SCCS-360

Which Applications Can Use High Performance Fortran and Fortran-D: Industry Standard Data Parallel Languages?

A. Choudhary, G. Fox, T. Haupt, S. Ranka
Syracuse University
Northeast Parallel Architectures Center

submitted to the Fifth Ausralian Supercomputing Conference
Melbourne, Australia, December 7–9, 1992

Abstract

In this paper, we present the first, preliminary results of HPF/Fortran-D language analysis based on compiling and running benchmark applications using a prototype implementation of HPF/Fortran-D compiler. The analysis indicate that the HPF is a very convenient tool for programming many applications on massively parallel and/or distributed systems. In addition, we cumulate experience on how to parallelise irregular problems to extend the scope of Fortran-D beyond HPF and suggest future extensions to the Fortran standard.

1. Introduction

Since its introduction over three decades ago, Fortran has been the language of choice for scientific and engineering programming. The advent of parallel computers made, however, ordinary Fortran77 of Fortran 90 obsolete, in a sense that codes written in these languages do not provide sufficient information to exploit the full capability of modern architectures. Moreover, programming distributed-memory parallel computers is difficult and the resulting codes are not portable.

To address these problems, the High Performance Fortran Forum was founded, a coalition of industrial and academic groups working to propose a new Fortran standard. It is expected that HPF will be offered by all major U.S. parallel computer vendors with an initial language definition, to be agreed in December 1992.

HPF is closely based on the research language Fortran-D, designed and implemented by research groups at Rice and Syracuse University [1]. We are currently implementing new features in Fortran-D which will handle irregular problems such as adaptive finite element calculations. These are outside the scope of the current HPF definition and we expect Fortran-D and similar research projects will suggest new features for later revisions of HPF.

In this paper, we discuss a set of applications with which we have experience in paralellizing using explicit message-passing. We show which category of applications can be expressed in the HPF language and which need extensions such as those in Fortran-D.

The paper is organized as follows. Section 2 presents an overview of the HPF design. A prototype implementation of Fortran-D is described in Section 3. In section 4, our experience of using Fortran-D to compile and run the benchmarking applications is presented. Finally, section 5 contains conclusions.

2. High Performance Fortran

The definition of HPF is still under discussion and the final draft of the proposal is expected to be agreed upon in December 1992. The detailed description of proposed HPF features can be found elsewhere [2]. This section briefly summarizes the current draft (September '92) of HPF proposal.

2.1 Goals and Scope of HPF

The goals of High Performance Fortran are to define language extensions and feature selection for Fortran supporting:

- Data parallel programming (defined as single threaded, global name space, and loosely synchronous parallel computation)
- High performance on MIMD and SIMD computers with non-uniform memory access costs
- Code tuning for various architectures

These goals are to be achieved by defining a minimal set of extensions to the current Fortran standard, minimizing direct conflict with Fortran-77 and Fortran-90. The proposed new standard will also define open interfaces to other languages and programming styles.

HPF Forum would like to make compiler availability feasible in the near term with demonstrated performance on an HPF test suite.

2.2 HPF Computational Model

The underlying intuition behind HPF computational model is that

- an operation on two or more data objects is likely to be carried out much faster if they all reside in the same processor
- it may be possible to carry out many such operations concurrently if they can be performed on different processors

Thus, the main mechanism for achieving parallelism in HPF is data partition. Distribution of control is then derived from the data distribution using the *owner compute* rule: each assignment is executed by processor(s) that own the assigned variable. However, a compiler may relax the owner rule, in order to improve performance.

2.3 Data Alignment and Distribution

The model of the allocation of data objects to processor memories adopted by HPF is that there is a two-level mapping of data object to abstract processors. Data objects (typically array elements) are first *aligned* with an abstract index space called a *template*; a template is then *distributed* onto a rectilinear arrangement of abstract processors. The implementation then uses the same number, or perhaps some smaller number, of physical processors to implement these abstract processors. This mapping of abstract processors to physical processors is implementation dependent.

2.4 Parallel Statements

There is no consensus yet on what new language features are necessary to explicitly express operations to be executed in parallel. Among submitted proposals are:

2.4.1 FORALL *FORALL* provides means to specify parallel loops in a deterministic manner. In a FORALL loop, each iteration *uses only values defined before the loop or within the current iteration* . When a statement in an iteration of the FORALL loop accesses a memory location, it will not get any value written by a different iteration of the loop. Instead, it will get the *old* value at that memory location (i.e. the value at that location before the execution of the FORALL loop) or it will get some new value written on the current iteration. Similarly, a merging semantics ensures that a deterministic value is obtained after the FORALL if several iterations assign to the same memory location.

Another way of viewing the FORALL loop is that it has copy-in/copy-out semantics. In other words, each iteration gets its own copy of the entire data space that exists before the execution of the loop, and writes its results to a new data space at the end of the loop. Since no values depend on other iterations, the FORALL loop may be executed in parallel without synchronization. However, communication may still be required before the loop to acquire non-local values, and after the loop to update or merge non-local values. Single-statement FORALL loops are identical to those supported in CM—Fortran [3].

2.4.2 INDEPENDENT DO *INDEPENDENT DO* asserts the compiler that the iterations of the loop may be executed independently, i.e. in any order, interleaved, or concurrently without changing the semantics of the program. The compiler is justified in producing a warning if it can prove otherwise.

2.4.3 Array Assignments and WHERE *array assignments* and *masked array assignments WHERE* are defined in Fortran-90 standard.

2.5 HPF Subset

An important goal for HPF is early compiler availability. In recognition of the fact that full Fortran-90 compilers may not be available in timely fashion on all platforms, and also that implementation of some of the HPF extensions proposed are more complex than others, a formal HPF extensions has been defined. Among selected features are Fortran-90 array language and dynamic storage allocation.

3. Implementation of HPF/Fortran-D

The current proposal of HPF is closely based on the research language Fortran-D, described in details elsewhere[1]. Actually there are two dialect of this language: Fortran-77D and Fortran-90D. Since HPF is based on Fortran-90, in this paper we concentrate here on Fortran-90D only. This chapter provides a general idea on how the compiler is implemented, and more detailed description can be found in [4].

3.1 Overall Strategy

The Fortran-D compiler consists of three parts. The Fortran-77D and Fortran-90D front ends process input programs into a common intermediate form. The Fortran-D back end then compiles the intermediate form to the SPMD (Single Program Multiple Data)

message-passing node program. We have several reasons for this strategy:

- Sharing a common back end for both the Fortran-77D and Fortran-90D avoids duplication of effort.
- Decoupling the Fortran-77D and Fortran-90D front ends allows them to become machine independent.
- Providing a common intermediate form helps us experiment with defining an efficient compiler/programmer interface for programming the nodes of a massively parallel machine.

3.1.1 Intermediate Form To compile both dialects of Fortran-D using a single back end, we must select an appropriate intermediate form. In addition to standard computation and control flow information, the intermediate form must capture three important aspects of the program:

- Data decomposition information, telling how data is aligned and distributed among processors.
- Parallelization information, telling when operations in the code are independent.
- Communication information, telling what data must be transferred between processors.

Finally, we believe that the primitive operations of the intermediate form should be relatively low-level operations that can be translated simply for single-processor execution.

We have chosen Fortran-77 with data decompositions, FORALL, and intrinsic functions to be the intermediate form for the Fortran-D compiler. We show later that this form preserves all of the information available in a Fortran-90 program, but maintains the flexibility of Fortran-77. Parallelism and communication can be determined by the compiler for simple computations, and specified by the user using FORALL and intrinsic functions for complex computations.

3.1.2 Node Interface Another topic of interest in the overall strategy is the node interface - the node program produced by the Fortran-D compiler. It must be both portable and efficient. In addition, the level of the node interface should be neither so high that efficient translation to object code is impossible, nor so low that its workings are completely opaque to the user. We have selected Fortran-77 with calls to communication and run-time libraries based on ParaSoft's Express communication library [5]. Evaluating our experiences with this node interface is the first step towards defining an ''optimal'' level of support for programming individual nodes of a parallel machine.

3.2 Fortran-90D Front End

The function of the Fortran-90D front end is to scalarize the Fortran-90D program, translating it to the intermediate form. For the Fortran-90D compiler we find it useful to view scalarization as three separate tasks:

- **Scalarizing Fortran-90 Constructs.** Many Fortran-90 features are not present in our intermediate form. They must be translated into equivalent Fortran-77 statements.
- **Fusing Loops.** Simple scalarization results in many small loop nests. Fusing these loop nests can improve the locality of data accesses, simplify partitioning, and enable other program transformations.
- **Sectioning.** Fortran-90 array operations allow the programmer to access and modify entire arrays atomically, even if the underlying machine lacks this capability. The Fortran-D compiler must divide array operations into *sections* that fit the hardware of the target machine [6,7].

We defer both loop fusion and sectioning to the Fortran-D back end. Loop fusion is deferred because even hand written Fortran-77 programs can benefit significantly [8]. Sectioning is needed in the back end because FORALL loops may also be present in Fortran-77D.

We assign to the Fortran-90D front end the remaining task, scalarizing Fortran-90 constructs that have no equivalent in the Fortran-D intermediate form. There are three principal Fortran-90 language features that must be scalarized: array constructs, WHERE statements, and intrinsic functions [9].

3.3 Fortran-D Back End

The Fortran-D back end performs two main functions - it partitions the program onto the nodes of the parallel machine and completes the scalarization of Fortran-D into Fortran-77. We find that the desired order for compilation phases is to apply loop fusion first, followed by partitioning and sectioning.

Loop fusion is performed first because it simplifies partitioning by reducing the need to consider inter-loop interactions. It also enables optimizations such as *strip-mining* and *loop interchange* [10]. In addition, loop fusion does not increase the difficulty of later compiler phases. On the other hand, sectioning is performed last because it can significantly disrupt the existing program structure, increasing the difficulty of partitioning analysis and optimization.

3.4 Run-time Library

Fortran-90 intrinsic functions represent computations (such as *transpose* and *matmul*) that may have complex communication patterns. It is possible to support these functions at compile time, but we have chosen to implement these functions in the run-time library instead to reduce the complexity and machine-dependence of the compiler.

The Fortran-D compiler translates intrinsics into calls to run-time library routines using a standard interface. Additional information is passed describing bounds, overlaps, and partitioning for each array dimension. The run-time library is built on top of the Express communication package to ensure portability across different architectures .

4. Lesson from using Fortran–90D Compiler

4.1 Benchmarking Suite

To validate our compiler a benchmarking suite has been developed. Currently, the suite contains about 30 application programs and is divided into several groups. More applications will be added to the suite soon. The applications come either from independent research at our lab or are adopted from existing benchmarking suites. The suite contains source codes written in at least four Fortran dialects: Fortran–77, Fortran–77 + hand coded message-passing (Express, PICL and/or NX), Fortran–90 (more precisely, Thinking Machine's CM-Fortran and/or Maspar's mpfortran) and Fortran–90D. The suite is expected to become an official HPF benchmarking suite and is available via anonymous ftp from *minerva.npac.syr.edu*

Two groups of applications (The General Section and The Purdue Set) collects applications designed for the initial test of the Fortran D compiler. The applications are simple, but diverse. They selectively address different aspects of parallel computing and thus enable systematic, clear test of the compiler at the development phase. Of particular value in this context is the Purdue Set, a benchmarking suite to test parallel language designs proposed at Purdue University [11]. Most of the computational problems included in the set have been extracted from larger computations and even though they are somewhat artificial by themselves, they do comprise a rich sample of practical computations.

The other groups of applications comprise a suite of complete, "real life" applications coming from independent research on parallel algorithms in linear programming, matrix algebra, computational physics, financial modeling, weather and climate modelling, and simulation of electromagnetic fields. The suite also includes applications from the NAS[12], GENESIS[13], and SPLASH[14] benchmarking suites. We are also working on paralellization of MOPACK[15].

4.2 Language Analysis

In this section we present preliminary results of HPF/Fortran–90D language analysis. The results are based on compiling applications from the Purdue Set, using the Fortran–90D compiler and running them an nCUBE-2, iPSC/860 and a network of SUN workstations. In addition, some observations has been made by comparing Fortran-90 versions to those coded in Fortran-77 with explicit message-passing, before converting them to Fortran-90D.

The first observation is that for many regular problems, HPF/Fortran–90D is a very convenient tool making programming both SIMD and distributed memory MIMD computers easy. Since (a subset of) Fortan-90 is already a language of choice for many SIMD machines, we concentrate on MIMD computers. The tedious and error prone coding of message passing is generated automatically by the compiler. It significantly reduces time necessary for coding, and allows for fast experimentation with data partitioning and alignment to get the maximum performance. This is very important, since compilers of the current generation are not yet expected to be able to find the optimal data distribution outomatically, and they will need programmer's hints (i.e. compiler directives).

The efficiency of Fortran–90D codes for simple applications of the Purdue Set is good: the execution time is only about 10% longer as compared to its equivalents with hand coded Express massage-passing. Thus, it can be expected that Fortran–90D codes will outperform those with explicit message-passing when all opportunities of optimization are fully exploit by the compiler.

Therefore, we recommend developing new applications in Fortran–90, as opposed to Fortran–77, leaving the option for easy parallelization of the codes open. Few commercial Fortran–90 compilers are already available on the market (NAG, ParaSoft, etc.), many others vendors announced the compiler availability soon.

Parallelization of the existing applications ('dusty decks') involves conversion of Fortran 77 codes into Fortran 90. For small applications (hundreds lines or so) it usually is not a difficult task, as in the case of the Purdue set. For larger applications it may be complicated. In some cases an automatic translators (like VAST and others) can be used, at least at the initial phase to save time of straight forward conversion of do loops into array syntax. Here, the situation is similar to the case of data partitioning: the ultimate goal is to produce fully automatic converters, but the present technology requires programmer assistance to handle more complicated cases.

Nevertheless, we expect that some of dusty decks will have to be completely rewritten before actual paralellization. The point is that an efficient parallelization requires understanding data dependency, in particular, the programmer must be aware of side effects generated by invoked procedures. This, in many cases, may prove to be very time consuming because of "careless" use of Fortran–77 features of sequence and storage associations like *COMMON*, *EQUIVALENCE*, *SAVE*, etc. On the other hand, it is a common experience that rewriting the code may lead to a significant improvement of efficiency even if the application is to be run on the same machine as before. The need for recoding is also necessary in the case of parallelization by explicit message passing. Since the future standards of Fortran, including HPF, will be based on Fortran–90, it is our recommendation to rewrite the code in Fortran–90.

An example of a troublesome application is MOPACK[15], used in computational chemistry. The profile of the code run on an example input data set indicates that roughly 50% of the execution time is used by only two routines DIAG (diagonalization of an array) and DENSIT (construction of electron density matrix). The rest of the execution is carried out by a large amount of repeatedly called small routines. The two time consuming routines can be parallelized, accepting some algorithmic challenges. However, the key issue for an efficient parallel implementation of MOPACK as a complete system is concurrent execution of the rest of the code. The difficulty arises from the fact that there are so many global variables declared as *COMMON BLOCK*s and/or *EQUIVALENCE*d to each other that tracing the data dependency and possible side effect generated by these routines seems as time consuming as rewriting the code from the beginning. Usage of Fortran 90 syntax may help to reshape the control scheme which would benefit not only when paralellizing the code but also when extending the scope of the application and/or refining the adopted model of molecular dynamics.

The other experience with coding the Purdue set applications in Fortran-90D is that some algorithms would be easier to express if some language features were extended. One example is nested *WHERE* constructs, not allowed either by Fortran-90 or current draft of HPF proposal. In practice, if type of data depen-

dency permits (as in case of interpolations), lack of nested *WHERE* construct force a programmer to introduce complicated array mask expressions corresponding to nested *IF* constructs in sequential Fortran-77. This is an example, where concern about feasibility of efficient implementation on an arbitrary architecture was found more important than creating a convenient tool for a programmer to generate easy to understand codes.

On the other hand, HPF Forum endorsed the proposal to accept *FORALL* statement even though it has been not accepted in Fortran-90 standard. The semantics of *FORALL* is very close to that associated with array syntax of Fortran-90, but it allows for non-conformable operands in the right hand side expressions. It can be shown that all applications of Purdue-set can be expressed in Fortran-90 syntax, that is, without *FORALL* statements. Nevertheless, *FORALL* is very convenient, and makes programming easier. Moreover, it usually helps the compiler to optimize the code.

Another extension to Fortran-90 standard, considered by HPF Forum is *INDEPENDENT DO* construct. Essentially, it provides means to tell the compiler that there are no loop-carried dependencies and, as a consequence, all iterations of the loop can be performed in arbitrary order, in particular in parallel. In practice, it is a very convenient tool for programming embarrassingly parallel problems. A very convincing example is the 'ep' kernel of the NAS set.

Not all applications of our benchmarking suite can by expressed in HPF syntax in the way which would guarantee efficient implementation of their algorithms on arbitrary computer architecture. All LAPACK[16] routines included in the suite (LU-, QR-, and Cholesky-factorizations) require different approaches for SIMD and MIMD machines. The most efficient MIMD implementations[17] are based on block algorithms with computing nodes arranged as a pipeline. Such arrangement of processors is not supported by HPF/Fortran-90D. Instead, HPF makes provision which makes possible to call a *foreign* procedure, i.e. a procedure written in other language than HPF, in particular, in Fortran-77 or Fortran–90 with explicit message-passing. In this way an HPF source can be linked with an architecture's optimized library, which in this case seems to be a reasonable solution. In our analyses of benchmarking applications we try to identify computational problems which are difficult or impossible to express in HPF. The cumulative experience allows us to experiment with our research language and compiler, and suggest extensions to the future Fortran standards.

5. Conclusions

In this paper we presented our preliminary results of analysis of High Performance Fortran/Fortran-D language designs, based on compiling and running applications from our benchmarking suite. The main conclusions can be summarized as follows:

- We developed a prototype HPF compiler, Fortran-90D compiler.
- Initial tests on small applications demonstrate good performance of resulting codes, even though not all of planned optimizations have been yet implemented.
- HPF seems to be a very convenient and efficient tool for data parallel programming for many applications.
- Parallelization of existing applications using HPF requires conversion from Fortran-77 to Fortran-90. We anticipate that for some of these applications the conversion may be time consuming and the effort may be similar to rewriting the code from the beginning. We estimate, that a direct parallelization of the Fortran-77 code with hidden data dependencies an unconscious side effects will cost a similar effort as rewriting the code in Fortran-90. Thus, in spite of the cost of doing the conversion, we recommend it. This will benefit not only in easy parallelization using HPF but it also will make the code easier to maintain and further development.
- We will recommend to the HPF Forum some additional extensions to the Fortran-90 standard, in particular, new parallel statements and/or compiler directives, as a conclusion of analysis of applications from our benchmarking suite.

- We are gathering experience on irregular problems which are difficult or impossible to express in HPF. We will test various solutions extending the scope of Fortran-D beyond HPF. This activity should lead to modifications and/or extensions of Fortran standard in the future.

Acknowledgments

We are grateful to the Fortran-D research groups for their assistance, and to Parasoft for providing the Fortran-90 parser and Express. This research was supported by the Center for Research on Parallel Computation (CRPC), a National Science Foundation Science and Technology Center. Use of the Intel iPSC/860 was provided by the CRPC under NSF Cooperative Agreement Nos.\ CCR-8809615 and CDA-8619893 with support from the Keck Foundation. This work was sponsored by DARPA under contract \# DABT63-91-C-0028. The content of this information does not necessarily reflect the position or policy of the government, and no official endorsement should be inferred.

References

1. G. Fox, S. Hiranandani, K. Kennedy, C.Koelbel, U. Kremmer, C. Tseng, and M. Wu, *Fortran D language specification.* Technical Report CRPC#TR90–141, Dept. of Computer Science, Rice University, December 1990.
2. High Performance Fortran Forum, *Draft of High Performance Fortran Language Specification*, September 1992, unpublished
3. Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, September 1989.
4. A. Choudhary, G. Fox, S. Ranka, S. Hiranandani, K. Kennedy, C. Koelbel, C. Tseng, *Compiling Fortran-77D and 90D for MIMD Distributed-Memory Machines*, Technical Report CRPC#TR92–203, Dept. of Computer Science, Rice University, December 1990, submitted to Frontiers '92.
5. Parasoft Corporation, *Express User's Manual*, 1989.
6. J. R. Allen, *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations*, Ph.D. Thesis, Rice University, April 1983.
7. J. R. Allen, K. Kennedy, *Vector Register Allocation*, Technical Report TR86–45, Dept. of Computer Science, Rice University, December 1986.
8. C. Koelbel, P. Mehrotra, *Compiling global name space parallel loops for distributed Systems*, IEEE Transactions on Parallel and Distributed Systems, 2(4):440–451, October 1991, and K.. S. McKinley, *Automatic and Interactive Parallelization*, Ph.D. Thesis, Rice University, April 1992.
9. ANSI X3J3/S8.115, *Fortran 90*, June 1990.
10. J. R. Allen, K. Kennedy, *Automatic translation of Fortran programs to Vector Form*, ACM Transactions on Programming Languages and Systems, 9(4):491–542, October 1987.
11. J. R. Rice, J. Jing, *Problems to Test Parallel and Vector Languages*, University of Rice, Technical Report CSD-TR-1016, 1990.
12. P.O. Frederickson, D.H.Bailey, *NAS Parallel Benchmark* , NASA Ames Research Center.
13. C. Addison et. al., *The Genesis Distributed Memory Benchmarks*, Electronics and Computer Science Department, Southampton University, England.
14. J. P. Singh, W. D. Weber, A. Gupta, *SPLASH: Stanford Parallel Applications for Shared Memory*, Computer Systems Laboratory, Stanford University.
15. M. Coolidge, J. Stewart, *MOPAC Manual*, F. J. Seiler Research Laboratory, United States Air Force Academy, December 1990.
16. J. Dongarra, F.G. Gustavson, A. Karp, *A Set of Level 3 Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, 16:1–17, March 1990
17. G. C. Fox, G. von Laszewski, N. T. Lin, A. G. Mohamed, M. Parashar, and N. Yeh, *High Performance Scalable Matrix Algebra Algorithms for Distributes Memory Architectures*, NPAC, Syracuse University, Technical Report SCCS-271, 1992.