Serie 4

This week you will go for your first parallel implementation. The objective of this serie is to go through the *Profiling-Optimization Cycle* presented during Week 4 by Nicola Varini. The steps are detailed here

- debug a bad serial implementation of the 2D Poisson problem
- profile a correct serial implementation
- Measure the difference between `icc`, `gcc` and the main optimization flags on the performance of this correct serial implementation
- parallelize the 2D Poisson problem using OpenMP

Exercise 1 (Debugging)**.**

- Update the pcpt-2015 repository.
- There is a new folder `bugy-poisson`, enter the folder and type `make`
- This will compile a version of poisson where bugs have been added. In order to do less loops the termination criteria has been changed to a loop on 200 iterations.

  In order to find the bugs you will use two sequential debugging tools, `valgrind` and `gdb`

  Notice that when you compile with the make command `-g` option is added. This is to add the debugging symbols in the executable.

  Try to concentrate on the bug one after the other. They are fairly easy to catch by reading the code, but the point of this exercise is to make you practice the tools more than catching this particular bugs.
- Run gdb
  - `$ gdb ./poisson-bug`
  - If you type `run` or `r` it will run the code up to the failure point and indicate the line where it stopped
  - At this point you can print the content of the different variables with the `print` command. (`print` or `p`)

    for example to print `i`: `p i` or the 10 first values of `u`: `p u[0][0]@10`
  - If at this point you did not figure out the first bug add a breakpoint before the loop at line 120 (`break 120` or `b 120`) or at the beginning of the main `break main` or `b main`

    Type `r` again to restart the code, this time the execution will stop at the breakpoint
  - Now you can advance step by step with with the `next` (`n`), and check the different values with `p`.
  - SPOILER: the first bug to find is in the loop bound, since we access to $i-1$ and $j-1$ at line 124, i and j cannot start at 0.

  - Once this is corrected compile again and run again. The second bug should show up. Here it is not a segfault but an infinite loop. And we have an hint the name of the image generated looks messed up after 127.

    So lets set a breakpoint at line 121 to stop at first line in the loop. But this time we will add a condition to the breakpoint. The command for this is `b 121 if k==127`
  - Try to advance in the execution and check what happen to the variable `k` when increased at the end of the loop
  - SPOILER: here the bug is a bound error. k is stored on a char of 8 bits so 256 values, but since it is signed it is from -126 to 127. In this case either you really want a 8bit storage `unsigned char` or go for a bigger space `short` or `int`
  - compile again and run again. Everything should run nicely, the results even look correct. But there is still 2 bugs. In particular one that will most probably not do anything as long as the problem is not big enough.

- Run valgrind
  - Run again the code with valgrind, It will complain right at the beginning something like

    ```
    ==14705== Invalid write of size 4
    ==14705==    at 0x400DE0: main (poisson-bug1.c:113)
    ==14705==  Address 0x54e0d30 is 0 bytes after a block of size 1,024 alloc'd
    ==14705==    at 0x4C28C20: malloc (vg_replace_malloc.c:296)
    ==14705==    by 0x400D49: main (poisson-bug1.c:104)
    ```

    This suggest that at line 113 there is a write access outside of an allocated zone in memory. An extra information is that this is right after an array allocated at line 104.
  - The other thing that valgrind tells us is at the end of the execution. It gives a `LEAK SUMMARY`, and since there are leeks it propose to rerun with the option `--leak-check=full`.

    If you do so it will give you the line of at which data are allocated and not freed.

    In our case it is not harmful, but if similar leaks happen in loops it would give a nasty bug that appears only on big cases. So even if your code run fine and give good results it is always a good idea to check for any leaks.

Exercise 2 (Flat profiling)**.** Once your code has been debugged and run without error, you are asked to profile it using gprof. Compile and link it with the `-pg` flag

```
gcc -pg  poisson.c -o poisson
```

then run it. The `gprof` utility is used to analyze a special file produced by the execution (`gmon.out`). It presents you the measured data in a flat profile view.

```
./poisson ; gprof ./poisson
```

Where does your code spend most of the time ?

Exercise 3 (`icc`, `gcc`, `optimization`, `vectorization` and other cool stuff)**.** From now on, you compiled and linked your code without any optimization flag. Try to change the compiler (`icc` or `gcc`), the optimization level (`-O1`, `-O2`, `-O3`) and the loop vectorizer (`-ftree-vectorize` for `gcc`, `-xHost` for `icc`) and complete the following array with the execution time in seconds. To do that : **remove the call to `write_to_file()` and increase the size to** `N = 1024` in order to measure representative timings (several seconds)

optimization level	icc	gcc
`-O0`		
`-O1`		
`-O2`		
`-O3`		
`-O3 -xHost`		

A simple way to measure the execution time is using `second()` :

```
#include <sys/time.h>
#include <time.h>
double second(){
        struct timeval tp;
        int i;
        i = gettimeofday(&tp,NULL);
        return ( (double) tp.tv_sec + (double) tp.tv_usec * 1.e-6 );
}
```

example :

```
double t = second()
 ... do some work here
printf("Execution time = %f [s] \n",(second()-t));
```

Exercise 4 (Parallelization with OpenMP)**.** Take the best combination [compiler,optimization flags] and parallelize the loops with OpenMP directives. To compile your OpenMP version, you need to add the flag `-fopenmp` if you choose `gcc`, `-openmp` if you choose `icc` at compilation and link phases. Vary the number of OpenMP threads, measure the speedup. Report the execution timings with respect to the number of threads of your parallelized version. The numerical results of your parallel version must be the same than the one of your serial version !! A simple verification can be done here with a comparison of the number of steps to reach `(l2` $\leq$ `eps)`.

`OMP_NUM_THREADS`	time [s]	speedup
1		1
2		
4		
8		
16		
32		

(facultative) Report the speedup in a log-log graph using R