A Portable Compiler For
High Performance Fortran

Jairo Panetta
Eduardo Voigt

Instituto de Estudos Avançados - Centro Técnico Aeroespacial
Celso Luiz Mendes
Instituto Nacional de Pesquisas Espaciais

INTRODUCTION

Programming current generation parallel machines is not an easy task. Although there have been many recent advances, producing high quality execution code with a reasonable programming effort has been an open problem for the last twenty years or so.

Currently, there are two methods to produce portable, efficient code for parallel machines. The first method is to write a sequential program for each processor and to use a standard library for data communication and process synchronization. The second method is to use a standard parallel language that has widely available compilers.

The ‘standard library’ approach usually requires a long development cycle and produces a program that is hard to maintain and modify. The ‘standard parallel language’ approach usually has a shorter development cycle and provides a better mechanism for source modification and maintenance. However, with the current stage of compiler technology, code produced by the first method generally executes faster than that produced by the second method.

One of the most promising parallel languages is High Performance Fortran[1]. HPF code is reasonably portable, due to a careful language standardization process and the availability of compilers for all major computing systems. The structure of HPF allows incremental conversion of a Fortran 90 (or Fortran 77) sequential program into a parallel program, by introducing HPF statements as compiler directives (comments in Fortran 90). The very few HPF language elements that are not recognized by Fortran 90 compilers (the for all construct and the pure procedure attribute) were included in the Fortran 95 draft international standard[2].

MOTIVATION

FINEP’s academic PAD subprogram is producing academic prototypes of parallel computer systems with basic system software. A portable compiler for a high level parallel programming language is required to test such systems, since it allows prospective users to move the same application program across multiple platforms. Furthermore, a portable HPF compiler can provide a test bed for compilation methods and language elements research.

The Portable HPF Compiler Project started at IEAv in 1995, as part of the ‘Hipersistemas Integráveis’ project of LSI-USP, with the central goal of producing an HPF compiler that could be used at each of the three major hardware PAD platforms. In order to achieve this goal, the compiler produces Fortran 90 code with either MPI[3] or PVM[4] procedure calls. These basic products should be available at each PAD hardware
platform. The compiler currently recognizes the minimum subset of HPF 2.0 and Fortran 95.

Project design decisions were highly influenced by the scarce availability of human resources. Due to that, simplicity has been the project guideline.

OVERVIEW OF HPF COMPILATION STRATEGIES

There are currently two classes of HPF compilers: ‘scalarizable’ compilers and ‘native’ compilers.

The ‘scalarizable’ HPF compilers use a previously existing ‘parallelizable’ Fortran 77 compiler. They operate in two steps. The first step translates HPF/Fortran 90 constructs into sequential Fortran 77 with run time library calls. This step translates Fortran 90 array constructs and the HPF for all statement into Fortran 77 do statements. The same step transforms HPF directives into data object definitions and declarations as well as run time library calls at appropriate program locations. The second step is to submit the code generated by the first step to the ‘parallelizable’ Fortran 77 compiler, allowing loop re-structuring and ‘parallelization’. This class contains IBM’s xlfhp compiler and APR’s xHPF compiler.

This compilation strategy allows re-utilization of the ‘parallelization’ technology included in the Fortran 77 compiler. The produced code may achieve a high level of parallelism, specially if source code has old Fortran 77 style loops. Its drawback is the ‘scalarization’ process of Fortran 90/HPF loops. This process modifies loop limits and array subscripts, usually transforming simple constant expressions into complex ones. The complexity of such expressions precludes many optimization techniques, which could have been directly applied to the original loop.

On the other hand, ‘native’ compilers work directly over the parallel statements and expressions of Fortran 90/HPF. These are transformed into parallel code, by the insertion of calls to run time libraries and the corresponding modification of loop bounds and array subscript expressions. The remaining source code is untouched, except for communicating distributed object elements. Program analysis is easier than the one required by the other strategy, since data dependency analysis can be performed at a higher level.

The main drawback of this strategy is that efficient parallelism exploitation still strongly depends on the form in which the user expresses the computation. Extensive use of array notation, the for all construct and the independent clause are required. With current generation compilers, many iterations of code modification, compilation and object code analysis are required to achieve efficiency.

THE ADOPTED STRATEGY

We decided to take the ‘native’ approach, due to the project’s simplicity constraint. The parallelization strategy is fully based on user-provided information, either as HPF directives or as strict compiler directives. There was no effort to do any automatic distribution of objects or parallelization of loops that refer to non-distributed objects. We concentrated our effort in the reliability of the compiler, producing correct code for any reference of objects distributed by any HPF form. Our compiler is written in C++, using Flex and Bison for lexical and syntactical analysis. The run time environment is written in Fortran 90 and uses a communication module that hides the underlying message passing library. There are both MPI and PVM implementations of this communication module.
COMPILER STRUCTURE

The compiler has the following four phases:

1. Source Code Recognition and Analysis
2. Data Mapping
3. Communication Generation
4. Pretty-printing

The first phase recognizes HPF source code and produces two outcomes:

- a native Fortran 90 program, by converting (or eliminating) non Fortran 90 elements such as for all and pure;
- a full program graph with the original source code information, for all to do transformations and corresponding tables.

The first product is submitted to the front-end of the native Fortran 90 compiler of the target machine, to flag semantic errors. Semantical analysis of HPF directives is performed on the second product, while collecting data distribution and alignment information.

The data mapping phase starts by generating, in the program graph, calls to run time routines that store distributed object attributes such as array bounds and distribution type. It also generates allocate and deallocate statements for creation and destruction of distributed objects.

The program graph is then scanned for references to distributed objects. Loop nodes in the graph are labeled sequential or parallel based only on loop header, data distribution references and user provided information. Array statements are also analyzed and transformed, keeping array notation as much as possible, to enable optimizations by target machine’s Fortran 90 compiler.

The annotated graph is then submitted to the communication generation phase. Alignment information is used to avoid unnecessary communication, but the default is to resolve references at run time. Each right-hand side reference not aligned with the left-hand side of a statement generates a call to run time routines to resolve processor ownership and to communicate non-local data, under the owner computes rule.

Special attention is devoted to parallel loops. Whenever possible, communication is factored out of the loop and vectorized, allowing fast execution of the inner part of the loop.

Finally, the pretty printer transforms the graph into a Fortran 90 program, to be submitted to the target compiler.

COMPILER DIRECTIVES

The user may interfere in the compilation process by using compiler directives. That mechanism allows users to express the locality of each reference, additional array locations required to accommodate array elements owned by other processors and a mechanism for scalar and vector expansion, similar to the new clause of the independent HPF directive.
By using this mechanism, we can keep the compiler simple, at the cost of user intervention. We plan to expand such mechanism to vectorize communication on complex loops and to provide data dependency information.

CONCLUSION

We presented a quite simple HPF compiler that trades intricate code analysis and parallelization by user intervention. Simplicity was a design decision due to the scarce availability of human resources. Nevertheless, the compiler was designed to allow future inclusion of modules that may substitute user intervention.

The current version has many limitations and requires extensive experimentation on moderate to large source programs. The experiments conducted so far, however, indicate that project goals are close to be achieved, allowing users to write parallel programs that are portable across PAD platforms.

BIBLIOGRAPHY