OpenMP parser for Ada

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 OpenMP parser for Ada, which is meant to make parallel programming in Ada simpler. We present different approaches to parallel programming and advantages of OpenMP solution. Next, implemented directives and clauses are described, and appropriated examples given. General look at parsing algorithm is presented in another part of this article. Finally, we present the source code of sequential program written in OpenMP Ada transformed into the parallel program by presented parser.

1. Introduction

Each parallel programming language must satisfy three basics aspects of parallel computing, namely it should specify parallel execution, communication between multiple threads, and synchronization between them. In most cases, it is achieved by extension to existing sequential languages. Different programming languages have presented different look at this subject. Some of them provide constructs expressing parallel communication, execution, etc. Others, like PVM, provide library routines.

OpenMP standard is based on directives, which may be embedded within sequential programs written in C, C++ or Fortran. Main advantage of this solution is that the same program may be executed on single – and multiprocessor platforms. In the first case OpenMP directives will be treated as comments and ignored by the language translator. In the second case when we have a compiler capable of understanding OpenMP directives, sequential program will be transformed into parallel program. Another benefit of this approach is easiness of creating parallel programs by simple adding parallel directives. For example, syntax of OpenMP directive for C/C++ is:

```
#pragma omp directive-name [clause] ...
```

---

[*] *E-mail address*: rkartaszynski@liza.umcs.lublin.pl
[**] *E-mail address*: przem@hektor.umcs.lublin.pl

If the compiler used to compile code with, for example `parallel` directive, supports OpenMP, code block after this directive will be "cloned" on n processes and executed n times in parallel. n is a number of available processors.

Why Ada? Because it provides powerful means for developing concurrent programs (tasks) or programs for parallel shared memory computers. However, those features are complicated in use, unlike to easy extensions to C or Fortran. It was the main reason why Ada did not become a popular parallel programming language [1].

The aim of this project is to write parser, which would convert the sequential program, written in Ada with embedded OpenMP directives, into a parallel program. This will greatly simplify parallel programming in this language. The idea of writing this program was put forward in [2]. It should be pointed out that there are some other extensions to Ada which also simplify parallel programming [1,3]. However, our approach is compliant with OpenMP – *de facto* standard for shared memory parallel programming, thus many shared memory parallel algorithms can be easily ported to Ada. Another benefit of our approach is that OpenMP-Ada programs are still correct "pure Ada 95" programs. One can compile such a program using the GNU Ada 95 compiler, which will produce a code with no syntactical errors. Finally, OpenMP Ada allows to simplify distributed memory programming using remote subprograms calls instead of complicated message passing [2].

2. OpenMP directives

According to the idea presented in [2], OpenMP directive syntax for Ada is as follows:

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

We have implemented the following directives:

```
- parallel
- parallel for
```

which are used in most cases when programming with OpenMP.
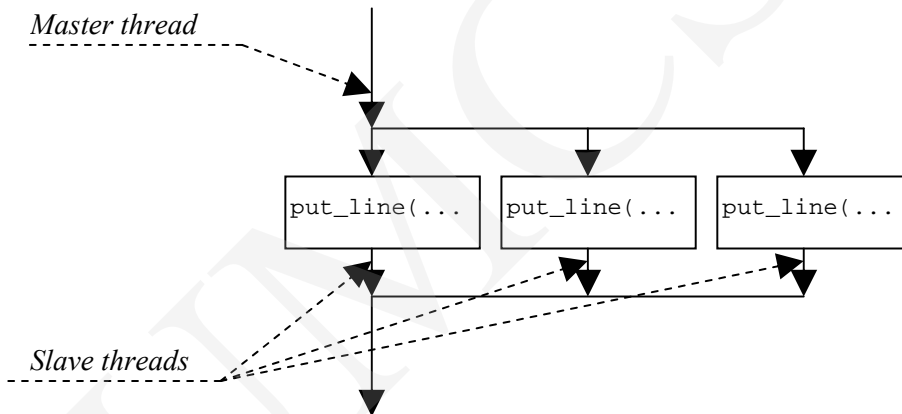
2.1. `parallel` directive

Opening and closing directive specifies a block of code to be executed by multiple threads. Precisely its copy will be executed by a group of threads. In the following example we will illustrate this behavior:

Example 1.

```
...
pragma omp; -- parallel
```

```
put_line("Hello world");
pragma omp; -- end
...
```

By default OpenMP program is run sequentially as a single thread. When beginning pragma is encountered, new slave threads are created and then called to print "`Hello world`" on standard output. The number of threads called depends on the number of processors available. After each thread finished executing, there is an implicit barrier. When all called threads have ended, the rest of main program is executed.

The following clauses are available for the `parallel` directive:

- **private(**list**)** - provides a list of variables, specifies that each thread has its private copy. They ought to be active in the thread block.
- **shared(**list**)** - list of variables to be shared between all threads
- **default(private** | **shared** | **none)** - switches data-sharing attributes of variables, `shared` is default. It may be changed to `private` or `none`. When it is `none` each variable within the parallel region must be named in `shared` or `private` clauses.
- **reduction(**operation **:** list**)** - provides a reduction operator and a list of variables

2.2. `parallel for` directive

In this case `for` loop is enclosed in directives. The iterations are divided among available threads, and each of them makes only part of all iterations. In the example below, if we have, for example, five processes, each of them will write "`Hello world`" twice, which will give total amount of ten lines written.

Example 2.

```
...
pragma omp; -- parallel for
for i in 1..10 loop
put_line("Hello world");
end loop;
pragma omp; -- end
...
```

The same clauses as for `parallel` directive are supported. Very important is `reduction` clause, used to identify variables used in reduction operations within the parallel region.

3. Implementation

Parser was written in C. We launch it with one or more arguments. The first is the path to `.adb` file containing program in Ada. The other are the options for `gnatmake`. After start, parser is trying to compile file with `gnatmake`. If no error occurred, main OpenMP parser is launched. Results are written to temporary file. When all necessary operations have been completed, adding suffix ".tmp" changes the source file name. Next `gnatmake` is called second time to compile new file.

Now we will concentrate on algorithm.

Program reads source file. All declarations of variables and their types, as well as positions in file, of beginning and ending OpenMP directives, are stored in arrays. Variables types will be necessary when writing declaration of local variables of task type. OpenMP parallel regions are not stored directly in memory, but are read from the source if needed. Remembered positions in file are used when the resulting file is being prepared.

Parallel program capacity is obtained by the use of task types. Task type executing code of directive is declared within the declaration section of structured block where this directive is placed. Name of type is defined by the program constant. When multiple directives are embedded within the same structured block, names of thread types are modified to guarantee their uniqueness. The first type name is not changed, but to next, appropriate numbers are added: "1" for second, "2" for third etc.. In a general type declaration may be written as follows:

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

Entry `Init` is used to initialize task variable and to give every instance of that type unique number. Entry `GetVals` and `RetVals` may not have parameters, in most trivial cases. We face such a situation, when there is no need of initializing local variables, or retrieving the results from threads. Simultaneously, need of calling `RetVals`, guarantees existence of implicit barrier. `GetVals` parameters are variables from `private` and `reduction` clauses, if `parallel for` directive is used, also bound, in which loop iterations are to be executed, must be provided. `RetVals` entry parameters are variables from `reduction` clause, returning parallel region execution results in the structured block, from which thread was called. Along with task types, arrays of pointers to those types are declared.

Entries `GetVals` and `RetVals` are to assign entries parameters to local variables. If the parameter list of these entries is empty they are to perform `null` operation.

For each parallel region, parser identifies used directive and its clauses. On that basis, array of variables, which have to be declared, is created. When writing the result to file those variables are inserted into appropriate places.

Function serving directives also create text of thread call. It will replace code between OpenMP directives. Creating and calling threads is obtained by the following code:

```
for Indexworker in 1..10 loop
    ArrWorker(Indexworker) := new worker;
end loop;
for Indexworker in 1..10 loop
    ArrWorker(Indexworker).Init(Indexworker);
end loop;
```

Rest of call depends on directive used. When it is `parallel` we have:

```
for Indexworker in 1..10 loop
    ArrWorker(Indexworker).GetVals(...);
end loop;
for Indexworker in 1..10 loop
    ArrWorker(Indexworker).RetVals(...);
end loop;
```

Names of `RetVals` parameters are variable identifiers from the `reduction` clause with the added „Loc" suffix.

When `parallel for` directive is considered, there is need of distribution of all `for` loop iterations between available processes. Following the code one computes bound for each thread:

```
LowIndex := (n - 1 + 1) / 10;
HighIndex := (n - 1 + 1) mod 10;
NumOfProcAlSt := 1;
```

Number 10 stands for the number of processes that we wish to call. (n – 1 + 1) is the number of all iterations of paralleled loop, which is ending index (here n) minus starting index (here 1) plus 1. Next appropriate parameters are sent to threads:

```
    for Indexworker in 1..HighIndex loop
        ArrWorker(Indexworker).GetVals(NumOfProcAlSt,
NumOfProcAlSt + LowIndex, ...);
        NumOfProcAlSt := NumOfProcAlSt + 1 + LowIndex;
    end loop;
    for Indexworker in (HighIndex + 1)..10 loop
        ArrWorker(Indexworker).GetVals(NumOfProcAlSt,
NumOfProcAlSt + LowIndex - 1, ...);
        NumOfProcAlSt := NumOfProcAlSt + LowIndex;
    end loop;
```

First two of GetVals parameters are bounds in which thread loop iterations are to be performed.

Source file code parts, as well as transformed code are written into the result file. This process is safe and produces correct results when one, or more OpenMP directives, are considered.

Error handling ensures immunity to most file format and system errors.

4. Results

We present below an example showing how a sequential program, with OpenMP directives, is parsed into parallel program.

We want to calculate the Euler constant described by the formula:

$$g_n = \left(\sum_{k=1}^{n} \frac{1}{k} - \ln n \right).$$

It is not difficult to write the following program in OpenMP Ada to handle needed operations:

```
with Text_io;              use Text_io;
with ada.integer_text_io;  use ada.integer_text_io;
with ada.float_text_io;    use ada.float_text_io;
with ada.numerics.aux;     use ada.numerics.aux;

procedure gamma is
    n : integer;
    i : integer;
    sum : float;

begin
```

```
    sum := 0.0;
    put_line("n: ");
    get(n);
    pragma omp; -- parallel for reduction(+ : sum)
     for i in 1..n loop
     sum := sum + 1.0/float(i);
     end loop;
    pragma omp; -- end

    sum := sum - float(log(double(n)));
    put("gamma = ");
    put(sum);
end gamma;
```

As we can see `for` loop is between `parallel for` directive. When we use OpenMP parser, code of our program will be modified as follows:

```
with Text_io;use Text_io;
with ada.integer_text_io; use ada.integer_text_io;
with ada.float_text_io;   use ada.float_text_io;
with ada.numerics.aux;    use ada.numerics.aux;

procedure gamma is
    n : integer;
    i : integer;
    sum : float;

    task type worker is
        entry Init(no : in natural);
        entry GetVals(StartPointLoc : in integer;
        StopPointLoc : in integer; sumLoc : in float);
        entry RetVals(sumLoc : out float);
    end worker;

    task body worker is
        sum : float;
        i : integer;
        number : integer;
        StartPoint : integer;
        StopPoint : integer;
    begin
        accept Init(no : in natural) do
            number := no;
        end Init;
        accept GetVals(StartPointLoc : in integer;
        StopPointLoc : in integer; sumLoc : in float) do
            StartPoint := StartPointLoc ;
```

```
                StopPoint := StopPointLoc;
                sum := sumLoc;
         end GetVals;

         for i in StartPoint..StopPoint loop
                sum := sum + 1.0/float(i);
         end loop;

         accept RetVals(sumLoc : out float) do
                sumLoc := sum;
         end RetVals;
    end worker;

    type TworkerPtr is access worker;
    Arrworker : array(1..10) of TworkerPtr;
    Indexworker : integer := 1;
    sumLoc : float;
    LowIndex : integer;
    HighIndex : integer;
    NumOfProcAlSt : integer;

begin
    sum := 0.0;
    put_line("n: ");
    get(n);

    pragma omp; -- parallel for reduction(+ : sum)
    for Indexworker in 1..10 loop
         ArrWorker(Indexworker) := new worker;
    end loop;
    for Indexworker in 1..10 loop
         ArrWorker(Indexworker).Init(Indexworker);
    end loop;

    LowIndex := (n - 1 + 1) / 10;
    HighIndex := (n - 1 + 1) mod 10;
    NumOfProcAlSt := 1;

    for Indexworker in 1..HighIndex loop
         ArrWorker(Indexworker).GetVals(NumOfProcAlSt,
         NumOfProcAlSt + LowIndex, sum);
         NumOfProcAlSt := NumOfProcAlSt + 1 + LowIndex;
    end loop;
    for Indexworker in (HighIndex + 1)..10 loop
         ArrWorker(Indexworker).GetVals(NumOfProcAlSt,
         NumOfProcAlSt + LowIndex - 1, sum);
         NumOfProcAlSt := NumOfProcAlSt + LowIndex;
    end loop;
```

```
    for Indexworker in 1..10 loop
        ArrWorker(Indexworker).RetVals(sumLoc);
        sum := sum + sumLoc;
    end loop;
    pragma omp; -- end

    sum := sum - float(log(double(n)));
    put("gamma = ");
    put(sum);
end gamma;
```

Execution of both programs will produce the same results. In the second case, ten parallel threads make loop iterations.

As we can see writing this program, solution to a trivial problem, in "pure" Ada would be a time consuming process. When we use OpenMP parser we save a lot of time, and what is important, our program can be executed sequentially or in parallel. 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.

We obtained a programming tool, which simplifies parallel programming with Ada. It should be pointed out that our tool works fast and no significant overheads of the transformation of OpenMP Ada programs into pure Ada take place.

5. Future work

We are planning to implement additional OpenMP directives and routines. We are also planning to add support for distributed shared memory. Using Ada for standardizing approach to parallel and distributed programming is very interesting [2] and will be part of our future research.

References

[1] Paprzycki M., Zalewski J., *Parallel computing in Ada: An overview and critique*, Ada Letters, 17(1997) 62.
[2] Stpiczyński, P., *Ada as a language for programming clusters of SMPs*, Annales UMCS Informatica, 1 (2003) 73.
[3] Paprzycki M., Zalewski J., *Ada in distributed systems: An overview*, Ada Letters, 17 (1997) 55.