Assignment 1 (Java Fork/Join framework) Documentation

Md Rafiqul Islam - 12123971

Framework used:

JAVA Fork/Join framework

How to run program:

1. Compile java class (with package):
   ~ "`javac -d . *.java`"

2. Execute java code:
   ~ "`java <package name>.<class name> <image name>.<image format>`"

   Example: ~ "`java filter.TestImageFilter IMAGE2.JPG`"

Code Structure:

There is total three java file in this project. They are:

- **ImageFilter.java**

  - This class contains the sequential image filtering process.

- **ParallelFJImageFilter.java**

  -This class contains some methods and one inner class called **InnerParallelFJImageFilter**. This is the class where parallel image filter has been done.

- **TestImageFilter.java**

  -This is the main class of the project. From this class the sequential and parallel image filtering has been called. The performance of sequential and parallel image filter also had been measured here. And finally, the comparison also had been done here between sequential and parallel image filtered file.

Threshold:

For parallel execution we need a proper threshold to limit the number of tasks generated. The equation of calculating dynamic threshold is

```
Threshold = width/ nThreads
```

Here, Threshold is an integer type number. Basically, for parallel image filter the height loop of the filter image has been split for Fork/Join task. In this situation for Fork/Join, the fraction of image width and nThreads (number of threads used for parallel execution) has been taken as threshold.

Number of tasks generated:

As it is told that here the Fork/Join framework's Recursive action has used for parallel image filter. The **ParallelFJImageFilter** class has an inner class called **InnerParallelFJImageFilter.** Basically, the Fork/Join has applied in **InnerParallelFJImageFilter** class. This class override a function called **compute ()**. Inside compute () function the tasks has been generated for parallel execution. There are two tasks (**task1** and **task2**) has been generated for each parallel execution. The **invoke ()** method forks the task and waits for the result and doesn't need any manual joining. It is easy to handle. That's why this method is used for forking the task.

Algorithm:

The class **ParallelFJImageFilter** implements a parallel, iterative nine-point image convolution filter working on a linearized (2D) image. In each of the NRSTEPS (=100) iteration steps, the average RGB-value of each pixel p in the source array is computed, considering p and its 8 neighbor pixels (in 2D) and written to the destination array. To compare the sequential image filter and parallel image filter here two different sources and destinations were used.

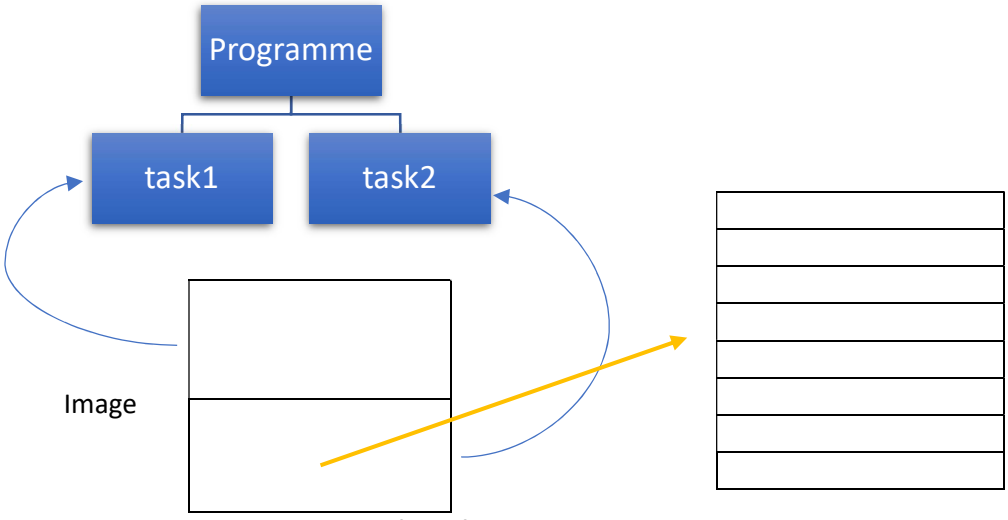

Fig-1: Parallel image filter fork join.

Steps:

- Create a class with constructor which takes src, dst, width and height of image.
- Define total number of threads which will be use in parallel execution.
- Calculate a suitable threshold.
- Create ForkJoinPool with nThreads.
- Define a method to do the work.
- Define a method to divide the work.
- Create tasks for parallel execution.
- Execute the tasks recursively.
- Properly synchronize the tasks
- Write the output image and check if it is correct or not also calculate speed up.

Description:

Here **ParallelFJImageFilter** class has been cleated with a constructor look like this **ParallelFJImageFilter (int [] src, int [] dst, int w, int h).** This constructor takes image source, destination, height, width as a parameter. **ParallelFJImageFilter** has an inner class called **InnerParallelFJImageFilter** with constructor **InnerParallelFJImageFilter (int [] src, int [] dst, int start, int end)**. Here start and end is basically the splitting image height's starting and ending point.

From main class when **apply (int nThreads)** is function called by the help of an instance of **ParallelFJImageFilter** class, then ForkJoinPool initiate with nThreads (nThreads is the number of threads we want to execute parallel program). After that the pool invoke the task which generated inside **compute ()** function in the **InnerParallelFJImageFilter** class. Mainly here is the parallel execution happened. Inside **compute ()** function, the program checks if the height of image is greater than threshold or not. If height is greater than threshold then it called function **filterProcess ()** which start the parallel image filtering. Otherwise, the program split the image height into 2 division and recursively called again and again up to finish the work. When execution has done, **invokeAll (task1, task2)** join and synchronize the tasks. This way the parallel image filter has been done.

Graph:

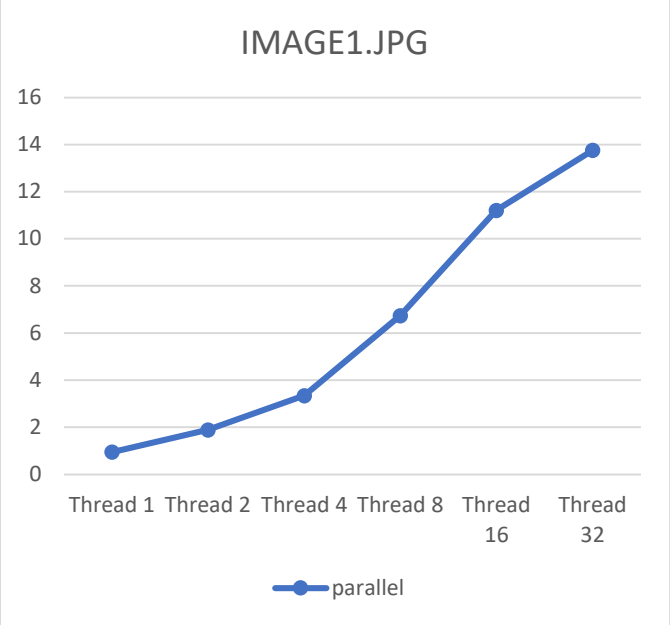

Fig-2: Image1 speed up curve.

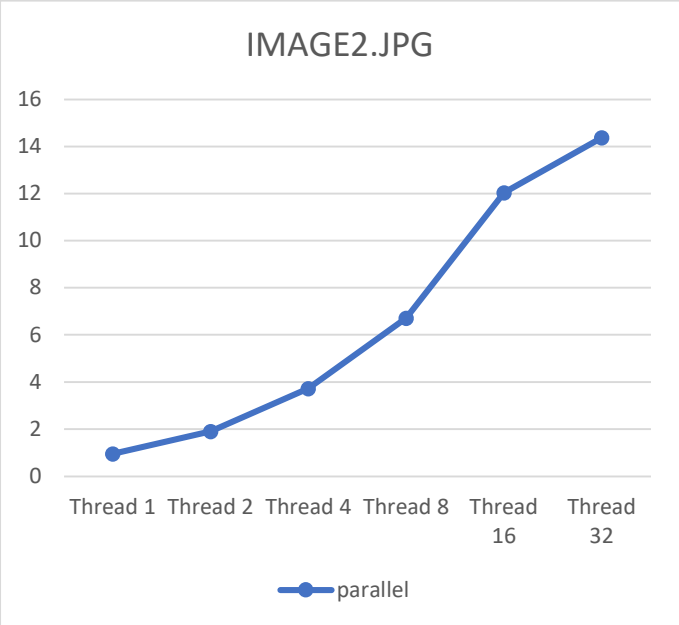

Fig-3: Image2 speed up curve.

Graph Description:

Fig-2 shows speed up curve of IMAGE1.JPG and Fig-3 shows for IMAGE2.JPG. In both graph the blue curve contains parallel speed up data. If we follow Fig-2 and Fig-3, we can see that the speed up curve of parallel execution, it is increasing with number of threads. That means the parallel execution is faster than sequential execution. Using 1 threads parallel execution the performance is almost same but better than sequential one. After that the speed up is increasing while the number of threads is increasing. The ALMA pc was used for these executions up to 32 threads. After using 16 thread the performance didn't increase so much.