Quantum Information and Computing

prof: *Simone Montangero*

Report of exercise 3

Alberto Bassi

October 26, 2020

Abstract

The aim of this exercise is to write a debugging module, which contains a logical type for setting the debugging mode ON or OFF, and some subroutines to check the correctness of the program's status. In particular, we shall specialize it for the matrix-matrix multiplication seen in week 1.

Theoretical introduction

Given two matrices $A \in M_{n,m}(\mathbb{R})$, $B \in M_{m,k}(\mathbb{R})$, then the product $C = AB \in M_{n,k}(\mathbb{R})$ for each entry reads

$$C(i,j) = \sum_{s=1}^{m} A(i,s) \cdot B(s,j) \quad \forall i = 1, \dots, n \quad \forall j = 1, \dots, k \,. \tag{0.1}$$

Since Fortran saves a generic matrix in a vector column by column (that is to say we will find the first column in the first positions, then the second, the third and so forth), the standard row par column loop for computing the matrix-matrix product is inefficient, since each time the program has to jump between positions which are not neighbours in the memory. Then, it appears to be convenient to perform the transposition of the first matrix first, which requires a negligible amount of time, and then compute the product column by column respecting the order in which the matrices are saved in the memory. So, the second loop for multiplication reads as follow:

$$C(i,j) = \sum_{s=1}^{m} A^T(s,i) \cdot B(s,j) \quad \forall i = 1, \dots, n \quad \forall j = 1, \dots, k \,. \tag{0.2}$$

After implementing a debugging module in order to check in runtime the status of the program, we will multiply two matrices in three different ways: the standard loop, the column par column multiplication and the built-in function MATMUL. The main program consists of two parts. In the first we will use the debugging module for multiplying two matrices of given sizes filled with random numbers. Then, we will perform an one hour simulation in order to compare the computation times for the three methods for square matrices as the size increases.

Code Development

The code contained in "debug.f" is divided into three parts. The debugging module, which includes subroutines for debugging, the multiplication module, in which we write the subroutines for matrix-matrix multiplication in the three different ways we have seen in week 1, and the main program.
The first thing we need in our debugging module is to implement a subroutine for initializing the debug logical variable. It asks the user if he wants to start the debugging mode and recognizes the input. If the answer is yes(y), then the debug variable is set to *.true.* and the other subroutines will be working. Otherwise, the program will be not debugged while running.

```
1         subroutine init_debug(debug)
2            implicit none
3            character :: yes, no, var
4            logical :: debug
5            yes="y"
6            no="n"
```

```
            print *, "Do you want to start the debugging mode?[y]/[n]"
            read *, var
            if (var.eq.yes) then
               debug=.true.
               print *, "=================="
               print *, "Debugging mode: ON"
               print *, "=================="
            else if(var.eq.no) then
               debug=.false.
               print *, "==================="
               print *, "Debugging mode: OFF"
               print *, "==================="
            else
               print *, "ERROR: insert a valid character"
               print *, "Program aborted"
               stop
            endif
         end
```

Then, debugging module includes some subroutines for matrix-matrix multiplication debugging. At first, we would like to check if the two matrices have positive dimensions. If this check is OK, then *check_dimensions* returns a positive feedback, while otherwise it stops the program and sends a message of error.

```
         subroutine check_dimensions(nn,mm,myname,debug)
            implicit none
            integer :: nn,mm
            logical :: debug
            character*10:: myname
            myname=trim(myname)
            if(debug.eqv..true.) then
               if (nn.ge.1 .and. mm.ge.1) then
                  print *,  "================================="
                  print *,  "Check dimensions of ",myname," :OK"
                  print *,  "================================="
               else
                  print *, "====================================="
                  print *, "Check dimensions of ",myname," :ERROR"
                  print *, "Proggram aborted"
                  print *, "====================================="
                  stop
               endif
            endif
         end
```

Second, we would like to check if the dimensions of the two matrices match correctly for multiplication, i.e. COL(matrix1)=ROW(matrix2). Even in this case, if the check is positive, the subroutine *match_dimensions* sends a message of no errors, otherwise claims an error and stops the program to run.

```
         subroutine match_dimensions(mm_1,nn_2,debug)
            implicit none
            integer :: mm_1,nn_2
            logical :: debug
            if (debug.eqv..true.) then
               if(nn_2.ne.mm_1) then
                  print *, "Dimensions matching: ERROR"
                  print *, "Program aborted"
                  stop
               else
                  print *, "Dimensions matching: OK"
               endif
            endif
         end
```

Moreover, a check on the successfully allocation of memory is needed. The subroutine *check_allocation* performs this checking and stops the program if the allocation fails.

```
         subroutine check_allocation(myname,stat,debug)
            implicit none
            integer :: stat
            character*10 :: myname
            logical :: debug
            if(debug.eqv..true.) then
               if(stat.ne.0) then
                  print *, "Allocation of ",myname,": FAILED"
                  print *, "Program aborted"
                  stop
               else
```

```
                print *, "Allocation of ",myname,": SUCCEEDED"
            endif
        endif
    end
```

Then, the subroutine *check_diff* checks if two matrices are equal element by element, provided that their dimensions (n, m) are already proved as equal. However, due to finite precision of Fortran, we have to set a threshold (here 0.00001). Let us say that the two matrices are different if at least one among the nm differences $mat1(i, j) - mat(i, j) > threshold$. This is valid as long as the entries of the two matrices are not too big (due to the problem of computing the difference of two big numbers in finite precision). As always, a message is provided and the program stops if there is a difference (in fact, by means of this subroutine we have to check whether the outcome of mat-mat multiplication is different for different methods).

```
    subroutine check_diff(A,B,nn,mm,nameA,nameB,treshold,debug)
        implicit none
        integer :: nn,mm
        real, dimension(nn,mm) :: A,B
        logical :: debug
        character*10 :: nameA,nameB
        real :: treshold,diff
        integer :: ii,jj
        integer :: counter
        treshold=0.00001
        counter=0
        if(debug.eqv..true.) then
            do jj=1,mm
                do ii=1,nn
                    diff=A(ii,jj)-B(ii,jj)
                    if(diff.gt.treshold) then
                        counter=counter+1
                    endif
                enddo
            enddo
            if(counter.eq.0) then
                print *, "Check equality: OK"
                print *, nameA, " and ",nameB,"are equal"
            else
                print *, "Check equality: ERROR"
                print *, nameA, " and ",nameB,"are different"
                print *, "Program aborted"
                stop
            endif
        endif
    end
```

At last, in the module debugging we implemented the subroutine *check_time*. It accepts as input a time threshold and a starting time along with the debug variable. Every time it is called, this subroutine checks whether the amount of time lasted since the starting is lower than the time threshold. Otherwise, it sends a message of error ("Computation time exceeded") and stops the program.

```
    subroutine check_time(start,treshold,debug)
        implicit none
        logical debug
        real :: start, finish,treshold
        if(debug.eqv..true.) then
            call cpu_time(finish)
            if(finish-start .gt. treshold) then
                print *, "Computation time exceeded"
                print *,"Ran in: ",finish-start
                print *, "Program aborted"
                stop
            endif
        endif
    end
```

As already stated, the module multiplication (here not reported due to the limited space) includes the three subroutines for the three mat-mat multiplication methods seen so far. All of them accept as input the two matrices to be multiplied along with their dimensions and compute the multiplication, storing the result and returning the computation time. We point out that a checking on the computation time is directly implemented inside the subroutines and the threshold set to 300 seconds. The computation is checked every time a new value of the product of obtained.

The main program (not included due to the limited space) works as follow. The user is asked to active the debugging mode. Then, in both cases, he is asked to insert the dimensions of the first matrix. Then, the

program performs the checking on the dimensions and stops itself if any error is encountered. After repeating this procedure for the second matrix, the program checks the dimension matching and stops the running with a message of error if there is no matching. Then, the memory allocation of the two input matrices and the three output matrices(each for each method) is checked. If the memory allocation succeeds, the matrices are filled with random numbers taken from a uniform distribution in $[0, 1]$ (so there are no problem when checking the differences). The computation times are printed on the terminal and the checking on the differences between the matrices are performed.
Moreover, the program is able to monitor the performances of the three algorithms for mat-mat multiplication. The user is asked if he wants to continue and the total computation time of the simulation has to be provided. The computation time is evaluated for the three methods for two square matrices of size n^2, until either the total computation time exceeds the threshold inserted by the user or at least one among the mat-mat multiplications exceed the built-in threshold. This assures us that the program will eventually stop. Staring from 1, each time n is incremented by 50 and the matrices are initialized with 1 in each entry. At the end, the total computation time, which is greater or equal the time inserted, is returned on the terminal if the computation does not stop due to the exceeding time of a multiplication function.

Results

We set the debugging mode ON and we ask for the multiplication of two 2x2 matrices, then we get:

```
Do you want to start the debugging mode?[y]/[n]
y
 ==================
 Debugging mode: ON
 ==================
 Insert the dimensions (n,m) of A
2,2
 ==================================
 Check dimensions of A          :OK
 ==================================
 Insert the dimensions (n,m) of B
2,2
 ==================================
 Check dimensions of B          :OK
 ==================================
 Dimensions matching: OK
Allocation of A         : SUCCEEDED
 Allocation of B         : SUCCEEDED
 Allocation of C1        : SUCCEEDED
 Allocation of C2        : SUCCEEDED
 Allocation of C3        : SUCCEEDED
 Standard mult. runs in:   1.60001218E-05
 Column par column mult. runs in:   1.50001142E-05
 The built_in mult. runs in:   3.00002284E-06
 Check equality: OK
 C1         and C2        are equal
 Check equality: OK
 C2         and C3        are equal

Do you want to monitor the performance?[y]/[n]
n
 Program aborted
```

We see that it works properly. The dimensions and the matching are correctly checked and the allocations of the memory succeeded. Moreover, the three algorithms compute correctly the mat-mat multiplications.
If, however, we ask for the multiplication of a 2x3 with a 4x2 matrix, then we get:

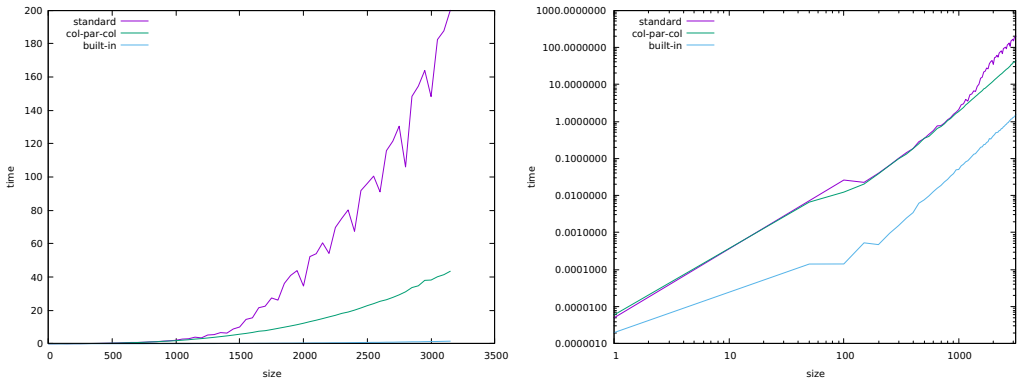
```
ab77@alberto:~/Documents/Quantum_Information/Report_week3/ex3$ ./debug.out
 Do you want to start the debugging mode?[y]/[n]
y
 ==================
 Debugging mode: ON
 ==================
 Insert the dimensions (n,m) of A
2,3
 ==================================
 Check dimensions of A          :OK
 ==================================
 Insert the dimensions (n,m) of B
4,2
```

```
 ==================================
 Check dimensions of B          :OK
 ==================================
 Dimensions matching: ERROR
 Program aborted
```

We see that the program correctly recognizes the mismatching of the two matrices and aborts. Even in this case, it seems to be working properly.
At last, we compiled "debug.f" with the optimization flag -O3 in "debug.out" in order to compare the performances of the three algorithms.

```
Do you want to monitor the performance?[y]/[n]
y
 Insert the computation time
3600
...
 Computation time exceeded
 Ran in:    3847.39230491
 Program aborted
```

The results for a simulation[1] of one hour are reported in **??**.



(a) Normal scale (b) Log-scale

Figure 1: The computation time vs the matrix size for three mat-mat multiplication algorithms with the -O3 optimization flag.

We can appreciate the polynomial growth for the computation time (which should go approximately as n^3). This polynomial behaviour is well emphasized in the log-scale graph on the right. We notice that all the three algorithms scale with approximately the same exponent, but different scaling constant. That is to say they goes like kn^α with approximately the same $\alpha \approx 3$, but with different k. However, a better result would be obtained with a size increasing equispaced in the logarithm.

Self Evaluation

In this exercise we have learned the importance of building a good debugging module, which can be very useful for checking the desired computations correctness. We have used this module for debugging the simple matrix-matrix multiplication algorithms seen in week 1. In particular, these three subroutines have been moved to a different module. It is very useful because it allows us to reuse them in different programs.
There are a few things that can be improved. First, even the type of the input dimensions must be checked (clearly, matrices of real dimensions do not exist). We did non succeeded in finding a suitable function that does this (*kind* only returns the precision of the integer type). However, Fortran itself recognizes when a real number is provided for an integer type sending a message of error. Second, while monitoring the performances a matrix of size n^2 is allocated, filled with ones, used and deallocated every single step. Clearly, this procedure slows down the program quite a lot. So, a slightly modification would be to initialize once and for all a matrix of size, let us say, 3000^2 and then performing the multiplications by running along its sub-matrices. Moreover, the increasing in the size of the matrices should be done equispaced in the logarithm in order to appreciate better the differences in growth. In the end, another problem arose. Even though the products are correctly computed when there is dimension matching, when there is not and the debugging mode is OFF the computation is performed anyway without Fortran errors.

[1] Performed by a MSI Leopard Pro (2017).