UNIVERSIDAD AUTÓNOMA DE MADRID

MASTER'S DEGREE IN RESEARCH AND INNOVATION IN COMPUTATIONAL INTELLIGENCE AND INTERACTIVE SYSTEMS (I^2-ICSI)
Numerical and Data-Intensive Computing (Course 2022/23)

MÁSTER UNIVERSITARIO EN INGENIERÍA INFORMÁTICA
Computación a Gran Escala (Curso 2022/23)

Laboratory 1: PROFILERS AND OPENMP

Deadline: ***October 20***

Compulsory assignment (8 points)

Proceed carefully through the following steps, completing the lab report (homework) as requested. The lab report must be a full technical document consisting of a front page, index and sections, which is to be delivered as a PDF file through the Moodle portal by the given deadline date.

1. Download associated material (profilers.tar.gz) from Moodle's course page into personal working directory.

2. Uncompress and untar associated material:

```
gunzip profilers.tar.gz
tar -xvf profilers.tar
```

PERF (I)

1. Go to directory task1:
   ```
   cd profilers/task1
   ```
2. Edit and understand example "task1.c":
   ```
   gedit task1.c &
   ```
3. Compile "task1.c":
   ```
   make
   ```
4. Run task1 with the performance analysis tool (`perf`), monitoring the following events:
   ```
   sudo perf stat -e cycles:u -e instructions:u \
     -e L1-dcache-loads:u -e L1-dcache-load-misses:u \
     -e L1-dcache-stores:u -e L1-dcache-store-misses:u \
     -e LLC-loads:u -e LLC-load-misses:u \
     -e LLC-stores:u -e LLC-store-misses:u ./task1
   ```
   a. CPU clock cycles at user level: `-e cycles:u`

   b. Machine code instructions at user level: `-e instructions:u`

   c. First-level data cache load hits at user level: `-e L1-dcache-loads:u`

   d. First-level data cache load misses at user level: `–e L1-dcache-load-misses:u`
   e. First-level data cache store hits at user level: `–e L1-dcache-stores:u`
   f. First-level data cache store misses at user level: `–e L1-dcache-store-misses:u`
   g. Last-level cache load hits at user level: `–e LLC-loads:u`
   h. Last-level cache load misses at user level: `–e LLC-load-misses:u`
   i. Last-level cache store hits at user level: `–e LLC-stores:u`
   j. Last-level cache store misses at user level: `–e LLC-store-misses:u`

5. Exchange indices in source code of "task1.c", such that array is now traversed by rows instead of by columns, hence taking advantage of both the row-major order used in C and the cache hierarchy:

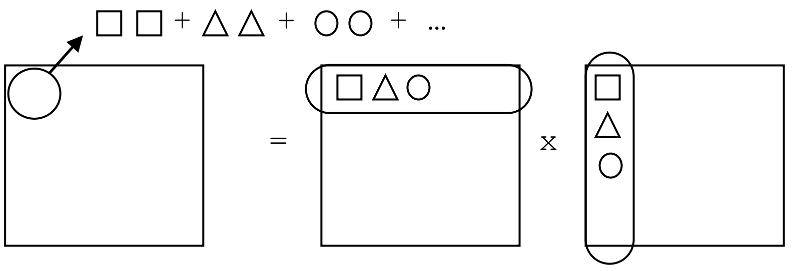
   `array[j][i]` ➔ `array[i][j]`

6. Compile "task1.c" and execute it through `perf`, writing down the same events as before.
7. Analyze the new results and compare them with the previous results.
8. Study the performance of the program without optimization (remove –O2 from the "Makefile") and different types of optimization (-O1, -O2, -O3).
9. Write down conclusions to lab report (homework).

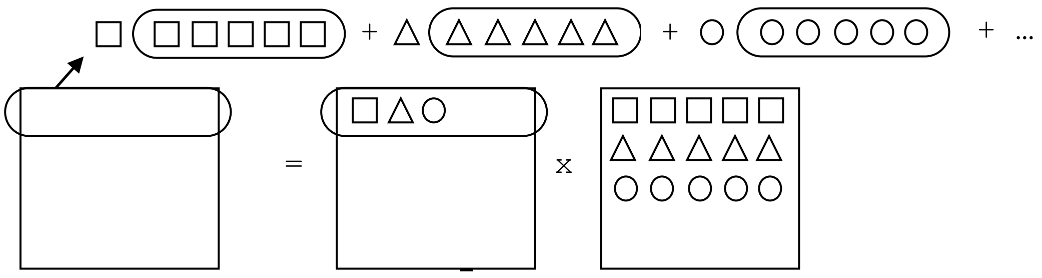
PERF (II)

1. Go to directory profilers/task2 (matrix multiplication).
2. Edit and understand example "task2.c".
3. Compile "task2.c".
4. Run task2 with the performance analysis tool (`perf`).
5. By following the next graphical schemes, implement a new version of the matrix multiplication function (Mult2) that takes advantage of both the row-major order used in C and the cache hierarchy.

   Standard matrix multiplication (Mult1)



   Optimized matrix multiplication (Mult2)



6. Run task2 and verify that both functions (Mult1, Mult2) yield the same result.
7. Comment call to Mult1 in "task2.c" and run task2 with the performance analysis tool (`perf`).
8. Compare the performance of both functions and write down conclusions to lab report, including the implementation of Mult2 (homework).

GPROF

1. Go to directory profilers/task3 (image processing algorithm).
2. Edit and understand structure of "Makefile". Option "-pg" at compile time forces compiler to generate profile data suitable for "`gprof`".
3. Compile program:

   ```
   make
   ```
4. Run program (it generates binary file "gmon.out" with profile data). Input and output bitmap images can be viewed with any image visualization program (`eog, gimp`, …)

   ```
   ./algi channel1.bmp channel2.bmp
   ```
5. Run "`gprof`" generating "profile" file:

   ```
   gprof algi > profile
   ```
6. Edit and understand the self-explained "profile" file.
7. Identify functions that consume a significant percentage of running time (bottlenecks) ➔ best candidates to be optimized / parallelized. Any improvement on them will have a significant impact on the overall running time:
   a. Large functions (big "self ms/call" value) with a big percentage of running time (big "% time" value).
   b. Small functions (low "self ms/call" value) that run very frequently (big "% time" value).
8. Write down conclusions to lab report, including short description of bottleneck functions (homework).

OpenMP (Compilation and basic directives)

Proceed through the following steps, completing the lab report (homework) as requested.

1. Obtain CPU information from the Linux kernel:

   ```
   cat /proc/cpuinfo > cpuinfo.txt
   gedit cpuinfo.txt
   ```

2. Identify the CPU's model in "cpuinfo.txt". Example:

   ```
   model name : Intel(R) Core(TM) i5 CPU  650  @ 3.20GHz
   ```
3. Identify the number of CPUs in "cpuinfo.txt". The number of CPUs is equal to the number of different physical identifiers of the available logical processors. Example for a single CPU Intel Core i5:

```
processor  : 0    (first logical processor)
      physical id: 0
processor  : 1
      physical id: 0
processor  : 2
      physical id: 0
processor  : 3
      physical id: 0
```

4. Identify number of cores per CPU in “cpuinfo.txt”. Example for Intel Core i5 with 2 cores:

   ```
   cpu cores  : 2
   ```

5. Identify number of hardware supported threads (i.e.: logical processors) per CPU in “cpuinfo.txt”. If the number of supported threads is N times the number of cores, the CPU supports hyper-threading and each core will be able to concurrently execute N of those threads by sharing its internal resources (ALU, FPU, etc.). Example for Intel Core i5 with 2 cores and hyper-threading:

   ```
   siblings   : 4
   ```

6. Identify what logical processors correspond to each CPU and core. Example for a single Intel Core i5 in which the first two logical processors are mapped to the first core and the last two logical processors to the second core, with a single CPU (physical processor):

   ```
   processor  : 0                   processor  : 2
         physical id     : 0              physical id     : 0
         core id         : 0              core id         : 2
   processor  : 1                   processor  : 3
         physical id     : 0              physical id     : 0
         core id         : 0              core id         : 2
   ```

7. Using a web browser, verify the number of cores and threads per core on the Internet according to the CPU’s model information. Write down conclusions to lab report (homework).

8. Download associated material (openmp1.tar.gz) from Moodle’s course page into personal working directory.

9. Uncompress and untar associated material:

   ```
   gunzip openmp1.tar.gz
   tar -xvf openmp1.tar
   ```

10. Go to directory openmp1/task1.

11. Edit and understand structure of “Makefile”. Option “-fopenmp” at compile time forces compiler to understand OpenMP directives. Option “-lgomp” at link time forces linker to include the OpenMP library for Linux (GOMP).

12. Edit and understand example “task1.c”.

13. Execute in a new terminal the run-time CPU monitor “`mpstat`” (if not available, execute “`gnome-system-monitor`” instead):

    ```
    xterm &              (create new terminal)
    mpstat -P ALL 1      (run from the new terminal)
    ```

“mpstat” shows statistical information about each available logical processor, including percentage of CPU load at the user level (%usr) and the system level (%sys).

14. In “task1.c”, set the number of OpenMP threads (constant “NUM_THREADS”) to 1. From the initial terminal, compile the program and execute it, writing down the wall time (real execution time). The latter is the minimum sequential time (Ts) of the algorithm. See how “mpstat” shows what logical processor is executing the program.
15. Set the number of threads to 2 in “task1.c”, recompile the program and run it, checking with “mpstat” what logical processors are executing both threads. Execute the program several times. The operating system automatically maps every thread to a different logical processor. Discard executions in which both logical processors belong to a same core (if the CPU supports hyper-threading). Write down the average wall time of all executions, which corresponds to the parallel time for two cores (Tp).
16. Compute speedup and efficiency for two cores.
17. Force the mapping of both OpenMP threads to the same logical processor. In case of several threads assigned to the same logical processor, the latter executes them with time-sharing. Example:

    ```
    export GOMP_CPU_AFFINITY=”3”
    ```
18. Run the program several times and compute speedup and efficiency for two threads executed by the same logical processor.
19. Release the explicit mapping of threads to specific logical processors, such that this mapping be left to the operating system again:

    ```
    unset GOMP_CPU_AFFINITY
    ```
20. Set the number of threads to 4 in “task1.c”, recompile the program and run it, checking with “mpstat” what logical processors are executing all threads. Execute the program several times. The operating system automatically maps every thread to a different logical processor. Discard executions in which several logical processors belong to a same core (if the CPU supports hyper-threading). Write down the average wall time of all executions, which corresponds to the parallel time for four cores (Tp).
21. Compute speedup and efficiency for four cores.
22. Write down results and conclusions to lab report (homework).

23. Go to directory openmp1/task2.
24. Edit and understand example “task2.c”.
25. Compile and run “task2” several times. Realize that the PID is always different at the beginning of the parallel body, and the same at its end. Analyze and interpret this behavior.
26. Declare variable “pid” as private. Compile the program and run it again several times, realizing that the PID now is always different. Analyze and interpret this behavior.
27. Write down conclusions to lab report (homework).
28. Run “task2” again and realize that private variable “limit”, which is initialized to -1 in its program declaration, is reset to zero at the begging of the parallel body, whereas it is set back to -1 when the master thread resumes its execution right after the parallel body. Analyze this behavior by considering that every thread within a parallel region has a local copy of all its private variables.
29. Change the “private” clause for “firstprivate”, which initializes the local copies of private variables to their original value. Compile and run the program again realizing the difference.

30. Write down conclusions to lab report (homework).

Optional assignment (1.5 points)

31. Go to directory openmp1/task3.
32. Edit and understand example “task3.c”.
33. Compile and run “task3” several times. Analyze why all threads run alternately.
34. Comment both the “omp_set_lock” and the “omp_unset_lock” function calls. Compile and run again several times. Analyze why the threads do not run alternately.
35. Include the critical region into a “critical” directive. Compile and run again several times. The result is the same as when locks are utilized. Example:

```
#pragma omp critical
{
      // Critical region: One thread at a time

      ...
}
```

36. Remove the “critical” directive. Insert a barrier synchronization right above the workload. Compile and run again several times. Analyze why the threads run alternately again. Since there is no critical region, the workload of both threads is running with global synchronization but without mutual exclusion. Example:

```
// Critical region: One thread at a time

#pragma omp barrier

// Workload
```

37. Go back to the original “task3.c” with the wait and signal semaphore calls. Comment the “omp_unset_lock” (wait but no signal). Compile and run it again. Analyze why the first thread runs only once and the program halts. Press Ctrl-C to stop the program.
38. Uncomment the “omp_unset_lock” and insert a barrier synchronization right above the workload. Compile and run again. Analyze why the program halts right from the beginning. Press Ctrl-C to stop the program.
39. Write down conclusions to lab report (homework).

Optional assignment (0.5 points)

40. Go to directory openmp1/task4.
41. Edit and understand example “task4.c”.
42. Compile and run “task4”. Justify the result and the need for the semaphore.
43. Comment both the “omp_set_lock” and the “omp_unset_lock” function calls. Compile and run again. Justify why the result is wrong.
44. Add a “reduction” clause to the “parallel for” directive. That clause performs a reduction operation on the variables that appear in its list. A private copy for each list variable is created and initialized for each thread. The reduction operation is applied to all private copies. At the end of the parallel section, the reduction operation is applied to all private copies and the final result is written to the shared variable. Compile and run again. The

result is the same as when locks are utilized, but the performance is much higher. Example:

```
#pragma omp parallel for reduction(+:k)
for (i=0; i<100000000; i++)
      k++;
```

45. Write down conclusions to lab report (homework).