Exercise 3
Checkpoints and Debugging

Michele Guadagnini - Mt. 1230663

October 27, 2020

Abstract

This exercise is about the usage of checkpoints for debugging. The assignment requires to implement a checkpoint subroutine to be used in the program created for the third exercise of week 1 assignment with added documentation, comments and some error handling.

1 Theory

Checkpoints are a useful tool when debugging a program, especially if it is big and complex. They are usually implemented as function that receives as input a logical variable to toggle the printing. In this way one can activate this debug functionality by simply changing the value of this variable. Checkpoint functions usually receives in input also other variables in order to be able to print a different message at every different call. If for some reason the program stops unexpectedly, the programmer can look at the last printed message to identify where the code failed and find a solution.

Another approach that helps the programmer to avoid run-time errors is to set some *pre-* and/or *post-conditions*. These conditions are assumptions that the programmer does on the program; if one of these conditions is not met, then there is an error and some dedicated code is executed in order to deal with the error and prevent the program to halt brutally.

2 Code Development

2.1 Design and Implementation

The code is implemented in two files: *"Ex3-Guadagnini-CODE.f90"* contains the matrix-matrix multiplication module and the main program, while *"Ex3-Guadagnini-Checkpoint-CODE.f90"* contains the *CheckPoint* subroutine.

The program is built to receive the shapes of the matrices as command arguments. Let C be the matrix resulting from the multiplication between A and B ($C = AB$); then the order for the size values is: # rows of A, # columns of A, # rows of B and # columns of B. It also receives a logical flag to activate the debug functionality, i.e. activate the printing of the subroutine *CheckPoint*.

An example of the command with all the arguments used is:

```
./MatrixTest.x 20 15 15 20 .TRUE.
```

Listing 1: Example command

The program can also be run with the simple command without arguments; in this case the matrices are set to be square with size 100 by calling a dedicated subroutine, *DefaultInit*. This subroutine is called also when the input arguments are invalid. To determine if the provided arguments are acceptable the program performs the following checks:

- firstly it counts the number of arguments and checks it to be 4, 5 or 0;
  - if # of arguments is 4 then it tries to read the matrices shapes through the intrinsic function *READ*. The reading success or failure is checked by the parameter *IOSTAT* of the function.
  - if # of arguments is 5 it does the same as the previous case but it tries also to read the logical flag for debugging.
  - if # of arguments is 0 it simply calls the default initialization.
  - if none of the previous case occurs then it prints an error message saying that the number of arguments is different from the expected value and calls the default initialization.
- it checks that the size of the matrices are strictly positive and that they are consistent with matrix-matrix multiplication. This is done by the subroutine *CheckShapes*.

The code in the main program executing all these checks is reported in the following listing:

```fortran
!   Checking # of input values and their consistence with variables type
    expargs = 4
    totargs = command_argument_count()
    IF (totargs == expargs) THEN
        DO idx = 1,totargs
            CALL get_command_argument(idx, args(idx))
        ENDDO
        READ(args,*,IOSTAT=ios) MatSizes
        IF (ios.ne.0) THEN  !if conversion of input fails, default initialization is called
            WRITE(*,"(A)") "Error: invalid input value provided"
            CALL DefaultInit(MatSizes)
        ENDIF
    ELSEIF (totargs == expargs+1) THEN
        DO idx = 1,totargs
            CALL get_command_argument(idx, args(idx))
        ENDDO
        READ(args,*,IOSTAT=ios) MatSizes,Debug
        IF (ios.ne.0) THEN
            WRITE(*,"(A)") "Error: invalid input value provided"
            CALL DefaultInit(MatSizes)
        ELSE
            CALL Checkpoint(Debug, "Input arguments accepted!", 157)
        ENDIF
    ELSEIF (totargs == 0) THEN
        CALL DefaultInit(Matsizes)
    ELSE
        WRITE(*,fmt="(A,I2,A,I2,A)") "Error: ", expargs," arguments expected, but ", totargs, " provided"
        CALL DefaultInit(MatSizes)
    ENDIF

!   Checking matrix dimensions to be positive and consistent with matrix-matrix multiplication
    CALL CheckShapes(MatSizes)
    CALL Checkpoint(Debug, "Matrices has consistent shapes.", 168)
```

Listing 2: Input checking

After all these checks the program allocates the memory for the three matrices, it initializes A and B with random numbers between 0 and 1 with double precision and performs the calculations.

The matrix-matrix multiplication module contains also two other subroutines:

- *PrintMatrix*, that prints a matrix if it contains less then 36 elements, otherwise it prints simply its shape;
- *PrintExecTime*, that checks the execution time to be positive and prints it on screen.

The subroutine *CheckPoint* receives as input the logical flag *Debug* and three optional variables:

- *Text*, a character array to be printed;
- *INumber*, an integer number used to pass the line number at which the *CheckPoint* subroutine is called;
- *RNumber*, a real variable to be printed.

It is called repeatedly in the main program with different messages to print. The subroutine implementation is reported in the following code listing:

```fortran
    SUBROUTINE Checkpoint(Debug, Text, INumber, RNumber)
        LOGICAL Debug
        CHARACTER(len=*), OPTIONAL :: Text
        INTEGER, OPTIONAL :: INumber     !optional line number variable
        REAL, OPTIONAL :: RNumber        !optional real variable

        IF (Debug .eqv. .TRUE.) THEN
            IF (PRESENT(Text) .and. PRESENT(INumber)) THEN
                WRITE(*,"(A,A,A,I4,A)") "--> ", Text, " [Line #", INumber, "]"
            ELSEIF (PRESENT(Text)) THEN
                WRITE(*,"(A,A)") "--> ", Text
            ENDIF

            IF (PRESENT(RNumber)) THEN
                WRITE(*,"(A,f8.5)") "--> ", RNumber
            ENDIF
        ENDIF

        RETURN
    END SUBROUTINE
```

Listing 3: CheckPoint subroutine

2.2 Debug and Test

The program has been tested by passing a wrong number of arguments and also by using different wrong values like float numbers for the size, negative numbers, characters and also numbers or characters to the logical debug flag. The program is able to catch these errors, print an error message and continue execution by using default values for size.

As an example in the following lines the outputs of some commands are reported:

```
$: ./MatrixTest.x 20 15 15 20 .TRUE.
   --> Input arguments accepted! [Line # 157]
   --> Matrices has consistent shapes. [Line # 168]
   --> Memory correctly allocated. [Line # 174]
   --> Matrices initialization complete. [Line # 180]
   Matrix A with shape: (     20,     15)
   Matrix B with shape: (     15,     20)
   --> First loop order multiplication successful. [Line # 190]
```

	1st loop order			**2nd loop order**			**MATMUL**		
Size	-O	-O2	-O3	-O	-O2	-O3	-O	-O2	-O3
100	0.0044	0.0019	0.0015	0.0019	0.0016	0.0004	0.0006	0.0006	0.0005
500	0.4929	0.2337	0.2173	0.1497	0.1383	0.0754	0.0337	0.0333	0.0347
1000	10.468	10.226	10.171	1.2102	1.1132	0.6206	0.2675	0.2580	0.2596
2000	98.487	93.288	93.293	9.4082	8.7364	5.4749	2.1231	2.1470	2.0659

Table 1: Algorithms timings for square matrices multiplication. Time is in seconds.

```
   Matrix C (1st loop order) with shape: (    20,    20)
   Exec. time [s]:   0.000056
   --> Second loop order multiplication successful. [Line # 198]
   Matrix C (2nd loop order) with shape: (    20,    20)
   Exec. time [s]:   0.000064
   --> MATMUL multiplication successful. [Line # 206]
   Matrix C (MATMUL function) with shape: (    20,    20)
   Exec. time [s]:   0.000052
   --> Execution is complete!
```

Listing 4: Example output with correct arguments. Lines starting with " − − > " are produced by the *CheckPoint* subroutine.

```
$ ./MatrixTest.x -4 2.3 -2.5 k
   Error: invalid input value provided
   The possible arguments are: # rows A, # cols A, # rows B, # cols B, (logical
    debug flag)
   Setting default values: size    100 square matrices with debug inactive
   Matrix A with shape: (   100,   100)
   Matrix B with shape: (   100,   100)
   Matrix C (1st loop order) with shape: (   100,   100)
   Exec. time [s]:   0.005859
   Matrix C (2nd loop order) with shape: (   100,   100)
   Exec. time [s]:   0.006431
   Matrix C (MATMUL function) with shape: (   100,   100)
   Exec. time [s]:   0.000491
```

Listing 5: Example output with wrong arguments.

3 Results

Three executable files have been created by compiling the code with three optimization flags: *-O*, *-O2* and *-O3*. The measured performances of the three multiplication methods are reported in Table 1 for different sizes of square matrices and with the different optimizations.

Looking at the table it is possible to see that 1^{st} *loop order* method and *MATMUL* function are quite insensitive to the optimization flags since timings vary very little. Instead, 2^{nd} *loop order* method halves the execution time passing from the *-O* flag to the *-O3* one. As expected, the *MATMUL* function is the fastest method, especially for big matrices.

There is also a huge difference between the two loop orders. The reason for this is that *FORTRAN* stores the matrices in memory in column-major order and so, because of how it is implemented, the 2^{nd} method causes less overwrites of the CPU cache.

Regarding the *CheckPoint* subroutine, as shown in the first example in the previous section, it works great by printing a message that helps the programmer to understand where the program has stopped in case something went wrong.

4 Self-evaluation

Things learned while completing this assignment are:

- what it is and how to implement a Checkpoint subroutine;
- how to define and use *OPTIONAL* variables;
- how to use the *IOSTAT* parameter of the intrinsic function *READ* to check if the reading operation was successful or not;

To improve this exercise solution one can create a subroutine to manage the input arguments in a better and clearer way instead of having nested *IF ELSE* statements in the main program. Also, it would have been better to take not the single execution time to test the performance, but run the program repeatedly and take the mean of the timings, since they are dependent on the load of the system, that is not constant, and also on the particular initialization of the matrices.