QUANTUM INFORMATION REPORT
Exercise 4

Giovanni Piccolo - 1242806

November 3, 2020

Abstract

This report contains the description of the performance time tests of the matrix multiplication exercise written during the week 1 of the course. The code was written using Fortran subroutines, while the dimensions N of the matrices were automatically generated by a Python script. A second Python script was used to automatize fitting and plotting through Gnuplot.

Theory

The theoretical computational concepts used to achieve the goals of this exercise are the ones related to Fortran functions and subroutines, and to the Fortran input-output procedures relative to text files. We also wrote for the very first time two scripts in Python using functions from the Numpy, os and subprocess modules.

Code Development and Results

In this section we will describe and analyze the codes used to solve the exercise highlighting the most significant lines of the files `matrixmul.f90`, `matrixmul.py` and `timefit.py`.

Matrixmul.f90 (FORTRAN)

We start to develop the code used to solve the exercise from an implementation of the previous version of the `matrixmul.f90` program: the first question of *exercise 1* asked to perform a $N \times N$ matrix multiplication, where N had to be read from file. In order to tell to the Fortran code which matrix size we wanted to probe we used the following script:

```
if(COMMAND_ARGUMENT_COUNT() < 1) then
    (...)
  nn = 100
else
```

```
  call GET_COMMAND_ARGUMENT(1, nnchar)
  read(nnchar,FMT=*, IOSTAT=stat)nn
    if((stat.NE.0).OR.(nn <= 0)) then
        (...)
        nn = 100
  end if
endif
```

The purpose of those lines of codes is to check if there is a given input for the matrix size `nn`: if not the code sets it to the default one, namely `nn=100`. If the `IOSTAT` specifier value is different from zero or if the matrix dimension `nn` read from the code is equal or less than zero we fall back into an invalid state and the matrix dimension is set to the default one again.

The `matrixmul.f90` code then prints line by line all the seven methods used to compute the matrix multiplication (six of them are the permutations of the three matrix indices `ii,jj,kk`, while the last one is the intrinsic Fortran `matmul` function):

```
!mode = ["IJK", "IKJ", "JIK", "JKI", "KIJ","KJI"]
do mode=0,5
call cpu_time(start)
res = MAT_MODES(mat1,mat2,nn,mode)
call cpu_time(finish)
print*, modes(mode+1),finish-start
end do

!measuring intrinsic fortran multiplication speed
call cpu_time(start)
res = matmul(mat1,mat2)
call cpu_time(finish)
print*, finish-start
```

Debug and error functions are still used to check inputs and provide checkpoints.

Matrix_mul.py (PYTHON)

Once the previous program is compiled (using the '`-o matrixmul.x`' flag) we can start describing the second program: the second script takes as inputs three command lines arguments, which correspond to:

- $\mathbf{N_{min}}$: it's the lowest matrix size to probe;
- $\mathbf{N_{max}}$: it's the highest matrix size to probe;
- $\mathbf{N_{count}}$: it counts the number of sizes,

as we can see from the following lines of code. This type of coding was possible thanks to the built-in Python module `sys`.

```
import sys
(...)
if(len(sys.argv) >= 4):
  N_min = int(sys.argv[1])
```

```
  N_max = int(sys.argv[2])
  N_count = int(sys.argv[3])
else:
  N_min=50
  N_max=1500
  N_count = 50
(...)
```

If not enough arguments are passed to the Python script, the default values for N_{min}, N_{max}, N_{counts} are namely 1500, 50 and 50.

The matrix sizes are computed using a logarithmic scale,

```
(...)
import numpy as np
(...)
sizes = np.logspace(np.log10(N_min),np.log10(N_max +1),num=N_count
    ,dtype=int)
(...)
```

while the execution times of the six modes and of the Fortran intrinsic function are stored as reported here.

```
import subprocess
(...)
times = np.zeros((len(sizes),7))
fName = 'matrixmul.x'
modes = ['ijk','ikj','jik','jki','kij','kji']
for i,sz in enumerate(sizes):
  print('%.2f' % float(i*100/len(sizes)),'%')
  print('Matrix size:',sz)
  result = subprocess.run(['./'+fName,str(sz)], stdout=subprocess.
    PIPE)
  splits = result.stdout.decode('utf-8').lstrip().split('\n')
  times[i,:]=np.array([float(tm) for tm in splits[0:7]])
(...)
```

The presence of the `subprocess` module is needed in order to launch the Fortran program with proper command lines, then the output is read and stored into different files. In the end, a *gnuplot* script plots all the results together in a final plot using a *for* loop, as we can see from Figure 1 ('*time_plot.png*'):

```
set xlabel "Matrix dimension"
set ylabel "Log execution time [s]"
array fnames[7]
fnames[1]="modeijk"
(...)
fnames[6]="modekji"
fnames[7]="intrinsic"
plot for [i=1:7] fnames[i] using 1:2 title fnames[i] with
    linespoints lw 2
set logscale y
set term png size 1024, 720
set output "time_plots.png"
replot
set term x11
```

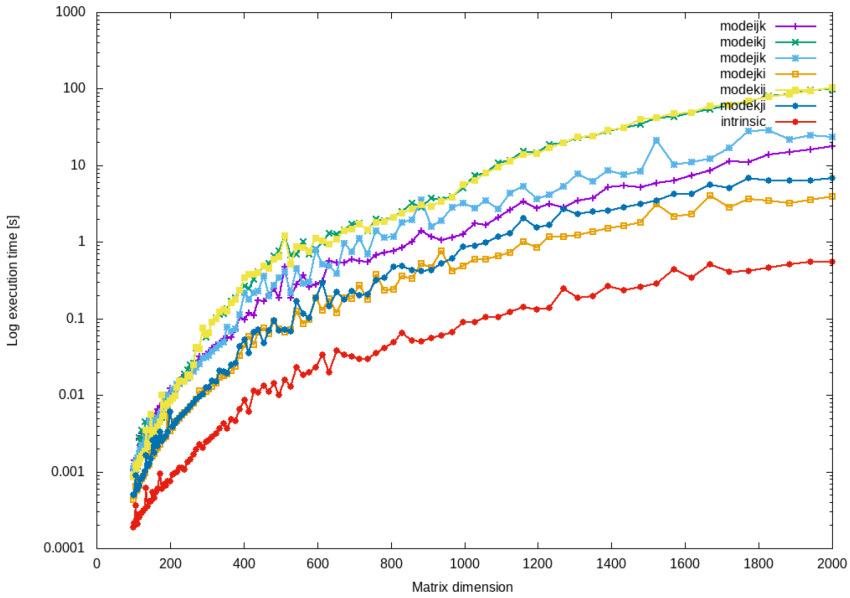

Figure 1: *Plot of the logarithmic behaviour of the execution times of the code as a function of the matrix dimension N.*

From Figure 1 we can also deduce that - as we expected - the Fortran intrinsic function `matmul` is way better scalable as the matrix dimension increases than the other plots.

Timefit.py (PYTHON)

The last Pyhon script was written to automatize fitting and plotting procedures by subsequently launching a *gnuplot* script (called '*timefit.plt*') directly from the `timefit.py` python code.

The *gnuplot* script reads from a file called *'file_to_fit.txt'* the name of the file containing the data to fit and outputs the parameters of the fit.

These lines of code make the procedure *automatic*: the modes' names are encoded in `fnames` and then, by using the subprocess module, the script launches the gnuplot one.

```
(...)
fnames = ["modeijk", "modeikj", "modejik", "modejki", "modekij", "
    modekji", "intrinsic"]
for fname in fnames:
  fl = open("file_to_fit.txt", "w+")
  fl.write(fname)
  fl.close()
  subprocess.call(["gnuplot",'timefit.plt'])
```

The fitting function used in the *gnuplot* script was a third order polynomial function `f(x)=a+b*x+c*x**2+d*x**3`.

In Figure 2 we show one of the fits we obtained as an example:

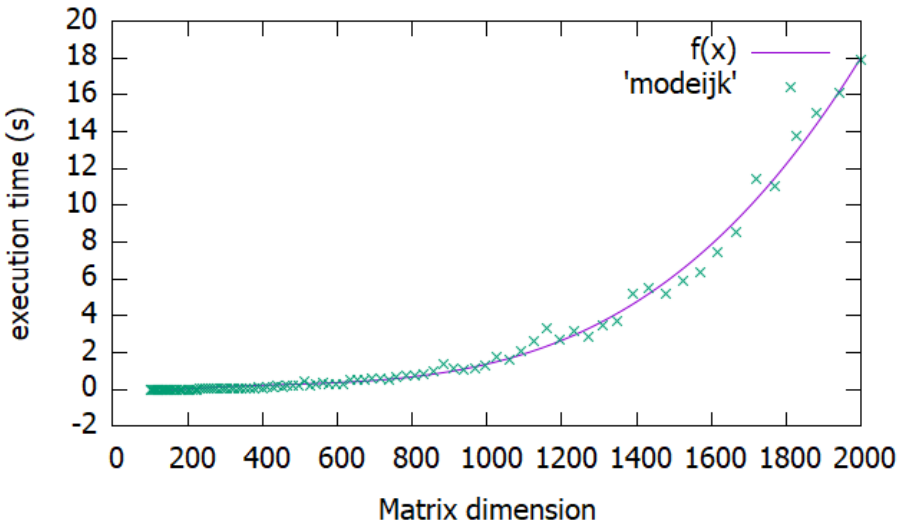

Figure 2: *Example of one of the automatic fits we obtained from the timefit.plt script. In the x axe we can see the matrix dimension N, while the y axe is the execution time in seconds, the value of the reduced χ^2 parameter is equal to 0.09, the agreement between the power law and the data is good..*

Self Evalutation

Writing this program in a very first time I have found some technical difficulties in coding a Python script but in the end I have learnt a way to pass arguments between different languages.