1 User manual for the util/TIMING module

1.1 Introduction

This module was developed by G. Tóth (2001-). It can be used for timing and profiling Fortran 90 codes.

It is platform and compiler independent and very easy to use. It can provide profiling information while the code is running. The amount and type of information can be easily manipulated.

Profiling with a 'real' profiler is compiler and platform dependent, it can only be done after the run is finished, and the amount and type of information is not always easy to manipulate. On the other hand a profiler may provide more accurate and detailed information than this timing module, and it does not require changes of the code.

1.2 Usage

The TIMING module can time anything identified by a name string. Here is a short example of usage:

```
call timing_version(on,vname,vnum)! check the version

call timing_comp_proc('GM',iProc) ! Set the component name and PE number
....
if(iProc==0) &
    call timing_active(.true.)    ! Activate timing on processor zero
call timing_step(0)               ! Initialize step value
call timing_start('main')         ! Start timing the main code
...
do nstep=1,100
   call timing_step(nstep)        ! Put step into timing module
   call timing_start('whatever')  ! Start timing
   ...                            ! Do whatever
   call timing_stop('whatever')   ! Stop timing
   call timing_show('whatever',1) ! Show last timing for whatever
   if (mod(nstep,10)==0) then
      write(*,*) &                ! Obtain and write speed
          'speed of whatever is',
          1./timing_func_d('sum/iter',1,'whatever','main'),&
  ' iterations/sec'
      call timing_report          ! Show tree of timings for last 10 steps
      call timing_reset_all       ! Reset timing
   end if
end do
call timing_stop('main')          ! Stop timing the main code
call timing_report_total          ! Show all timings as a sorted list
call timing_report_style('tree')  ! Change report style
call timing_report_total          ! Show all timings in calling tree
```

The timing_version returns three values: the first logical variable 'on' is true for a functional timing module, and false for the empty timing module. The second string variable 'name' (of length 40) returns the name and the author of the module, and the last real variable 'number' returns the version number.

The timing_comp_proc subroutine sets the name of the component and the processor rank (with respect to some MPI group). This information is needed if there are multiple components using the timing utility. Both the compnent name and processor number will be shown in the timing reports. If more components do timings on the same processor, the timings will be reported together with the name of the component which called timing_comp_proc last.

The timing is activated by timing_active. For parallel runs one should usually activate the timing module for one processor only. For timing multiple components the root processor of each component can be used, for example. When the timing module is inactive, the timing commands are executed but do not time and do not provide output.

The timing is done by a pair of timing_start and timing_stop calls. The string name arguments of the two calls must match. Make sure that if the timing_start is called then the corresponding timing_stop call is also executed. Timings can be inside loops, and nested arbitrarily. Note, however, that timing inside a recursive procedure does not work. The timings are distinguished by the name as well as by the nesting level.

The timing clocks can be reset, and the results of timings can be printed to the screen or returned into variables in various formats as discussed in the following sections.

1.2.1 Clocks and resets

There are 3 clocks started and stopped by timing_start and timing_stop. Clock 1 always measures the latest timings, clock 2 measures cumulative timings since the last reset, while clock 3 typically measures cumulative timings for the whole run.

The clocks can be reset by

```
call timing_reset('whatever',2)
```

which resets clocks 1 and 2 for the timing of 'whatever'. The first string argument is 'name', and the second integer argument is 'nclock', i.e. the number of clocks to be reset starting with clock 1.

If the 'name' argument is set to '#all', then the clocks 1 to nclock are reset for all names. The particularly useful and typical call

```
call timing_reset('#all',2)
```

is identical with the shortcut version

```
call timing_reset_all
```

Note that active timings (started but not yet stopped) are not stopped by the reset, but the start time is overwritten so that only the time after the reset is measured.

Beside measuring cumulative timings, clocks 2 and 3 also count the number of 'calls', and the number of 'iterations' for each timing entry. The iterations are distinguished by the current step number (a monotonically increasing positive integer) which can be set by calling timing_param_put_i or timing_step, which are described in the next subsection.

1.2.2 Putting parameters: `timing_param_put_i, timing_step, timing_depth, timing_report_style`

The integer parameters for the timing module can be set with the generic subroutine call

```
call timing_param_put_i('depth',2,error)
```

where the first string argument is the name of the function, the second integer is the value, and the third integer argument returns 0 if the parameter was set successfully, or -1 if it failed.

There are only two integer parameters for the timing module: 'step' gives the current step, while 'depth' is the maximum depth of nested timings. For these two parameter settings the following short cuts are provided:

```
timing_step(value)  ! same as timing_param_put_i('step',value,error)
timing_depth(value) ! same as timing_param_put_i('depth',value,error)
```

The default value for 'step' is 0. It is expected to be set to a positive integer value which is monotonously increasing in successive calls.

The default value of 'depth' is -1, which means that the timings can be nested arbitrarily deep. If depth is set to 0, then no timing is done at all, while if depth is set to 1, only the main code is timed.

The style of the report shown by the `timing_report` and `timing_report_total` subroutines is determined by the `report_style`. The default style is 'cumu', which produces cumulative timings sorted by the timing values. The 'list' style also gives sorted timings, but timings with different calling parents are distinguished. Finally the 'tree' style gives the timings in the format of a nested calling tree.

1.2.3 Reading the timings: `timing_func_d`

The current timing value of clock 2 for 'whatever' called from 'main' can be obtined by the function call

```
timing_func_d('sum',2,'whatever','main')
```

The first string argument 'func_name' determines the function to be returned. The available values are 'sum', 'sum/iter' and 'sum/call'. The latter two functions only make sense for clocks 2 and 3. The second integer argument 'iclock' selects the clock. The third string argument 'name' selects the timing, which is further specified by the last string argument 'parent_name'. The parent is the timing that was started last but not stopped when the timing for 'whatever' is started. The parent of the first timing is itself, so

```
write(*,*)'Elapsed time=',timing_func_d('sum',1,'main','main')
```

prints out the total time spent by 'main' since the last reset. The parent is needed to distinguish between timings called from different places. If the names do not match, no output is produced.

1.2.4 Show individual timings: `timing_show`

Results for a certain timing can be printed with the timing_show command. The first string argument 'name' is the name of the timing to be shown, the second integer argument 'iclock' is the selected clock number.

For clock 1 the name, the calling parent, and the very last timing are shown:

```
call timing_show('calc_gradients',1)
Last timing for calc_gradients (advance_expl):    0.01 sec
```

For clock 2 the cumulative timing since the last reset is given. All timings matching the name (but called from different parents) are shown. The timing per iteration and per call and the percentage with respect to the parent are also shown:

```
call timing_show('calc_gradients',2)
Timing for calc_gradients from step      15 to      20 :
    0.55 sec,    0.111 s/iter   0.011 s/call   26.66 % of advance_expl
Timing for calc_gradients from step      15 to      20 :
    0.01 sec,    0.008 s/iter   0.008 s/call    0.32 % of timing_test
```

For clock 3 the total timing is reported:

```
call timing_show('calc_gradients',3)
Timing for calc_gradients at step      20 :
    1.11 sec,    0.111 s/iter   0.011 s/call   26.69 % of advance_expl
Timing for calc_gradients at step      20 :
    0.01 sec,    0.008 s/iter   0.008 s/call    0.16 % of timing_test
```

1.2.5 Timing reports and profiling: `timing_sort, timing_tree, timing_report`

For most purposes one can use the following two generic subroutines

```
timing_report
! same as timing_sort(2,-1,.true.)  if style is 'cumu'
! same as timing_sort(2,-1,.false.) if style is 'list'
! same as timing_tree(2,-1)         if style is 'tree'

timing_report_total
! same as timing_sort(3,-1,.true.)  if style is 'cumu'
! same as timing_sort(3,-1,.false.) if style is 'list'
! same as timing_tree(3,-1)         if style is 'tree'
```

In the following the general timing_tree and timing_sort subroutines are described in detail.

The timings of all or some of the subroutines can be reported in various ways. The most complete information is obtained by

Table 1: Output of `timing_tree(2,-1)`

TIMING TREE from step 15 to step 20

name	#iter	#calls	sec	s/iter	s/call	percent
timing_test	1	1	2.54	2.536	2.536	100.00
advance_expl	5	5	2.02	0.404	0.404	79.65
calc_gradients	5	50	0.50	0.100	0.010	24.79
calc_facevalues	5	50	1.02	0.203	0.020	50.32
#others			0.50	0.101		24.88
calc_gradients	1	1	0.01	0.008	0.008	0.31
save_output	1	1	0.20	0.201	0.201	7.92
#others			0.32	0.315		12.42

```
call timing_tree(2,-1)
```

where 2 is the clock number, and the second argument is the maximum depth of the tree to be shown (-1 means to show the whole tree). The output is shown in Table 1. The header indicates that the timing tree was generated at time step 20 by clock 2 which was restarted at step 15. So the timings refer to 5 time steps.

The table consists of seven columns and several rows:

1. The 1st column gives the name of the timing. The very first row is the top of the tree, usually refers the main program. The names below the first row are indented according to the calling depth: timings called directly from the top timing are not indented, timings called from these are indented by 2 spaces, timings called from these are indented by 4 spaces, etc.

2. The 2nd column gives the number of iterations when a timing call was maede.

3. The 3rd column gives the number of timing calls for an item.

4. The 4th through 6th columns give the actual timings in seconds: total time, time/iteration and time/call.

5. The 7th column gives the percentage with respect to the calling 'parent'. The consecutive lines at the same indentation level should always add up to 100%, because the last row with name '#other' contains the untimed part of any given level.

In the example presented in Table 1 timing_test took 2.54 seconds to run from step 15 to 20. Roughly 80% of the time was spent in advance_expl, and 8% on save_output. Advance_expl itself took 2.01 seconds or 0.4 sec/step. 50%

Table 2: Output of `timing_sort(1,-1,.true.)`

```
-------------------------------------------------------------------------------
SORTED TIMING at step=      20
name                     sec percent
-------------------------------------------------------------------------------
timing_test             2.54  100.00
-------------------------------------------------------------------------------
advance_expl            0.40   15.77
save_output             0.01    0.31
calc_facevalues         0.02    0.87
calc_gradients          0.01    0.34
initialize              0.00    0.00
-------------------------------------------------------------------------------
```

of this time was spent on calc_facevalues, 25% on calc_gradients, and 25% on other things. Note that calc_gradients occurs twice, because it is called from the main program and advance_expl as well.

The amount of detail can be decreased by giving a maximum depth. For example

```
call timing_tree(2,2)
```

will produce a table without the indented 3rd to 5th rows. The timing_tree cannot be used with clock 1, because clock 1 does not accumulate timings, which makes the information of the table rather difficult to interpret. Clock 1 timings are better presented by the 'timing_sort' subroutine, which is discussed next.

Another way of representing the timing results is

```
call timing_sort(1,-1,.true.)
```

which shows the full uniquely sorted timings for clock 1. The first argument 'iclock' selects the clock, the second argument 'show_length' defines the maximum number of timings shown (-1 means show all), and the third argument 'unique' determines whether the timings for identical names but different calling parents should be added up or not. When clock 1 is used the timings are given for the very last call. A sample output is shown in Table 2. The percentages are with respect to the longest timing in the first row.

If clock 2 or 3 is used, the table contains the cumulative timings and the number of steps and calls are also indicated. For example the first four of the uniquely sorted timings of clock 2 can be obtained with

```
call timing_sort(2,4,.true.)
```

which gives an output as shown in Table 3. Note that calc_gradients was called 51 times altogether. The last row with name '#others' contains the sum of

Table 3: Output of `timing_sort(2,4,.true.)`

```
--------------------------------------------------------------------------------
SORTED TIMING from step=      15 to step=      20
name                         sec percent   #iter  #calls
--------------------------------------------------------------------------------
timing_test             2.71  100.00       1       1
--------------------------------------------------------------------------------
advance_expl            2.14   78.88       5       5
calc_facevalues         1.04   38.17       5      50
calc_gradients          0.60   21.96       6      51
#others                 0.20    7.40
--------------------------------------------------------------------------------
```

Table 4: Output of `timing_sort(3,-1,.false.)`

```
--------------------------------------------------------------------------------
SORTED TIMING at step=      20
name                 (parent)                  sec percent   #iter  #calls
--------------------------------------------------------------------------------
timing_test          (timing_test)            5.21  100.00       1       1
--------------------------------------------------------------------------------
advance_expl         (timing_test)            4.23   81.23      10      10
calc_facevalues      (advance_expl)           2.05   39.38      10     100
calc_gradients       (advance_expl)           1.16   22.26      10     100
initialize           (timing_test)            0.30    5.76       1       1
save_output          (timing_test)            0.20    3.85       1       1
calc_gradients       (timing_test)            0.01    0.22       1       1
--------------------------------------------------------------------------------
```

timings that were not included into the first 4 rows. Also note that the total percentage exceeds 100% since the timings at different depths overlap.

Finally the original timings can be sorted without adding up values for the same subroutine. In this case the parents are also indicated, so that timings with identical names can be distinguished:

```
call timing_sort(3,-1,.false.)
```

results in Table 4.

1.3 List of subroutines and functions

See the reference manual for a complete and documented list.

```
option_timing(on,name,number)

timing_active(value)

timing_comp_proc(value1,value2)

timing_param_put_i(name,value,error)
timing_step(value)  ! == timing_param_put_i('step',value,error)
timing_depth(value) ! == timing_param_put_i('depth',value,error)

timing_report_style(value)

timing_start(name)

timing_stop(name)

timing_reset(name,nclock)
timing_reset_all    ! == timing_reset('#all',2)

timing_show(name,iclock)

timing_sort(iclock,show_length,unique)

timing_tree(iclock,show_depth)

timing_report       ! == timing_sort(2,-1,.true.)  for style 'cumu'
                    ! == timing_sort(2,-1,.false.) for style 'list'
                    ! == timing_tree(2,-1)         for style 'tree'
timing_report_total ! == timing_sort(3,-1,.true.)  for style 'cumu'
                    ! == timing_sort(3,-1,.false.) for style 'list'
                    ! == timing_tree(3,-1)         for style 'tree'

real*8 function timing_func_d(func_name,iclock,name,parent_name)
```

1.4 Files and make targets

A complete list of make targets in the src and doc directories can be listed with

```
make help
```

The actual TIMING module consists of

```
src/ModTiming.f90
src/timing.f90
src/timing_cpu.f90
```

The last file contains the call to the actual timing function MPI_WTIME, but it could be replaced with a platform specific function or SYSTEM_CLOCK. The three source files can be compiled into one library module:

```
libTIMING.a
```

with the command

```
cd src
make LIB
```

Compiler options can be edited in the main directory in

```
Makefile.conf
```

which is included into the Makefile. The empty version of the TIMING module is defined by

```
srcEmpty/timing_empty.f90
```

When the TIMING module is not needed for the main code, libTIMING.a should be produced in the srcEmpty directory. which can be compiled with

```
cd srcEmpty
make LIB
```

The empty module does not use any memory, most subroutine calls return directly to the caller without any output. The exceptions are timing_version, which tells the calling program that the empty timing routine is not functional, and timing_active, which writes a warning message if an attempt is made to activate the empty timing module.

The use of the TIMING module is fully demonstrated in

```
src/timing_test.f90
```

which can be compiled both to a serial and a parallel code, both with the real and the empty timing module. These four combinations provide 4 tests, which can be all executed with

```
cd src
make tests
```

A sample output can be found in

```
src/tests.log
```

which was obtained by

```
cd src
make tests > tests.log
```

This manual was produced from

```
doc/MAN_TIMING.tex
doc/TIMING.tex
```

with

```
cd doc
make MAN
```

The src, srcEmpty and doc directories can be cleaned with

```
make clean
make distclean
```