Parallel computing with Ada programming language

Deividas Gineitis

Dominykas Pošiūnas

The history of Ada (1/2)

- The Department of Defense (DoD) study in the early and middle 1970s indicated that enormous saving in software costs (about $24 billion between 1983 and 1999) might be achieved if the DoD used one common programming language for all its applications instead of 450 programming languages and incompatible dialects used by its programmers.
- An international competition was held to design a language based on DoD’s requirements.

The history of Ada (2/2)

- The winning proposal was a programming language, originally developed in the early 1980s by a team led by Dr. Jean Ichbiah in France. With some minor modifications, this language, referred to as **Ada**, was adopted as an American National Standards Institute (ANSI) standard in 1983 (Ada 83).
- Major Ada versions include Ada 83, Ada 95, Ada 05 and Ada 12 (the most recent).
- The name “Ada” is not an acronym. It was chosen in honor of Augusta Ada Lovelace (1815-1852), a mathematician who is sometimes regarded as the world’s first programmer.

Ada major features

- Ada is a structured, statically (and strong) typed, imperative, and multi-paradigm high-level programming language.
- Designed for embedded and real-times systems, focused on making bugs almost non-existent.
- The big five structural elements:
    - Packages - groups, units of compilations
    - Subprograms - procedures, functions - reusable sequences of instructions
    - Generics - arbitrary type packages that meet some requirement
    - Tasks - operations done in parallel
    - Protected objects - coordinate shared data

Common uses

- Avionics (e.g. Airbus A380 flight control and navigation systems).
- Railways (e.g. Paris Metro).
- Space Technology (e.g. European Space Agency satellite control software).
- Military (e.g. Weapon systems, radar, and control systems).
- Banking (e.g. Transaction Processing).

Hello World

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Greet is

begin
   Put_Line ("Hello, World!");

end Greet;
```

Concurrency in Ada

Ada has a built-in concurrency feature that is referred to as **Tasking**.

Tasking consists of two main parts:

1. Tasks
2. Protected objects

Tasks in Ada

- A thread of execution in Ada is called a task, and it is declared using the keyword `task`.
- The implementation of the task is defined in a `task body` block.
- The main application is itself considered a task and is often referred to as the **master task**.
- Tasks can synchronize with the main application (master task) or other tasks via **entries**, but they can also operate independently, processing information concurrently without interaction.

Task declaration

Simple task:

```
task T; --  Simple task declaration
```

```
task T is --  Simple task declaration
   --  declarations of exported identifiers
end T
```

Task type:

```
task type TT; --  Task type declaration
```

```
task type TT is --  Task type declaration
   --  declarations of exported identifiers
end TT;
```

The difference between simple tasks and task types is that task types don't create actual tasks.

Simple task example in Ada (1/2)

- master and T tasks runs concurrently.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Simple_Task is
   task T;

   task body T is
   begin
      Put_Line ("In task T");
   end T;
begin
   Put_Line ("In main");
end Show_Simple_Task;
```

Simple task example in Ada (2/2)

- master, T and T2 tasks runs concurrently.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Simple_Tasks is
   task T;
   task T2;

   task body T is
   begin
      Put_Line ("In task T");
   end T;

   task body T2 is
   begin
      Put_Line ("In task T2");
   end T2;

begin
   Put_Line ("In main");
end Show_Simple_Tasks;
```

Task type example in Ada (1)

- master and T tasks also runs concurrently.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Simple_Task_Type is
   task type TT;

   task body TT is
   begin
      Put_Line ("In task type TT");
   end TT;

   A_Task : TT;
begin
   Put_Line ("In main");
end Show_Simple_Task_Type;
```

Data race condition in Ada

Data race condition - two or more threads or processes access the same shared memory or data simultaneously.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Data_Race is
   Shared_Counter : Integer := 0;

   task T1;

   task body T1 is
   begin
      for I in 1 .. 10 loop
         Shared_Counter := Shared_Counter + 1;
         Put_Line ("Task T1 incremented counter to: " & Integer'Image(Shared_Counter));
      end loop;
   end T1;

begin
   Put_Line ("In main, counter starts at: " & Integer'Image(Shared_Counter));
   delay 1.0;
   Put_Line ("In main, final counter value: " & Integer'Image(Shared_Counter));
end Data_Race;
```

Synchronization

- While master task and its subtasks are executed separately, the master task does not terminate until all of its subtasks have finished executing.
- This provides simple synchronization between master task and subtasks.
- The master task will wait for tasks in packages to execute before terminating.

Synchronization example (1/2)

- All tasks will run when main procedure starts (no start).
- The main terminates only if all tasks terminate (no join).

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Simple_Sync is
   task T;
   task body T is
   begin
      for I in 1 .. 10 loop
         Put_Line ("hello");
      end loop;
   end T;
begin
   null;
end Show_Simple_Sync;
```

Synchronization example (2/2)

```
package Simple_Task is -- Package specification
   task T;
end Simple_Task;
```

```
with Ada.Text_IO; use Ada.Text_IO; -- Package body implementation

package body Sub_Task is
   task body T is
   begin
      for I in 1 .. 10 loop
         Put_Line ("hello");
      end loop;
   end T;
end Sub_Task;
```

```
with Sub_Task;

procedure Main is
begin
   null;
end Main;
```

Custom synchronization 'entry-accept' (1/3)

- Task may export something for other task to use (to call).
- This can be done with custom synchronization points using the keyword `entry`.

```
task T is
   entry Start;
end T;
```

Custom synchronization 'entry-accept' (2/3)

- For each `entry` point there is a corresponding `accept` statement.
- In the task body, you specify where the task will accept entry calls by using the keyword `accept`.
- The accept block can be reffered as the ***rendezvous*** section.

```
task body T is
begin
   accept Start;
   --  This is rendezvous section
   Put_Line ("In T");
end T;
```

Custom synchronization 'entry-accept' (3/3)

Tasks run independently until either:

- An accept statement.
  Waits for someone to call this entry, then proceeds to the *rendezvous* section.
  After this, both tasks execute their ways.
- An entry call.
  Waits for corresponding task reaching its accept statement, then proceeds to the rendezvous section. After this, both tasks execute their ways.

This is synchronous communication.

'entry-accept' synchronization example (1/2)

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Rendezvous is
   task T is
      entry Start;
   end T;

   task body T is
   begin
      accept Start;
      Put_Line ("In T");
   end T;

begin
   Put_Line ("In Main");
   T.Start;
end Show_Rendezvous;
```

'entry-accept' synchronization example (2/2)

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Float_Text_IO; use Ada.Float_Text_IO;

procedure Temperature_Converter is
   Celsius_Temperature : Float;
   Fahrenheit_Temperature : Float;

   type Temperature_Array is array (1 .. 5) of Float;
   Fixed_Temperatures : Temperature_Array := (0.0, 20.0, 37.5, 15.0, 30.0);

   task Producer is
      entry Produce (Temp : out Float);
   end Producer;

   task Consumer is
      entry Consume (Temp : Float);
   end Consumer;

   task body Producer is
   begin
      for I in Fixed_Temperatures'Range loop
         accept Produce (Temp : out Float) do
            Celsius_Temperature := Fixed_Temperatures (I);
            Put ("Produced Temperature: ");
            Put (Celsius_Temperature, Fore => 1, Aft => 1, Exp => 0);
            Put_Line (" °C");
            Temp := Celsius_Temperature;
         end Produce;
      end loop;
   end Producer;

   task body Consumer is
   begin
      for I in Fixed_Temperatures'Range loop
         accept Consume (Temp : Float) do
            Fahrenheit_Temperature := (Temp * 9.0 / 5.0) + 32.0;
            Put ("Consumed Temperature: ");
            Put (Temp, Fore => 1, Aft => 1, Exp => 0);
            Put (" °C -> ");
            Put (Fahrenheit_Temperature, Fore => 1, Aft => 1, Exp => 0);
            Put_Line (" °F");
         end Consume;
      end loop;
   end Consumer;

begin
   for I in Fixed_Temperatures'Range loop
      declare
         Temp : Float;
      begin
         Producer.Produce (Temp);
         Consumer.Consume (Temp);
      end;
   end loop;

   Put_Line ("Temperature conversion completed.");
end Temperature_Converter;
```

Protected objects

- In Ada, protected objects are a concurrency control mechanism designed to safely encapsulate and manage shared data.
- Protected objects encapsulate data in their private part. Data is shared among tasks but can only be accessed via protected operations.
- Ada ensures that when a task is executing a protected operation, no other task can interfere, making it thread-safe by default.
- Protected procedures and functions are similar to getters and setters in object-oriented programming.

Protected object example (1/2)

```ada
protected Obj is
    procedure Set(Value : Integer);
    function Get return Integer;
private
    Local : Integer;
end Obj;

protected body Obj is
    procedure Set(Value : Integer) is
    begin
        Local := Value;
    end Set;

    function Get return Integer is
    begin
        return Local;
    end Get;
end Obj;
```

Protected object example (2/2)

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Protected_Object is

   protected Obj is
      procedure Set (V : Integer);
      entry Get (V : out Integer);
   private
      Local  : Integer;
      Is_Set : Boolean := False;
   end Obj;

   protected body Obj is
      procedure Set (V : Integer) is
      begin
         Local := V;
         Is_Set := True;
      end Set;

      entry Get (V : out Integer)
        when Is_Set is
      begin
         V := Local;
         Is_Set := False;
      end Get;
   end Obj;

   N : Integer := 0;

   task T;
   task body T is
   begin
      Put_Line ("Task T will delay for 4 seconds...");
      delay 4.0;
      Put_Line ("Task T will set Obj...");
      Obj.Set (5);
      Put_Line ("Task T has just set Obj...");
   end T;
begin
   Put_Line ("Main application will get Obj...");
   Obj.Get (N);
   Put_Line ("Main application has retrieved Obj...");
   Put_Line ("Number is: " & Integer'Image (N));

end Protected_Object;
```

Terminate and delay

- The `terminate` statement terminates the task
  that executes this terminate statement.
- The `delay` statement suspends the task for at least seconds provided.
- The `delay` statement might introduce time drift. In those cases `delay until` statement, which accepts a precise time for the end of the delay.

The select statement (1/3)

The `select` statement in Ada is a control structure that manages communication between tasks, handling concurrency and synchronization.

It allows a task to wait for multiple possible communications and choose between them based on availability.

The select statement (2/3)

```
select
   -- select_alternative
or
   -- select_alternative
or
   -- select_alternative
else
   -- sequence_of_statements
end select
```

- `or` and `else` blocks are optional in the `select` statement.

The select statement (3/3)

- Each *select_alternative* may be an `accept`, a `delay` followed by some other statements, or a `terminate`.
- A *select_alternative* shall contain at least one `accept`.
- In addition, *select_alternative* can contain (1) at most one `terminate`, (2) one or more `delay`, or (3) an `else`. These possibilites are mutually exclusive (e.g. if you use `delay` you should not use `terminate` or `else`).
- If several `accept` blocks are available, one of them is selected arbitrarily.
- The `delay` is selected when its expiration time is reached if no other `accept` or `delay` can be selected prior to the expiration time. The `else` part is selected and its sequence of statements are executed if no `accept` can immediately be selected.

The main forms of the select statement

There are multiple commonly used forms of the select statement:

1. Selective accept.
2. Timed entry call.
3. Asynchronous transfer of control.
4. Terminating select statement.

Selective accept

Allows a task to accept one of several possible entries.

```
select
   accept Request1 do
      -- Handle Request1
   end;
or
   accept Request2 do
      -- Handle Request2
   end;
end select;
```

Timed entry call

Allows a task to wait for an entry call but only for a specified amount of time.

```
select
   accept Some_Entry do
      -- Handle entry call
   end;
or
   delay 5.0;
   -- Handle timeout case
end select;
```

Asynchronous transfer of control

Allows a task to perform an ongoing activity and react if a specified event occurs.

```
select
   delay 10.0;
   -- After delay, perform some action
then abort
   -- Some ongoing activity that might be aborted
   Do_Something;
end select;
```

Terminating select statement

Allows a task to choose among alternative actions, including the ability to terminate itself when certain conditions are met.

E.g. This statement allows master task to automatically terminate the subtask when the master construct reaches its end.

```
select
   -- alternative actions
or
   terminate;
end select;
```

Terminating select statement example (1/2)

Problem: There's no limit to the number of times an entry can be accepted. We could even create an infinite loop in the task and accept calls to the same entry over and over again. An infinite loop, however, prevents the subtask from finishing, so it blocks its master task when it reaches the end of its processing.

Terminating select statement example (2/2)

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Show_Rendezvous_Loop is

   task T is
      entry Reset;
      entry Increment;
   end T;

   task body T is
      Cnt : Integer := 0;
   begin
      loop
         select
            accept Reset do
               Cnt := 0;
            end Reset;
            Put_Line ("Reset");
         or
            accept Increment do
               Cnt := Cnt + 1;
            end Increment;
            Put_Line ("In T's loop ("
                      & Integer'Image (Cnt)
                      & ")");
         or
            terminate;
         end select;
      end loop;
   end T;

begin
   Put_Line ("In Main");

   for I in 1 .. 4 loop
      --  Calling T's entry multiple times
      T.Increment;
   end loop;

   T.Reset;
   for I in 1 .. 4 loop
      --  Calling T's entry multiple times
      T.Increment;
   end loop;

end Show_Rendezvous_Loop;
```

References

[1] https://www.adacore.com/about-ada

[2] https://learn.adacore.com/courses/intro-to-ada/chapters/tasking.html

[3] https://youtu.be/YPD9U4Wuh5A?si=YxNWNLj57tAQolne

[4] https://www.youtube.com/watch?v=ZcdCDEhkbjU

[5] https://www.youtube.com/watch?v=RjbrUbp1Xo4