14 Mar 2025

Some simple ideas for multitasking of for-loops in Ada.

Leo Brewin

School of Mathematical Sciences
Monash University, 3800
Australia

Preface and Disclaimer

This is a short summary of my journey in learning how to use Ada's multitasking tools to handle for-loops in large scale numerical computations. Yes, this is a rather narrow target but it reflects my simple needs as I do not need the vast and impressive machinery that Ada offers.

I have written this document more to test my understanding rather than to provide a comprehensive account of Ada's multitasking capabilities. That is a job better left to those far more informed on Ada than myself. An excellent account (and my go-to reference for all things Ada) can be found in the classic textbook by John Barnes[a].

Given the brevity and narrow focus of this document it is likely that it will appeal only to a narrow audience. Most probably people like myself with a professional or academic interest in large scale scientific computation. Others may find it useful as an introduction to the field.

In the following sections I have provided fragments of Ada code as a way to introduce the main ideas. I have also provided (on this GitHub site)complete working codes. These are much the same codes as I used when learning about these matters. You may find them useful – or maybe not :). As they say – your kilometerage might vary.

There are a couple points regarding notation that need to be said. Here is a typical code fragment similar to those scattered throughout this document.

```
for i in 1 .. very_big loop                                                   -- sample-code
   --> do something useful
end loop;
```

Note the Ada comment, `-- sample-code`, in the top right corner. This will be taken as the name of the example. It will be useful when referring back to this or other examples. Note also the use of `-->` in the body of the for-loop. This is a shorthand way to hide one or more Ada statements that would be needed in a proper program.

[a]Barnes, Programming in Ada 2022 (CUP, ISBN 9781009564786)

uuid:920DBAE0-5BFE-4932-9956-7C8A75803A0A

Setting the context

I like to dabble with computational aspects of general relativity. Many of my Ada codes have loops similar to the following

```
loop                                                       -- three-for-loops

   for i in 1 .. very_big loop
      --> do something useful
   end loop;

   for i in 1 .. very_big loop
      --> do more useful things
   end loop;

   for i in 1 .. very_big loop
      --> and yet more amazing things
   end loop;

   exit when --> some exit condtion

end loop;
```

The outer loop might run for many thousands of iterations while the inner loops might have millions of iterations. These programs can run for a week or more (on a single cpu). So I have a strong interest in using Ada's multitasking to spread the computational load across all of the 20 cores of my computer. If I can get a 20-fold reduction in execution time I think I'd be quite happy.

Which loops in the above example are amenable to being run in parallel? If I wasn't interested in getting correct answers then the answers is simple – every loop can be run in parallel. But life is not so simple. Journal editors have a quaint preference for correct results (one of their many annoying habits). The physics and mathematics on which the above loops where built dictate that the outer loop must be run in strict sequence while for the three inner loops it's full steam ahead.

This clears the air a smidgen – the Ada tasks must be assigned to just the three inner loops.

Here is a simple question – When and where in the program should the tasks be created (and how long should they live)? Should they be created (and destroyed) repeatedly for each inner loop *inside* the outer loop? Or should they be created just once *outside* the outer loop? The former approach, using *transient* tasks, is quiet easy to implement while the latter approach, using *persistent* tasks, requires a bit more work (to ensure that the serial-parallel workflow is observed).

The advantages of the transient model is that it naturally observes the serial-parallel workflow. But at a cost of creating and destroying all tasks three times for every single iteration of the outer loop (which may run for many thousands of steps – so that could be a significant problem).

Here is another matter to ponder – How should the workload of any one of the inner loops be assigned to the tasks? The standard approach (as used by Open-MPI and Light-Weight Threading) is to *evenly* split the loops into *disjoint* parts and have each part processed by individual tasks. This minimalist approach is commonly known as a *work sharing* model. But it does have one potential weakness. As each task completes its allocated work it will sit idle until all other tasks have completed their work. Having tasks twiddle their thumbs while waiting for others to catch up is not ideal. Why not put them to work? Other parallel models do exactly that. A task manager oversees all tasks and doles out more work as each task finishes its present body of work. The intention is to keep all tasks busy with minimal idle time. Such models are known as *work seeking* or *work stealing.* Is this idea useful for our simple three for-loops example? Not really. The computational complexity of the lines inside each for loop is independent of the loop index[b]. Thus, when the data space is evenly spread amongst the tasks, it is extremely unlikely that any task will finish significantly ahead of any other task, ergo, little or no idle time.

By my own painful experience, I know that throwing tasks at loops can get a tad tricky. For example, there is this pesky business of getting correct results. If the tasks are not carefully monitored, then some tasks could start processing parts of the second loop before others have finished work on the first loop. The program will run but it will give incorrect results (not my preferred outcome). So a key objective is to ensure that the tasks are coordinated so that the correct serial-parallel workflow is observed. Another all too common problem (in my limited experience) is that, without due care, it is possible for tasks to stall (for example, a pair of tasks each waiting on the other to do something) or for tasks to become lost (when communication between it and the main program is broken). The end result is that the program comes to a grinding halt and does nothing – it doesn't crash (usually) but rather it stalls while waiting for the tasks to sort out their problems or for absent tasks to rejoin the team (impossible once they have gone AWOL). And let's not mention debugging parallel programs.

A brief summary of the rendezvous model

Any program that uses more than one task must have some mechanisms to control the flow of execution of those tasks (lest there be anarchy in the workspace). How do tasks announce that they are ready to do work or that they have finished their allocated jobs? How are tasks told what to do and when? When should a task be created? When should it die? All of these questions lead back to one simple question – How can tasks communicate with each other?

The Ada language provides two distinct methods by which tasks can communicate – by *rendezvous* calls (the subject of this short summary) or by using *protected objects* (summarised in a later section).

As with other constructs in Ada, the definition of a task consists of two parts – a specification part and a body part.

A task specification will declare one or more entry points. These will be matched by corresponding accept statements in the task body. The body of work that the task is meant to do is scattered between the accept statements. Here is a simple example containing a pair of entry points, their matching accept statements and some executable statements.

[b]This is not true for all for-loops. It just happens to be true for the class of for-loops that I am interested in.

```
 1  task foo is                                                    -- task-spec-body
 2     entry start_task;
 3     entry stop_task;
 4  end foo;
 5
 6  task body foo is
 7  begin
 8     accept start_task;
 9     --> one or more statements to do some work
10     accept stop_task;
11     --> cleanup and prepare to die
12  end foo;
```

These entry/accept statements are used by other tasks (including the main task) to control the flow of each task.

When a task hits an accept statement, it will wait for a corresponding entry call to be made by another task. Until that matching post is made, that task will sit still while twiddling its digital thumbs. Conversely, when a task posts an entry call, it will wait until the nominated task accepts that call. These exchanges that occur between pairs of tasks are the basis of the rendezvous model.

The entry points (in the specification) shows what posts the task will be listening for. The accept statements (in the body) shows the waypoints (the rendezvous) that the task will visit while it waits for the matching entry call.

In the above code fragment there are two entries, `start_task` and `stop_task` (lines 2 and 3). The task comes alive (is created) at the `begin` statement (line 7) in its body. In this case the task `foo` will immediately hit the `start_task` accept statement (line 8). It will then wait until a `start_task` call is posted directly to it. That call is made by another task using the standard dot notation, that is, `foo.start_task`. At that point the rendezvous between the pair of tasks is complete and both tasks can resume doing useful work.

What happens if some task (say `cat`) calls `foo.stop_task` ahead of another task (say `dog`) calling `foo.start_task`? Nothing out of the ordinary. Task `cat` will be forced to wait while `foo` does its work. When `foo` hits its `stop_task` statement then `cat` will resume its work. This is good. However, had task `dog` made both calls, first to `foo.stop_task` and later to `foo.start_task`, then `dog` will wait forever. This is not good.

Out of sequence entry calls are to be expected for the simple reason that all tasks involved are running, largely, independently of each other and thus there is some uncertainty as to when various entry calls are made. The above simple example shows that this could cause problems though Ada does offer a `select-or` block (as described below in the section on task types) that can ameliorate this class of problems.

In a correctly written program, the task `foo` will accept calls to both `start_task` and `stop_task`. When `foo` hits the `end foo` statement it will be terminated with prejudice – RIP `foo`. This completes the lifespan of `foo`.

This all sounds nice and dreamy but the assumption of *a correctly written program* might be be false. How so? Imagine that `foo` is sitting waiting for a call to `stop_task`. What if that call is never made (by poor program design or by programmer fatigue)? The result is that the task `foo` will wait forever and likewise the program unit that contains `foo` will also wait forever. A similar problem can arise if two tasks make calls to `foo.start_task`. One call will be accepted while leaving the other stranded, waiting for an accept that can never be made. it's a bit ironic that a programming scheme designed to speed up a program might in fact cause it to run forever! Clearly such aberrant behaviour must be avoided and that requires careful attention to the interplay between tasks.

Task types

The plan as stated earlier was to arrange for a set of tasks to cooperatively do the job of loops such as the following

```
for i in 1 .. very_big loop                                    -- simple-loop
   --> do something useful
end loop;
```

In this picture, each task will be doing the same computations but over disjoint subsets of the index range (from 1 to very_big). Each task differs only in which subset of indices it is working on. So it makes sense to have a way of specifying a template for each task – this is where a task type enters the game.

Here is an example of task type.

```
task type my_task is                                              -- task-type
   entry start_task (i_beg, i_end : Integer);
   entry stop_task;
end my_task;

task body my_task is
   the_beg, the_end : Integer;
begin
   accept start_task (i_beg, i_end) do
      the_beg := i_beg;
      the_end := i_end;
   end start_task;
   --> do something useful
   accept stop_task;
   --> goodbye cruel world
end my_task;
```

The key point to note here is that the start_task now carries two parameters, i_beg and i_end. The values for i_beg and i_end are posted in the call to start_task and are copied to local variables, the_beg and the_end, in the execution of the matching accept statement. This is standard method for passing operational data between tasks.

This template can be used to create one or more actual tasks, for example

```
foo, bah, cat, dog : my_task;
```

This creates four tasks all of which, at this point, will be sitting idle at their start_task statements. Suppose that very_big equals 400 (which is hardly very big but it serves as a demonstration). Then the four tasks can be set loose by calls like the following

```
foo.start_task (  1, 100);
```

```
bah.start_task (101, 200);
cat.start_task (201, 300);
dog.start_task (301, 400);
```

The main thread (or which other task is running the show) will also need to make `stop_task` calls to ensure that all four task die a respectful death. That can be done using

```
foo.stop_task;
bah.stop_task;
cat.stop_task;
dog.stop_task;
```

If all goes well (as it should) the four tasks will do their work, in parallel, collectively processing the data from index 1 to 400. This should show a 4-fold reduction in wall clock time.

That all seems quite nice but what about the case of, say, 20 tasks? It would be tedious to have to create 20 named tasks (not to mention how silly the code would look). The simple solution is to create an array of tasks, as follows,

```
the_tasks : Array (1..20) of my_task;
```

This will create 20 tasks all based on the `my_task` type. Starting these tasks can be done with a simple loop over `the_tasks` array with (obviously) some extra logic to correctly set the index sub-range for each task. Here is a code fragment that generalises the above specific example. It starts each of the `num_tasks` tasks and sets the sub-ranges to cover the complete range from 1 to `num_data`.

```
 1  the_tasks : Array (1..num_tasks) of my_task;                    -- task-arrays
 2
 3  declare
 4
 5     i_step : Integer;
 6     i_beg, i_end : Integer;
 7
 8  begin
 9
10     i_step := (num_data - 1) / num_tasks;
11
12     i_beg := 1;
13     i_end := min (i_beg + i_step, num_data);
14
15     for t in 1 .. num_tasks loop
16
17        the_tasks (t).start_task (i_beg, i_end);
18
19        i_beg := i_end + 1;
20        i_end := min (i_beg + i_step, num_data);
21
22     end loop;
23
24  end;
```

The tasks are created and spring to life in line 1 and they sit idly by on their `accept start_task` statement waiting for the call to `start_task`. This occurs on line 17. Once that start loop is complete (lines 15 to 22) all of the tasks are up and running and are doing useful work. Once each task has finished its allocated work it will sit idle at its accept stop_task statement. To kick start the task once again, another set of calls, this time to stop_task, is needed. This is really easy to do (as there are no arguments required by stop_task). Here is a code fragment that does the job

```
for t in 1 .. num_tasks loop                                       -- stop-tasks

   the_tasks (t).stop_tasks;

end loop;
```

This not only stops each task but, importantly, each task now terminates (look back at the task body, tasks die when they exit their body – the ultimate out-of-body experience).

Since tasks will die a natural death when they exit their body (i.e., they reach their end statement in the task body), it is sometimes easier (for the minimalist coder) to *not* include the stop entry/accept statements in the task's specification and body. But in most cases there are good reason to have explicit calls to stop_task. This will allow the task to do any last minute jobs (e.g., write final summaries, close files etc.) in an orderly fashion before the task dies.

Despite having just made the point that stop entry/accept statements *should* be used, the following few examples ignores that advice. Why? It's simple – the space on these pages is limited so brevity wins the day. Feel free to add these elements back into the code :).

Comparing codes

The simple code described above targets simple loops like the following

```
very_big : Constant := 100_000;                                     -- serial-01

for i in 1 .. very_big loop
   --> do something useful
end loop;
```

For reasons that (should) become clear soon, this simple loop can also be implemented as follows.

```
declare                                                             -- serial-02

   very_big : Constant := 100_000;

   procedure job1 (i_beg, i_end : Integer) is
   begin

      for i in i_beg .. i_end loop
         --> do something useful
      end loop;

   end job1;

```

```
14 begin
15    job1 (1, very_big);
16 end;
```

The `for loop` has been buried in a procedure `job1` (lines 5 to 12) while the execution of the loop occurs with the call to that procedure (on line 15). Yes this variation does seem a bit crazy but have faith – this formulation will make a bit more sense after looking at the multitasking version.

Actually, the name `job1` should provide a clue as to where this discussion is headed – to make it easier to adapt the code to cases with more than one simple loop. Those extra loops will be written as procedure, `job2`, `job3` etc., all based on `job1`, and will be executed with calls similar to that for `job1`.

The corresponding multitasking version of this loop is somewhat more involved (are we surprised?).

```
declare                                                          -- parallel-01

   very_big  : Constant := 100_000;
   num_tasks : Constant := 16;

   procedure job1 (i_beg, i_end : Integer) is
   begin

      for i in i_beg .. i_end loop
         --> do something useful
      end loop;

   end job1;

   procedure run_parallel (run_serial : access procedure
                                           (i_beg, i_end : Integer);
                           i_min, i_max : Integer;
                           num_tasks    : Integer)
   is

      task type My_Task is
         entry start_task (i_beg, i_end : Integer);
      end My_Task;

      task body My_Task is
         my_i_beg : Integer;
         my_i_end : Integer;
      begin

         accept start_task (i_beg, i_end : Integer) do
            my_i_beg := i_beg;
            my_i_end := i_end;
         end;

         run_serial (my_i_beg, my_i_end);

      end My_Task;
```

```
38
39          the_tasks : Array (1..num_tasks) of my_task;
40
41       begin
42
43          declare
44             i_step : Integer;
45             i_beg, i_end : Integer;
46          begin
47
48             i_step := (i_max - i_min) / num_tasks;
49
50             i_beg := i_min;
51             i_end := min (i_beg + i_step, i_max);
52
53             for t in 1 .. num_tasks loop
54
55                the_tasks (t).start_task (i_beg, i_end);
56
57                i_beg := i_end + 1;
58                i_end := min (i_beg + i_step, i_max);
59
60             end loop;
61
62          end;
63
64       end run_parallel;
65
66    begin
67
68       run_parallel (job1'access, 1, very_big, num_tasks);
69
70    end;
```

All (of the meat and veggies) of the multitasking code is hidden inside the procedure run_parallel. The remaining parts of this code are very similar to the previous (serial) code. The for-loop appears as the body of the procedure job1. That procedure, along with the loop limits, are passed as arguments to run_parallel (line 68).

A little package

A small saving on space can be had by plonking the procedure run_parallel in a package. The package specification would be

```
package Parallel is

   procedure run_parallel (run_serial    : access procedure
                                             (task_id      : Integer;
                                              i_beg, i_end : Integer);
                           i_min, i_max  : Integer;
                           the_num_tasks : Integer);
```

```
end Parallel;
```

While the corresponding body would be

```
package body Parallel is

   procedure run_parallel (run_serial    : access procedure
                                              (task_id      : Integer;
                                               i_beg, i_end : Integer);
                           i_min, i_max  : Integer;
                           num_tasks     : Integer)
   is

      task type My_Task is
         entry start_task (task_id : Integer; i_beg, i_end : Integer);
      end My_Task;

      task body My_Task is
         my_task_id : Integer;
         my_i_beg   : Integer;
         my_i_end   : Integer;
      begin

         accept start_task (task_id : Integer; i_beg, i_end : Integer) do
            my_task_id := task_id;
            my_i_beg   := i_beg;
            my_i_end   := i_end;
         end;

         run_serial (my_task_id, my_i_beg, my_i_end);

      end My_Task;

      the_tasks : Array (1..num_tasks) of my_task;

   begin

      declare
         i_step : Integer;
         i_beg, i_end : Integer;
      begin

         i_step := (i_max - i_min) / num_tasks;

         i_beg := i_min;
         i_end := min (i_beg + i_step, i_max);

         for t in 1 .. num_tasks loop

            the_tasks (t).start_task (t, i_beg, i_end);

            i_beg := i_end + 1;
            i_end := min (i_beg + i_step, i_max);
         end loop;

      end;

   end run_parallel;

end Parallel;
```

The previous example, `parallel-01`, can now be rewritten as

```
with parallel;   use parallel;                                 -- parallel-02

declare

   very_big  : Constant := 100_000;
   num_tasks : Constant := 16;

   procedure job1 (i_beg, i_end : Integer) is
   begin

      for i in i_beg .. i_end loop
         --> do something useful
      end loop;

   end job1;

begin

   run_parallel (job1'access, 1, very_big, num_tasks);

end;
```

The reason for rewriting the simple for-loop example, `serial-01`, as the seemingly overblown for-loop in `serial-02` should now be apparent – the extension to one or more loops is trivial. For example, the original `three-for-loops` example can now be run using

```
with parallel;   use parallel;                                 -- parallel-03

declare

   very_big  : Constant := 100_000;
   num_tasks : Constant := 16;

   --> define procedures job1, job2 and job3

begin

   loop

      run_parallel (job1'access, 1, very_big, num_tasks);
      run_parallel (job2'access, 1, very_big, num_tasks);
      run_parallel (job3'access, 1, very_big, num_tasks);
      exit when --> some exit condtion

   end loop;

end;
```

This meets the basic objectives for a parallel version of the original loop, three-for-loops, namely, to run each inner for-loop across a set of tasks while ensuring that no task starts work on a subsequent for-loop until all tasks have finished their work on the current for-loop. This last part comes for free from the structure of the procedure run_parallel. Every task lives and dies in each instance of run_parallel. Thus no task survives in the transition form one inner-for loop to the next. Hence the earlier description of this as a model using *transient* tasks.

One possible problem with this transient model is that the tasks are repeatedly created and destroyed (three times) for each iteration of the outer loop. As that outer loop might run for many hundreds of thousands of iterations that might incur a non-trivial operational cost. This would only be a concern if the cost of creating and destroying the tasks in one instance of run_parallel was comparable to the computational cost of the for-loop in that instance of run_parallel. The safe bet is that if the tasks are kept busy (e.g., very_big is truly a very big number), then this apparent concern really is not worth worrying about.

Persistent tasks

The overheads of using transient tasks are almost certainly very small when compared to the actual computational costs for the work done across all three inner for-loops.

Even so, it is worth asking the question – How can the same objectives be met using *persistent* tasks? In this case the tasks are created once, outside the main outer loop. Each task then has to do its work, in turn, on all three inner for-loops. But importantly, as before, no task can begin work on a subsequent for-loop until all tasks have finished their work on the current loop. This requires care in controlling the tasks. Explicit code must be included to pause a task (for those that finished their current work ahead of other tasks) and to restart each task (after all tasks are paused, that is, after all tasks have finished their work). This pause and restart sequence must be repeated three times (once for each for-loop) for each iteration of the outer loop.

This simple analysis gives a few clues as to how such a work flow might be expressed in Ada. There will be the familiar entry statements start_task and stop_task and two new entries, pause_task and restart_task. Here is a skeleton version of a task specification and body for the persistent tasks.

```
task type My_Task is                                          -- persistent-spec-01
   entry pause_task;
   entry restart_task;
   entry stop_task;
   entry start_task (...);
end My_Task;
```

```
task body My_Task is                                          -- persistent-body-01
   --> some local variable
begin
```

```
10
11     accept start_task (...) do
12        --> collect index bounds: i_beg, i_end
13     end;
14
15     loop
16
17        select
18
19           accept restart_task;
20              --> do work for 1st inner for-loop
21           accept pause_task;
22
23           accept restart_task;
24              --> do work for 2nd inner for-loop
25           accept pause_task;
26
27           accept restart_task;
28              --> do work for 3rd inner for-loop
29           accept pause_task;
30
31        or
32
33           accept stop_task;
34              --> prepare to die
35           exit;
36
37        end select;
38
39     end loop;
40
41  end My_Task;
```

The main thing to notice here is that the various `accept` statements are embedded in an outer loop. This is an essential part of a persistent task – without a loop of some kind the task will hit its `end My_Task` statement thus enjoying an untimely death (not much persistence in that).

Look closely at the body of this task type. It closely mimics the structure of the original for-loop in `three-for-loops`. The loop (lines 15 to 39) mimic's the outer loop (lines 1 to 17) of `three-for-loops` while each pair of restart/pause accept statements captures the inner for-loops of `three-for-loops`.

The task body also contains a `select-or` block. This is an Ada construct specific to tasks. Its purpose is to simplify the code needed to control the task. How this works may not seem clear at this point but will become so soon – put this point aside for the moment.

How can a set of such tasks be driven to do the job (as per `three-for-loops`)? The code would contain an array of tasks as well as a loop to call the appropriate restart/pause accept statements. That code would look something like the following

```
declare                                                   -- persistent-driver-01

   very_big  : Constant := 100_000;
   num_tasks : Constant := 16;

   the_tasks : Array (1..num_tasks) of My_Task;

   --> define procedures job1, job2 and job3

   procedure run_parallel is
   begin

      for t in 1 .. num_tasks loop
         the_tasks (t).restart_task;
      end loop;

      for t in 1 .. num_tasks loop
         the_tasks (t).pause_task;
      end loop;

   end run_parallel;

begin

   for t in 1 .. num_tasks loop
      --> set values for i_beg & i_end
      the_tasks (t).start_task (t, i_beg, i_end);
   end loop;

   for n in 1 .. very_big loop

      run_parallel;  -- all tasks running job1
      run_parallel;  -- all tasks running job2
      run_parallel;  -- all tasks running job3

   end loop;

   for t in 1 .. num_tasks loop
      the_tasks (t).stop_task;
   end loop;

end;
```

The intent here is quite clear – one loop to start the tasks, one loop to do the work and one loop to stop the tasks. This simple structure is in part due to the `select-or` block in the task body. The `select-or` block allows the task to respond to out-of-sequence entry calls. For example, a task with a body such as

```
select
   accept foo;
   accept bah;
or
   accept moo;
or
   accept cow;
end select;
```

could complete the `select-block` with entry calls to both `foo` and `bah` (in that order) or one call to either `moo` or `cow`. In contrast, if the task body had been

```
accept foo;
accept bah;
accept moo;
accept cow;
```

then the task would progress only after all four entries had been called in strict sequence.

A `select-or` block is not essential but it does make the code a tad more readable (in keeping with the Ada spirit). As a little exercise, try rewriting the above code *without* using a `select-or` block.

Problems with shared data

The discussion to this point has avoided one of the major headaches in multitasking – tasks that access shared data. For example, consider two tasks, `foo` and `bah`, that have access to a global data item named `xyz`. There are a whole host of problems that can arise when `foo` and `bah` attempt to manipulate `xyz` at the same time (or within a few cpu clock cycles). Here is a classic example (known as a *race condition*). Suppose `xyz` represents an integer with a current value of zero. Suppose also that this pair of tasks do nothing more than increment `xyx` by one. On a single cpu computer, the net effect, after `foo` and `bah` have done their work, will be that `xyx` now has a value of two.

Sounds simple enough. However, things get a bit tricky when multiple tasks are in play. The simple business of adding one to a variable actually involves a handful of cpu instructions. This provides a short window of opportunity for one task to read the *original* value of `xyx` *before* the other has had chance to return its updated value. So both tasks could read zero for `xyx` in which case the final value for `xyx` would be one. A less than ideal outcome.

If both tasks do not overlap in time then the correct result will be obtained. This improved outcome might arise from fortunate programming where `foo` always runs well ahead of `bah`. It might also be that `bah` was, on this occasion, delayed because it was serving other jobs (not our jobs but other jobs the computer happens to be running). On other occasions `foo` and `bah` might be running in-step and thus produce the incorrect result for `xyz`. The upshot is that the program might produce different results from one run to the next.

Here is another well known problem. Suppose now that `foo` is set the job of computing `x := x + y;` while `bah` computes `y := x - y;`. The programmer decides, to avoid a race condition (on `x` and `y`), that `foo` owns `x` while `bah` owns `y`. This means that `bah` will wait for `foo` to update `x`. Likewise, `foo` will wait for `bah` to update `y`. Thus each task sits still waiting for the other to complete its work. Both tasks will be blocked and the whole program comes to a silent standstill.

Do either of these problems exist in any of the previous examples? No, and for one simple reason – the tasks share no data. Recall that when each for-loop was farmed out to a set of tasks, the range of that for-loop was split into *disjoint* sub-ranges with one task per sub-range. Thus no pair of tasks could share any data. Hence no race-conditions nor any task blocking. That may be good news but the problem is that there are far more situations where task do share data and where the programmer must avoid race-conditions and blocked tasks. Ada provides an elegant solution to this problem in the form of *protected objects*. Here is a short account of protected objects.

Protected objects

Race-conditions and blocking are just two examples of problems that can arise when tasks manipulate shared data. The result is that the execution of such programs is non-deterministic. One way to regain determinism is to use Ada's *protected objects*. Each protected object is a collection of *protected operations* (comprised of entries, procedures and functions) and its *private data*. A task can engage with the protected object's private data only through the protected operations. The important point to note is that Ada uses some behind-the-scenes magic to ensure that no two or more tasks can manipulate any private data at the same time. It does this by way of conceptual locks (i.e., locks that are hidden from the programmer). When a single task requests access to the private data, that task is given the lock for that protected object. Only the task with the lock can manipulate the private data. When the task has finished working on that private data, it surrenders the lock (again, hidden from the programmer, it's part of the behind-the-scene magic). The private data is then available for use by other tasks. If a task requests access to private data that is currently locked with another task, then Ada places that task in a queue. The queue provides an orderly access to the private data (as each prior task surrenders its' lock). Note that for each protected object there is only one lock in play at any time.

There is more that needs to be said for protected objects, but for the moment, here is a trivial example of a protected object.

```
 1  protected addition is                                    -- protected-object-01
 2     procedure add_one;
 3     function get_total return Integer;
 4  private
 5     total : Integer := 0;
 6  end addition;
 7
 8  protected body addition is
 9     procedure add_one is
10     begin
11        total := total + 1;
12     end add_one;
13
14     function get_total return Integer is
15     begin
16        return total;
17     end get_total;
18  end addition;
```

The protected object `addition` contains two protected operations (`procedure add_one` and `function get_total`) and one piece of private data, `total`. The full declaration of the protected object has the familiar structure of a specification part (lines 1 to 6) and body part (lines 8 to 18). The protected procedure `add_one` does just one thing – it adds one to the protected data `total` while the protected function `get_total` provides read access to `total`. Yes, this really is a trivial example :).

How can this protected object be set to work? By simply using the dot notation to call the appropriate protected operations. Here is a simple example.

```
declare
   sum : Integer;
```

```
begin
   for i in 1 .. 50 loop
      addition.add_one
   end loop
   sum := addition.get_total;
end;
```

The result is that `sum` is set equal to 50. That same job could be split across two tasks, one task counts from 1 to 25 and the other from 26 to 50. The expected result is 50 and that is exactly what this pair of tasks will report (how nice). However, had this pair of tasks *not* used the above protected object they would do their running sum using a shared data item and would thus run the real risk of a race-condition. The magic of the protected object's lock mechanism guarantees that such race conditions can not happen in the above example. This is good news.

Barrier conditions and protected entries

One of the key protected operations of a protected object are the *protected entries*. These look like protected procedures but with one crucial addition – they each carry a *barrier condition*. These are boolean expressions that control the execution of the protected entries. When a task places a call on a protected entry the first action is to test the barrier condition. If its value is `True` then the call succeeds and the body of the protected entry is executed (and control returns to the caller). Otherwise, the entry call (and its caller) are paused and placed in a queue (for this entry). Any subsequent calls to this paused entry will also be added to the end of that queue. At some later point, as other tasks go about their work, the barrier condition may change from `False` to `True`. This releases the paused entry and allows exactly one queued call to proceed. But what of the other calls waiting in the queue? Are they also released? No, and for one very good reason – the execution of the released entry call might flip the barrier condition back to `False`. It might also flip the barrier condition on *other* paused entries. Thus the barrier conditions of *every* entry must be retested on completion of *any* successful entry call. This retesting must also be applied for any successful call on a protected procedure (because they too might flip one or more barrier conditions). No such retesting is required when protected functions are called for the simple fact that functions can only read data and thus can not change any barrier conditions.

Enough chatter – what use can be made of these protected entries? Here is a simple example based on the previous problematic computation of `x := x + y;` and `y := x - y;`. Its purpose is to show how easy it is to force the `x := x + y;` computation to *always* be done ahead of that for `y := x - y;`. It's a simple example of how to avoid blocked tasks.

```
declare                                                  -- protected-object-02

   x : Integer := 0;
   y : Integer := 1;

   protected plus_minus is
      entry x_minus_y;
      procedure x_plus_y;
   private
      done_x_plus_y : Boolean := False;
   end plus_minus;

```

```
13    protected body plus_minus is
14
15       entry x_minus_y when done_x_plus_y is
16       begin
17          y := x - y;
18          done_x_plus_y := False;
19       end x_minus_y;
20
21       procedure x_plus_y is
22       begin
23          x := x + y;
24          done_x_plus_y := True;
25       end x_plus_y;
26
27    end plus_minus;
28
29 begin
30
31    --> in task foo:
32    plus_minus.x_minus_y;
33
34    --> in task bah:
35    plus_minus.x_plus_y;
36
37 end;
```

The code is rather easy to read. There are the obvious parts that deal with the pure numerics (lines 17 and 23) and the calls for those computations from two tasks (lines 32 and 35). But the more interesting part is the `when done_x_plus_y` part of the body of `x_minus_y` (line 15). That is the barrier condition. It states that the body of the entry should be executed only when the boolean variable `done_x_plus_y` is `True`. But the only place where `done_x_plus_y` is set to `True` is in the final line of `procedure x_plus_y`. So even if a call to `entry x_minus_y` is made *ahead* of a call to `procedure x_plus_y` that call will be forced to wait until *after* the `procedure x_plus_y` has been completed. Note that `entry x_minus_y` also resets `done_x_plus_y` to `False` in preparation for a later round of calls for `y := x-y;` and `x := x+y;`.

So, regardless of which task ran ahead of the other, the final values for `x` and `y` will be `x=1` and `y=0`.

The three for-loops using protected objects

Is it possible to create a protected object that suits the demands of the simple `three for-loop` example? Indeed it is and the logic is quite simple. The protected object has the job of controlling the tasks as they do their for each of the three inner loops. The protected object will at times need to pause the tasks (so that other tasks may catch up) and at other times to allow the tasks to resume useful work. This suggests that the protected object will have (at least) two protected entries, `pause_task` and `resume_tasks`. A corresponding workflow might be as simple as follows. Each task, as it completes its allotted work, calls `pause_task`. Later, once all tasks are paused, the main task will call `resume_tasks` to kickstart the tasks forward to their next job. Each pause and resume cycle fits with the three inner for-loops with one cycle for each for-loop. Consider now the barrier conditions. These are the secret sauce that enables the protected object to works its magic. Here is a sketch of a protected object that can do the job. It is adapted form the second solution to exercise 2 in section 20.9 of Barnes[c].

```
num_tasks : Constant := --> some integer                   -- protected-object-03

protected control is
   entry pause_task;
   entry resume_tasks;
   entry resume_main;
end control;

protected body control is

   entry pause_task when resume_tasks'count = 1 is
   begin
      null;
   end pause_task;

   entry resume_tasks when pause_task'count = 0 is
   begin
      null;
   end resume_tasks;

   entry resume_main when pause_task'count = num_tasks is
   begin
      null;
   end resume_main;

end control;
```

This is remarkably simple (as noted by Barnes). Notice the use of `'count` in each barrier. This is part of the internal bookkeeping for protected objects. Each protected entry has an associated queue and the `'count` attribute returns how many calls are present in the queue.

The main loop in the (persistent) task body would look something like the following:

```
loop                                                        -- protected-object-04

   job1 (...);   control.pause_task;
   job2 (...);   control.pause_task;
   job3 (...);   control.pause_task;

   exit when --> some stopping crietria;

end loop;
```

A *single* iteration of the above task-loop can be had by including lines like the following

```
-- for job1                                                 -- protected-object-05
control.resume_main;
control.resume_tasks;

-- for job2
```

[c]Barnes, Programming in Ada 2022 (CUP, ISBN 9781009564786)

```
control.resume_main;
control.resume_tasks;

-- for job3
control.resume_main;
control.resume_tasks;
```

in the main task (i.e., the main program).

So how does this protected object and its friends work? Once all the work of creating and initialising the tasks has been done, those tasks will be merrily doing work on `job1`. Meanwhile, the main task will have issued its call to `control.resume_main` and will immediately be paused because the corresponding barrier will be `False` (as no tasks are paused at this point). When a task completes its current job, it calls `control.pause_task`. At that point the main task has yet to call `control.resume_tasks` and thus the barrier on `control.pause_task` is `False`. Consequently, the task will be paused. Other tasks will also, at some point, make their calls to `control.pause_task` and they too will be paused. Once *all* tasks are paused, then the main task can complete its call to `control.resume_main`. The main task will resume and then place a call to `control.resume_tasks`. This now sets `resume_tasks'count` to 1 and thus the barrier on `control.pause_task` is now `True`. This triggers a cascade of resumed tasks. The main task will, during this cascade, be paused until the final task has been released. This fact follows from the choice of barrier on `control.resume_tasks`. That barrier is `True` only when there are no paused tasks.

This simple interplay between the barriers does the job of marshalling the tasks to move from one job to the next without any task racing ahead or falling behind.

There is one last piece of the puzzle to sort out – how to gracefully stop the tasks and the main program. The proposed task-loop given above does have an as yet incomplete exit condition. The main task will at some point communicate to the protected object that it is time to stop each task. It could do this by calling a protected entry to set a protected Boolean exit flag to `True`. Each task task would, at a later point, read that flag (using a protected function) and thus exit the task-loop. This logic requires some small changes to the protected object. Here is an (abbreviated) updated version.

```
num_tasks : Constant := --> some integer                     -- protected-object-06

protected control is
   entry pause_task;
   entry resume_tasks;
   entry resume_main;
   entry request_stop;
   function should_stop Return Boolean;   -- used by each task
private
   stop_tasks : Boolean := False;         -- the exit flag
                                          -- written by entry request_stop
                                          -- read by each task
end control;

protected body control is

   --> bodies for the above spec's

end control;
```

The main task-loop would now be something like

```
loop                                                     -- protected-object-07

   job1 (...);   control.pause_task;
   job2 (...);   control.pause_task;
   job3 (...);   control.pause_task;

   exit when control.should_stop;

end loop;
```

The main task can request each task to stop by issuing a call to `control.request_stop`. That seems quite simple but it also raises a potential problem. Unless some is care is taken in choosing when to make that call, it is possible for the program to be blocked due to an earlier race condition. How might this happen? There is a short window of time when all the tasks are resuming work after their paused state. In that short time some fast starting tasks may be ahead of the call to `control.request_stop` while the slow starting tasks will be behind that call. Thus the fast tasks will continue their work in the main task-loop while the stragglers will exit. And therein lies the problem – the barrier for `control.resume_main`, namely, `pause_task'count = num_tasks` can never be `True`. The fast tasks will never terminate and so too for the main program.

A simple solution to this problem is to ensure that the call to `control.request_stop` is made when the tasks are *all paused*. This snippet would do the job.

```
loop                                                     -- protected-object-08

   -- for job1
   control.resume_main;
   control.resume_tasks;

   -- for job2
   control.resume_main;
   control.resume_tasks;

   -- for job3
   control.resume_main;
   if time_to_stop then
      control.request_stop;
   end if;
   control.resume_tasks;

   exit when --> are we done having fun?

end loop;
```

Notice that the main outer loop (as per the original three-for-loops example) has been brought back into the picture. Notice also how the call to `control.request_stop` was placed *between* the calls to `control.resume_main` and `control.resume_tasks`. Had it been placed *before* `control.resume_main` then the same race condition as described above could occur.

Another way to avoid the race condition is to chose a barrier on `control.request_stop` identical to that for `control.resume_main`. This allows the call to `control.request_stop` to be moved anywhere before the final call to `control.resume_tasks`. So the above loop could now be written as

```
loop                                                         -- protected-object-09

   if time_to_stop then
       control.request_stop;
    end if;

   -- for job1
   control.resume_main;
   control.resume_tasks;

   -- for job2
   control.resume_main;
   control.resume_tasks;

   -- for job3
   control.resume_main;
   control.resume_tasks;

   exit when --> are we done having fun?

end loop;
```

Both versions work but this later version seems a bit cleaner than the previous version.

Sample codes

I have included a number of sample codes that demonstrate the ideas outlined above. These fully functional codes can be found in the ... directory. They can be compiled either by running the shell script `build.sh` or by running `make` in the directory ...

There are six different examples. The first four are based on the skeleton codes described above. The last pair of codes, `lwt1` and `lwt2`, are examples using the Light-Weight Threading library (see below for more details).

ex1:	Transient tasks using rendezvous calls for multitasking.
ex2:	Persistent tasks using rendezvous calls for multitasking.
ex3:	As per **ex2** but with small changes in the task body to use `Ada.Synchronous_Barriers`.
ex4:	Persistent tasks using protected objects.
lwt1:	A Light-Weight Threading example using OpenMP
lwt2:	As per **lwt1** but using pure Ada.

There are also two versions of each of the six examples. Version “a” (e.g., `ex3a`) provides copious output that demonstrates that the code is doing what it is meant to do (look carefully at the output and notice that all tasks finish a job before starting the next job). In contrast, the version “b” codes (e.g., `ex3b`) are stripped bare of any output (except the LWT examples). This serves two purposes. First, the code is much cleaner as the various `put_line`’s in the version “a” codes are a distraction from the underlying structure of the code. The second reason is that the bare code is more suited to timing experiments – the time lost due to the put_line’s could skew any time measurements.

Each code can be run using a matching shell script (e.g., ex2a.sh for ex2a).

So which of the codes has the shortest execution time? None – they all run at much the same speed. Here are my simple benchmark results.

Benchmark times

They say that the proof is in the pudding. In this case the pudding is the set of example codes while the proof would be to show that their execution times are reduced as the number of tasks are increased. The best that could be expected, when using N tasks, would be an $\mathcal{O}(1/N)$ behaviour in the timings (i.e., each doubling of N should see a halving in the execution time).

Each of the sample codes uses rather dull for-loops like the following

```
procedure job1 (...) is
begin

   for i in i_beg .. i_end loop
      delay Duration (rnd_scale*rnd_real);
   end loop;

end job1;
```

The purpose of the delay statement is to simulate a prolonged mathematical computation. The variable rnd_scale is a convenient way to adjust the extent of the duration. The *function* rnd_real returns uniform random reals in the range 0 to 1. The for-loop limits, i_beg..i_end, represent the subset, drawn from 1..Num_Data, of data being processed by this for-loop.

The following table really does show that there is ample proof in the pudding. Up until the full 20 cores (on this computer) are used, the execution times show a very consistent $\mathcal{O}(1/N)$ behaviour. No gains can be expected beyond $N = 20$ for the simple fact that there are no more cpu cores available to call into action.

Other codes

There are a handful of other Ada projects that appear suitable for large scale scientific computations. However, the only active project appears to be the Light-Weight Threading library. The others date from around 2013 – that might explain why I was unable to compile any of these codes (other than the LWT library) on my 2022 Mac Studio (Mac Silicon running macOS 15.3.1 with version 14.2.0 of gcc-gnat and friends).

ParaSail: Light-Weight Threading Library

The proposed Ada 2022 standard provides support for parallel execution of many parts of an Ada program. The Light Weight Threading (LWT) project is a library that brings many of

N	ex1	ex2	ex3	ex4	lwt1	lwt2
1	178.8	178.1	178.2	178.6	178.2	177.9
2	88.7	88.7	88.9	88.8	88.4	88.4
4	44.2	44.3	44.3	44.2	44.3	44.3
8	22.2	22.1	22.1	22.1	22.2	22.2
16	11.3	11.3	11.3	11.3	11.3	11.3
32	10.6	10.6	10.6	10.6	10.5	10.5
64	10.9	10.9	10.9	10.8	10.7	10.8

Table 1: Execution times (in seconds) for the example codes for various choices of the number of tasks, N. The above times were obtained using `Num_Loop=1`, `Num_Data=1_000_000`, and `rnd_scale=0.0001`. This table shows clearly that for $N \leq 16$ the execution times are consistent with an $\mathcal{O}(1/N)$ behaviour. The flat execution times for $N > 16$ is expected since the host computer on which these examples were run only had 20 cpu cores.

those features to existing pre-Ada2022 codes. It does so by providing an API that implements the parallel constructs using either the OpenMP library or by use of existing Ada tools (tasks, rendezvous and protected objects). The complexities of the implementation are mostly hidden behind the API resulting in clear and simple codes. With a simple for-loop like the following

```
declare

   Num_Data    : Integer := 100_000;

begin

   for i in 1 .. Num_Data loop
      --> some useful computation
   end loop;

end;
```

the LWT OpenMP version would have the following basic structure

```
with LWT.Parallelism;             use LWT.Parallelism;
with LWT.OpenMP;                  use LWT.OpenMP;

declare

   Num_Data    : Integer := 100_000;
   Num_Chunks : Integer := 10;        -- optional

   Control : OMP_Parallel (Num_Threads => 6);

   procedure job1 (...) is
   begin
      for i in ... loop
```

```
         --> some useful computation
      end loop;
   end job1;

begin

   Par_Range_Loop (Low        => Longest_Integer (1),
                   High       => Longest_Integer (Num_Data),
                   Num_Chunks => Num_Chunks,    -- optional
                   Loop_Body  => job1'Access);

end;
```

This example uses the OpenMP library (which is a standard inclusion in all modern Ada compilers). The LWT also offers a *work stealing* model that uses standard Ada constructs for multitasking rather than the OpenMP library. The changes in going from an OpenMP code to the work stealing code are trivial – just two changes are required. First

```
-- with LWT.OpenMP;                   use LWT.OpenMP;
   with LWT.Work_Stealing;            use LWT.Work_Stealing;
```

and second

```

-- Control : OMP_Parallel (Num_Threads => 6);
   Control : WS_Parallel  (Num_Servers => 6, Options => null);
```

The LWT library is part of, but wholly independent of, a larger project, ParaSail. It, LWT, can be found on GitHub at https://github.com/parasail-lang/parasail/tree/main/lwt.

Neither of the following codes compile on my Silicon Mac. I have included them here because some people might be inclined to bring these codes back to life (my limited Ada skills are not up to the task). Both offer work-stealing and work-sharing models. The source code, despite the compilation issues, might provide insights on how to implement such models.

Paraffin

Authors:	Brad Moore
Version:	4.3 (June 2013)
Source:	https://sourceforge.net/projects/paraffin

Magpie

Authors:	Marc Criley
Version:	0.11 (Jan 2011)
Source:	https://sourceforge.net/projects/magpie-mc