Scientific Software
Session 4: Fortran 95, Advanced loops and performance

Emil Løvbak, Wouter Baert & Pieter Appeltans Leuven, October 29, 2020

Introduction

This session aims for you to develop an understanding of the following concepts:

- overloading procedures;
- passing arguments via the command line;
- timings (both wall clock time and CPU time);
- optimization flags;
- advanced loops.

Exercises

1. As some of you may already have noticed, `print *, A` prints the elements of `A` in the order they are stored in memory, i.e. in column major order. In this exercise you therefore have to write a procedure, named `show_matrix`, that prints matrices in a more readable format. On the first line, print the dimensions of the matrix and the kind parameter of the type of the matrix elements. On the subsequent lines print the elements of the matrix row by row in scientific format with 5 decimal places. Your interface has to work for both single and double precision real arrays.
   Hint: `write(*,fmt=???,advance='NO')` will prevent the writing position from advancing to the next line after finishing the write operation.

   Link preparation: Scientific software development with Fortran - 3.10 (Overloading Procedure)

2. Modify the `tensor` program from the previous exercise session (a model implementation can be found on Toledo) such that the desired size can be passed as a *commandline argument* instead of reading it from the standard input stream. Use `read(string,*) N` to convert the character string `string` to an integer that will be stored in `N`. Next, implement the following five functions that each calculate the sum of the elements in the tensor:

   - **builtinsum**: uses the intrinsic routine `sum`
   - **blocksum**: uses the intrinsic routine `sum`, but in blocks of N numbers. First take the sum along the first dimension, then along the second dimension, and finally along the last dimension (*Hint*: `sum(sum(sum(T,1),1),1)`).
   - **smallest**: iterate with explicit loops from the smallest till the biggest element and accumulate the numbers
   - **biggest**: iterate with explicit loops from the biggest till the smallest element and accumulate the numbers
   - **kahan**: use the method of Kahan [1], described in pseudo-code below:

[1] attributed to the Canadian mathematician and computer scientist William "Velvel" Morton Kahan (1933), awardee of the Turing Award in 1989, for more information see `https://en.wikipedia.org/wiki/Kahan_summation_algorithm`.

Algorithm 1: The Kahan summation algorithm.

```
Input: my_array
Output: total
Initialize total = 0;
c = 0;
for i = 1:length(my_array) do
    y = my_array(i)-c;
    temp = total + y;
    c = (temp - total) - y ;
    total = temp;
end
```

Add a second command line argument to your program. Call one of the functions above depending on the value of this second command line argument and print the relative error compared to the exact sum $\sum_{i=1}^{N^3} i = N^3(N^3+1)/2$. For example, `./tensor 111 builtinsum` should print the relative error for the sum of the elements of a $111 \times 111 \times 111$ tensor computed using the intrinsic sum function. Next, execute the program for $N = 111$ for the five methods given above and compare their relative error.

- Which method has the smallest error? Why?
- Are floating point additions commutative?

Link preparation: the intrinsic function `COMMAND_ARGUMENT_COUNT` and subroutine `GET_COMMAND_ARGUMENT` as described in section 13.7 of the Fortran 2003 standard.

3. Modify your code of the previous question such that your program also prints the elapsed wall clock time[2] and the cpu time [3]. Compare the wall clock time and cpu time of the five summation methods for $N = 300$ and answer following questions.

   - Does the execution time vary over different runs with the same method and size? Is it a good idea to average your timings over multiple runs?
   - Compare your timings with the output of the unix commando `time` [4](eg. `$ time ./tensor 300 kahan`)
   - Which method is the fastest?
   - When will the wallclock time and the cpu time be very different?

   Important: make sure that your timings are compiler independent.
   Link preparation: the intrinsic subroutines `CPU_TIME` and `SYSTEM_CLOCK` as described in section 13.7 of the Fortran 2003 standard.

4. Download `prime.f95`, which calculates the first 500000 prime numbers, from Toledo. Compile this program twice using gfortran. First with the flag `-O0` and subsequently with the flag `-O3`. Compare their execution time using `time`. What do you notice?

5. Write a subroutine `play_with_array` that contains no explicit `do`-loops and is equipped with the following arguments and respective functionality:

   - a `real` matrix `x`: when leaving the subroutine, all elements of `x` that are in the interval $[0.3, 0.65)$ should be replaced by 0.45;
   - an output `integer` array `v` that indicates whether the i-th column of `x` contains a number greater than 0.75: if so, `v(i)` gets the value 42, otherwise `v(i)` should equal -12;
   - a logical output value `w` indicating whether all columns of the original `x` contain an element greater than 0.75;

[2]the *real world* time elapsed between the start and the end of the execution of the program
[3]the time the CPU spent on the program
[4]`https://en.wikipedia.org/wiki/Time_(Unix)` explains the difference between the real, user and system time.

- an output `integer` array `c` that counts the number of elements smaller than 0.25 in each row of `x`: `c(i)` is equal to the number of elements in row i that are smaller than 0.25.

Generate a 7×10 matrix of random numbers between 0 and 1 and print it. Next, call `play_with_array(x, v, w, c)` and print the resulting `x`, `v`, `w` and `c`. For printing the matrices, use the procedure you wrote in the first exercise.

Link preparation: Scientific software development with Fortran: section 4.7 - `ANY`, `ALL` and `COUNT` as described in section 13.7 of the Fortran 2003 standard.