Francis J. Patron Fidalgo

CIIC 4030 – 036

3/21/2021

Parallel vs Serial: Median Filter

Introduction

A parallel process means that several tasks can be done concurrently. This practice is very useful for optimizing run times of some algorithms and even some processes do not work without it. But is it always better than solving sequentially? Both serial and parallel approaches to solving problems have their advantages in each individual case. There is no general superior technique, as famous author once said: “quote”. A representation of a parallel process & serial can be seen in figure 1.0.

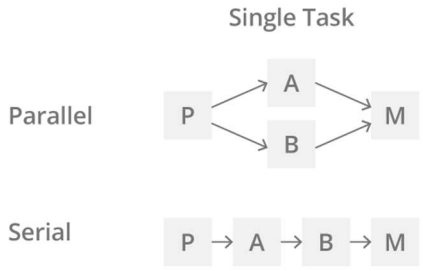


Figure 1.0

The median filter is a non-linear digital filtering technique, often used to remove noise from an image or signal. In this demonstration, we will use two simple implementations of a median filters in Scala, first in serial and then in parallel. Each of these will produce a filtered image (reduced noise) and run time.

The Code

The demonstration will be a simulation of an application that includes a server and a client. The client will send an image and then the server will return the output images and a report of execution times. The application uses the akka library.

- **The Server:** The server will first ask for the image to be filtered, then it creates two *Actor Systems*, one for each filtering implementation.

```
// System 1
val actorSystem = ActorSystem("ActorSystem")
val serialActor = actorSystem.actorOf(Props[serialServer], name = "serialActor")
val serialFuture = serialActor ? Serial(inputImage)

// System 2
val actorSystem2 = ActorSystem("ActorSystem2")
val parallelActor = actorSystem2.actorOf(Props[parallelServer], name = "parallelActor")
val parallelFuture = parallelActor ? Serial(inputImage)
```

Then, after each process has finished filtering the image, it saves the images filtered and shows the run time results.

- **Serial Implementation:** The serial implementation takes the input image and applies the median filter method. This method goes through each pixel in the image and saves all the neighboring pixel's RGB values. Then it replaces the original pixel with the median value of these neighbor pixels (see figure 1.2).

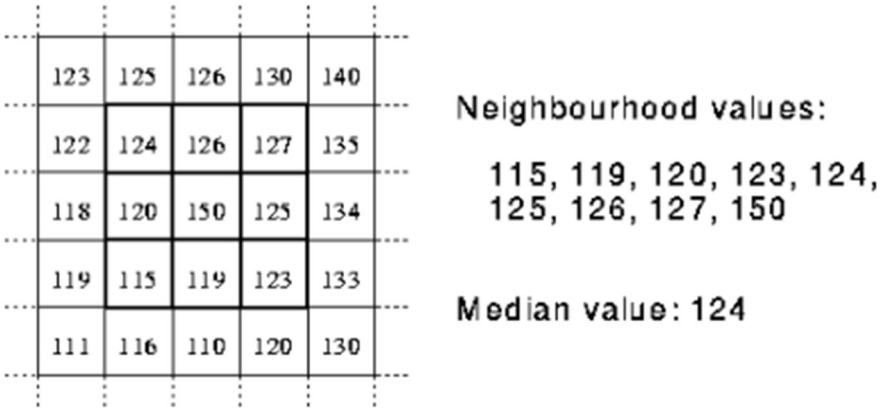


Figure 1.2: Obtaining median RGB value from neighbor pixels.

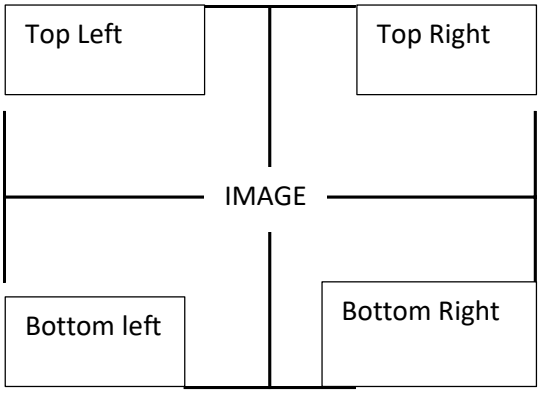
Here is the implementation of the median filter method.

```
def medianFilter(img: BufferedImage,left: Int, right: Int, up: Int, down: Int): BufferedImage = {
  // contains all RGB values of nearby pixels
  val neighbors = new Array[Color](9)
  // contains separate RGB values to sort & get medium
  val redValues, greenValues, blueValues = new Array[Int](9)

  for (x <- left until right){
    for ( y <- up until down){
      neighbors(0) = new Color(img.getRGB(x,y))
      neighbors(1) = new Color(img.getRGB(x+1,y+1))
      neighbors(2) = new Color(img.getRGB(x+1,y-1))
      neighbors(3) = new Color(img.getRGB(x-1,y-1))
      neighbors(4) = new Color(img.getRGB(x-1,y+1))
      neighbors(5) = new Color(img.getRGB(x,y-1))
      neighbors(6) = new Color(img.getRGB(x-1,y))
      neighbors(7) = new Color(img.getRGB(x+1,y))
      neighbors(8) = new Color(img.getRGB(x,y+1))

      for (i <- neighbors.indices) {
        redValues(i)=neighbors(i).getRed
        greenValues(i)=neighbors(i).getGreen
        blueValues(i)=neighbors(i).getBlue
      }
      //change RGB value
      img.setRGB(x,y,new Color(redValues.sortWith(_<_)(4),
            greenValues.sortWith(_<_)(4),
            blueValues.sortWith(_<_)(4)).getRGB)
    }
  }
  img
}
```

- **Parallel Implementation:** For the parallel method, we first divide the image in four sections:



Then, the same median filter method used in the serial process is applied to each divided section concurrently. When the server starts the actor for the parallel process, four additional child actors are created to execute the median filter. When each actor is finished processing their section of the image, the parent actor returns the results to the server. Here is the implementation with the four actors.

```
class topLeft extends Actor{
  def receive : PartialFunction[Any, Unit] = {
    case Serial(inputImage) =>
      val result = medianFilter(inputImage,  left= 1,inputImage.getWidth()/2-1, up= 1,inputImage.getHeight()/2-1)
      sender() ! result
  }
}
class topRight extends Actor{
  def receive : PartialFunction[Any, Unit] = {
    case Serial(inputImage) =>
      val result = medianFilter(inputImage,inputImage.getWidth()/2-1, inputImage.getWidth-1,  up= 1, inputImage.getHeight()/2-1)
      sender() ! result
  }
}
class botRight extends Actor{
  def receive : PartialFunction[Any, Unit] = {
    case Serial(inputImage) =>
      val result = medianFilter(inputImage, inputImage.getWidth()/2-1, inputImage.getWidth-1,inputImage.getHeight()/2-1, inputImage.getHeight()-1)
      sender() ! result
  }
}
class botLeft extends Actor{
  def receive : PartialFunction[Any, Unit] = {
    case Serial(inputImage) =>
      val result = medianFilter(inputImage,  left= 1, inputImage.getWidth/2-1,inputImage.getHeight()/2-1, inputImage.getHeight()-1)
      sender() ! result
  }
}
```

Results

- **Quality**

These are the images used and their output using their parallel & serial methods.

First Image:

Original

Parallel Serial

Second Image:

Original

Parallel Serial

As you can see, the image noise is reduced greatly. Both methods produce an image of similar quality. But if you look closely, you can see that the median filter does not work as well on the edges of the picture. This is due to the edge pixels not having pixels next to them. These are not able to get a full RGB medium of neighbor pixels like the center ones can. Additionally, exclusive to the parallel method, the line where the picture was divided suffers from the same effect. This is due because the filter is applied to each section of the image individually. This can be remedied by using the original image in conjunction with the partitioned image when calculating the RGB values.

- **Run Time**

The runtime is where the parallel and serial method differ the most. These are the results for the images above.

	Parallel Time (ms)	Serial Time (ms)
First Image (size: 514x511)	297	727
Second Image (size 300x300)	178	324

These results show that the serial method is slower than the parallel one at filtering the image. The parallel method is 2.45x faster in the first image and 1.82x in the second image. By observing the image sizes, we can conclude that the larger the image, the faster the parallel method is when comparing it to a serial approach. To better demonstrate this effect, two more images will be filtered. These will be extreme cases to further show the difference between these methods.

	Parallel Time (ms)	Serial Time (ms)
100 x 100 image	76	85
7680 x 4320 image	27037	82714

In the 100x100 image, the difference is now very small. But when the image gets extremely large, the serial method is now 3.1 times faster than serially. This proves that the parallel method is superior as the image size scales up.

Conclusion

When implementing a median filter, both parallel and serial approach have their benefits. The serial implementation is easier, and it results in a cleaner image. The parallel implementation is much faster as the image size increases at the cost of simplicity and image quality. This demonstration shows an instance for comparing a parallel approach vs a serial one, both are valid tools when considered their strengths and weaknesses.