Parallel Image Analysis

Lab session 2:
Using OpenMP to create multi-threaded applications

In this lab you will be adding OpenMP constructs to serial programs to make them multi-threaded. You will then run them on UPPMAX as a node job, and compare their execution time with the serial version.

Start by obtaining the serial source code and example data:
`http://www.cb.uu.se/~cris/ParallelImageAnalysis/lab2.zip`

The ZIP file contains two subdirectories: `filter` and `label`. Each subdirectory belongs to one of the two exercises below.

1 Exercise 1: filter

As discussed in class, filters are rather simple to parallelize. This should be an easy exercise!

The `filter` directory contains a Makefile, some test images (with a `ui8` extension, these files only contain pixel values, no metadata), and the following source files:

- `filter.c`: contains the function `main()`.
- `gaussf.c`: contains the filtering function `filter_gauss()` and subfunctions.
- `image.c`: contains image reading and writing functions.
- `filter.h`: contains the declaration of all the functions and data types.

The function `main()` parses the input parameters, loads the image into memory, calls the function `filter_gauss()`, times the execution of this function, and writes the output to a file. The function `filter_gauss()` implements a Gaussian convolution (as two one-dimensional filters), using a mirrored boundary condition (i.e. the image is extended beyond its borders by mirroring the pixels inside the border). The file `gaussf.c` is the only one that we'll need to modify to make the filtering multi-threaded.

Compile the program by typing

```
make
```

(note the '`-fopenmp`' argument to `gcc`, both for compiling and linking) and run it with

```
./filter cermet.ui8 cermet.out  256  256 5
```

As you can see, the program has 5 input arguments: input file name, output file name, size of the input image in pixels, and the sigma of the Gaussian filter. To best observe the speedup of parallelization, use the `europe.ui8` image, which is 5800 by 9100 pixels. Also, you can increase the filter's size (sigma) to increase the cost of the operation.

After testing your multi-threaded program, try running it with different number of threads:

```
export OMP_NUM_THREADS=4
./filter ...
```

How well does the speedup behave with respect to the number of threads?

If you want to see the output of the function, for example to see if it is correct, use the following sequence of commands in MATLAB:

```
fid = fopen('cermet.out','rb');
a = fread(fid,[ 256, 256],'uint8');
fclose(fid);
imagesc(a)
```

2 Exercise 2: label

This next exercise should be a bit more complex. We have a sequential labeling algorithm, which uses propagation to identify connected components in a binary image.

The `label` directory contains a Makefile, some test images (with a `bin` extension, these files only contain pixel values, no metadata), and the following source files:

- `label.c`: contains the function `main()`.
- `simple_label.c`: contains the labeling function `simple_label()` and subfunctions.
- `compound_label.c`: contains the function `compound_label()` and a subfunction.
- `image.c`: contains image reading and writing functions.
- `compare.c`: contains a function to compare two labelings.
- `filter.h`: contains the declaration of all the functions and data types.

The function `main()` parses the input parameters, loads the image into memory, calls the functions `simple_label()` and `compound_label()`, times the execution of these functions, compares their output, and writes one of the outputs to a file.

The function `compound_label()` is my attempt at making a labeling routine suitable for parallelization. It splits the image into two, and labels the two halves separately (either by calling `compound_label()` recursively, or by calling `simple_label()` if the desired number of recursion steps has been reached). Next, it analyses the labels on each side of the split, and decides which labels in each of the two halves should be merged. This step produces two look-up tables, which are then used to change pixel values in the two halves to produce a coherent labeling. Add OpenMP constructs to the function `compound_label()` to make it multi-threaded. Note that this will become a nested multi-threaded program, and you will need to enable the nesting explicitly.

Compile the program by typing

```
make
```

and run it with

```
./label cermet.bin cermet.out  256  256
```

The input arguments are like in the first exercise, except there is no filtering parameter. The file `images.txt` contains a list of binary images available, and their sizes. The bigger the image, the larger the potential benefit of multi-threading.

As before, try your multi-threaded program with different number of threads. How many threads do you need to reap the benefit? Can you come up with a more efficient way of parallelizing this operation?

If you want to see the output of the function, for example to see if it is correct, use the following sequence of commands in MATLAB:

```
fid = fopen('cermet.out','rb');
a = fread(fid,[256,256],'uint32');
fclose(fid);
imagesc(a)
```

Important information

We have several nodes on Kalkyl reserved during each of the lab sessions. To use the reserved nodes (and not have to wait in the queue with all the people doing actual work), add `--reservation=g2011040` to your `sbatch` and `interactive` commands.

The project number for this course is `g2011040`. This goes after the `-A` option of the `sbatch` and `interactive` commands.