Università degli Studi di Padova

Quantum Information and Computing 2021/2022

Prof. Montangero

Campesan Giulia

2027592

1st week assignment

November 7, 2021

Exercise 1: Setup

(a) **Create a working directory.**

(b) **Open emacs/your favorite editor and write your first program in FORTRAN.**
I just wrote a simple program that prints the 'Hello World!' string on the terminal:

```
program HelloWorld
    implicit none
  print *, 'Hello, World!'
end program HelloWorld
```

(c) **Submit a test job.**
What I get when compiling and executing the 'HelloWorld.f90' file is what expected:

```
(base) giulia@Air-di-Giulia week1 % gfortran HelloWorld.f90 -o HelloWorld.out
(base) giulia@Air-di-Giulia week1 % ./HelloWorld.out
 Hello World!
(base) giulia@Air-di-Giulia week1 %
```

Figure 1: 'HelloWorld.f90' execution

(d) **Connect to the cluster spiro.fisica.unpd.it via ssh and repeat the execution**
When connecting to the cluster via the command line
'ssh campesgi@spiro.fisica.unipd.it -oKexAlgorithms=+diffie-hellman-group1-sha1'
and executing the program 'HelloWorld.f90', the same output is obtained.

```
-bash-3.2$ gfortran HelloWorld.f90 -o HelloWorld.out
-bash-3.2$ ./HelloWorld.out
 Hello World!
-bash-3.2$
```

Figure 2: 'HelloWorld.f90' execution on cluster via ssh

Exercise 2: Number precision

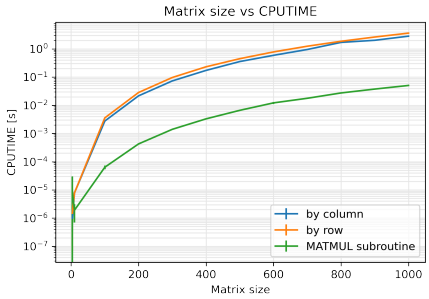
Integer and real numbers have a finite precision. Explore the limits of INTEGER and REAL in Fortran.

(a) **Sum the numbers 2.000.000 and 1 with INTEGER*2 and INTEGER*4**
I report underneath the result obtained when executing the program 'sum.f90' that carries out the required task:

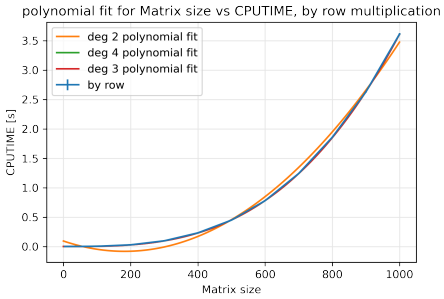
```
(base) giulia@Air-di-Giulia week1 % gfortran sum.f90 -fno-range-check -o sum.out
(base) giulia@Air-di-Giulia week1 % ./sum.out
 2*10^6+1 with integer*2 precision:     -31615
 2*10^6+1 with integer*4 precision:      2000001
 pi*10^32+sqrt(2)*10^21 in single precision:    3.14159278E+32
 pi*10^32+sqrt(2)*10^21 in double precision:    3.1415927578462317E+032
```

Figure 3: 'sum.f90' execution

When performing $2 * 10^6 + 1$ in integer*2 precision, the result we get is -31615: this is clearly wrong. In fact, the range we can encode in 2 bytes is [-32.768, 32.767], which $x = 2000000$ exceeds by far. Then, after encoding x as a binary number $x_2 = 11110100000010010000000$, the program truncates it and it just keeps its 16 LSBs $x_T = 1000010010000000$. Finally, the sum $x_T + 1 = 1000010010000001$ is performed, that returns exactly -31615 in signed 2's complement.
On the other hand, when performing the same operation with integer*4 variables, we get to the correct result: the range we can encode in that precision is $[-2^{31}, 2^{31} - 1]$, in which x is contained.



(a) n vs CPUTIME



(b) polynomial fit, by row multiplication

deg=2	deg=3	deg=4
$7.0 * 10^{-3}$	$8.1 * 10^{-6}$	$7.7 * 10^{-6}$

Table 1: Mean Squared Error for polynomial fit of n vs CPUTIME

(b) **Sum the numbers $\pi * 10^{32}$ and $\sqrt{2} * 10^{21}$ in single and double precision.**
The result is reported in fig. 3: we can see that the decimal digit up to which the number is exact clearly depends on the precision, that sets the level at which the approximation will be performed.

Exercise 3: Test performance

(a)(b) **Write explicitely the matrix-matrix multiplication loop in two different orders. Use the Fortran intrinsic function**

The 'performance.f90' program takes in input two matrices $A_{(n,k)}$ and $B_{(k,m)}$ and performs their multiplication in two different ways, as requested:

- performing the multiplication looping along the rows of the first matrix ('by row' multiplication)
- performing the multiplication looping along the columns of the first matrix ('by column' multiplication)

The difference between the two is obtained by switching the two loops in the code.
Finally, we compare the results with the built-in Fortran subroutine MATMUL.

(c) **Increase the matrix size and use the Fortran Function CPUTIME to monitor the code performance.**
In fig. 4a are displayed the trends of the execution time, monitored via the native CPUTIME subroutine, for different sizes of square random matrices, for the 3 different matrix multiplication subroutines. The data are obtained by performing each operation 10 times and computing the mean and standard deviation of the obtained execution time samples, that are reported in the graphs. It is interesting to notice that multiplying matrices 'by columns' is faster than doing it 'by rows': this is due to the fact that Fortran matrices are stored in memory by columns, so the first methods exploits subsequent elements in the cache.

In order to retrieve how the CPUTIME scales with the matrix size n, I have performed a polynomial fit on the data obtained from the 'by row' multiplication (fig. 4b) with curves of degrees 2,3 and 4 and I have compared the obtained mean squared errors reported in table 1.

a_2	a_1	a_0
$5.3 * 10^{-6}$	$-1.9 * 10^{-3}$	$9.3 * 10^{-2}$

Table 2: Coefficient obtained from fit with $y = a_2 * x^2 + a_1 * x + a_0$

a_3	a_2	a_1	a_0
$3.6 * 10^{-9}$	$6.0 * 10^{-8}$	$-2.1 * 10^{-5}$	$6.4 * 10^{-4}$

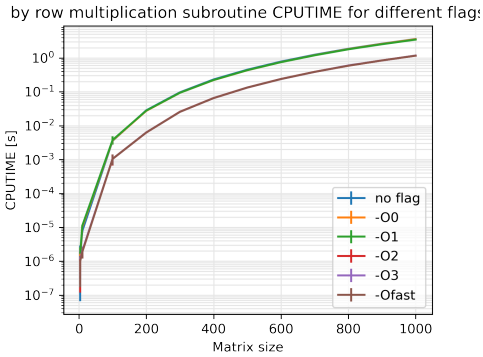
Table 3: Coefficient obtained from fit with $y = a_3 * x^3 + a_2 * x^2 + a_1 * x + a_0$

a_4	a_3	a_2	a_1	a_0
$-1.1 * 10^{-13}$	$3.8 * 10^{-9}$	$-8.5 * 10^{-8}$	$7.3 * 10^{-6}$	$-1.3 * 10^{-5}$

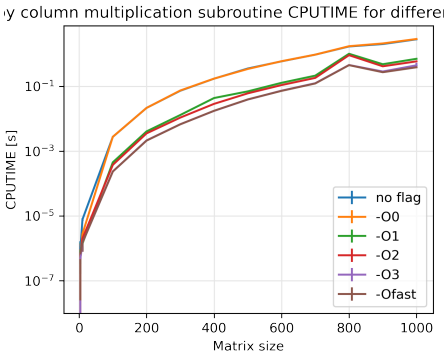
Table 4: Coefficient obtained from fit with $y = a_4 * x^4 + a_3 * x^3 + a_2 * x^2 + a_1 * x + a_0$

As we can see, despite the fact that the MSE for the deg= 4 polynomial fit is slightly smaller than the deg= 3 one, if we consider that the a_4 coefficient in the deg= 4 fit (table 4) is 4 order of magnitude smaller than a_3, we can assume that the CPUTIME scales as $O(n^3)$.

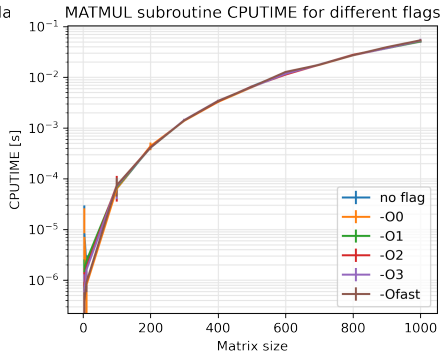
(d) **Use the compiler different optimization flags and monitor the performances.**
In fig 5 are reported the trends of the matrix dimension n vs CPUTIME, for the three different subroutines and for several flags: '-O0', '-O1', '-O2', '-O3', '-Ofast'



(a) by row multiplication



(b) by column multiplication



(c) MATMUL subroutine

Figure 5: Optimization flags