Challenges

José Fuentes
Parallel programming

In this document we present six challenges to be solved in a multicore environment. It is part of the course Multicore programming at University of Chile. There are some rules to be considered to solve the challenges properly:

- At least **three** challenges must be completely solved. If a student solves more than three challenges, the three challenges with the best marks will be considered. The deadline to solve the challenges is the last week of the semester.
- The solution for each challenge must be **reproducible**. For each solved challenge, the student must provide the code and datasets to reproduce the results.
- The solution for each challenge must include a **report**. The report must include a description of the methodology, the complexity of the solution (if it is needed), the experimental results and conclusions.

1 Performance of multiple multicore algorithms

Description: In this challenge we will measure the impact of running multiple multicore algorithms in the same multicore machine. The student have to select several parallel implementation and execute them at the same time, measuring time and cache misses.

At least two experiments must be performed: Change the number of available threads (`CILK_NWORKERS`) of each multicore algorithm, and change the core affinity of each multicore algorithm. To change the core affinity, the **taskset**[1] must be used.

[1] taskset - retrieve or set a process's CPU affinity: `https://linux.die.net/man/1/taskset`

For example, if `./pa` is a parallel algorithm implemented in Cilk Plus, then the following command will execute `./pa` using 4 threads over the cores 0, 1, 2 and 3

```
CILK_NWORKERS=4 taskset -c 0-3 ./pa
```

and the following command will execute `./pa` using 2 threads over the cores 4, 5, 6 and 7

```
CILK_NWORKERS=2 taskset -c 4-7 ./pa
```

To see the list of cores, execute the command `lstopo` and see the substrings *Core L#*.

Evaluation: To solve this challenge, the students must select at least three parallel algorithms implemented in Cilk Plus. Additionally, the students must perform at least the two experiments mentioned above. Each experiment must be repeated at least 5 times and the median must be reported. The running time must be measured using the C function `clock_gettime` and the cache misses must be measured using `perf`.

2 Histogram of an image

Description: In this challenge we will obtain the image histogram of an RGB image. The image histogram consist in the number of ocurrence of each color (from 0 to 255) on each layer (red, green and blue).

For example, if the following three matrices represent a RGB image

$$R = \begin{bmatrix} 0 & 255 & 128 & 0 \\ 255 & 0 & 0 & 255 \end{bmatrix}$$

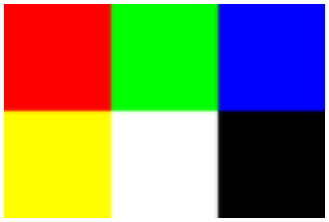
$$G = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 255 & 255 & 255 & 255 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 & 255 & 0 & 255 \\ 255 & 0 & 255 & 0 \end{bmatrix}$$

its image histogram will be:

Red:	Green:	Blue:
0: 4	0: 4	0: 4
128: 1	255: 4	255: 4
255: 3		

In this challenge we will use the **portable pixmap format (PPM)**[2] to visualize the datasets. As an example, the following image



is represented as follows in the PPM format

```
P3
3 2
255
255 0 0
0 255 0
0 0 255
255 255 0
255 255 255
0 0 0
```

Where the first three line are the header: `P3` means that the format is a RGB color image in ASCII, `3 2` is the width and height of the image in pixels and `255` is the maximum value for each color. The rest of the lines represents the color of each pixel

Evaluation: To complete this challenge, the student must implement a parallel solution using Cilk Plus. The student must perform an experimental evaluation to show the scalability of its solution. In the experiments, the student must vary the number of threads and the size of the images. To test

[2] `https://en.wikipedia.org/wiki/Netpbm_format`

the solution, the student can use the C-code of the file `random_rgb_image.c`.

Note: The solution with the best scalability will have extra points.

3 Comparison of parallel sorting algorithms

Description: In this challenge we will compare the performance of different parallel sorting algorithms. The comparison must include at least four parallel algorithms to sort arrays of integers. Most of the algorithms will be provided in the classes.

Evaluation: To complete this challenge the student must measure the running time and cache misses of, at least, four parallel sorting algorithms. The student must vary the number of threads and the size of the arrays. The running time must be measured using the C function `clock_gettime` and the cache misses must be measured using `perf`.

4 Subtree size

Description: In this challenge we will compute the size (number of nodes) of each subtree in a tree. Given an adjacency list representation of a tree, the student must implement a parallel algorithm to compute the size of each subtree, storing the corresponding size on each node.

Hint: Review the solution for the Euler Tour problem

Evaluation: To complete this challenge, the student must to provide a parallel implementation using Cilk Plus. To test the solution, the student must use the datasets available at in `www.josefuentes.cl/datasets/trees.php`. Additionally, the student must measure the running time and cache misses of its solution, varying the number of threads and the number of the nodes of the tree. The running time must be measured using the C function `clock_gettime` and the cache misses must be measured using `perf`.

5 Parentheses representation of a tree

Description: In this challenge we will compute the parentheses representation of a tree. Given an adjacency list representation of a tree, the student must implement a parallel algorithm to compute the parentheses representation efficiently.

Hint: Review the solution for the Euler Tour problem

Evaluation: To complete this challenge, the student must to provide a parallel implementation using Cilk Plus. To test the solution, the student must use the datasets available at in `www.josefuentes.cl/datasets/trees.php`. Additionally, the student must measure the running time and cache misses of its solution, varying the number of threads and the number of the nodes of the tree. The running time must be measured using the C function `clock_gettime` and the cache misses must be measured using `perf`.