QUANTUM INFORMATION AND COMPUTING

ASSIGNMENT 1

Physics of Data

Turci Andrea

October 24th, 2024

THEORY

1. SETUP:
   - Language: FORTRAN and Python
   - Code Editor: Visual Studio Core

2. NUMBER PRECISION: testing precision of operations with different data types.

   a) Limits of INTEGER in Fortran
   - INTEGER*2 (uses 2 bytes): can represent numbers from 32,768 to 32,767.
   - INTEGER*4 (uses 4 bytes): can represent numbers from -2,147,483,648 to 2,147,483,647

   b) Limits of REAL in Fortran. For real numbers:
   - Single Precision (REAL) uses 32 bits, i.e. around 7 significant decimal digits
   - Double Precision (REAL*8) uses 64 bits, i.e. around 15-16 significant decimal digits of precision

When dealing with very large numbers like $\pi \cdot 10^{32}$ and $\sqrt{2} \cdot 10^{21}$, single precision may fail to capture both terms accurately, and the smaller term might be lost due to limited precision. Double precision should handle the sum more accurately.

THEORY

3. Testing performance of the Matrix-matrix multiplication using different techniques.
   I implement $C = A \times B$, given A, B $n \times n$ matrices, in different ways:
   - Raw-Major order (i-j-k): implementing the loop order firstly looping through raws of A, then columns of B
   - Column-Major order (i-k-j): implementing the loop order firstly looping through raws of A, then iterate over columns in B
   - Using the intrinsic function MATMUL of Fortran

   Then I vary the matrix size and record the time it takes for the different multiplication methods, using CPU_TIME
   - If I plot the CPU time versus the matrix size, I expect a trend in the order of $O(n^3)$.

   Another way to test the performance is to compare the results with different optimization flags. Common optimization flags for gfortran are:
   - -O1: Basic optimization
   - -O2: Moderate optimization (most loops optimized)
   - -O3: Aggressive optimization (includes vectorization and additional optimizations).

CODE DEVELOPEMENT

- Test Job:
  - A simple subroutine with I/O interactions was used to verify the environment setup.

- Number Precision:
  - Integer Precision: Calculated 2,000,001 using INTEGER*2 and INTEGER*4. Compiled with -fno-range-check flag to prevent overflow in INTEGER*2.
  - Real Precision: Computed $\pi \cdot 10^{32} + 2 \cdot 10^{21}$ in single and double precision to compare results.

- Performance Testing:
  - Matrix Multiplication: Implemented three methods, i.e. row-by-column, column-by-row, and Fortran matmul function.
  - Time Measurement: Used cpu_time before and after each computation to measure performance. Matrix size (n) is user-defined for flexibility.
  - Optimization Levels: Compared -O1, -O2, and -O3 compiler flags, with -O3 yielding the most notable results shown.

RESULTS

- Summing 2’000’000 and 1 yields:

```
Sum with INTEGER*2: -31615
Sum with INTEGER*4:     2000001
```

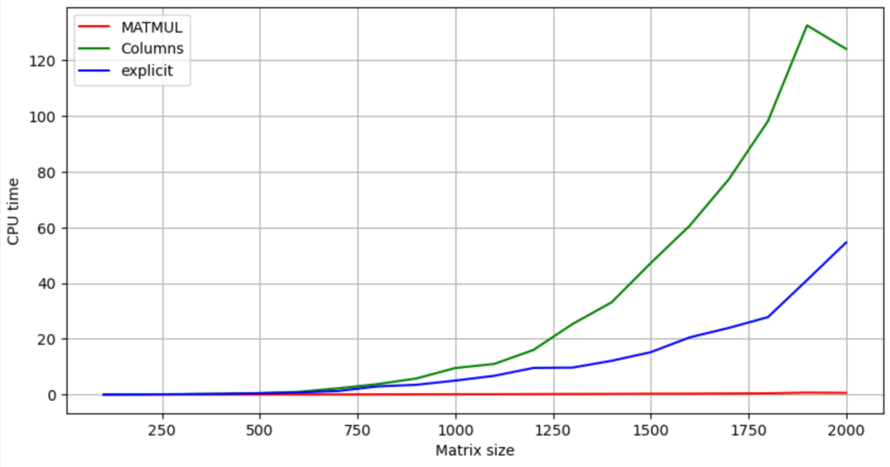
meaning that for INTEGER*2, the result causes an overflow, and the value is not correct. For INTEGER*4, the result is the expected one .

- Summing $\pi \cdot 10^{32}$ and $\sqrt{2} \cdot 10^{21}$ yields:

```
Sum with single precision:   3.14159278E+32
Sum with double precision:   3.1415927410267153E+032
```

With Single Precision (REAL) the result loses accuracy, particularly with the smaller term $\sqrt{2} \cdot 10^{21}$ which is rounded away. Double Precision (REAL*8): The result preserves the contribution of both terms, as expected.

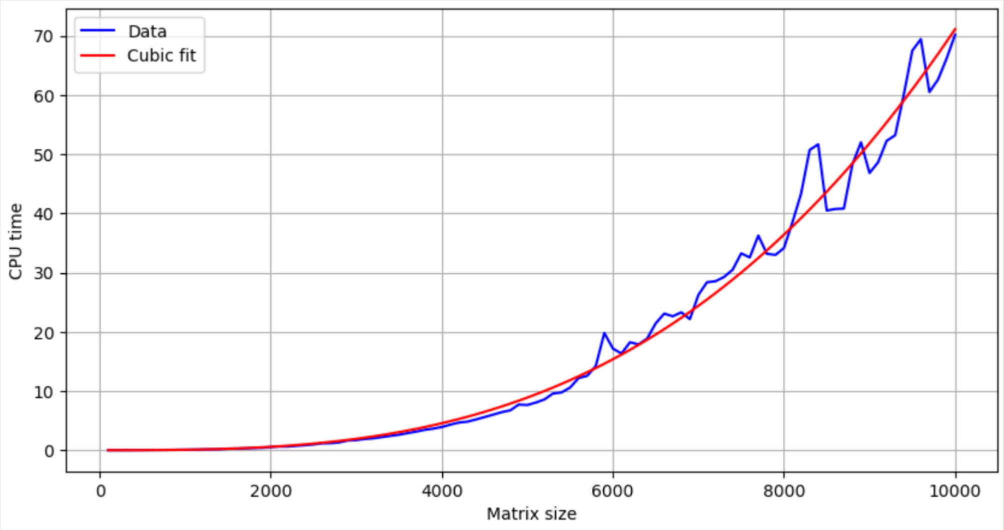
- Using the 3 different methods described before, increased the matrix size from 100 to 2000 in step of 100, and computed the CPU time, averaging over multiple runs



- ➢ The most efficient algorithm is MATMUL
- ➢ A cubic trend starts to emerge for the other two

RESULTS

- Now I try to highlight the exponential trend of the CPU time. To do so, I just use the MATMUL module of Fortran, since it is the fastest one, and I scale the size of the matrix up to $n = 10000$:
  - ➢ A cubic trend is obtained and well fitted



- In order to perform a comparison of the results using different flags (-O1, -O2, -O3), I use the row-by-column order (since it is the one not already optimized by Fortran, but that explicitly computes the matrix) and compare the CPU time.
  - ➢ Using this module, the flags appear to be comparable in terms of CPU time, despite flag –O2 appears to be slightly the fastest one