Australian
National
University

DISTRIBUTING SERVER

Week 7 Laboratory for Concurrent and Distributed Systems

Uwe R. Zimmer based on material by Alistair Rendell

Pre-Laboratory Checklist

- ❏ **You have read this text before you come to your lab session.**
- ❏ **You understand and can utilize message passing.**
- ❏ **You have a firm understanding of memory based synchronization.**
- ❏ **You successfully completed all your previous labs in full.**

Objectives

This is an optional lab for advanced students. No concepts in here are required to pass the course, and if you struggle with basic concepts from previous labs, then it is wise to focus on their full resolution first.

The objective of this lab is to gain confidence and experience in more complex server setups, especially load balancing servers.

Interlude: Load balancing server

A classical pattern for a load balancing server is to re-distribute incoming calls based on the current situation of the computation nodes, while fully releasing any connection with the originating caller. The original caller may or may not be aware of the re-direction.

We will illustrate this principle based on the example of calculating the Mandelbrot Set by employing the currently-free, computational resources. The example is interesting in the sense that the time which a specific node will take to determine divergence is almost impossible to predict.

The techniques which will be used during this example are select statements on the calling as well as on the receiving side, conditional entries and requeue statements. This is a powerful set of tools which will require some time for you to fully analyse.

Relations between the individual parts of the system are apparently circular, yet Ada (despite being an Algol-style language which insists on declarations and elaborations before any usage) will be able to express this, due to the separation of specifications and implementations.

Have a look at the following pattern for a server which accepts requests to calculate one column inside a Mandelbrot Set:

```
task body Server is

begin
   loop
      select
         accept Compute (Job : Jobs; Diverge_Column : out Natural_Array) do
            for Node of Compute_Nodes loop
               select
                  Node.Responsive;
                  requeue Node.Compute;
               else
                  null;
               end select;
            end loop;
            requeue Busy_Nodes.Hold;
         end Compute;
      or
         terminate;
      end select;
   end loop;
 exception
    when E : others => Put_Line (Exception_Information (E));
 end Server;
```

The server has knowledge about all active compute nodes and probes each node to see whether it is currently responsive (by attempting to call it). If it finds a node to respond to such a call, it assumes that the node is currently available for business and redirects the original call to this node. This is done by a `requeue` statement, i.e. that the call is completely disassociated with the server and taken over by the chosen compute node.

If no responsive compute node can be found then this client call is re-directed to the blocking entry `Hold`. If the load situation changes then this entry may open and the call is directed back to the server for another attempt to find a compute node.

You may have noticed that this is a strategy to produce maximal throughput – not to guarantee fairness. Think how you could change the process in order to provide a first-in-first-out order of processing. Would this result in the same performance level?

Next, check how the compute nodes would interact in such a setup:

```
task body Compute_Node is
 begin
    loop
       select
          accept Responsive;
       or
          accept Compute (Job : Jobs; Diverge_Column : out Natural_Array) do
             declare
                Divergent : constant Real := 2.0;
             begin
                Busy_Nodes.Inc;
                for y in Diverge_Column'Range loop
                   declare
                      C : constant Complex :=
                      Job.Origin + (Re => 0.0, Im => Real (y) * Job.Resolution);
                      Z :          Complex := C;
                   begin
                      Diverge_Column (y) := Divergence_Limit;
                      Iterate : for Iteration in 0 .. Divergence_Limit loop
                         if abs (Z) > Divergent then
                            Diverge_Column (y) := Iteration;
                            exit Iterate;
                         end if;
                         Z := C + Z ** 2;
                      end loop Iterate;
                   end;
                end loop;
                Busy_Nodes.Dec;
             end;
          end Compute;
       or
          terminate;
       end select;
    end loop;
 exception
    when E : others => Put_Line (Exception_Information (E));
 end Compute_Node;
```

Such a compute node will determine the divergence-stage of a specific point in the Mandelbrot set by iterating the function $Z = C + Z^2$ (with C and Z being initialized the `x` and `y` coordination in question in their real and imaginary parts respectively). The mathematics behind are irrelevant for this example – the fact that divergence is hard to predict with respect to the original `x` and `y` coordinates is not. It means that some compute nodes will complete their job and become available again much earlier than others. This makes a classical "lock-step" parallel strategy of concurrent programming less efficient and we have a chance to rather apply a flexible strategy which considers the actual computational load at all times.

The compute node will alter between two states: being blocked in its `select` statement (and this being responsive on the `Responsive` entry) or being busy in the Compute entry (and thus being unresponsive on the `Responsive` entry).

The server as well as the compute nodes made references to Busy_Nodes, hence let's have a look what's happening over there:

```
protected body Busy_Nodes is

   entry Hold (Job : Jobs; Diverge_Column : out Natural_Array)
     when No_of_Busy_Nodes < Utilized_Cores is

   begin
      requeue Server.Compute;
   end Hold;

   procedure Inc is

   begin
      No_Of_Busy_Nodes := CPU_Range'Succ (No_Of_Busy_Nodes);
   end Inc;

   procedure Dec is

   begin
      No_Of_Busy_Nodes := CPU_Range'Pred (No_Of_Busy_Nodes);
   end Dec;

 end Busy_Nodes;
```

Turns out this a simple protected object which will hold calls which could not go through the server directly to the compute nodes in the first attempt. Calls will be held there until the number of busy compute nodes is less than the total number of cores. It will then requeue the call back to the server and the server will process it as if it would have arrived for the first time.

I assume that you begin to see why I warned you about circular dependencies in this example.

If you analyse the sequence of operations carefully you will also find that there is no starvation of tasks, as if a task has been redirected to the Hold point, because the server missed a compute node becoming responsive before the end of the probing loop inside the server, then this call will immediately come back to the server.

So far, this whole infrastructure is independent of a concrete Mandelbrot Set and in fact we can use this server to render many Mandelbrot Set displays at the same time. In order to make use of the setup for a specific section of the Mandelbrot Set we need Agents which will manage the processing of individual columns inside such a set. This could for instance look like that:

```
task body Agent is

 begin
    loop
       select
          accept Process (x : Pix_Range_x) do
             Server.Compute
               (Job => (Origin     => (Re => Min_x * Resolution * Real (x),
                                       Im => Min_y * Resolution),
                        Resolution => Resolution),
                Diverge_Column => Mandelbrot_Set (x));
          end Process;
       or
          terminate;
       end select;
    end loop;
 exception
    when E : others => Put_Line (Exception_Information (E));
 end Agent;
```

Instances of those Agent tasks will then take those calls to the server and ultimately to the compute nodes.

A procedure to instantiate the whole operation could then be:

```
procedure Dynamic_Servers is

   Agents      : array (Core_Range) of Agent;
   Agent_Index : Core_Range := Core_Range'First;

begin
   for x in Pix_Range_x loop
      Agents (Agent_Index).Process (x);
      Agent_Index := Succ_Mod (Agent_Index);
   end loop;

exception
   when E : others => Put_Line (Exception_Information (E));
end Dynamic_Servers;
```

Every column of this Mandelbrot Set display is handed over to a set of Agent tasks and everything runs at maximal performance.

… or does it? … see the next section …

Exercise 1: Analyse, Optimise and Enjoy

I cheated and attempted to talk you into believing that the above structure will result in maximal concurrency on your specific machine. Download the provided project from the course site, run it and see for yourself that your computer will be rather under-utilized. If you have trouble to see the actual CPU load then go to the Spec package and crank up the `Divergence_Limit` from `1_000` to say `10_000`. You should now see some significant sweating of your computer, yet not on all cores. Analyse what is going on, repair (or even optimise) the structure and resubmit it under the same archive name to the *SubmissionApp* under "Lab 8 Balancing Server" for a detailed code review by your peers and by us.

**Make Sure You Logout
to Terminate Your Session!**

Outlook

You are now very well prepared to take on your first major assignment – make the best of it and enjoy the swarming.