Quantum Information and Computing

prof: *Simone Montangero*

Report of exercise 4

Alberto Bassi

November 2, 2020

Abstract

The purpose of this week is to write a python script to control a Fortran program aimed to compare the different efficiency in multiplication of two square matrices for the three methods seen in week 1. Moreover, the scripts is able to run a Gnuplot script to make fits and graphs.

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

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

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:

$$C(i,j) = \sum_{s=1}^{m} A^T(s,i) \cdot B(s,j) \quad \forall i = 1, \dots, n \quad \forall j = 1, \dots, k \ . \tag{2}$$

We will perform[1] a nearly 27 minutes simulation in order to compare the efficiency of the three algorithms (the third is the built-in function MATMUL). As we will see, we will compute the matrix-matrix multiplication for the three different methods four matrices each time through parallelization in Python. Each time we initialize the matrices with random numbers in single precision (4 bytes each). Since the last matrices which are initialized simultaneously have dimensions of 3400,3500,3600 and 3700 respectively, it is straightforward to compute the maximum amount of memory needed for initialization. We get

$$Memory_{init} = (3400^2 + 3500^2 + 3600^2 + 3700^2) * 4/10^6 \simeq 201.84 \ Mb \ . \tag{3}$$

Since the Ram of the computer's simulation is about 16 Gb, we see that the memory required for initialization is not too big in comparison. In fact, the maximum size for a square matrix to be initialized (and only initialized) in single precision is:

$$Size = \sqrt{16 \cdot 10^9/4} = 63245 \ , \tag{4}$$

where in double complex precision (16 bytes) it would be 31622.
Clearly, we have also to take into account the memory required for calculations. Since for each loop we have to initialize 3 matrices (A,B such that AB=C), we get

$$Memory = Memory_{init} * 3 \simeq 605.5 \ Mb \ . \tag{5}$$

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

Code Development

In order to build a suitable dataset, we have written a python script that computes four parallel matrix-matrix multiplication each time. Even though our Cpu has 8 cores, it has been noticed that we gained in performance by only using four of them (i.e. four processes). Then, this loop is repeated as many times as the user decides by inserting the minimum dimension N_min, the maximum dimension N_max and the number of steps N_step. With this method, the computation is not terminated until all the groups of four are computed. For instance, if it was that $N\_min = 100$,$N\_max = 150$ and $N\_step = 50$, then the program would compute the matrix-matrix multiplication for sizes 100,150,200 and 250.
The program for matrix-matrix multiplication is "multiplication.f", here not reported due to space limitations. It is a simplified version of last week program. It consists of module named multiplication, which includes the three subroutines for multiplication algorithms and a main program, which reads the matrix sizes from a txt file, initializes them with single precision uniform random numbers in [0,1] and calls the subroutines in the module above, writing the computation time in different files according to the method used. Last, the memory is deallocated. We have compiled four executable "multiuplicationN.txt", where N=1,2,3,4 stands for the process, with the optimization flag -O3. Every one of them looks for the matrix size in the correspondent txt file "diemensionsN.txt", in order to avoid problems with the simultaneous opening of the same file from different processes. In the end, the total computation time and the plots (normal and log scale) are given. This algorithm gives us the power to add data if required, without modifying the already acquired ones. Through parallelization, we expect that the total computation time is much less than the sequential programming time (not one fourth because the time scales with the largest matrix in each group of four).

```python
#"mult_proc.py"
import multiprocessing as mp
import subprocess
import time
#We ask for the parameters
b=input("Insert N_min \n")
a=input("Insert N_max \n")
c=input("Insert N_step\n")
N_max=int(a)
N_min=int(b)
N_step=int(c)

def f(ii,jj):
  name="dimensions"+str(jj)+".txt" #name of the file in which to read the matrix size
  exe="./multiplication"+str(jj)+".out" #the executable to be called
  out=open(name,'w')
  out.write(str(ii))
  out.close()
  subprocess.call(exe)

start=time.time()
for kk in range(N_min, N_max,4*N_step):
  processes=[mp.Process(target=f,args=(kk+(jj-1)*N_step,jj,)) for jj in range(1,5)]
  [process.start() for process in processes]#we start the processes
  [process.join() for process in processes]#we do not go on in the cycle for until all the
    four processes are completed
finish=time.time()
print('Total computation time (s): ' +str(finish-start))
var=input("Do you want to plot and fit the results?[y]/[n]")
if var=='y':
  subprocess.call("gnuplot 'fit.plt'",shell=True)#make fits
```

After the simulation, the user is asked if he wants to fit the data. If so, the gnuplot script "fit.plt" is loaded. An extract of that file is reported below. The graphs are directly saved in .png files.

```gnuplot
#"fit.plt"
...
set fit logfile 'logfile.log' quiet errorvariables nocovariancevariables errorscaling prescale
    nowrap v5
f1(x)=a1+b1*x**c1 #polynomial function
f2(x)=a2+b2*x**c2
f3(x)=a3+b3*x**c3
GNUTERM = "qt"
...
#initial parameters
...
#checks on fit output
...
#initial parameters
...
```

```
fit f1(x) "std_mult.txt" u 1:2 via a1,b1,c1 #polynomial fit
fit f2(x) "clmn_mult.txt" u 1:2 via a2,b2,c2
fit f3(x) "built_in.txt" u 1:2 via a3,b3,c3
g1(x)=h1+k1*x #linear function
g2(x)=h2+k2*x
g3(x)=h3+k3*x
GNUTERM = "qt"
...
#initial parameters
...
#checks on fit output
...
#initial parameters
...
fit  g1(x) "std_mult.txt" u (log($1)):(log($2)) via h1,k1 #linear fit in log-log scale
fit  g2(x) "clmn_mult.txt" u (log($1)):(log($2)) via h2,k2
fit g3(x) "built_in.txt" u (log($1)):(log($2)) via h3,k3
set term png
set autoscale
set output "fit.png"
p "std_mult.txt" u 1:2 w p ls 1 ti "Standard-mult", f1(x) ti "" ls 1,"clmn_mult.txt" u 1:2 w p
     ls 2 ti "Col-par-col",  f2(x) ti "" ls 2, "built_in.txt" u 1:2 w p ls 3 ti "Built-in", f3
    (x) ti "" ls 3
set ylabel "Residual(s)"
set output "res_fit.png"
p "std_mult.txt" u 1:($2-f1($1)) w p ls 1 ti "Standard-mult", "clmn_mult.txt" u 1:($2-f2($1))
    w p ls 2 ti "Col-par-col", "built_in.txt" u 1:($2-f3($1))w p ls 3 ti "Built-in"
set output "log.png"
set ylabel "log(Time)"
set xlabel "log(Size)"
p "std_mult.txt" u (log($1)):(log($2)) w p ls 1 ti "Standard-mult",g1(x) ti "" ls 1, "
    clmn_mult.txt" u (log($1)):(log($2)) w p ls 2 ti "Col-par-col",g2(x) ti "" ls 2, "built_in
    .txt" u (log($1)):(log($2)) w p ls  3 ti  "Built-in", g3(x) ti "" ls 3
set ylabel "Residual(log(Time))"
set autoscale
set output "res_log.png"
set key right
p "std_mult.txt" u (log($1)):(log($2)-g1(log($1))) w p ls 1 ti "Standard-mult", "clmn_mult.txt
    " u (log($1)):(log($2)-g2(log($1))) w p ls 2 ti "Col-par-col", "built_in.txt" u (log($1))
    :(log($2)-g3(log($1))) w p ls 3 ti "Built-in"
#    EOF
```

Results

Parallel vs Sequential programming

We have compared the written at the very beginning sequential script "run.py" with the parallel programming script "mult_proc.py". The result for the matrix-matrix multiplication with seizes from 1 to 1000 with steps of 2 are reported below.

```
ab77@alberto:~/Documents/Quantum_Information/Report_week4/mult_processes$ python3 mult_proc.py
Insert N_min
1
Insert N_max
1000
Insert N_step
2
Total computation time: 197.5223844051361
Do you want to plot the normal scale?[y]/[n]y
Do you want to plot the log scale?[y]/[n]y
```

```
ab77@alberto:~/Documents/Quantum_Information/Report_week4/ex4$ python3 run.py
Insert N_min
1
Insert N_max
1000
Insert N_step
2
Total compuation time is: 533.2975504398346
```

We notice that we more than halved the time needed for the simulation. However, these data are not included in the analysis below. If he wanted, one could find them in the folder "old_file", which contains a previous simulation.

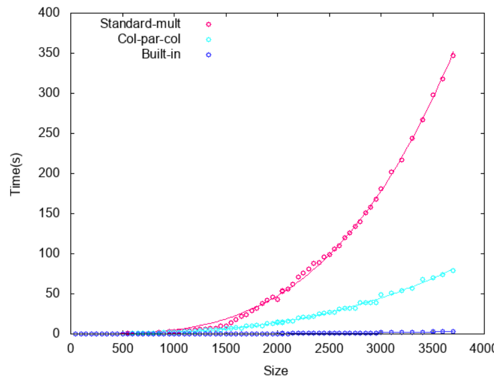
Polynomial growth

After proving the more efficiency of the parallel programming, we completed the simulation, which lasted less than 27 minutes, through various steps.

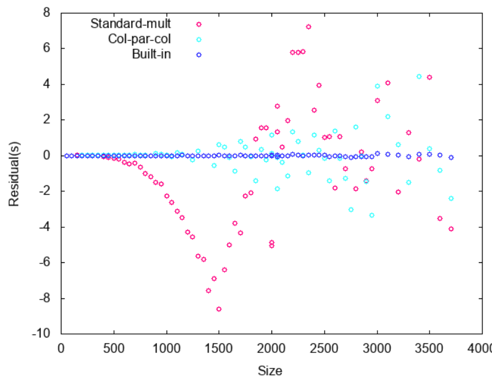
```
ab77@alberto:~/Documents/Quantum_Information/Report_week4/ex4$ python3 mult_proc.py
Insert N_min
50
Insert N_max
3000
Insert N_step
50
Total computation time (s): 881.628309209234
Do you want to plot and fit the results?[y]/[n]y
```

```
ab77@alberto:~/Documents/Quantum_Information/Report_week4/ex4$ python3 mult_proc.py
Insert N_min
3000
Insert N_max
3600
Insert N_step
100
Total computation time (s): 733.3130824565887
Do you want to plot and fit the results?[y]/[n]y
```

At first, we fit the computation time vs the matrix size with a polynomial $t = a + bN^c$, where t is the computation time, N is the matrix size and a, b, c three parameters. For each method we have different functions along with their coefficients. Since the time needed for fitting is lower if we choose initial values near the experimental ones, first we set $a1 = a2 = a3 = 0.0001$, $b1 = b2 = b3 = 1$ and $c1 = c2 = c3 = 3$, since we expect a polynomial increasing to the third power. In Figure 1 the polynomial fis are reported, while the fitting parameters are reported in Table 1.



(a) Fit plot.



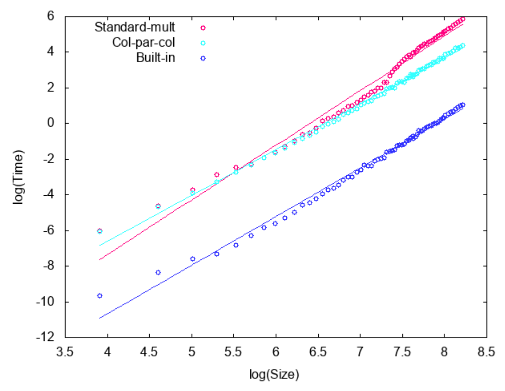
(b) Residual plot.

Figure 1: Multiplication time vs matrix size.

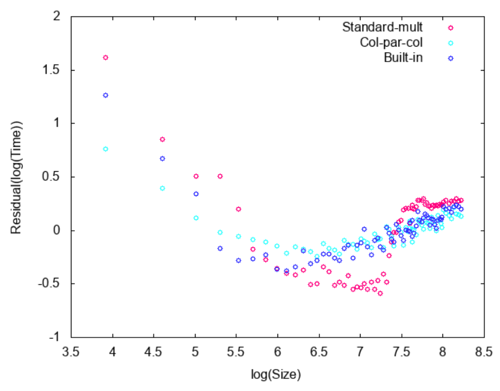
	a	b	c
Standard-mult	9.99e-06	9.16e-10	3.25
Col-par-col	1.14e-05	7.29e-09	2.82
Built-in	9.99e-06	3.35e-11	3.06

Table 1: Fit values for polynomial growth.

From the residual plot, we notice that the fit is good for the column par column method and for the built-in function MATMUL. Instead, the standard multiplication loop shows some sort of bias which ruins the growth. From Table 1, we see that all the three methods have nearly the same exponential factor, while they have different pre-factors. From this, we explain the more efficiency of MATMUL for low sizes. In fact, the column par column method seems to have the lowest growing exponential constant c.
However, for a better comparison a log-log scale viewing should be made. Thus, we have linearised the polynomial growth and got $\ln(t) = h + k \ln(N)$. We now expect $h \simeq -20$ and again $k \simeq 3$. In Figure 2 the linear fits are reported, while the fitting parameters are reporetd in Table 2.



(a) Fit plot.



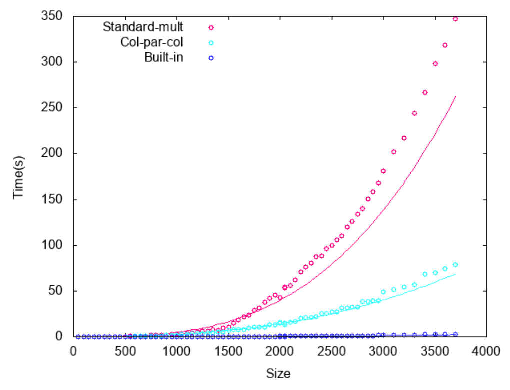
(b) Residual plot.

Figure 2: The log of multiplication time vs the log of matrix size.

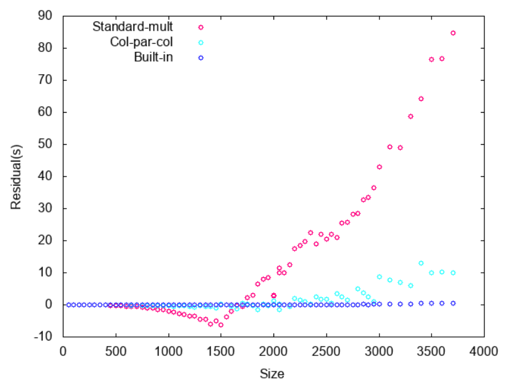
	h	k
Standard-mult	-19.60	3.06
Col-par-col	-16.87	2.57
Built-in	-21.61	2.73

Table 2: Fit values for linear growth in the log-log scale.

Even in this case, although MATMUL is the more efficient and stable algorithm in the range considered, column par column algorithm seems to scale with a lower exponential factor. Moreover, even though is not as stable as MATMUL in the range considered, it exhibits a greater stability than the standard algorithm, which is on the other hand more efficient for low sizes (≤ 140). Now, the bias of the standard multiplication is evident from the graph, at least for sizes greater than about 2000. In fact, if we invert the linear relation and plot the polynomial curves along with the residuals, we notice that the accordance for the first method is really bad, while it improves a little for the column par column multiplication and a great deal for MATMUL, which is thus the more stable algorithm overall.In Figure 3 the data are plotted along the polynomial functions obtained by inverting the linear relation.



(a) Fit plot.



(b) Residual plot.

Figure 3: Multiplication time vs matrix size.

Self Evaluation

In this week, we have learned how to use Python for running pre-compiled Fortran programs and Gnuplot's scripts. Also, we have learned how to write Gnuplot's scripts for making fits and plots automatically. Moreover, we have learned how to parallelize computation in Python through the library Multiprocessing. However, we have to pay attention while doing parallel calculus, because the memory required may easily exceed available resources.