The `timer_tree_type` module

Neil N. Carlson

Abstract

The `timer_tree_type` module implements a lightweight method for creating and maintaining a nested tree of timers. The tree is automatically generated via the nested starting and stopping of named timers. Because timers are distinguished by name and position in the tree, a start/stop pair will give rise to different timers when the code containing it is executed within different timer nestings. As a result, the timing of shared code can be easily and automatically partitioned according to its use.

1 Introduction and Basic Usage

The `timer_tree_type` module implements a lightweight method for creating and maintaining a nested set, or tree, of timers. The tree is automatically generated via the nested starting and stopping of named timers. The most basic use of the timers requires no more than calling `start_timer` and `stop_timer` in matching pairs with an arbitrary name string as illustrated in Figure 1. The only restriction is that the timers be nested: the timer most recently started (and still running) is the only one eligible to be stopped. Starting A, then B, and then stopping A is not allowed, for example. Any incorrect nesting of timers will be detected by `stop_timer` at run time. The calls, of course, can be in different program units; they need only have the logical nesting illustrated.

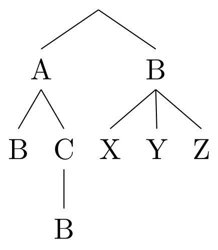
The same name may be used for multiple start/stop pairs, and start/stop pairs with the same name *and nesting* are regarded as a single timer and their elapsed time accumulated accordingly. Otherwise they are regarded as distinct timers. An example of the latter is timer B. It is started and stopped at three different positions within the call nesting, and this is reflected in the associated timer tree. Each pair is regarded as a distinct timer. Timer A, on the other hand, is an example of the former case. It is started and stopped twice, but at the same position within the call nesting, and thus appears in the timer tree only once. The elapsed time for each call pair is accumulated in the single timer as expected.

Finally, a call to `write_timer_tree` (at any time) will write the time accumulated thus far for each timer, using indentation to express the nested structure of the timer tree, as shown in Figure 1.

Because timers are distinguished by name and position in the tree, a start/stop pair will give rise to different timers when the code containing it is executed within different timer nestings. As a result, the timing of a portion of shared code can be easily and automatically partitioned according to its use. This is a distinctive feature of this implementation.

For portability the Fortran intrinsic subroutine `cpu_time` is used to acquire the processer time, and thus the resolution of the timers is limited by the resolution of this subroutine,

```
use timer_tree_type
call start_timer ("A")
  call start_timer ("B")
  call stop_timer  ("B")
  call start_timer ("C")
    call start_timer ("B")
    call stop_timer  ("B")
  call stop_timer ("C")
call stop_timer  ("A")
call start_timer ("B")
  call start_timer ("X")
  call stop_timer  ("X")
  call start_timer ("Y")
  call stop_timer  ("Y")
  call start_timer ("Z")
  call stop_timer  ("Z")
call stop_timer  ("B")
call start_timer ("A")
call stop_timer  ("A")
call write_timer_tree (unit=6, indent=2)
```



Sample Output

```
A: 4.90000E-02
  B: 1.00000E-02
  C: 1.90000E-02
    B: 9.00000E-03
B: 2.80000E-02
  X: 9.00000E-03
  Y: 1.00000E-02
  Z: 9.00000E-03
```

Figure 1: A sequence of `start_timer`/`stop_timer` calls and the corresponding timer tree that it generates. Also a sample of the type of output generated by `write_timer_tree`.

which varies from one system to another but is typically not very fine. As a result, these timers are not well suited to timing computationally short bits of code.

For most cases this covers everything one needs to know to in order to use the global timer tree managed by the `timer_tree_type` module. The next section describes some additional functionality that may be useful, and in section that follows it, the `timer_tree` derived type that underlies the implementation is described.

2 The global timer tree

The following procedures operate on the global timer tree.

`call start_timer (name [,handle])`

starts the timer with the specified character string `name` that is a child of the current timer. If no such child exists, one is created with this name. This child timer then becomes the current timer. If the optional integer argument `handle` is specified, it returns a handle to the timer which can be used as an argument to `write_timer_tree` or `read_timer`.

`call stop_timer (name [,stat [,errmsg]])`

stops the current timer. The current timer's parent becomes the new current timer.

It is an error if the current timer does not have the specified name. If the optional integer argument `stat` is present, it is assigned the value 0 if no error was encountered; otherwise it is assigned a non-zero value. In the latter case, the allocatable deferred-length character string `errmsg`, if present, is assigned an explanatory message. If `stat` is not present and an error occurs, the error message is written to the preconnected error unit and the program is stopped.

`call write_timer_tree (unit, indent [,handle])`
writes the accumulated time for each timer to the specified logical unit, using indentation to express the nested structure of the timer tree. The incremental number of spaces to indent for successive tree levels is given by `indent`. If an optional integer `handle` returned by `start_timer` is specified, only the accumulated times for that timer and its decendents are written.

`call read_timer (handle, cpu)`
returns, in the default real argument `cpu`, the elapsed time for the timer associated with the `handle` returned by `start_timer`. The timer may be running or stopped.

`call reset_timer_tree`
resets the timer tree to its initial, empty state.

2.1 Accessing the timer tree data

Sometimes more direct access to the timer tree data is needed than is provided by either `read_timer` or `write_timer_tree`. For example, the data may need to be communicated between processes in a parallel simulation, or it may need to be written in a format that can be easily read and used to initialize the timer tree. The following subroutines provide such functionality.

`call serialize_timer_tree (tree, name, cpu)`
returns the current state of the timer tree in flat arrays. Timers may be running or stopped and their state is unaltered. The allocatable, deferred-length character array `name` and allocatable default real array `cpu` return the timer names and elapsed cpu times indexed by tree node number. The allocatable default integer array `tree` returns the structure of the tree as a sequence of node numbers: node numbers appear in matching pairs, like opening and closing parentheses, with the intervening sequence describing the trees of its children, recusively. The nodes are numbered so that the initial node of the pairs appear in sequential order. This enables a simple reconstruction of the tree. All three allocatable array arguments are allocated by the subroutine; they are reallocated if necessary.

`call deserialize_timer_tree (tree, name, cpu)`
defines the state of the timer tree using the `tree`, `name`, and `cpu` arrays as returned by `serialize_timer_tree`. This can be used to initialize the timer tree with results from a previous simulation, for example. Note that timer handles are not preserved.

```
m = 0 ! the high-water mark of node indices encountered
do j = 1, size(tree)
  n = tree(j)
  if (n > m) then ! first encounter of this node index
    print *, '<TIMER NAME="', trim(name(n)), '" CPU="', cpu(n), '">'
    m = n
  else
    print *, '</TIMER>'
  end if
end do
```

Figure 2: Using the output of `serialize_timer_tree` to reconstruct the timer tree.

2.1.1 Using the data from `serialize_timer_tree`

The code fragment shown in Figure 2 illustrates how the output of `serialize_timer_tree` can be used to reconstruct the tree by writing the tree using nested XML tags that reproduce the structure of the tree. The integer `m` records the high-water mark of the tree node numbers encountered, which, because of the way the nodes are numbered, can be used to distinguish the intial and final node of each pair.

3 The `timer_tree` derived type

The timer tree used thus far is in fact an object of derived type `timer_tree` that is stored as a private module variable in the `timer_tree_type` module. All of the procedures described above operate on this single global instance. There may be circumstances where it is preferable for applications to declare and use their own `timer_tree` variables. The trade off is that the `timer_tree` variables must be carried around. The derived type has the following type bound subroutines that have the same interface and effect as those described above, except that they operate on the specific instance rather than the global timer tree.

`start(name [,handle])` has the same interface as `start_timer`.

`stop(name [,stat [,errmsg]])` has the same interface as `stop_timer`.

`write(unit, indent [,handle])` has the same interface as `write_timer_tree`.

`read(handle, cpu)` has the same interface as `read_timer`.

`serialize(tree, name, cpu)` has the same interface as `serialize_timer_tree`.

`deserialize(tree, name, cpu)` has the same interface as `deserialize_timer_tree`.

Objects of this derived type:

- should not be used in assignment statements; only the default intrinsic assignment is available, and its semantics are unlikely to be what is desired.
- are properly finalized when the object is deallocated or otherwise ceases to exist.

4 Bugs

Send bug reports and suggestions for improvements to `neil.n.carlson@gmail.com`.