primitives for irregular communications
- A library of intrinsic functions

The beta release of the compiler is scheduled for June 1993. 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 Fig. 3 for a Gaussian elimination problem.

5. Scientific Applications

The highest profile applications in computational science are the so-called "Grand Challenges". [12] At the last count, there are 39 "official" grand challenge applications in the areas of chemistry, biology, biochemistry, physics, fluid dynamics, the environment, weather forecasting, geophysics, space science and astronomy, and artificial intelligence, which are deemed to be of national importance. Most grand challenge problems involve partial differential equations, particle dynamics, multidisciplinary integration, with some visualization and artificial intelligence. These are all amenable to parallel processing, and can all be expressed in HPF.

Most of the basic problems of this kind, for example solving differential equations, can be done using regular and often local algorithms. HPF can express these types of algorithms very well, and they will generally run very efficiently on MIMD, SIMD and vector architectures. However in many cases, the traditional algorithms have been superseded by modern, more effective algorithms, which are often non-local (e.g. multi-grid) and/or irregular (e.g. adaptive mesh). However the properties of non-locality and irregularity which help make these algorithms so effective, also make them difficult to implement efficiently on a vector or parallel supercomputer. In some cases it is also not obvious how to express this kind of algorithm in HPF.

This is an important issue for HPF: what applications does it support, and what extensions are needed to cover those applications not currently supported? [7, 14] A survey of approximately 400 scientific applications [11] gave the following classification of problems:

Problem Class	Number	HPF Support
Synchronous	40%	Now
Embarrassingly Parallel	14%	Now
Loosely Synchronous	36%	Future
Asynchronous	10%	No

HPF can currently handle synchronous problems, using data parallel arrays manipulated using intrinsic functions (shift, multiply, add, sum, etc.). Examples include:

- Regular grid partial differential equation solvers (e.g. finite difference)
- Regular "crystalline" particle interactions (e.g. lattice models of high energy physics theories such as quantum chromodynamics, spin models of magnets)
- Particle dynamics using $O(N^2)$ N-body algorithms
- Fast Fourier Transforms
- Dense matrix algorithms
- Low level image processing

The INDEPENDENT construct in HPF can be used for embarrassingly parallel problems, such as:

- Database searching (e.g. DOWQUEST [17])
- Monte Carlo calculations summed over independent simulations
- Analysis of independent data sets (e.g. real time analysis of millions of events from a particle collider using a MIMD farm)
- Calculation of matrix elements for computational chemistry (e.g. using MOPAC, GAUSSIAN)

Irregular loosely synchronous problems should be implementable in HPF, for example by using the FORALL statement, the PARTI routines for handling irregular communications and data distribution, [18] or library functions. These kinds of problems can usually only be implemented efficiently on MIMD machines, and in some cases even this is very difficult. Examples include:

- Particle dynamics with cutoff forces and irregular spatial structure (e.g. CHARMM, AMBER)
- Unstructured finite element meshes (static and adaptive)
- Connected component labeling (e.g. image processing, Monte Carlo simulation of spin models)

Problems which probably cannot be expressed in HPF in its current form include extremely irregular problems such as:

- Multiple phase problems (e.g. particle in cell, unstructured multigrid)
- Some $O(N \log N)$ or $O(N)$ fast multipole methods for N-body particle dynamics
- Direct methods for sparse matrix solvers (general methods are very hard to parallelize and give modest performance)

Asynchronous problems such as transaction analysis and event driven simulation are outside the current (and anticipated future) scope of HPF.

Some irregular applications cannot currently be expressed in HPF, or at least not nearly as efficiently as in message passing languages. However the more common irregular applications such as sparse matrix solvers will be incorporated into HPF using software libraries, so the difficulties of the parallelism are hidden in a library call.

Some example applications are described in the following sub-sections. These algorithms are included in a suite of applications codes used to help validate and evaluate the Fortran 90D compiler. [31] This is being expanded into an HPFF application suite, which is being used as "experimental data" to test and evaluate the

HPF language design, and may eventually evolve into a test suite to certify an HPF compiler.

5.1. Component Labeling

Connected component labeling [19] has important applications in image processing and computer vision. [20] It is also the main computational requirement in Monte Carlo simulations of spin models of magnetism using cluster algorithms, in which clusters of spins are updated at once. [21] These non-local algorithms are much more effective than standard Monte Carlo methods such as the Metropolis algorithm. [22]

The Metropolis algorithm is regular and local and can therefore be easily and efficiently parallelized using standard domain decomposition. This is easy to express in HPF and should be handled efficiently by an HPF compiler for any architecture.

The clusters in spin models are highly irregular in both shape and size, and also highly non-local (there is usually one very large cluster). The cluster algorithms are therefore very difficult to parallelize, especially on a SIMD machine. However parallel algorithms exist which can be expressed in data parallel Fortran. [24] These algorithms use general irregular communication routines, and may also benefit from intrinsic functions such as the *scan* routine on the Connection Machine, which scans data along a row or column of an array (like a parallel prefix operation in HPF). Much more efficient algorithms are available on MIMD machines using message passing, [23] but it is difficult to see how these could be expressed in HPF.

5.2. N-Body Particle Dynamics

N-body simulations play a crucial role in theoretical astrophysics. Recently hierarchical methods such as the Barnes-Hut algorithm [25] have been introduced which reduce the time required for N-body simulations from $O(N^2)$ for standard algorithms to $O(N \log N)$ or even $O(N)$. These methods work by reducing an M-body force calculation for a cluster of stars to a single calculation using the center of mass of the cluster. This involves constructing an irregular, hierarchical tree-like data structure (shown in Fig. 4), where each particle is in a single cell. Clusters of particles may be represented by the center of mass of a coarser cell.

Using the standard $O(N^2)$ algorithm, simulations are limited to $O(10^4)$ particles. Using clustering methods, much more realistic simulations of $O(10^6)$ particles are possible. These methods have been used for simulations of the evolution of structure in the Universe following the Big Bang, and for the study of galactic structure caused by collisions of galaxies.

Due to the irregularity of the problem and the complexity of the data structure, this problem is not well expressed in parallel (or even sequential) Fortran, and is difficult to parallelize. In spite of the irregular and complex nature of this problem, John Salmon and Mike Warren have implemented a parallel Barnes-Hut algorithm using message passing on MIMD machines which has given extremely good performance (approximately 5 GFlops on the Intel Delta). [26] This work was acknowledged by the 1992 Gordon Bell Prize.

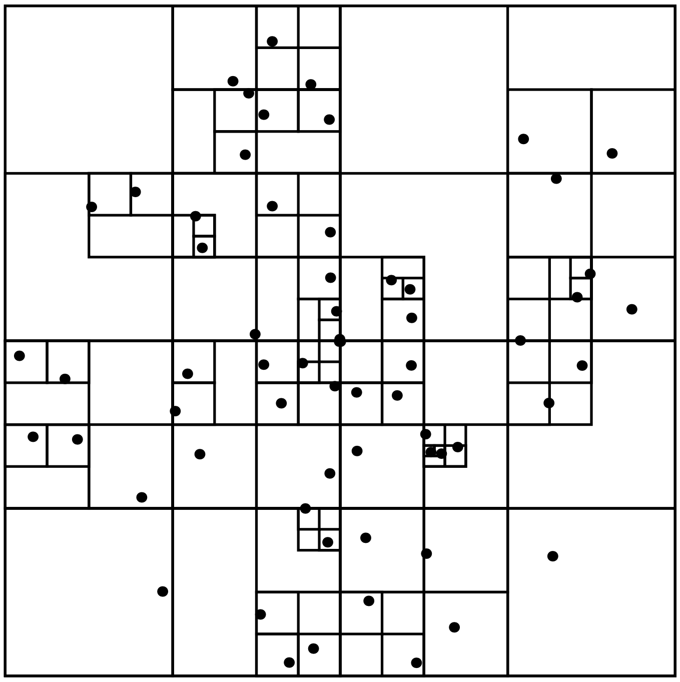

Figure 4: Decomposition of particles into a hierarchical tree structure for a Barnes-Hut N-body particle dynamics algorithm.

In spite of the apparent difficulties in expressing this kind of algorithm in HPF, Warren and Salmon have recently reformulated the parallel Barnes-Hut algorithm in a form which appears suitable for incorporation into an extended HPF compiler as a generalization of the PARTI methods used to parallelize irregular grids. [18]

5.3. Random Surface Simulations

Quantum gravity and string theories of fundamental forces can be simulated using models based on dynamically triangulated random surfaces. [27] For example, the two dimensional worldsheet of a one dimensional string can be discretized as an irregular triangular mesh. Calculations involve integrating over all possible worldsheets, which can be approximated by a Monte Carlo sum over a large number of different meshes. These are obtained by making random changes to the positions and connections of nodes in the mesh throughout the calculation using the standard Metropolis algorithm, [22] resulting in a dynamically triangulated random surface. A number of simulations of string theories have been performed using this method. [29]

Random surface models can also be used to simulate systems which have real fluctuating surfaces, such as biological and chemical membranes, lipid bilayers, microemulsions, and phase boundaries. [27, 28] This application has much in common with solving PDEs using irregular, adaptive grids.

Since these problems require a Monte Carlo simulation, it is possible to use independent parallelism, with a different mesh and different random numbers on each processor, and then average the results over processors. This is supported by HPF, and works well as long as the problem size is small enough so that it fits into the memory of a single processor, and the equilibration of the Monte Carlo algorithm is relatively fast. However for large mesh sizes one or both of these constraints will generally be violated, so a data parallel algorithm where the mesh is distributed over processors is necessary.

The triangulated random surface can be represented as a graph. Two major components of the parallel algorithm are graph partitioning to find a good domain decomposition which balances load and minimizes communication, and graph coloring so that neighboring vertices and edges are not updated in parallel, which would invalidate the Monte Carlo procedure.

Due to the dynamic and irregular nature of the problem, efficient parallelization is difficult, and probably not possible on SIMD machines. A dynamic mesh means that even if the domain decomposition is initially optimal, neighboring vertices will move to other processors during the simulation, thus requiring substantial non-local communication. So dynamic redistribution of data is required to minimize communication overhead. This is available in HPF using the REDISTRIBUTE directive.

It is interesting to note that optimization of performance by improving data locality is not limited to distributed memory parallel machines, but is also crucial in sequential computers. Modern RISC microprocessors are so fast that access to external memory has a large overhead, so good performance is only obtained by optimal use of cache memory. This is analogous to the difference in access times between local memory and off-processor memory in a distributed memory parallel computer.

Maintaining data locality can be difficult for irregular problems such as the dynamically triangulated random surface, and requires the equivalent of dynamic domain decomposition for a parallel machine. For the sequential code, this means constantly updating a secondary data structure which holds the data for neighboring points, rather than just accessing that data from irregular sections of the primary array. For the sequential random surface code, optimizing the data locality in this way gave substantial performance enhancements, up to a factor of 2 for the Intel i860 processor, which has a small cache. [30] The program performs much better on more modern processors with large cache memory.

6. Conclusions

Parallel computing currently offers the best price/performance ratio, and this advantage will increase in the future. Along with improved systems software, language standards, improved compilers, a new generation of computer users educated in parallel computing, and the entrance of large computer vendors such as Digital, Hewlett-Packard and IBM into the field, parallel computing will in the next few years generate enough "headroom" to become the dominant technology.

Much progress is being made on high-level parallel languages such as High Performance Fortran, which will accelerate the transition to parallel computing. HPF can express the majority (perhaps 90%) of computational science problems, including many irregular problems. More than half of applications are synchronous or embarrassingly parallel, and these problems should be mapped very efficiently to different architectures by an HPF compiler. The challenge for compiler writers will be to generate efficient code for irregular problems, which may be difficult or impossible for vector or SIMD architectures. Even for MIMD architectures, the expression of certain types of irregular loosely synchronous problems in HPF, and the generation of efficient MIMD code by an HPF compiler, are still the subject of much research. However if this cannot be done using standard HPF, there is an escape route whereby sections of the code may be called as EXTRINSIC functions, which may be implemented more efficiently using explicit message passing.

We also have a challenge for computational scientists: is your application expressible in HPF? If not, we would like to hear from you!

One final point to note is that scientific applications are certainly important, but are only a limited market for a computer vendor. Most commercial uses of computers involve information processing, decision support, economic (and other complex system) modeling, network simulation, scheduling, manufacturing, education and entertainment. [32] Parallel computing needs to make an impact in these areas, since it will become the dominant technology if and only if it can benefit industry. Industrial applications typically have much larger codes than scientific applications (sometimes on the order of millions of lines), which will only be ported to parallel computers when improved, portable and maintainable software, such as HPF, becomes available.

7. Acknowledgements

Our work on parallel languages and HPF is supported in part by the Center for Research on Parallel Computation with NSF cooperative agreement No. CCR-9120008, and by DARPA under contract #DABT63-91-C-0028.

8. References

1. G.C. Fox and D. Walker, "Concurrent supercomputers in science", Caltech Technical Report C3P-646, in *Proc. of the Conference on Computers in Physics Instruction*, E.F. Redish and J.F. Risley eds., (Addison-Wesley, Reading, Mass., 1989).
2. G.C. Fox *et al.*, *Solving Problems on Concurrent Processors, Vol. 1*, (Prentice-Hall, Englewood Cliffs, NJ, 1988).

3. M. Creutz, *Quarks, Gluons and Lattices*, (Cambridge University Press, Cambridge, 1983); R. Kenway, *Rep. Prog. Phys.* **52** (1989) 1475.
4. G.C. Fox *et al.*, *Parallel Computing Works*, to be published by Morgan Kaufman.
5. G.C. Fox, "Parallel Computing", Caltech Technical Report C3P-830, submitted to *Encyclopedia of Physical Science and Technology 1991 Yearbook*, (Academic Press, New York, 1991).
6. W.D. Hillis, *The Connection Machine*, (MIT Press, Cambridge, Mass., 1985); W.D. Hillis and G. Steele, *Comm. ACM* **29** (1986) 1170.
7. G.C. Fox, "FortranD as a Portable Software System for Parallel Computers", NPAC Technical Report SCCS-91, in *Proc. of Supercomputing USA/ Pacific 91 Conference*, Santa Clara, CA, (June 1991).
8. G.C. Fox, "The Architecture of Problems and Portable Parallel Software Systems", NPAC Technical Report SCCS-134.
9. A. Choudhary *et al.*, "A classification of irregular loosely synchronous problems and their support in Scalable Parallel Software Systems", NPAC Technical Report SCCS-255, Proc. of the DARPA Software Technology Conference, (April 1992).
10. G.C. Fox, "The Use of Physics Concepts in Computation", NPAC Technical Report SCCS-237, in *Computation: the Micro and Macro View*, B.A. Huberman ed., (World Scientific, River Edge, NJ, 1992).
11. G.C. Fox, "What have we learnt from using real parallel computers to solve real problems?", Caltech Technical Report C3P-522, in *Proc. of the Third Conference on Hypercube Concurrent Computers and Applications, Vol. 1*, ed. G.C. Fox (ACM Press, New York, 1988); I. Angus *et al.*, *Solving Problems on Concurrent Processors, Vol. II*, (Prentice-Hall, Englewood Cliffs, NJ, 1990); P.J. Denning and W.F. Tichy, *Science* **250** (1990) 1217; G.C. Fox, "Lessons from Massively Parallel Applications on Message Passing Computers", NPAC Technical Report SCCS-214, in *Proc. of 37th IEEE International Computer Conference*, San Francisco, CA (February 1992).
12. "Grand Challenges: A Report by the Committee on Physical, Mathematical and Engineering Sciences", (1991), in *The FY 1992 U.S. Research and Development Program.*
13. The High Performance Fortran Forum, "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.
14. A. Choudhary, G. Fox, S. Ranka, and T. Haupt, "Which Applications Can Use High Performance Fortran and FortranD – Industry Standard Parallel Languages?", NPAC Technical Report SCCS-360, in *Proceedings of the Fifth Australian Supercomputer Conference*, Melbourne, (December 1992).
15. G.C. Fox *et al.*, "Fortran D Language Specifications", NPAC Technical Report

SCCS-42C (1990); A. Choudhary *et al.*, "Compiling Fortran 77D and 90D for MIMD Distributed Memory Machines", NPAC Technical Report SCCS-251 (1992).

16. Information on the Message Passing Interface specification can be obtained by sending the following message to netlib@ornl.gov: send index from mpi.
17. D. Waltz and C. Stanfill, "Artificial intelligence related research on the Connection Machine", in *Proc. of the International Conference on Fifth Generation Computer Systems, Vol. 3*, (OHMSHA Ltd., Tokyo, 1988).
18. J. Saltz *et al.*, "The PARTI parallel runtime system", *Proc. of the SIAM Conference on Parallel Processing for Scientific Computing*, Los Angeles, CA (1987); J. Saltz *et al.*, *Concurrency: Practice and Experience* **3** (1991) 573.
19. E. M. Reingold, J. Nievergelt and N. Deo, *Combinatorial Algorithms: Theory and Practice* (Prentice-Hall, Englewood Cliffs, NJ, 1977); E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, (Computer Science Press, Rockville, Maryland, 1978).
20. A. Rosenfeld and A. C. Kak, *Digital Picture Processing*, (Academic Press, New York, 1982).
21. A. D. Sokal, in *Computer Simulation Studies in Condensed Matter Physics: Recent Developments*, eds. D. P. Landau *et al.* (Springer-Verlag, Berlin-Heidelberg, 1988); J.-S. Wang and R. H. Swendsen, *Physica A* **167** (1990) 565.
22. *Monte Carlo Methods in Statistical Physics*, Ed. K. Binder (Springer-Verlag, Berlin, 1986); H. Gould and J. Tobochnik, *An Introduction to Computer Simulation Methods*, (Addison-Wesley, Reading, Mass., 1988).
23. C.F. Baillie and P.D. Coddington, *Concurrency: Practice and Experience* **3** (1991) 129.
24. J. Apostolakis, P. Coddington and E. Marinari, "New SIMD Algorithms for Cluster Labeling on Parallel Computers", NPAC Technical Report SCCS-279 (1992), to be published in *Int. J. Mod. Phys. C.*
25. J. Barnes and P. Hut, *Nature* **324** (1986) 446.
26. J. Salmon, "Parallel Hierarchical N-body Methods", Center for Research on Parallel Computation Technical Report 90-14, Caltech, December 1990; M. Warren and J. Salmon, "Astrophysical N-body Simulations Using Hierarchical Tree Data Structures", in *Proc. of Supercomputing '92*, (IEEE Computer Society, Los Alamitos, 1992).
27. F. David, in *Two Dimensional Quantum Gravity and Random Surfaces*, eds. D.J. Gross, T. Piran, and S. Weinberg, (World Scientific, Singapore, 1992); J. Ambjørn, B. Durhuus and J. Frölich, *Nucl. Phys.* **B257** (1985) 433.
28. *Statistical Mechanics of Membranes and Surfaces*, eds. D. Nelson, T. Piran, and S. Weinberg, (World Scientific, Singapore, 1989).
29. S. Catterall, *Phys. Lett.* **220B** (1989) 207; C. Baillie, D. Johnston and R. Williams, *Nucl. Phys.* **B335** (1990) 469; J. Ambjørn *et al.*, *Phys. Lett.*

275B (1992) 295; M. Bowick *et al.*, "The Phase Diagram of Fluid Random Surfaces with Extrinsic Curvature", NPAC Technical Report SCCS-357, to be published in *Nucl. Phys.* **B**.

30. L. Han, "Optimization of a Dynamic Random Surface Code for RISC Processors", to be published in *Proc. of the SIAM 1993 Meeting*, Philadelphia, PA (July 1993).
31. A.G. Mohamed *et al.*, "Application Benchmark Set for Fortran-D and High Performance Fortran", NPAC Technical Report SCCS-327 (1992). The benchmark suite is available via anonymous ftp from minerva.npac.syr.edu in directory benchmark.ftp.
32. G.C. Fox, "Parallel Computing in Industry – An Initial Survey", NPAC Technical Report SCCS-302b, in Supplemental Proceedings of the Fifth Australian Supercomputer Conference, Melbourne, (December 1992).