MASTER'S DEGREE IN PHYSICS
Academic Year 2020-2021

QUANTUM INFORMATION

Student: Giorgio Palermo
Student ID: 1238258
Date: November 2, 2020

EXERCISE 4

In this report I will describe the details on how I tested the performance of my machine relative to matrix multiplication using Fortran subroutines and Python scripts; I will briefly evaluate trend of the data I obtained for matrices of sizes spanning from 10 to 2500 units.

1 Theory

The basic concepts I used to build this program are relative to Fortran functions and subroutines, to the Fortran input-output procedures relative to text files and to the communication between Fortran programs and bash. I also used some basic commands of Python scripting language, including functions from NumPy and Os modules.

Code Development

To develop the code I used to solve this exercise I started from the `MatTest` subroutine I implemented for ex2 and I modified it. In the first place `MatTest` performed a matrix multiplication test in an almost automatic mode: given the maximum size of the matrix it tested the computation time for different algorithms increasing the matrix size by 100 units steps. For this exercise the test couldn't be automatic anymore: the matrix size had to be specified arbitrarily iteration by iteration. For this reason I implemented a new version of `MatTest` (reported below) with the following characteristics:

- it measures the computation time for a $N \times N$ matrix multiplication between random matrices, with N given as input
- it outputs the results on different files for each method; the file name is passed as input; an optional live output of the results is implemented
- the result is passed to the calling program using a one-dimensional array containing matrix size and time results for each iteration.

```fortran
subroutine MatTest(filename, size_input, verbose, result)
        use Functions
        use Debug
        implicit none
        ! Local scalars
        character(*), intent(in) :: filename
        character(*), intent(in) :: verbose
        character(len=50) :: filename_in, filename_alpha, filename_bravo, message
        double precision, intent(in) :: size_input
        integer :: size
        integer :: nn=5, ii=0, status
        integer :: oo=6 ! to suppress screen output set oo=something
        double precision :: start=0, finish=0, sum=0
        ! Local Arrays
        double precision, dimension(4) :: result
        double precision, dimension(:,:), allocatable :: A,B,C,C1,C2
        double precision, dimension(1,3) :: time
        character(len=30), dimension(:), allocatable :: args
        size=floor(size_input)
        ! Select if verbose
        select case(verbose)
        case("y")
            oo=6
        case("n")
```

```
            oo=3456
        case default
            oo=3456
        end select

        write(oo,*)
        write(oo,*) "   *** Matrix multiplication test program ***   "

        ! filename_in = filename
        filename_in = filename //"_ByRows"// ".txt"
        filename_alpha = filename //"_ByCols" // ".txt"
        filename_bravo = filename // "_Intrinsic" // ".txt"

        ! Testing section
        open(unit=10,file=filename_in,action='write',position="append",status='unknown',iostat=status)
        open(unit=20,file=filename_alpha,action='write',position="append",status='unknown',iostat=status)
        open(unit=30,file=filename_bravo,action='write',position="append",status='unknown',iostat=status)
        write(oo,*)
        write(oo,*) "Running test..." ! courtesy message
        write(oo,*) "Size    ", "     ByRows[s]    ","      ByCols[s]    ","     Intrinsic[s]   "

        allocate(A(size,size),B(size,size),C(size,size),C1(size,size),C2(size,size))
        call random_number(A)
        call random_number(B)

        call cpu_time(start)
        call LoopMult(A,B,C)     ! by rows
        call cpu_time(finish)
        time(1,1)=finish-start

        call cpu_time(start)
        call LoopMultColumns(A,B,C1) ! by columns
        call cpu_time(finish)
        time(1,2)=finish-start

        call cpu_time(start)
        call IntrinsicMult(A,B,C2)  ! Intrinsic
        call cpu_time(finish)
        time(1,3)=finish-start
        deallocate(A,B,C,C1,C2)

        write(oo,'(i5,3G15.5)') size,time(1,1),time(1,2),time(1,3)
        write(10,'(i10, G15.5)') size, time(1,1) ! Write output on different files
        write(20,'(i10, G15.5)') size, time(1,2)
        write(30,'(i10, G15.5)') size, time(1,3)
        result(1) = dble(size)
        result(2) = time(1,1)
        result(3) = time(1,2)
        result(4) = time(1,3)
        write(oo,*) "Done"
        close(10,iostat=status)
        close(20,iostat=status)
        close(20,iostat=status)

    end subroutine MatTest
```

To import the grid of points containing the matrix sizes to test I implemented the `ReadGrid` subroutine: this piece of code reads as inputs the name of the file to read from (`filename`) and the grid size (`grid_dim`); the output is a one-dimensional array containing the points to test:

```
subroutine ReadGrid(filename,grid_dim, grid)
        implicit none
        character(*), intent(in) :: filename
        character(len=100) :: msg
        integer, intent(in) :: grid_dim
        double precision, dimension(:), allocatable,intent(out) :: grid
        integer :: ii=1, ios

        open(unit=100,file=filename,iostat=ios,iomsg=msg)
        if(ios/=0) then
```

```
            write(*,*) msg
            stop
        end if

        allocate(grid(grid_dim))
        read(100,*,iostat=ios, end=997,iomsg=msg) grid
        997 print*, "Grid read with exit status: ", ios
        ! print*, grid(:)
        close(100)
    end subroutine ReadGrid
```

The real testing is implemented in `program cos`. This program calls repeatedly the `MatTest` subroutine, giving as input a different value of N for each call. The grid is read from file using the `ReadGrid` function, while the output results are collected in a two dimensional array `results_all`. In order to be easily scripted with python, I wrote this program to take an input from bash (`filename`), which I used to pass to the program the name of the file to store the results in.

```
program cos
    use Functions
    use cosmod
    implicit none

    double precision, dimension(:,:), allocatable :: results_all
    double precision, dimension(:), allocatable :: grid
    double precision, dimension(4) :: result
    character(len=30), dimension(:), allocatable :: args
    integer grid_dim,ii,ios, num_args
    character(len=30) :: filename

    num_args = command_argument_count()
    allocate(args(num_args))
    call get_command_argument(1,args(1))
    print*, args(1)
    filename = "result"// trim(args(1))// ".dat"
    print*, filename
    grid_dim=10
    allocate(grid(grid_dim))
    allocate(results_all(grid_dim,4))

    call ReadGrid("grid.dat", grid_dim,grid)
    open(unit=78,file=filename,status="unknown",iostat=ios)
    write(*,*) "Testing matrix of size: "
    do ii=1,grid_dim,1
        call MatTest("time", grid(ii), "n", result)
        write(*,*) floor(grid(ii))
        results_all(ii,:) = result(:)
        write(78,*) results_all(ii,:)
    end do
    write(*,*) "Done!"
    close(78)

end program cos
```

To test the effective computing performace of my machine with different optimization flags, I wrote a Python script which has the following characteristics:

- it defines a grid of points which range with exponential spacing from N_{min} and N_{max} arbitrarily chosen; this is made to have equally spaced points on the log-scaled graphs that will be discussed below
- saves the grid on file
- it calls the compiler for `cos.f90`
- it repeats the operation for all optimization flags available
- it calls a `gnuplot` script to plot results.

The Python script `program.py` is the following:

```
import os
import numpy
```

```python

fid=open("grid.dat","w+")   #Open file
numbers=numpy.logspace(1.,numpy.log10(2500),num=10) # Create the grid
numbers=numpy.floor(numbers)
for item in numbers:
    fid.write("%s\n" % item)
fid.close()

opt_flags = ["","-O1", "-O2", "-O3", "-Ofast"]
for item in opt_flags:      #cycle through opt flags
    comp_comm="gfortran cos.f90 -o cos.out "+item
    os.system(comp_comm)    #Call the compiler
    command = "./cos.out " + item
    os.system(command)  #Run the program

os.system("gnuplot plotres.gp") #Call gnuplot for graphs
```

This `gnuplot` script (`plotres.gp`) fits and plots all the computation times; this is done on different images for each optimization flag used. The results of the fits are displayed on terminal.

```
... line style definition ...
... set terminal and labels ...
f(x) = a*x**b # Fitting functions
g(x) = c*x**d
h(x) = e*x**p
...
fit f(x) "result.dat" using 1:2 via a,b
fit g(x) "result.dat" using 1:3 via c,d
# fit h(x) "result.dat" using 1:4 via e,p
plot "result.dat" using 1:2 title 'StdLoop' with points linestyle 1, \
     "result.dat" using 1:3 title 'InvLoop' with points linestyle 2, \
     "result.dat" using 1:4 title 'Intrinsic' with points linestyle 3,\
     f(x) with line linestyle 4,\
     g(x) with line linestyle 4,\
     # h(x) with line linestyle 4 # bad results! not plotted
... repeat for all opt flags ...
```

Results and self evaluation

Using Python script and `program cos` I tested the computation times for matrices with sizes spanning from 10 to 2500. I collected these data into graphs, which can be seen in figure 1 and figure 2. The first graph contains data collected from an unoptimized algorithm, while data on the remaining ones were collected from algorithms with increasing optimization.

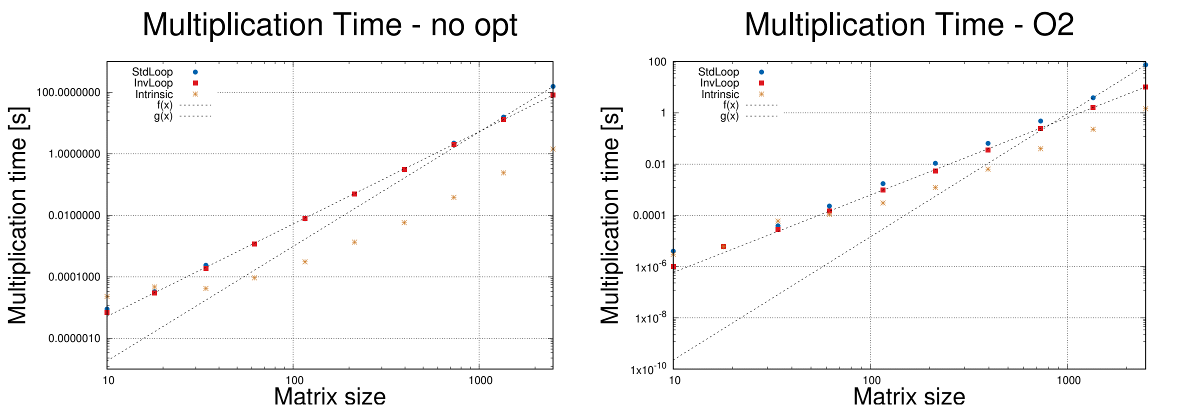

Figure 1: Computation time as a function of the matrix size, without opt flags and with -O2.

On graph I plotted power laws $f(x) = ax^b$, with a, b parameters retrieved from the data via fit; these power laws look like straight lines on these plots because of the double logarithmic scale. The

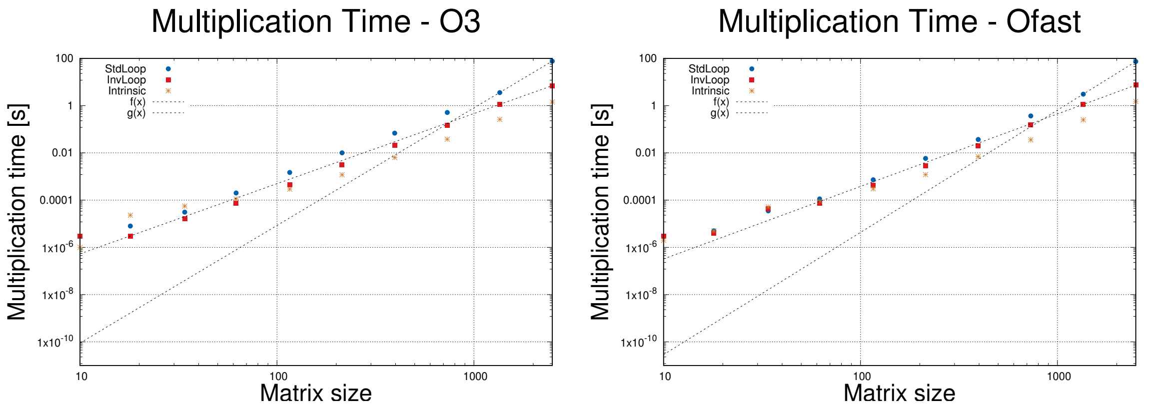

Figure 2: Computation time as a function of the matrix size, with -O3 and -Ofast.

agreement between data and the power law is good for the "InvLoop" method (non-optimized), while is poor for the others; fit results for "Intrinsic" multiplication method are not shown since this data present an evident nonlinear behavior in both log-log plots.
These graphs suggest that the computation times scale as power laws with the matrix size, at least for user-implemented methods. This statement might be verified building a more complex test involving repeated runs of the Python script, the elaboration of the data on a statistical base and the construction of confidence intervals for each data point.

Building this program I learned for the first time how to write a Python script and some ways to pass arguments between different languages.