A Parser for F77 Array Declarations and HPF Directives *

Alan Louis Scheinine
Internet email: scheinin@crs4.it
Centro di Ricerca, Sviluppo e Studi Superiori in Sardegna

Abstract.
The parser of Ftnchek (principal author Robert Moniot) was modified so that the output includes the bounds of arrays. The syntax rules for a useful subset of HPF (High Performance Fortran) directives were also added to the parser. The output of Ftnchek was modified to have a highly regular structure and a Perl script was written to read the output. A group of Perl scripts was written so that, given a section of F77 code with HPF directives, Perl arrays would be returned containing (for each module, i.e. main program, subroutine, function, or blockdata) a list of entry points, subprograms called, statement functions, common blocks, namelists, variables, HPF objects, and HPF directives. In this report the parser will be called "the HPF parser" though the nomenclature is a bit of an exaggeration.

In particular, the array information (bounds and type) and HPF information could be useful for developing macros that can be expanded into message-passing directives using a preprocessor. There are several much more advanced parallelization packages available in the public domain, for example Chameleon and Adaptor. Nonetheless, the parser described herein may be useful for certain niche applications that are not served by other packages. An example of such a niche application is given in this report, the creation of a ParView description file.

*Lavoro eseguito con il contributo finanziario della Regione Autonoma Sardegna.

Contents

1 Introduction **3**

2 Why is a parser needed? 4

3 Generation of a Description File for ParView 4
- 3.1 Introduction 4
- 3.2 Usage details 4

4 Concluding Remarks **7**
- 4.1 Why F77, why not F90? 7
- 4.2 Macro Expansion 7

5 Acknowledgements **7**

6 Appendix: What is a parser? **8**

1 Introduction

The description of an algorithm for a message-passing computer, such as the SP-1, is substantially different from what would be used for a SIMD computer, such as the CM-200. In more general terms, the most recent version of Fortran (Fortran 90) has a data parallel syntax and yet many parallel computers have distributed memory and communicate with message passing. In some cases, the vendor has written a compiler that can hide the message-passing routines, but in other cases the user is expected to specify every call to message-passing subroutines. The value of a computer program is enhanced if it can be used on either a message-passing or a data-parallel computer with a minimum number of changes done by the programmer, leaving most of the work to a preprocessor.

In many cases, a few lines of Fortran 90 instructions requires a page or two of message passing instructions. Consequently, porting a program to a message-passing computer can be time-consuming and the result can be difficult to read and understand. Various computer vendors, software houses, and academic research groups are developing compilers or preprocessors that can start with a Fortran 90 program, perhaps with HPF (High Performance Fortran) directives, and generate a program that will run on a message-passing computer. The parallel computer group at CRS4 is developing a compiler for Fortran 90 with HPF directives (contact Dr. Pietro Rossi for more details) and we have a portable C++ parallel library called ParLib++ (contact David Greco for more details).

Suppose the program developer would like to insert a "macro" into a program, which could be expanded by a preprocessor into message-passing code that meets his or her particular needs. Typically, the preprocessor needs to know the size of the arrays and the layout of the arrays on the processors. This report describes a parser written at CRS4 (evolved from Ftnchek) that will find array declarations, passing the information to a Perl script that can be used for generating new Fortran code or other files. The parser also recognizes a useful subset of HPF directives.

The parser is based on the public domain package called Ftnchek written by Robert Moniot (Fordham University, New York, NY). His parser was extended by having it provide as output the bounds of declared arrays and also by having it output HPF declarations for PROCESSORS, TEMPLATES, and DISTRIBUTE directives. The Ftnchek parser is for Fortran 77 (and the most widely implemented extensions). If the package described in this report proves to be useful, it could be expanded by using a Fortran 90 parser.

This "HPF parser" is intended to be complementary to the compiler being developed within the Parallel Computer group — aside from the present restriction (of the parser described herein) to Fortran 77. CRS4's compiler will use HPF directives rather than cumbersome subroutine calls that are specific to a particular package. To be consistent, this parser was given the ability to recognize HPF directives.

2 Why is a parser needed?

The information about an array can be distributed among several declaration statements, as shown in the following example.

```
PARAMETER(NX=1000)
PARAMETER(NY=1000)
REAL A
COMMON /COM/ A(NX, NY-1: NY*2)
```

Consequently, a simple Perl script could not find the bounds and type of the array `A`. We need to use a parser that understands Fortran. If the parser can also find information expressed by HPF directives such as the following,

```
!HPF$ PROCESSORS PROCS(10,20)
!HPF$ DISTRIBUTE A(BLOCK,BLOCK) ONTO PROCS
```

then we have enough information to automatically generate some message-passing statements.

The information can also be used for other purposes. For example, the visualization interface called ParView needs to have an input file that describes how data is distributed among parallel processes. This file can be generated automatically given the array declarations and the HPF directives.

3 Generation of a Description File for ParView

3.1 Introduction

A Perl script was written to demonstrate the creation of a ParView description file from a Fortran program with HPF directives. The Perl script has the path name `/u/scheinin/pub/PARSE/parview/pvdist.perl` . At the present time the ParView description file must be typed-in by hand, which is time-consuming and error-prone. The pvdist.perl program shows how this could be done automatically.

The format of the pvdist.perl output is not a complete ParView description file. In fact, ParView operates at a higher level of abstraction and depends on a user-supplied subroutine to translate between the variables — the fields — in the program and the fields drawn by ParView. Moreover, the ParView description file can tesselate space with diverse sizes of bricks, whereas the pvdist.perl output is limited to a regular grid that corresponds to the BLOCK distribution. Nonetheless, the pvdist.perl program demonstrates a promising direction towards integrating the ParView package with parallel programs.

3.2 Usage details

In the directory `/u/scheinin/pub/PARSE/parview` one can find the Perl script `pvdist.perl` and an example program called `pvtest.f` . The pvdist.perl program is interactive and can be started by typing its name. As a point of reference in describing its operation, the help information printed by the program is shown on the following page.

```
 parse <filename>  Parse a fortran file
 list m[odules]  Lists the program units of the file
 list a[rrays] [<modulename>]  Lists arrays [ in <modulename> ]
               or default module
 list d[istribute] [<modulename>]  Lists distribute
    directives [in <modulename>] or default module
 set m[odule] [<modulename>]  Changes default module
 select <arrayname> [ <modulename> ]  Looks for a DISTRIBUTE
    directive [in <modulename>] or default module
 help  Print this help text
 quit  Quit this Perl script
```

The first step is to parse a program, as shown below.

```
scallion$ pvdist.perl
comandami> parse pvtest.f
pvtest.f Bytes read: 1897
 Warning, if this program crashes, you may need
 to type "  rm /u/scheinin/pub/PARSE/parview/delete_me*  " .
```

The user's directory serves as scratch space for the execution of Ftnchek. The main program, and each subroutine or function is a separate "module." The scope of an array definition is restricted to the module in which the array is declared, of course. For the "list" and "select" commands an optional module name can be given as the third argument. If the module name is not given, the default value is used, which is initially the first module in the file. To set a different module as the default, the user can type `set module` *modulename*. An example is shown below.

```
comandami> list modules
Module: pvtest ,  Type: prog
Module: pvsub1 ,  Type: subr
comandami> set module pvsub1
Default module is pvsub1
comandami> list arrays
Default choice, module: pvsub1  [Note the default value.]
Variables:
 real* ABC( 1: 150, 1: 500, -199: 200 ) rank = 3, size = 30000000
 dcpx HIGHPREC( 1: 150, 1: 500, -199: 200 ) rank = 3, size = 30000000
 real PRESSR( 1: 150, 1: 500, -199: 200 ) rank = 3, size = 30000000
   [Some text delete for clarity.]
Processors and Templates:
 proc PROCDEF( 3, 4, 5 ) rank = 3, size = 60
comandami> list distribute
Default choice, module: pvsub1
DISTRIBUTE directives:
DISTRIBUTE (BLOCK,BLOCK,BLOCK) ONTO PROCDEF :: TEMPR
DISTRIBUTE (BLOCK,BLOCK,BLOCK) ONTO PROCDEF :: PRESSR
   [Some text delete for clarity.]
```

The above example shows how the user can list the array declarations (which includes HPF directives that declare array-like objects) or can list the DISTRIBUTE directives.

The Perl script pvtest.perl can only handle one array — it is only a demonstration. The user chooses the array with the `select` command. The information shown to the

user is similar to the description file, but is more verbose (except that is does not show the list of all the blocks). For the given array, pvdist.perl combines the array declaration, the DISTRIBUTE command in which it is found, and the PROCESSORS declaration, to generate the ParView description file. For the blocks, the interactive output shows how many and of what size. If the final block is smaller than the others, that is shown. (That is to say, the last node in any given dimension may have a smaller piece of the array than the other nodes, when the array size is not an exact multiple of the number of processors in the given dimension.) The data type is given in the C language, based on the translations appropriate for IBM RS/6000 computers.

```
comandami> set m pvtest
Default module is pvtest
comandami> select velocity
Default choice, module: pvtest
DISTRIBUTE (BLOCK,BLOCK,BLOCK,*)
 float VELOCITY( 1: 240, 1: 500, -199: 200, 1: 3 ) rank = 4, size = 144000000
PROCESSORS PROCS( 2, 2, 4 ) rank = 3, size = 16
RANK   = 3  (1:240,1:500,-199:200)
ORIGIN =   1  1  -199
DIM    =   240  500  400
BLOCKS = 16
axis) block_size*num_blocks [ + last block_size ]
 1) 120 * 2
 2) 250 * 2
 3) 100 * 4
FIELDS = 1   (one, per forza in this example program)
FIELD
        NAME = VELOCITY
        FIELDTYPE = VECTOR (1:3)
        DATATYPE = float
Name of output file [none] ? pvtest.descript
Wrote file:  pvtest.descript
```

In the above example, one of the dimensions in the distribute command had an asterisk. Consequently, there is a one-dimensional set of data that is not distributed, hence the field type is "vector." As a final example, below is shown the case of having two dimensions not distributed.

```
comandami> select stress
Default choice, module: pvtest
DISTRIBUTE (BLOCK,*,BLOCK,BLOCK,*)
 double STRESS( 1: 240, 1: 5, 1: 500, -199: 200, 1: 3 ) rank = 5,
 size = 183129088 [The "size = 183129088" is overflow in the Ftnchek program.]
PROCESSORS PROCS( 2, 2, 4 ) rank = 3, size = 16
RANK   = 3  (1:240,1:500,-199:200)
ORIGIN =   1  1  -199
DIM    =   240  500  400
   [Some text deleted.]
FIELDS = 1   (one, per forza in this example program)
FIELD
        NAME = STRESS
        FIELDTYPE = TENSOR of rank 2 (1:5,1:3)
        DATATYPE = double
```

4 Concluding Remarks

4.1 Why F77, why not F90?

Well, it seemed like a good idea at the time.

A few months ago I asked some people in the Computers and Networks group: which of our computers have a Fortran 90 compiler? The answer was: none. (To be more accurate, the CM-200 uses a language that is similar to Fortran 90.) At CRS4 we have IBM RS/6000, Silicon Graphics, Sun, Hewlett-Packard, DEC Alpha, Thinking Machines, and Parsytec computer. Yet only one computer could understand even a subset of F90 syntax, the CM-200. Consequently, for the development of tools for portable parallel programs, Fortran 77 was a better choice. Moreover, there was a public-domain parser available, Ftnchek.

The situation has changed rapidly. DEC and IBM have announced F90 compilers, and the purchase of an F90 compiler is in the CRS4 budget for this year. Moreover, there are now at least two public-domain F90 parsers. Those parsers are part of the research projects Sage and Adaptor. Moreover, rapid progress is being made on an F90 compiler being written in-house.

4.2 Macro Expansion

Work on this project was initially motivated by an exercise in porting a CM Fortran program onto a message-passing computer. It became obvious that in many places a page of message passing commands could be easily implemented with preprocessor (making the code more readable and more portable) if only array bounds, type, and distribution onto nodes was available to the preprocessor. Speaking in the broadest of terms, the goal has been to develop a foundation of tools at CRS4 that will allow us to be among the leading centers of parallel computation. Taking into account work being done at other research centers, it is clear that the original goal of "hiding" the message passing calls for the most commonly used cases would be a duplication of work that has already been done by others. So no examples of macro expansion will be given in this report.

On the other hand, some people at CRS4 may have particular parallel algorithms that have not been anticipated by presently available parallelization packages. Algorithms that could be made portable by using this parser to develop a software package for hiding the machine specifics.

5 Acknowledgements

I want to thank Dr. Pietro Rossi for giving me the opportunity to pursue this work. I am indebted to David Greco for his explanation of parsing and related topics during several conversations; and for his recommendations for improving the text of this report. Moreover, the high quality of the Ftnchek program has been essential. The program Ftnchek was written by Dr. Robert Moniot, with help from Michael Myers, Lucia Spagnuolo, Lois Bigbie, Brian Downing, and John Quinn.

6 Appendix: What is a parser?

This report is not intended to explain the operation of parsing a Fortran program. Nonetheless, the reader might be interested in seeing what is meant by the term "parser." A few extracts from the syntax file are described in this appendix. Without necessarily understanding the details, it is hoped that the reader will be able to see how the formal description of a syntax can be combined with C subroutines to store information during parsing. This is the underlying tool for the utilities described in this report.

In many cases the structure of a phrase in some language can be described as a tree. The set of all possible phrases can be succinctly described by giving rules for forming the branches of the tree. For a computer language, the entire program can be represented as a tree. Every legal program should fit one unique tree representation and finding that tree representation is *parsing*. Storing an explicit tree representation of a program is not the primary goal of parsing; but rather, during the process of matching the branching rules (and terminal symbols) to the program, information is obtained concerning the program.

An example will clarify the meaning of the branching rules. Below is a small section of the Fortran 77 rules for a common block.

```
labeled_common_block:   common_block_name common_variable_list
                ;
common_block_name:      '/' symbolic_name '/'
                |       '/'  '/'                /* block with no name */
                |       tok_concat              /* "//" becomes this */
                ;
common_variable_list:   common_list_item
                |       common_variable_list common_list_item
                ;
common_list_item:       common_entity
                |       common_entity ','
                ;
common_entity   :       symbolic_name
                |       array_declarator
                ;
symbolic_name   :       tok_identifier
                |       tok_array_identifier
                ;
```

Most of the symbols (the ones that do not begin with "tok_" have no meaning aside from making connections to other rules. But all rules must eventually lead to terminal symbols (tokens) that are given meaning by the *lexer*. The above format is understood by a program called Yacc which converts the table of rules into a C program (y.tab.c) that can traverse the user's program. Reading the actual text of the user's program is done by the "lexer" which returns tokens (the terminal symbols) to y.tab.c.

The syntax rules given to Yacc must allow a program to be parsed when the lookahead across the statement line is only one symbol beyond the symbol being *reduced*. (To "reduce"a symbol means to go deeper into the tree.) It is not simple to find a description of a language that is unambiguous with only one step of lookahead. There is an additional complication related to the Fortran language. The keywords of Fortran, for

example COMMON, can also be variable names. The lexer for Fortran tries to resolve that ambiguity.

Even though the non-terminal symbols have no intrinsic meaning, being at a certain rule while parsing does have meaning. In particular, C code can be inserted into the Yacc rule table. In the previous example of the rule table, all of the C code was eliminated so that it could be read easily. During parsing, if the parser arrives at the common_entity rule, then the names that are found are variables declared in common. That fact is noted by calling the C program def_com_variable().

```
common_entity   :       symbolic_name
                        {
                             def_com_variable(&($1));
                        }
                |       array_declarator
                        {
                             def_com_variable(&($1));
                        }
                ;
```

Every symbol has a variable associated with it during the parsing (typically, the variable is a C structure). The variables are called "tokens," which is a different concept from the "tokens" that are terminal symbols. The same word is used because terminal symbols generate token variables — and non-terminal symbols pass token variables. The token variables are temporary while a particular symbol is being reduced. To use a specific example, one data item in the token variable (for a typical Fortran parser) is the position (table entry) in a global table of symbols containing information for any variable that has been found. (The symbol table also has information about other symbols such as common names, subroutine names, etc.) That information is returned by the lexer when a tok_identifier or a tok_array_identifier is found. (See the rule for *symbolic_name*.) In the C statements under the common_entity rule, the symbol $1 refers to the token variable returned from symbolic_name or array_declarator, which is then passed to the routine def_com_variable(&($1)). In particular, the routine reads the symbol table entry for the variable and stores it in a list of variables found in this particular common variable list. After a particular common declaration statement has been parsed, the tokens used by the parser for that statement disappear. But the information that the user feels is important has been grabbed by C routines inserted into the rules.

The Yacc syntax rules for Fortran 77 and the associated lexer that is part of Ftnchek (written by Robert Moniot at Fordham University, New York, NY) were modified to include some of the syntax rules of High Performance Fortran compiler directives. Also, the output of Ftnchek was modified to display the array bounds of all arrays that are declared within each program unit — and, of course, to display (in a standardize way) the HPF directives that are found.

A rule was added to recognize the PROCESSORS compiler directive of HPF. The symbolic_name and dim_bound_list Yacc symbols shown below are the original Ftnchek definitions. The global symbol table uses a number to represent the various kinds (class and type) of Fortran symbols (variables versus function names, etc.) and so it was possible to introduce a new kind of symbol; in particular, the class HPF and various types, such as type_PROCESSORS. The routine declare_type_storage_class adds the class and type tags

to the symbolic_name and the already existing routine def_array_dim is used to store the array bounds of the processors object. This is shown in the following example.

```
processors_decl_list:   processors_decl
                |       processors_decl_list ',' processors_decl
                ;
processors_decl :       symbolic_name
                        {
                            declare_type_storage_class(&($1),
                                type_PROCESSORS,1,(int)class_HPF);
                        }
                |       symbolic_name '(' dim_bound_list ')'
                        {
                            def_array_dim(&($1),&($3),
                                &bounds_list_temp,(int)class_HPF);
                            declare_type_storage_class(&($1),
                                type_PROCESSORS,1,(int)class_HPF);
                        }
                ;
```

In sum, because the software is public domain, other people at CRS4 are invited to copy the parser (in `/u/scheinin/ftnchek02`) and add new functionality to the parser or the Perl scripts (in `/u/scheinin/pub/PARSE/perl`).