Pobrane z czasopisma Annales AI- Informatica **http://ai.annales.umcs.pl**
Data: 06/11/2024 00:55:08

Annales UMCS Informatica AI 1 (2003) 73-79

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI

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

Ada as a language for programming clusters of SMPs

Przemysław Stpiczyński*

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

Abstract

This paper presents a new idea of developing parallel programs for clusters of SMP nodes using the Ada programming language. We show how to implement OpenMP in Ada and simplify programming of distributed memory applications using remote subprograms calls instead of complicated message passing.

1. Introduction

While the largest computers in the world are still built for the highest performance, they still cost tens of millions of dollars. Clusters made high performance parallel computing available to institutions with much smaller budgets. Now it is possible to use several workstations (for example PC's) connected by Fast Ethernet, Gigabit or Myrinet, as a single powerful computational resource. The Parallel Virtual Machine (PVM) system helps to develop parallel programs for such distributed memory architecture [1]. In 1993 Message Passing Interface (MPI) has been developed by researchers from Argonne National Laboratory and became a de-facto standard for message passing parallel computing. MPI provides a large set of communication subroutines including point-to-point communication, broadcasting and collective communication [2].

Often, nodes of such clusters are SMP (symmetric multiprocessing) machines, which means that their architecture is based on tightly-coupled identical processors with access to a shared memory. The parallel nature of such machines is hidden from the user: an operating system is responsible for allocation of processor time to the programs when scheduling them to run. Moreover, such a kind of parallel architectures is easy to program. Until quite recently each vendor has provided its own set of commands to support writing parallel programs. All these approaches were quite similar with directives for

* *E-mail address:* przem@hektor.umcs.lublin.pl

PDF created with FinePrint pdfFactory Pro trial version www.pdffactory.com

Pobrane z czasopisma Annales AI- Informatica **http://ai.annales.umcs.pl**
Data: 06/11/2024 00:55:08

managing parallel code execution; e.g. loop parallelizing directives, locks, barriers and other synchronization primitives inserted into codes written in Fortran or C. Recently OpenMP [3] emerged as a standard for code parallelization for shared memory parallel computers.

Unfortunately, people who want to utilize fully clusters of SMPs have to combine two different standards: OpenMP for shared memory and MPI for distributed memory, which makes programs complicated. In this paper we show how to simplify developing programs for the clusters of SMPs using the Ada 95 programming language [4] with the Distributed System Annex [4, 5] and GLADE [6].

2. Ada and OpenMP

"The language Ada was primarily designed for the production of large portions of readable, modular, portable, and maintainable software for real-time applications" [7], so why not use it for developing parallel programs? The answer is quite simple: when Ada was being designed [4, 8], the parallel computing was not popular and this explains why Ada does not provide constructs which would simplify parallel programming [9, 10]. However, Ada provides a very powerful mechanism for concurrent programming (tasks and rendezvous for synchronization) which can also be used for developing programs for parallel shared memory computers. Unfortunately, the use of tasks is rather complicated in comparison with simple extensions to Fortran and C provided by vendors producing parallel computers.

OpenMP provides support for three basic aspects of parallel computing: *specification of parallel execution, communicating between multiple threads, expressing synchronization between threads*. In Fortran, OpenMP directives satisfy the following format:

```
!$omp directive name   optional clauses
```

Such an approach allows to write the same code for both single-processor and multiprocessor platforms. Simply, compilers which do not support OpenMP directives or that are working in a single-processor mode treat them as comments.

Example 1 *Let us consider the following code for numerical integration:*

$$\int_a^b f(x)dx \approx\sim h\left(\frac{f(x_o)}{2}+f(x_1)+...+f(x_{n-1})+\frac{f(x_n)}{2}\right),$$

where $h=\frac{b-a}{n}$ *and* $x_i=a+ih,\ i=0,...,n$. *An OpenMP code will look like this:*

PDF created with FinePrint pdfFactory Pro trial version www.pdffactory.com

Pobrane z czasopisma Annales AI- Informatica **http://ai.annales.umcs.pl**
Data: 06/11/2024 00:55:08

```
        h=(b-a)/n
!$omp parallel do private(x) reduction(+:sum)
        do j=1,n-1
          x=a+j*h
          sum=sum+f(x)
        end do
        sum=h*(sum+0.5*(f(a)+f(b)))
```

In this example the `reduction(+:sum)` clause is used. It instructs the compiler that the variable `sum` is the target of a sum reduction operation. The OpenMP uses the fork-join model of parallel execution. A program starts execution as a single process, called the *master thread* of execution, and executes sequentially until the first parallel construct is encountered. Then the master thread spawns a specified number of threads and becomes a "master" of the team. All statements enclosed by the parallel construct are executed in parallel by each member of the team.

It is clear that OpenMP standard can be easily adopted to Ada and it should support the same functionality as OpenMP in Fortran and C. Thus, our proposal is to use the following format for OpenMP directives in Ada:

```
pragma omp; -- directive name   optional clauses
```

Now let us consider a few examples of *OpenMP-Ada* constructs. Note that other "non loop-based" OpenMP constructs like *sections* and explicit synchronization can be easily translated into Ada 95.

Example 2 *The* `parallel` *constructs instructs a compiler to create a parallel region to execute lines between* `begin` *and* `end` in parallel:

```
pragma omp; -- parallel
begin
  -- lines of code to be executed in parallel
  -- .......
end;
```

Example 3 *The* `parallel for` *construct instructs a compiler to parallelize the execution of a* `for` *loop*:

```
h:=(b-a)/float(n);
pragma omp; -- parallel for private(x) reduction(+:sum)
for j in 1..n-1 do loop
  x:=a+float(j)*h;
  sum:=sum+f(x);
```

PDF created with FinePrint pdfFactory Pro trial version www.pdffactory.com

Pobrane z czasopisma Annales AI- Informatica **http://ai.annales.umcs.pl**
Data: 06/11/2024 00:55:08

```
  end loop;
  sum:=h*(sum+0.5*(f(a)+f(b)));
```

Analogously we can adopt all OpenMP constructs. It should be pointed out that our proposal for *OpenMP-Ada* (just like official OpenMP for Fortran and C) defines only a "potential" parallelism. The above examples can be complied with a standard Ada compiler and it produces no errors. We only get warnings that the pragma *omp* is unknown. Now let us observe that each *OpenMP-Ada* construct can be translated into well known pure Ada constructs which support concurrent programming (just like tasks synchronized by rendezvous).

Example 4 *The parallel region construct from Example 2 should be translated into the following code:*

```
declare
  task type ompT1 is      -- define a local task type
   entry Init(nr:in integer); -- which accepts rendezvous
  end ompT1;
  task body ompT1 is -- body of the task
    myid : integer;
  begin
    accept Init(nr:in integer) do -- get the number
      myid:=nr;
    end Init;
    -- lines of code to be executed in parallel
    -- .......
  end ompT1;
  type refompT1 is access ompT1;
  tref : refompT1;
begin
  for i in 0..AdaOpenMP.NPROCS-1 loop
    tref:=new ompT1;   -- create a new task
    tref.all.Init(i);  -- give the number to the task
  end loop;
end;
```

In the above example we define a local task type and use it to create a number of tasks (in the *for* loop). Each task gets its unique number and then starts to execute the code intended to be executed in parallel. It is clear that the *OpenMP-Ada* construct is much simpler.

3. Distributed computing with DSA

As it was mentioned above, if we want to develop programs for clusters we have a choice: PVM or MPI. Both of the systems are based on the message passing. They are rather complicated and cannot be used together with Ada. On

PDF created with FinePrint pdfFactory Pro trial version www.pdffactory.com

Pobrane z czasopisma Annales AI- Informatica **http://ai.annales.umcs.pl**
Data: 06/11/2024 00:55:08

the other hand, Ada 95 offers much simpler way of writing distributed programs called *remote subprogram calls* defined in the Distributed System Annex (DSA) which is a part of the language definition [4]. A typical distributed application in Ada consists of several partitions executed on remote hosts. Each partition comprises a number of Ada packages categorized with simple Ada constructs. Such packages define "services" provided by remote hosts [5, 6]. From a programmer's point of view, calls to "remote subprograms" are quite similar to simple subprogram calls. It should be pointed out that DSA provides several interesting mechanisms just like distributed shared memory [6, 11].

Example 5 *The categorized package* `Host_Pkg` *provides the function* `Integral`. *In the evaluation part of the package body one calls the routine Register from the package* `Main_Pkg`} which is a part of the main partition.

```
with funct;
package Host_Pkg is
   pragma Remote_Types;
   type Host_handler is tagged limited private;
   type RefToHandler is access all Host_handler'Class;
   function Integral(h:Host_handler;a,b:in float;
     n: in integer; f: funct.RefToFunct) return Float;
private
   type Host_handler is tagged limited
     record
       myid:integer;
     end record;
end Host_Pkg;
```

The main partition contains the following package:

```
with Host_Pkg;
package Main_Pkg is
   pragma Remote_Call_Interface;
   procedure Register(h:access Host_Pkg.RefToHandler);
   ......... -- other serveces
end Main_Pkg;
```

The routine `Integral` *can be called as follows:*

```
with Host_Pkg; with Main_Pkg;
procedure Simple is
begin
   .......
   value:= Host_Pkg.Integral(handler,a,b,n,f);
```

PDF created with FinePrint pdfFactory Pro trial version www.pdffactory.com

Pobrane z czasopisma Annales AI- Informatica **http://ai.annales.umcs.pl**
Data: 06/11/2024 00:55:08

```
   .......
end Simple;
```

In the above example, a distributed application starts with the main partition which contains the package `Main_Pkg`. Each remote partition "registers" its services to the main partition by providing a remote reference to its local data of the type `Host_handler`. When the main partition calls to the routine `Integral`, the request is dispatched to the partition which contains `handler`, where the routine is called. Then the result goes back to the main partition.

Now let us consider the situation when a distributed application consists of several partitions with the package `Host_Pkg}`. Thus each partition can be responsible for computing the integral on a subinterval. Calls to remote copies of the subprogram `Integral` should be parallelized. Thus we can simply mix the proposed *OpenMP-Ada* with DSA.

Example 6 *In this example we show how combine the proposed* OpenMP-Ada *with the mechanism of remote subprograms calls.*

```
h:=(b-a)/float(AdaOpenMP.NHOSTS);
pragma omp; -- parallel for shared(h)
pragma omp; -- private(xa,xb) reduction(+:sum)
for j in 0..AdaOpenMP.NHOSTS-1 do loop
  xa:=a+float(j)*h;
  xb:=xa+h;
  sum:=sum+Host_Pkg.Integral(handler(j),xa,xb,n,f);
end loop;
```

4. Conclusions and future work

We have presented a new idea of developing parallel programs for clusters of SMP nodes using the Ada programming language. We have shown how to implement OpenMP in the pure Ada and simplify programming of distributed memory application using remote subprogram calls instead of complicated message passing. As the future work, we are planning to write *OpenMP-Ada* compiler using *Aflex* and *Ayacc*, which are Ada versions of well known tools *lex* and *yacc*.

References

[1] Dongarra, J., et~al., *PVM: A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, (1994).
[2] Pacheco, P., *Parallel Programming with MPI*, Morgan Kaufmann, San Francisco, (1996).
[3] Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Menon, R., *Parallel Programming in OpenMP*, Morgan Kaufmann Publishers, San Francisco, (2001).
[4] *Ada 95 Reference Manual*, Intermetrics, (1995).

PDF created with FinePrint pdfFactory Pro trial version www.pdffactory.com

Pobrane z czasopisma Annales AI- Informatica **http://ai.annales.umcs.pl**
Data: 06/11/2024 00:55:08

[5] Pautet, L., Tardieu, S., *What future for the distributed systems annex?*, ACM SIGADA Ada Letters, 19 (1999) 77.
[6] Pautet, L., Tardieu, S., *GLADE User's Guide*, Free Soft Foundation (2001).
[7] Kok, J., *Parallel programming with Ada*, Int. J. of Supercomp. Applic., 2 (1988) 100.
[8] *Ada Reference Manual*, Intermetrics, (1983).
[9] Paprzycki, M., Zalewski, J., *Ada in distributed systems: An overview*, Ada Letters, 17 (1997) 55.
[10] Paprzycki, M., Zalewski, J., *Parallel computing in Ada: An overview and critique*, Ada Letters, 17 (1997) 62.
[11] Kermarrec, Y., Pautet, L., *A distributed shared virtual memory for Ada 83 and Ada 9X applications*, In Engle, Jr., C.B., ed.: Proceedings of the Conference on TRI-Ada, Seattle, WA, USA, ACM Press, (1993) 242.

PDF created with FinePrint pdfFactory Pro trial version www.pdffactory.com