Annales UMCS Informatica AI 5 (2006) 47-57

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI

http://www.annales.umcs.lublin.pl/

Professionally designed and developed OpenMP parser for the Ada programming language using flex

Rafał Henryk Kartaszyński[*], Przemysław Stpiczyński[**]

Department of Computer Science, Maria Curie-Skłodowska University,
Pl. M. Curie-Skłodowskiej 1, 20-031 Lublin, Poland

Abstract

This paper describes a new version of OpenMP parser for Ada. AdaOMP consists of: OpenMP compiler for Ada and Ada package with OpenMP routines and variables. We show how compiler writing can be improved by the use of professional tool – flex – lexical analyzer, for kernel creation. In this paper we focus on describing steps leading to parser creation. We will explain some implementation details and present the results of using the OpenMP parser on sequential programs.

1. Introduction

OpenMP (Open specifications for Multi Processing via collaborative work between interested parties from the hardware and software industry, government and academia) is Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism [1]. It comprises the following elements: Compiler Directives, Runtime Library Routines, Environment Variables. Directives can be embedded within the sequential program written in C/C++ or Fortran, on multiple system architectures (UNIX, Windows NT, …).

Ada is a programming language designed to support the construction of long-lived, highly reliable software systems. It also, what is most important, includes facility for the support of real-time, parallel and distributed programming [2,3,4]. Unfortunately, it did not become a popular parallel programming language [5], because these features are very time consuming and complicated in use.

To cope with these problems an idea of implementing OpenMP parser for Ada was born [6] and resulted in creation of AdaOMP. The previous version of

[*]*E-mail address:* hatamoto@goblin.umcs.lublin.pl
[**]*E-mail address:* przem@hektor.umcs.lublin.pl

parser, presented in [7], had many drawbacks. Most of them were caused by non effective approach to compiler programming, i.e. writing its kernel, directly under C. Version 2.0 has restructured kernel based on professional tool – `flex`. Owing to this, all the faults were eliminated and new functionality was added. Therefore we would like to present AdaOMP: OpenMP parser and Ada package containing OpenMP functions and procedures. Implemented directives, clauses and functions are sufficient enough to provide a powerful tool to make parallel programming under Ada fast and simple.

2. AdaOMP syntax, supported directives and library functions

Let us remind syntax, presented in [6] (which become basis for developing OpenMP parser for Ada), of OpenMP directives:

```
pragma omp; -- directive-name [clause] ...
block of code
pragma omp; -- end
```

Till now the following directives have been recognized and interpreted by AdaOMP parser:

- **parallel**
  Code block enclosed in parallel directive (parallel region) is executed simultaneously by multiple threads,
- **parallel for**
  is used to split execution of `for loop` iterations between multiple threads. It is required that `parallel for` pragma preceeds `for loop` directly.

In most practical applications of OpenMP these two directives are sufficient enough to satisfy most programmer needs. In addition, each directive has also a set of clauses. The following clauses are available:

- **private(**variable **[,** variable**])**
  variables provided in this clause are set as private in the parallel region, and are used as scratch storage there,
- **shared(**variable **[,** variable**])**
  behaviour of this clause is opposite to the previous one. Named variables will be shared among all threads. Variables can be accessed and modified by all thread,
- **default(private** | **shared** | **none)**
  by default all variables not declared as private are assumed to be shared. This can be changed by setting default data-sharing to `private` or `none`. When it is `none` each variable within the parallel region must be named in `shared` or `private` clauses.
- **reduction(**operation **:** variable **[,** variable**])**

used to identify `variables` and reduction operations used in parallel regions.

Apart from OpenMP parser, AdaOMP also provides package including additional functions and procedures. Package consists of two files `ada_openmp.ads` and `ada_openmp.adb`, and must be included in every program using Ada OpenMP. This can be achieved by writing the line

```
with ada_openmp; use ada_openmp;
```

at the beginning of Ada program code.

The following functions / procedures are made available:

- `function OMP_GET_THREAD_NUM return integer;`
  returns `thread` identifier, which is different for each thread within the parallel region.
- `procedure OMP_SET_NUM_THREADS(t : in integer);`
  by default number of threads that are started is determined by `OMP_NUM_PROCS` environment variable. If this variable cannot be read 10 is set as default. Before each parallel region this procedure can be used to set a number of threads within the next parallel regions.

3. New approach to parser implementation

In this article we would like to focus our attention on presenting implementation of Ada OpenMP parser. Because, as mentioned before, version 1.0 was not flexible enough to assure simple add of new directives and functionality, we have decided to use `flex`, lexical analyzer, to ensure this. Another disadvantage of the previous version, i.e. lack of support of nested OpenMP directives, was also eliminated by this new approach.

When programming any kind of compiler or parser, for any language, we are facing three general problems:

1. *finding lexical elements of this language (key words: "begin", "end"…; identifiers; numbers; …),*
2. *recognising and checking language grammar expressions – the way lexical symbols coexist in a particular language,*
3. *interpreting expressions found in program code, according to suitable grammatical rules.*

Of course each problem requires different approach and solution to above points can differ. In our case – OpenMP parser – first point is solved by using `flex` [8,9].

Lexical analyzer scans standard input (setting `yyin` variable to `FILE` pointer can change this behaviour) for strings matching defined patterns. By providing a simple set of lexical rules for `flex`, OpenMP directives, clauses, etc. can be easily found in Ada program code. For example, following rules can be used to find OpenMP `parallel` and `parallel for` directives with accompanying clauses:

DIGIT	[0-9]
LETTER	[a-zA-Z]
BLANK	[\t]
IDENT	("_"|{LETTER})({LETTER}|"_"|{DIGIT})*
IDLIST	{IDENT}({BLANK}*","{BLANK}*{IDENT})*
PRAGMA	"pragma"{BLANK}+"omp"{BLANK}*";"
END	"end"
PRAGMAEND	{BLANK}*{PRAGMA}{BLANK}*"--"{BLANK}*{END} {BLANK}*"\n"
PARALLEL	"parallel"
PARALLELFOR	"parallel"{BLANK}+"for"
PRIVATE	"private"
SHARED	"shared"
NONE	"none"
DEFAULT	"default"
REDUCTION	"reduction"
PRIVATEDIRECTIVE	{PRIVATE}{BLANK}*"("{BLANK}*{IDLIST}{BLANK}*")"
SHAREDDIRECTIVE	{SHARED}{BLANK}*"("{BLANK}*{IDLIST}{BLANK}*")"
DEFAULTDIRECTIVE	{DEFAULT}{BLANK}*"("{BLANK}*({PRIVATE}|{SHARED}| {NONE}){BLANK}*")"
REDUCTIONDIRECTIVE	{REDUCTION}{BLANK}*"("{BLANK}*{OPERATION}{BLANK}* ":"{BLANK}*{IDLIST}{BLANK}*")"
PARALLELDIRECTIVES	({PRIVATEDIRECTIVE}|{DEFAULTDIRECTIVE}|{SHAREDDIRECTIVE}|{REDUCTIONDIRECTIVE}{BLANK}*
PRAGMAPARALLEL	{BLANK}*{PRAGMA}{BLANK}*"--"{BLANK}*{PARALLEL}(({BLANK}+{PARALLELDIRECTIVES}*"\n")|"\n")
PRAGMAPARALLELFOR	{BLANK}*{PRAGMA}{BLANK}*"--"{BLANK}*{PARALLELFOR} ((({BLANK}+{PARALLELDIRECTIVES})*"\n")|"\n")
PRAGMAOMP	({PRAGMAPARALLEL}|{PRAGMAPARALLELFOR})

Matched strings are then returned as tokens (numeric representations of strings and simplified processing) and variable `yytext` points to found string [8,9]. However, we must be conscious of some of `lex's` limitations. Because it is based on FSA (Finite State Automaton) it cannot be used, for example, to recognize nested structures such as parentheses or OpenMP directives. In such cases a stack must be incorporated to solve this problem. Whenever scanner encounters starting pragma it is pushed on the stack. When ending pragma is encountered it is matched with the top of the stack, and the stack is popped.

In most cases the answer to the second point is the use of syntax analyzer (`yacc - bison`), which generates syntax tree according to tokens returned by `lex`. Because we are not interested in all grammar rules of the Ada language, but only OpenMP syntax, the use of this tool is pointless. All directives, clauses, variables, etc. can be easily extracted using the "powerful" C function `sscanf` and some simple string handling tricks [10]. Pragmas syntax is trivial, and is automatically checked while matching to the above lexical rules.

While scanning, the file is divided into blocks. Three kinds of blocks are needed to correctly interpret OpenMP directives: *variables declarations*, *parallel regions* (code between pragma begin and end) and *rest of program code*. These program parts are stored in one array, in order that they are encountered and each of them has its identifying number. Directives and clauses are stored in another structure. Each directive has a number of blocks it refers to, incorporated with it. Such organisation of data makes interpretation rather a straightforward process, although we can run into some problems occasionally.

Before we describe the process of interpreting OpenMP directives, let us remind how parallelism in Ada is carried out [2,3,7]. In Ada, tasks are objects. Each task has a unique type, which is specified in an object declaration or allocator (an expression of the form "`new ...`") that causes the creation of the task. Each task type is declared in two separate parts: *a task specification* and *a task body*. The specification has a sequence of entry declarations, which define the communications interface of tasks of that type. The body has the rest of the description of the task type. Over time, tasks proceed through various states. A task is initially inactive; upon activation, and prior to its termination it is either blocked (as part of some task interaction) or ready to run. While ready, a task competes for the available execution resources that it requires to run.

A task type can be regarded as a template from which actual tasks are created. Task objects and types can be declared in any declarative part, including task bodies themselves. For any task type, the specification and body must be declared together in the same unit, with the body usually being placed at the end of the declarative part.

In our case task type specification can be written as [7]:

```
task type Worker_Thread is
   entry Init(no : in natural);
   entry SetVals(...);
   entry RetVals(...);
end Worker_Thread;
```

and task body as:

```
task body Worker_Thread is
   --declaration of local variables
```

```
begin
   accept Init(no : in natural) do
      -- set task identifier
   end Init;

   accept SetVals(...) do
      -- init appropriate local variables
   end SetVals;

   -- execute parallel region operations

   accept RetVals(...) do
      -- return appropriate local variables (results)
   end RetVals;
end Worker_Thread;
```

In addition, task of this type must be organised into an array to ensure multi-thread execution:

```
type TWorker_ThreadPtr is access Worker_Thread;
Worker_Thread_Array0 : array(1..ADAOMP_THREADS_COUNT) of
TWorker_ThreadPtr;
Worker_Thread_Index0 : integer := 1;
```

Now the only thing left is to call threads, which is done in the following manner (for parallel directive):

```
-- allocate array elements
for Worker_Thread_Index0 in 1..ADAOMP_THREADS_COUNT loop
   Worker_Thread_Array0(Worker_Thread_Index0) := new Worker_Thread;
end loop;

-- init thread (give them distinct identifiers)
for Worker_Thread_Index0 in 1..ADAOMP_THREADS_COUNT loop
   Worker_Thread_Array0(Worker_Thread_Index0).Init(Worker_Thread_Index
0 - 1);
end loop;

-- set, if needed, local variables of the threads
for Worker_Thread_Index0 in 1..ADAOMP_THREADS_COUNT loop
   Worker_Thread_Array0(Worker_Thread_Index0).SetVals(...);
end loop;

-- get, if needed, local variables – results - from threads
for Worker_Thread_Index0 in 1..ADAOMP_THREADS_COUNT loop
   Worker_Thread_Array0(Worker_Thread_Index0).RetVals(...);
end loop;
```

After presenting ideology of achieving parallelism in Ada, let us return to parser analysis. The interpretation process can be shortly described as follows:

for each OpenMP directive **do**
begin
Locate declaration block (**DB**) corresponding to the program block OpenMP directive is in;
Add specification of appropriate task type to **DB**;
Add declaration of appropriate task body to **DB** and move blocks (**PB**), current directive is referring to, to declarative part of task body (between SetVals and RetVals entries);
Add declaration of array of task pointers to **DB**;
Insert threads call section into place **PB** was located;
end;

This ends Ada OpenMP file parsing. The only thing left to do is to write the result to the appropriate file and compile it with gnatmake to check for errors.

When called AdaOMP compiles given Ada program file (.adb) with gnatmake. If there are no errors, the file is parsed by OpenMP compiler and the results overwrite a given file (however, the old file is written under a different name – „.tmp" suffix is added to the file name). The last step is running gnatmake again to check if parsing was correct. AdaOMP calling conversion is as follows:

adaomp [argument] [ada_file] [arguments_for_gnatmake]

where argument is one of:
-o – [ada_file] is not altered and the results are written on the standard output
-h – displays help.

4. Results

As an example of using Ada OpenMP parser, let us discuss the Ada version of an OpenMP parallel program for solving the Helholtz equation [11]. The example will also show how nested directives are interpreted. The following fragment of the sequential program includes OpenMP pragmas (some obvious code fragments are replaced with ". . ."):

```
---------------------------------------------------------------------------------
-- Solves poisson equation on rectangular grid assuming :
-- (1) Uniform discretization in each direction, and
-- (2) Dirichlet boundary conditions
--
-- Jacobi method is used
--
```

```
-- Input : n,m    Number of grid points in the X/Y directions
--         dx,dy  Grid spacing in the X/Y directions
--         alpha  Helmholtz eqn. coefficient
--         omega  Relaxation factor
--         f(n,m) Right hand side function
--         u(n,m) Dependent variable/Solution
--         tol    Tolerance for iterative solver
--         maxit  Maximum number of iterations
--
-- Output : u(n,m) - Solution
--------------------------------------------------------------------------------
...
k := 1;
   while (k <= maxit) and (error > tol) loop
      error := 0.0;
      -- Copy new solution into old
      pragma omp; -- parallel private(i)
        pragma omp; -- parallel for private(k)
        for j in 1..m loop
          for i in 1..n loop
            uold(i,j) := u(i,j);
            error := error;
          end loop;
        end loop;
        pragma omp; -- end
        -- Compute stencil, residual, & update
        pragma omp; -- parallel for private(resid) reduction(+ : error)
        for j in 2..m - 1 loop
          for i in 2..n - 1 loop
            -- Evaluate residual
            resid := (ax * (uold(i - 1, j) + uold(i + 1, j))
            + ay*(uold(i, j - 1) + uold(i, j + 1))
            + b * uold(i, j) - f(i, j)) / b;
            -- Update solution
            u(i,j) := uold(i,j) - omega * resid;
            -- Accumulate residual error
            error := error + resid * resid;
          end loop;
        end loop;
      pragma omp; -- end
   pragma omp; -- end
   k := k + 1;
   error := sqrt(error) / float(n * m);
```

Main region

Region I

Region II

```
end loop; -- End iteration loop
...
```

After parsing the above code with AdaOMP we will get its parallel version:

```
...
task type Worker_Thread is -- Computations of Main Region
   entry Init(no : in natural);
   entry SetVals(iLocal : in integer);
   entry RetVals(iLocal : out integer);
end Worker_Thread;
task body Worker_Thread is
   ... -- local variables declaration

   -- Computations of Region I
   task type Worker_Thread1 is
      entry Init(...);
      entry SetVals(...);
      entry RetVals(...);
   end Worker_Thread1;
   task body Worker_Thread1 is
      ... -- local variables declaration
   begin
      accept Init(...) do ...
      accept SetVals(...) do ...
      ... -- execute operations from Region I
      accept RetVals(...) do ...
   end Worker_Thread1;
   ... -- Worker_Thread1 array and local variables declaration

   -- Computations of Region II
   task type Worker_Thread2 is
      entry Init(...);
      entry SetVals(...);
      entry RetVals(...);
   end Worker_Thread2;
   task body Worker_Thread2 is
      ... -- local variables declaration
   begin
      accept Init(no : in natural) do ...
      accept SetVals(...) do ...
      ... -- execute operations from Region II
      accept RetVals(...) do ...
```

```
    end Worker_Thread2;
    ... -- Worker_Thread2 array and local variables declaration

begin -- Worker_Thread
    accept Init(...) do ...
    accept SetVals(...) do ...
    ... -- create, init, lunch and get results from parallel processes:
    ... -- Worker_Thread1 and Worker_Thread2 executing operations
    ... -- from Region I and Region II respectively
    accept RetVals(...) do ...
end Worker_Thread;
... -- Worker_Thread array and local variables declaration


... -- Main program
k := 1;
while (k <= maxit) and (error > tol) loop
    error := 0.0;
    ... -- create, init, lunch and get results from parallel process
    ... -- Worker_Thread executing operations from Main Region
    k := k + 1;
    error := sqrt(error)/float(n*m);
end loop; -- End iteration loop
...
```

Execution of both programs will produce the same results. In the second case, multiple thread will be used to execute the parallel region code. Writing such a parallel program in Ada [12] would be time consuming and difficult, whereas the use of OpenMP simplifies this process. In addition, we do not need to rewrite our old programs to execute them in parallel, we simply put OpenMP pragmas in appropriate places in the source code and OpenMP parser makes all necessary transformations.

5. Future work

Due to the fact that the recognized directives and clauses as well as library functions provide means to face most of programmers' expectations we want to postpone adding new directives and clauses. Instead we would like to focus our attention on standardizing approach to parallel and distributed programming with Ada [6,13].

References

[1] Chandra R., Dagum L., Kohr D., Maydan D., McDonald J., Menon R., *Parallel Programming in OpenMP*, Morgan Kaufmann Publishers, San Francisco, (2001).
[2] Kok J., *Parallel programming with Ada*, Int. J. of Supercomp. Applic., 2 (1988) 100.
[3] *Ada 95 Reference Manual*, Intermetrics, (1995).
[4] Aho A.V., Sethi R., Ulman J.D. *Kompilatory. Reguły, metody i narzędzia*, WNT, (2002), in Polish.
[5] Paprzycki M., Zalewski J., *Parallel computing in Ada: An overview and critique*, Ada Letters, 17 (1997) 62.
[6] Stpiczyński, P., *Ada as a language for programming clusters of SMPs*, Annales UMCS Informatica, 1 (2003) 73.
[7] Kartaszyński R., Stpiczyński P., *OpenMP parser for Ada*, Annales UMCS Informatica, 2 (2004) 125.
[8] Niemann T., *A Compact Guide to Lex & Yacc*, epaperpress.com
[9] Lex online manual: http://dinosaur.compilertools.net/lex/index.html
[10] Kernighan B.W., Ritchie D.M., *The C Programming Language*" Second Edition.
[11] http://www.openmp.org/drupal/samples/jacobi.html
[12] Huzar Z., Fryźlewicz Z., Dubielewicz I., Hnatkowska B., Waniczek J., *Ada 95*, Helion, (1998), in Polish.
[13] Paprzycki M., Zalewski J., *Ada in distributed systems: An overview*, Ada Letters, 17(1997) 55.