Ouverture

Parallel Programming Patterns

```
Greetings from task World.
Greetings from procedure Hello.
```

or

```
GGrreeeettiinnggss ffrroomm tparsokc eWdourrled .H
ello.
```

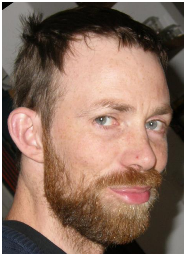
Parallel Programming Patterns

Jacob Sparre Andersen

Jacob Sparre Andersen Research & Innovation

March 10th 2013

Jacob Sparre Andersen



Currently:

- Independent consultant.
- Co-founder of AdaHeads K/S.
- Co-owner of Koparo Ltd.
- Software architect at AdaHeads and Koparo.

Background:

- PhD & MSc in experimental physics.
- BSc in mathematics.
- Has taught mathematics, physics and software engineering.
- Worked with bioinformatics, biotechnology and modelling of investments in the financial market.

jacob@jacob-sparre.dk
www.jacob-sparre.dk

Outline

1. Introduction
2. My favourite patterns
3. Low-level patterns
4. Higher-level patterns

An example

```
with Ada.Text_IO;
procedure Hello is
   task World;
   task body World is
   begin
      Put_Line ("Greetings from task World.");
   end World;
begin
   Put_Line ("Greetings from procedure Hello.");
end Hello;
```

An example (continued)

With the GCC version of the Ada standard library we get:

```
Greetings from task World.
Greetings from procedure Hello.
```

Using a different – but still correct – version of the Ada standard library we get:

```
GGrreeeettiinnggss ffrroomm tparsokc eWdourrled .H
ello.
```

In reality we would like `Put_Line` in the example program to be an **atomic operation.**

Logical parallelisation

Imagine a PBX management system[1].

```
task Connection_Manager;

task PBX_Message_Receiver;

task type Call_Manager (ID : Channel);

Call := new Call_Manager (Incoming_Channel);
```

[1] Could be `https://github.com/AdaHeads/Alice`

Implicit parallelisation

Give GCC the flag `-ftree-vectorize` and see what happens.

This can easily speed up a calculation with a factor of 6, and I've seen a factor of 8 reported.

Tasks

```
--  Singleton declaration:
task Name;

--  Type declaration:
task type Name;

--  Implementation:
task body Name is
   -- declarations
begin
   -- statements
end Name;
```

Protected objects

```
protected type Name is
   function Read_Some (Data : in Some_Type) return Data_Type;
   --  multiple parallel reads

   procedure Modify (Data : in out Some_Type);
   --  exclusive read/write

   entry Modify_With_Barrier (Data : in out Some_Type);
   --  exclusive read/write, possibly with barriers blocking
private
   Internal_Data : Some_Type;
end Name;
```

Rendezvous

```
task type Name is
   entry Rendezvous (Data : in out Data_Type);
end Name;

task body Name is
   -- declarations
begin
   -- statements
   accept Rendezvous (Data : in out Data_Type) do
      -- statements / copying of data
   end Rendezvous;
   -- statements
end Name;

--  Making a rendezvous with Name:
Name.Rendezvous (Data => Some_Data);
```

Semaphores

```
protected type Semaphore (Initial_Value : Natural) is
   procedure Signal;
   entry Wait;
private
   Count : Natural := Initial_Value;
end Semaphore;

protected body Semaphore is
   procedure Signal is
   begin
      Count := Count + 1;
   end Signal;

   entry Wait when Count > 0 is
   begin
      Count := Count - 1;
   end Wait;
end Semaphore;
```

Semaphores (continued)

We could have used a semaphore to ensure that each of the two calls to `Put_Line` in the very first example was running as an atomic operation:

```
procedure Hello is
  Exclusive_Output :  Semaphore (Initial_Value => 1);
  task World;
  task body World is
  begin
    Exclusive_Output.Wait;
    Put_Line ("Greetings from task World.");
    Exclusive_Output.Signal;
  end World;
begin
  Exclusive_Output.Wait;
  Put_Line ("Greetings from procedure Hello.");
  Exclusive_Output.Signal;

end Hello;
```

Barrier

```
protected type Barrier (Group_Size : Positive) is
   entry Wait;
private
   Gate_Open : Boolean := False;
end Barrier;

protected body Barrier is
   entry Wait when Wait'Count = Group_Size or Gate_Open is
   begin
      if Wait'Count > 0 then
         Gate_Open := True;
      else
         Gate_Open := False;
      end if;
   end Wait;
end Barrier;
```

Broadcast

```
protected type Broadcast is
   procedure Send (Message : in Message_Class);
   entry Tune_In (Message : out Message_Class);
private
   Current_Message : Message_Class;
   Has_Message     : Boolean := False;
end Broadcast;

protected Broadcast is
   procedure Send (Message : in Message_Class) is
   begin
      if Tune_In'Count > 0 then
         Current_Message := Message;
         Has_Message := True;
      end if;
   end Send;

   entry Tune_In (Message : out Message_Class) when Has_Message is
   begin
      Message := Current_Message;
      Has_Message := Tune_In'Count > 0;
   end Tune_In;
end Broadcast;
```

Queue

```
protected type Queue is
   procedure Insert (Message : in Message_Class);
   entry Get (Message : out Message_Class);
private
   Messages : ...;
end Queue;

protected Queue is
   procedure Insert (Message : in Message_Class) is
   begin
      ...
   end Insert;

   entry Get (Message : out Message_Class) when not Messages.Is_Empty is
   begin
      ...
   end Get;
end Queue;
```

Worker tasks

```
   Work_Queue : Queue;

   task type Worker;
   task body Worker is
      Job : Task_Description;
   begin
      loop
         Work_Queue.Get (Job);
         Process (Job);
      end loop;
   end Worker;

   Workers : array (1 .. CPUs) of Worker;
begin
   ...
   Work_Queue.Insert (New_Job);
```

Contact

Jacob Sparre Andersen
Jacob Sparre Andersen Research & Innovation
jacob@jacob-sparre.dk
http://www.jacob-sparre.dk/

Jacob Sparre Andersen
+45 21 49 08 04

A small warning: Not all source code in this presentation is complete and compilable.