System Operation Monitors

Aleksandar Donev

February 2000

Contents

1 Module System Monitors
   1.1 Initialization of Hardware Monitors ........................................... 2
   1.2 Timer Operations ............................................................................ 3
   1.3 (De)Allocation Monitoring ............................................................... 4

2 Formatting rules for HPF/F90 files ......................................................... 6
1 Module System_Monitors

This module provides some utilities for monitoring and bookkeeping of system operations, such as memory allocation and timing. For timing, the Stop_Watch public-domain module from NIST is used, but I hide it from the library by making simple wrappers StartTimer, StopTimer, ResetTimer and ReadTimer. In fact, I decided to provide internal timers to be used by the program (currently 100 of them), so that the user does not have to worry about the dynamic clock creation of Stop_Watch. For memory allocation, I monitor the total explicitly allocated memory in RecordAllocation and watch if it ever exceeds max_usable_memory—all in bytes—and abort if it does. Also, the peak allocated memory is monitored and can be reported via a call to MemoryUsage. The routines CreateMonitors and DestroyMonitors should be called before or after using this module respectively.

"WEAVE.f90" 1 ≡

MODULE System_Monitors
  USE Precision // Kind parameters
  USE Standard_Types, ONLY: BIT_SIZE
  USE StopWatch
  USE Error_Handling
  IMPLICIT NONE
  PUBLIC :: CreateMonitors, DestroyMonitors, StartTimer, StopTimer, ResetTimer, ReadTimer,
             RecordAllocation, MemoryUsage
PRIVATE

  TYPE (WATCHTYPE), DIMENSION (100), PUBLIC :: timers
     // An array of timers that we can use later
  INTEGER, PUBLIC, SAVE :: nflops = 0, nreads = 0, nwrites = 0 // Operation counters
  INTEGER, PUBLIC, SAVE :: allocated_memory = 0, max_allocated_memory, max_usable_memory = 10^8
     // About 100 MB by default
  (GenericInterfaces 1.3.2)

CONTAINS

  (Initialization 1.1)
  (TimerOperations 1.2)
  (AllocationMonitoring 1.3.1)

END MODULE System_Monitors
1.1 Initialization of Hardware Monitors

The routine `CreateMonitors` simply initializes some monitors, allocates memory, etc. `DestroyMonitors` deallocates the timers.

(Initialization 1.1) ≡

```fortran
subroutine CreateMonitors()
    implicit none
    call option_stopwatch (io_unit_print = message_log_unit, io_unit_error = error_log_unit,
                            print_errors = .t., abort_errors = (non_critical_action ≡ "S"))
    call create_watch (timers)  // Reset all timers
    n_flops = 0
    n_reads = 0
    n_writes = 0
    allocated_memory = 0
    max_allocated_memory = 0
end subroutine CreateMonitors

subroutine DestroyMonitors()
    implicit none
    call destroy_watch (timers)  // Release the timers
end subroutine DestroyMonitors
```

This code is used in section 1.
1.2 Timer Operations

I decided to avoid using `StopWatch` directly inside the code unless needed (such as when actually optimizing the code). Therefore I make some simple wrappers available. The routines are very simple and self-explanatory:

\[ \text{Timer Operations 1.2} \equiv \]

**SUBROUTINE** `StartTimer(timer)`

```fortran
IMPLICIT NONE
INTEGER, INTENT (IN) :: timer
CALL START_WATCH(timers(timer), err = error_status)
END SUBROUTINE StartTimer
```

**SUBROUTINE** `StopTimer(timer)`

```fortran
IMPLICIT NONE
INTEGER, INTENT (IN) :: timer
CALL STOP_WATCH(timers(timer), err = error_status)
END SUBROUTINE StopTimer
```

**SUBROUTINE** `ResetTimer(timer)`

```fortran
IMPLICIT NONE
INTEGER, INTENT (IN) :: timer
CALL RESET_WATCH(timers(timer), err = error_status)
END SUBROUTINE ResetTimer
```

**FUNCTION** `ReadTimer(timer) RESULT(elapsed_time)`

```fortran
IMPLICIT NONE
INTEGER, INTENT (IN) :: timer
REAL :: elapsed_time
CALL READ_WATCH(READ_RESULT = elapsed_time, WATCH = timers(timer),
    CLOCK = "cpu", err = error_status)
END FUNCTION ReadTimer
```

This code is used in section 1.
1.3 (De)Allocation Monitoring

Since this library is very memory intensive it makes sense to monitor memory problems. The routine `RecordAllocation` is a generic routine that does bookkeeping of allocated memory. The type of the allocated data is passed via the argument `mold`, so that the user need not worry about the number of bytes each type and kind takes. I use the portability-project extended routine `BIT_SIZE` to calculate this size with no hassle. The routine checks for the success of the allocation in the `stat` argument to `ALLOCATE alloc_status` and aborts if an allocation was not successful. `caller` can be any identifier telling the user where the error occured, usually the name of the calling program unit.

```
"WEAVE.F90" 1.3 ≡
@m _Record_De_Allocation(_type, _kind)

SUBROUTINE RecordAllocation[_&_kind](n_elements, mold, caller, alloc_status)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: n_elements
  _type(KIND = _kind), INTENT(IN) :: mold
  CHARACTER(LEN = *), INTENT(IN), OPTIONAL :: caller  // Where did the error occur?
  INTEGER, INTENT(IN), OPTIONAL :: alloc_status

  allocated_memory = allocated_memory + n_elements * BIT_SIZE(mold) / 8  // Add to counter
  max_allocated_memory = MAX(max_allocated_memory, allocated_memory)
  IF (allocated_memory > max_usable_memory) THEN  // We are getting too tight
    CALL MemoryUsage()
    CALL NonCriticalError("Allocated memory has exceeded the usable capacity")
  END IF

  IF (PRESENT(alloc_status)) THEN
    IF (alloc_status /= 0) THEN
      CALL MemoryUsage()
      IF (PRESENT(caller)) THEN
        CALL CriticalError(message = "Memory could not be (de)allocated", caller = caller)
      ELSE
        CALL CriticalError(message = "Memory could not be (de)allocated")
      END IF
    END IF
  END IF
END SUBROUTINE  // RecordAllocation
```
Now we make specific instances of the generic routine:

\[
\text{AllocationMonitoring} \equiv
\begin{align*}
\text{_Record_De_Allocation}(\text{INTEGER}, \text{i\_byte}) \\
\text{_Record_De_Allocation}(\text{INTEGER}, \text{i\_short}) \\
\text{_Record_De_Allocation}(\text{INTEGER}, \text{i\_sp}) \\
\text{_Record_De_Allocation}(\text{INTEGER}, \text{i\_dp}) \\
\text{_Record_De_Allocation}(\text{REAL}, \text{r\_sp}) \\
\text{_Record_De_Allocation}(\text{REAL}, \text{r\_dp}) \\
\text{_Record_De_Allocation}(\text{LOGICAL}, \text{l\_short}) \\
\text{_Record_De_Allocation}(\text{LOGICAL}, \text{l\_word}) \\
\text{_Record_De_Allocation}(\text{CHARACTER}, \text{c\_ascii})
\end{align*}
\]

**subroutine MemoryUsage()** // If the processor allows, use a real memory allocation monitor

**implicit NONE**

**write(unit = message_print_unit, fmt = *)** "At present allocated: ", allocated\_memory, " bytes, maximum allocated at one time: ", max\_allocated\_memory, " bytes, still available: ", max\_usable\_memory \- allocated\_memory

**write(unit = message_log_unit, fmt = *)** "At present allocated: ", allocated\_memory, " bytes, maximum allocated at one time: ", max\_allocated\_memory

**end subroutine MemoryUsage**

This code is used in section 1.

I overload the above allocation monitor routine generically with all possible kinds of mold arguments:

\[
\text{GenericInterfaces} \equiv
\begin{align*}
\text{interface RecordAllocation} \\
\text{module procedure RecordAllocation\_i\_byte} \\
\text{module procedure RecordAllocation\_i\_short} \\
\text{module procedure RecordAllocation\_i\_sp} \\
\text{module procedure RecordAllocation\_i\_dp} \\
\text{module procedure RecordAllocation\_r\_sp} \\
\text{module procedure RecordAllocation\_r\_dp} \\
\text{module procedure RecordAllocation\_l\_short} \\
\text{module procedure RecordAllocation\_l\_word} \\
\text{module procedure RecordAllocation\_c\_ascii}
\end{align*}
\]

**end interface**

This code is used in section 1.
2 Formatting rules for HPF/F90 files

These are just same auxiliary formatting rules and useful macros I use from time to time.

```
@m GENERICINTERFACE(generic_name, ...)  
   INTERFACE generic_name  
   MODULE PROCEDURE #.  
END INTERFACE generic_name  
@m DECLARE_L_WORD(...)  
   INTEGER :: #.  
@m DECLARE_L_WP(...)  
   INTEGER (kind = i_wp) :: #.  
@m DECLARE_R_WP(...)  
   REAL (kind = r_wp) :: #.  
@m DECLARE_R_SP(...)  
   REAL (kind = r_sp) :: #.  
@m DECLARE_R_DP(...)  
   REAL (kind = r_dp) :: #.  
@m FULLEXTENT(rank) : DO (DIM, 2, rank) { ; }  
@m VARSequence(variable, start, end)  
   variable #: start : DO (DIM, $eval(start + 1), end) { , variable@DIM }  
@m NESTEDLOOPSTART(variable, array, rank)  
   DO (DIM, rank, 1, -1) { DO variable@DIM = LBND(array, DIM), UBND(array, DIM) }  
@m NESTEDLOOPEND(rank) : DO (DIM, 1, rank) { END DO }  
@m DUMMY(...)  
@m DISPLAY_ARRAY(message, array)  
   IF (SIZE(array) <= 20) THEN  
      WRITE(message_print_unit, "(A)") message  
      WRITE(message_print_unit, "(20d5.2)") array  
   END IF
```