High Performance Computing
Exercise Sheet 5

FS 21
dr. Douglas Potter

http://www.ics.uzh.ch/
Teaching Assistants:
Jozef Bucko, jozef.bucko@uzh.ch
Hugues de Laroussilhe, hugues@ics.uzh.ch

Issued: 26.03.2021
Due: 31.03.2021

In this exercise session, we will practise some basics usage of Regex as well as learn more about how to analyse the performance of our code. We will get more familiar with one of many codes offering such an analysis, namely *Cray Performance Measurement and Analysis Tool*.

Exercise 1 [Regex - `grep`]

Commit : The command `grep` can be used to search a file for a specific expression. On the course repository you can find the file `binary.txt`, which contains a list of binary numbers. Give a regular expression that will match only the binary strings that satisfy the condition:

1) end with `00`
2) start and end with `1`
3) contain the pattern `110`
4) contain at least three times a `1`
5) contain at least three consecutive `1`s

In addition, you can find another file called `measurements.txt` in the course repository containing temperature and humidity data taken at different times. Write a simple bash (or python) script which stores `out.txt` file containing two columns: `TEMPERATURE` and `TIME` (so humidity data are skipped). In addition, the script should print the average measured temperature.

Exercise 2 Introduction to `Perftools-Lite`

Man pages introduce the `Perftools-Lite` as follows: "Perftools-Lite is a simplified, easy-to-use version of the Cray Performance Measurement and Analysis Tool set. Perftools-Lite

provides basic performance analysis information automatically, with a minimum of user interaction, and yet offers information useful to users wishing to explore their program's behavior further using the full CrayPat tool set...". In this exercise, we will use this tool to analyse the run of the code `nbody.cpp` which can be found on the well-known git repository `https://github.com/hlasco/hpc_esc_401` in folder `exercise_session_05`. During this exercise, we will again work on Piz Daint (do not forget to use project `uzg2` for now).

- log in to Daint and load the `perftools-lite` module
- compile the code `nbody.cpp` (do not forget about `-g` flag). Do you observe any changes in the terminal output compared to usual compilation during the previous sessions? Run the produced executable using a jobscript (you do not need to include any changes into it).
- once the job is successfully finished, inspect the produced files/folder(s) and read few details about the output structure in the manual (`man perftools-lite`)
- which parts of the code took the most CPU time? Can you make sense of why?

Commit : Push the log file provided by the perftools-lite tool and comment briefly on its content and what you can learn about the performance of your code.

In the last part, let's revise the way how to separate the code into more modules.

- extract the function `forces()` into a separate file `forces.cpp` and function `ic()` into file `init_conditions.cpp`. Do not forget to create one header file (.h) for each of the .cpp file. (You might need to put `#include <cmath>` in `forces.cpp`)
- make sure you have `perftools-lite` loaded and try to compile and link the codes. Do you get any error? What about the new type `particles`? Should it also be defined in a separate file?
- run the compiled code and see the `perftools-lite` output. Is the execution time comparable with the simple case when the whole code was placed within a single .cpp file?

Commit : Provide all the relevant files you used to compile and link your code. Discuss the runtime comparison of the different code running.

Exercise 3 Time Complexity

- Let's first write an OpenMP version of `nbody.cpp`. This code is slightly different from what you already encountered as it contains nested `for` loops. Here you can decide to add OpenMP parallel directive to the first, the second, or to both loops. Which one is the most efficient solution, why ?
- The execution time of the code `nbody.cpp` will obviously increase with the number of particles. What is the expected scaling relation between the code execution time t and the number of particles N ? Can you justify your answer ?

  Let's check that numerically:
- Edit the code `nbody.cpp` in order to be able to specify the number of particles with a command line argument. For example `./nbody 1000` should execute the code with 1000 particles. Does your implementation still work when not specifying any command line argument ? If not, try to include safety nets while reading the command line arguments.
- Plot the execution time of the serial code with

  $$N = [100, 500, 1000, 5000, 10000, 20000, 50000]$$

  with 1 node on the uzg2 partition (using a bash script is a good idea). Repeat the experiment with your OpenMP implementation using 12 threads. Can you find a good fit to your results ?
- BONUS: Is it possible to reduce the time complexity of N-body force calculations ?

Commit : Push your answers, implementation, plots and fits to your repository.