

A Concurrent Image Filtering Project for Undergraduate Computer Science Education

Kenny Hunt, Ph.D.
Computer Science Department
The University of Wisconsin - La Crosse
La Crosse, Wisconsin, 54601
hunt.kenn@uwlax.edu

Abstract

Parallel and distributed computing has emerged as an increasingly important topic in Computer Science education driven largely by the predominance of multi-core systems, cloud computing systems and the use of GPU's for general computation. As fundamental concepts of concurrency become more tightly integrated with undergraduate education there is a heightened need for projects that have both a practical and accessible parallel solution. This paper presents a parallel programming project using Java that is based on median filtering digital images. The project is designed for use in an undergraduate computer science curricula and has been used in a programming languages course. The project has appealing features since 1) the problem domain lies within the field of multimedia and digital image processing 2) the problem domain is easily understood 3) the sequential solution is straightforward to implement 4) the output can be visualized since the output is a digital image and 5) parallelization provides a clear speedup where the controlling parameters can be tuned to exhibit both the cost and benefit of synchronization.

Image processing is a field of study within the broader domain of signal processing where a digital image, commonly referred to as the source image, is processed to produce an output digital image, which is commonly referred to as the destination image. Digital images are often filtered to produce an image that is more suitable for later computation or to produce an image that is more aesthetically or artistically pleasing than the source image. Median filtering is a non-linear process based on a straightforward statistical analysis of a local sub-region of an image. This paper gives an overview of median filtering digital images, provides a naive sequential solution, a design for parallelization, a discussion of the performance gains and concludes with a discussion of issues related to using the project in an undergraduate educational setting.

1 Background

Computer science educators have begun to reconsider how parallel computing is taught at the undergraduate level. While parallel computing has most often been taught as an upper-level undergraduate elective, if at all, many educators now advocate weaving parallel computing topics into even the CS0 or CS1 curriculum. Parallel computing has emerged as an increasingly important [1, 5, 6] due largely to the predominance of multi-core systems, cloud computing, and using GPU processing for general purpose computation. It seems inevitable that single cpu systems will only exist in either embedded systems or small portable devices such as cell phones or PDAs. Since programming concurrent systems requires skills and techniques that extend beyond those involved in sequential processing it is vital that CS educators inject concurrent programming concepts throughout the curriculum in appropriate ways and at appropriate levels of complexity and abstraction.

This paper presents a parallel programming project using Java that is based on median filtering digital images. The project is designed for use in an undergraduate computer science curricula and has been used in a programming languages course. The project has appealing features since 1) the problem domain lies within the field of multimedia and digital image processing 2) the problem domain is easily understood 3) the sequential solution is straightforward to implement 4) the output can be visualized since the output is a digital image and 5) parallelization provides a clear speedup where the controlling parameters can be tuned to exhibit both the cost and benefit of synchronization. In technical parlance, the project requires understanding of fork/join in addition to synchronization on a single method of a single shared object among the threads of execution.

2 Median Filtering

Image processing is a field of study within the broader domain of signal processing where a digital image, commonly referred to as the source image, is processed to produce an output digital image, which is commonly referred to as the destination image. Digital images are often filtered to produce an image that is more suitable for later computation or to produce an image that is more aesthetically or artistically pleasing than the source image.

Rank filtering is a non-linear process based on a statistical analysis of neighborhood samples. Rank filtering is typically used, like image blurring, as a noise-reduction pre-processing step in a multi-stage processing pipeline. The central idea in rank filtering is to make a list of all samples within the neighborhood and sort them in ascending order. The term rank means just this, an ordering of sample values. The rank filter will then output the sample having the desired rank.

The most common rank filter is the median filter where the output is the element having median rank; the middle element in the sorted list of samples. For an $M \times N$ rectangular neighborhood (where M denotes the width of the region and N denotes the height) there are MN samples in the neighborhood. These samples can be sorted by using ranks in

$[0..(MN - 1)]$ such that the median sample has an index, or rank, of $\lfloor MN/2 \rfloor$.

Throughout this paper we adopt the convention that the origin of the image is the element at the upper-left corner of the image, the column index, increases towards the right of the image, the row index increases downward, indexing is zero-based and that elements are referenced by their column followed by row. Figure 1(a) shows a 5×5 source image I that when median filtered using a 3×3 region produces the destination image, I of Figure 1(c). Figure 1(b) shows that the destination element $I(2, 2)$ is computed by first determining the 9 neighbors surrounding the source element $I(2, 2)$ and then determining the median of those 9 elements. In this example, the nine elements are, when sorted, 50, 52, 53, 55, 59, 60, 61, 65, 253 and have 59 as the median value. Hence, the destination element at location $(2, 2)$ is given as 59.

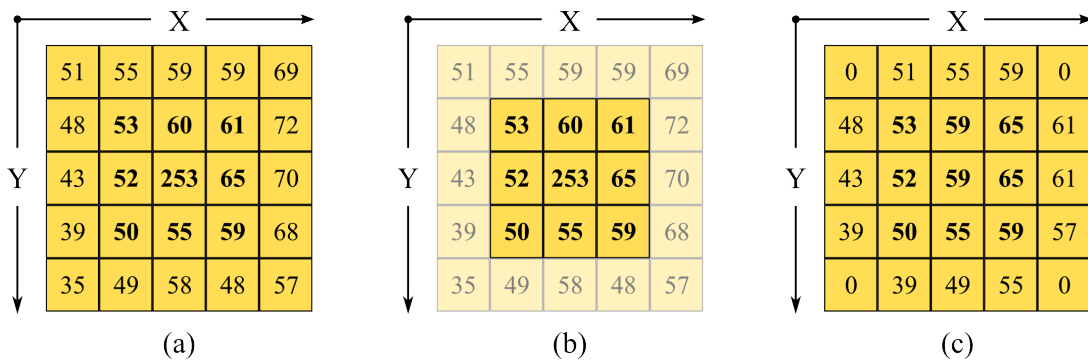


Figure 1: Median Filter Example.

Salt-and-pepper noise occurs when pixels of an image are incorrectly set to either their maximum (salt) or minimum (pepper) values. Median filters are exceptionally good at reducing this type of noise since they do not smear the noise across the image as blurring filters do. Salt and pepper noise is common in imaging systems that use CMOS or CCD based image sensors and commercial digital cameras often correct for such hardware defects by applying a median filter to the raw image data prior to shipping the image to the end user.

3 Project Description

Writing a sequential method to median filter a source image is a trivial task. Assume, for example, that the image is represented as a two-dimensional array of integers and that the size of the mask is give as a parameter. The Java methods gives a complete solution for median filtering a source image.

Listing 1: Sequential Filtering Method

```

1 public int getMedian(int[][] src,
2                     int x, int y,
3                     int m, int n, int[] reg) {
4     int counter = 0;
5     for(int dx = -m/2; dx <= m/2; dx++){

```

```

6     for(int dy = -n/2; dy <= n/2; dy++){
7         reg[counter++] = get(src, x+dx, y+dy);
8     }
9 }
10 Arrays.sort(reg);
11 return reg[reg.length/2];
12 }
13
14 public int[][] medianFilter(int[][] src, int m, int n) {
15     int width = src[0].length;
16     int height = src.length;
17     int[][] dest = new int[height][width];
18     int[] region = new int[m*n];
19     for(int y = 0; y < height; y++){
20         for(int x = 0; x < width; x++) {
21             dest[y][x] = getMedian(src, x, y, m, n, region);
22         }
23     }
24     return dest;
25 }

```

In this solution, the `medianFilter` method scans the entire array using a raster scan which proceeds from top-to-bottom and left-to-right within each row. For each element in the array, the median value of the $M \times N$ region surrounding the element is determined and placed into the destination array. The `getMedian` method is responsible for taking a single element of the source and determining the median at that element. Although not shown, the `get` method used in `getMedian` is a thin wrapper around the source array that effectively extends the source array infinitely in all directions such that all index values are valid. The `get` method returns a value of 0 if either of the two indices is invalid and returns the appropriate element of the source otherwise.

This algorithm is a naive implementation and other well-known approaches, while beyond the scope of this paper, obtain significant speed-ups by eliminating the need to sort regional elements and by leveraging the redundant computation that exist between overlapping regions of adjacent elements in the source [2, 3, 4]. Nonetheless, this algorithm can be easily understood at both the conceptual and implementation level but undergraduate CS students.

3.1 Threading without dynamic load balancing

Median filtering can achieve speedups that scale well with the number of system processing units. Consider, for example, a system having C cores or processing units. A parallel implementation might create C processes each of which has access to the source and destination arrays. The source image can then be partitioned into C tiles such that each of the processes is responsible for median filtering the elements within exactly one of these tiles. Under this scenario, although each process shares access to two objects, there is no possibility of deadlocks or race conditions since 1) the source image is a read-only data structure and 2) no two processes will write into the same destination element. Synchronization between processes is not required and hence the implementation is a straightforward refactoring of the sequential code.

In Java, the Thread class is used to represent a lightweight process or thread of control. The Thread class is designed to be subclassed such that 1) the subclass encapsulates the data and methods available to the thread as a normal Java class and 2) the run method is overridden and represents the code of the thread. Each thread has a start method which allocates resources for the thread, places the thread into the JVM scheduler and then invokes the run method. It is important that students understand that the run method should never be directly executed by the programmer but rather indirectly invoked through the start method. In addition, each thread has a join method that causes the calling thread to wait for the called thread to complete prior to proceeding.

Figure 2 shows the states in which a Java thread can exist during its lifetime. Each box of this figure depicts a state while the labelling of the arrows denotes the methods or operations which cause a transition between states. Figure 2 is not a detailed view of the thread life cycle but presents a summary of the life cycle. When a thread is first constructed it is placed in a 'new thread' state which corresponds to the start-state of Figure 2. When the start method of thread is invoked, the JVM allocates resources for execution of the thread and moves the thread into the runnable state. At some point, the run method is invoked by the JVM and the JVM scheduler is then responsible for moving the thread between runnable and running states as the run method progresses. Program-specific operations (i.e. blocking, sleeping or waiting) will cause the thread to move between the 'not runnable' and 'running' states. When the run method terminates, the thread has fulfilled its responsibilities and moves into the 'dead' state.

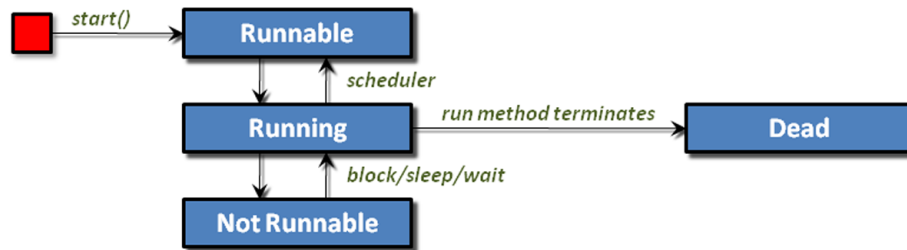


Figure 2: Life cycle of a Java thread.

Consider the filter method in the MedianFilter class of code listing 2. In this method we are given a source image and a mask size which is denoted by variables m and n. The method creates a destination array and partitions the source into four 500×500 regions each of which is represented as a Rectangle object. The Rectangle class is defined in the awt package and is part of the standard Java API. We then create four threads such that each thread is associated with one of the tiles and is responsible for median filtering only the elements of the source that fall within the tile. Each thread is then started after which we wait for them all to complete by joining on each thread. Once all of the threads are finished, the method returns the destination array.

Listing 2: Parallel Filtering Without Load Balancing

```

1 public class MedianFilter {
2   public static int[][] filter(int[][] src, int m, int n) {
3     // Create the destination array
  
```

```

4     int[][] dest = new int[src.length][src[0].length];
5
6     // Create a list in which to store the threads
7     List<MedianThreadNoLoadBalancing> threads =
8         new ArrayList<MedianThreadNoLoadBalancing>();
9
10    // Partition the image into four disjoint tiles
11    Rectangle[] tiles = {new Rectangle(0,0,500,500),
12                        new Rectangle(500,0,500,500);
13                        new Rectangle(0,500,500,500);
14                        new Rectangle(500,500,500,500) };
15
16    // Create four threads. One for each tile.
17    for(int i=0; i<rects.length; i++) {
18        threads.add(new MedianThreadNoLoadBalancing(src, dest, m, n, tiles[i]));
19    }
20
21    // start each thread
22    for(MedianThreadNoLoadBalancing t : threads) {
23        t.start();
24    }
25
26    // wait for each thread to complete prior to returning the destination
27    for(MedianThreadNoLoadBalancing t : threads) {
28        try {
29            t.join();
30        } catch(Exception e) {
31        }
32    }
33
34    return dest;
35 }
36 }

```

The filter method serves as the driver for the parallel solution. We must also write a MedianThreadNoLoadBalancing thread which will do the work of median filtering a single tile of the source image. This code is shown in listing 3.

Listing 3: Parallel Filtering Thread Without Load Balancing

```

1 public class MedianThreadNoLoadBalancing extends Thread {
2     private int[][] src, dest;
3     private int[] buffer;
4     private int m, n;
5     private Rectangle tile;
6
7     public MedianThreadNoLoadBalancing(int[][] src,
8                                       int[][] dest,
9                                       int m, int n,
10                                      Rectangle tile) {
11
12         this.src = src;
13         this.dest = dest;
14         this.m = m;
15         this.n = n;
16         this.tile = tile;
17         this.buffer = new int[m * n];
18     }
19
20     public int get(int x, int y) {
21         try {
22             return src[y][x];
23         } catch (ArrayIndexOutOfBoundsException e) {
24             return 0;
25         }
26     }
27 }

```

```

26
27 public int getMedian(int x, int y) {
28     int counter = 0;
29     for (int dx = -m / 2; dx <= m / 2; dx++) {
30         for (int dy = -n / 2; dy <= n / 2; dy++) {
31             buffer[counter++] = get(x + dx, y + dy);
32         }
33     }
34
35     Arrays.sort(buffer);
36     return buffer[buffer.length / 2];
37 }
38
39 public void run() {
40     for(int x = tile.x; x < tile.x + tile.width; x++) {
41         for(int y = tile.y; y < tile.y + tile.height; y++) {
42             dest[y][x] = getMedian(x, y);
43         }
44     }
45 }
46 }

```

The above code is limited in two important ways. First, the implementation creates only four threads regardless of the computing system on which it is running. On a two-core platform the creation of four threads serves only to thwart the computation since at least two of the threads will always be waiting while the others are performing their computation. Second, the implementation assumes that the source image is of dimension 1000×1000 and that each tile is then 500×500 . We will first consider how the code can be improved through more general tiling after which we will consider how to dynamically select an appropriate number of threads.

Tiling an image can be made more general by creating a class that is responsible for tiling a source of arbitrary dimension using tiles of arbitrary, though constant, dimension. We will construct a `Tiler` class that takes a rectangular region and tiles it into tiles. The tiles are given in raster-scan order as a series of `Rectangle` objects that are provided one at a time through a method known as `nextTile`. The implementation is given in code listing 4.

Listing 4: Tiler class for arbitrary tiling of a source image.

```

1 public class Tiler {
2     private int tileWidth, tileHeight;
3     private int nextTileX, nextTileY;
4     private Rectangle sourceDimension;
5
6     public Tiler(int[][] data, int tW, int tH) {
7         tileWidth = tW;
8         tileHeight = tH;
9         sourceDimension = new Rectangle(0,0,data[0].length, data.length);
10        nextTileX = nextTileY = 0;
11    }
12
13    public boolean hasMoreTiles() {
14        return nextTileX < sourceDimension.width &&
15            nextTileY < sourceDimension.height;
16    }
17
18    public Rectangle nextTile() {
19        if(!hasMoreTiles()) return null;
20
21        Rectangle result = new Rectangle(nextTileX, nextTileY, tileWidth, tileHeight);

```

```

22     nextTileX += tileWidth;
23     if(nextTileX >= sourceDimension.x + sourceDimension.width) {
24         nextTileX = 0;
25         nextTileY = nextTileY + tileHeight;
26     }
27
28     result = result.intersection(sourceDimension);
29     return result;
30 }
31 }

```

Note in listing 4 that given a source image (represented as a two dimensional array) and a tile dimension, the tiler will return a sequence of `Rectangle` objects through sequential calls to `nextTile`. Note that the `nextTile` method trims tiles to an appropriate size when the tile would otherwise extend beyond the source image boundary. The `intersection` method is an exceptionally convenient tool for this necessary computation. When the entire source image has been partitioned, subsequent calls to `nextTile` will return nulls.

Given the `Tiler` class, the `filter` method of the `MedianFilter` class can now be rewritten as shown below. Note that the code is still encumbered by the assumption that the source image will be completely partitioned into four distinct regions by selecting a 500500 tile size.

Listing 5: MedianFilter driver that uses the Tiler class.

```

1  // Partition the image into four disjoint tiles of 500 by 500 size
2  Tiler tiler = new Tiler(src, 500, 500);
3
4  // Create four threads. One for each tile.
5  for(int i=0; i<rects.length; i++) {
6      threads.add(new MedianThreadNoLoadBalancing(src, dest, m, n, tiler.nextTile()));
7  }

```

This leads us to consider how many threads to construct and how the source image can be partitioned in an efficient manner. Any solution that divides the entire problem space into a pre-determined number of partitions is not likely to perform well on systems running under heavy load. Consider, for example, a quad core machine which is running a computationally expensive algorithm unrelated to median filtering on one of the four cores. If median filtering is performed on this machine, the likely result is that the four threads (each of which is responsible for approximately 1/4 of the problem space) are assigned to each of the four cores and hence three of the threads will complete long before the thread that has been assigned execution on the busy core. Even if the JVM attempts to load balance the computation, there are essentially five processes vying for four cores on which execute.

4 Threading with dynamic partitioning

A preferable solution to the technique described above is to create one thread per core but *not pre-partition the problem space*. We will instruct the threads to filter a series of small

partitions such that each thread requests a relatively small tile of the source image, filters that tile, and repeats this process until there are no more tiles left to filter. Faster threads are then able to process more tiles than slower threads and hence the workload is more evenly distributed across the system when considered as a whole.

Java provides a convenient method for determining the number of cores on a system which is accessible through the Runtime class. The `availableProcessors()` method of the JVM Runtime object will return the number of processors (or cores) available to the application. Our final design utilizes three cooperating classes as shown in the UML class diagram of Figure 3. The `MedianFilter` class serves only as the public interface for the parallel filtering algorithm. The `filter` method constructs a `Tiler` object and then constructs one `MedianThread` per processor such that each thread shares the source image, the destination image, and the tiler object.

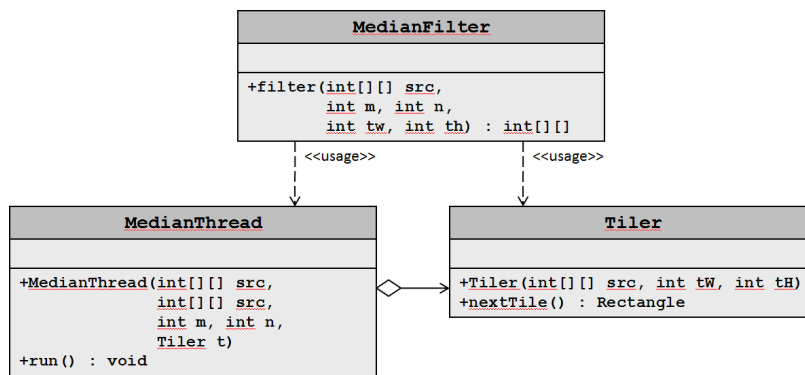


Figure 3: UML class diagram for the parallel solution.

The only real change required of the `MedianThread` of code listing 3 is that the `run` method should now loop over all available tiles. This modification is shown below in listing 6. Of course, the constructor must now accept a `Tiler` object in place of a single `Rectangle` object since the thread is filtering a sequence of tiles (represented by the `Tiler`) rather than a single tile.

Listing 6: `MedianThread` that uses dynamic partitioning.

```

1 public void run() {
2     Rectangle tile = null;
3     while((tile = tiler.nextTile()) != null) {
4         for(int x = tile.x; x < tile.x + tile.width; x++){
5             for(int y = tile.y; y < tile.y + tile.height; y++){
6                 medianFilter(x, y);
7             }
8         }
9     }
10 }
  
```

The only change required of the `Tiler` is that the `nextTile` method be synchronized since it is a shared object whereby multiple threads will be competing for access to the next tile. The only change to the `MedianFilter` driver is that a single `Tiler` object be created

(rather than creating a series of tiles) and this tiler object is then given to each thread that is constructed. This modification is shown in listing 7.

Listing 7: MedianFilter that uses dynamic partitioning.

```
1 // Create a Tiler
2 Tiler tiler = new Tiler(src, tileWidth, tileHeight);
3
4 // Create the threads. One per core.
5 for(int i=0; i<Runtime.getRuntime().availableProcessors(); i++) {
6     threads.add(new MedianThreadNoLoadBalancing(src, dest, m, n, tiler));
7 }
```

5 Conclusion

Students were asked to complete the project by developing the MedianFilter and the MedianThread classes. Students were provided with the sequential solution and a sketch of the Tiler class. They were asked to test their implementation by varying 1) the number of threads created 2) the mask size and 3) the tile size. The interplay between these three parameters exhibits very interesting behavior that exposes central concepts of any parallel processing task.

Increasing mask sizes will always require increased processing time regardless of the number of threads or the tile size. Increasing the number of threads will yield speedups until the thread count exceeds the available processors. The amount of speedup is, however, also dependent on the tile size since too large a tile size reduces to an essentially sequential implementation while too small a tile size also reduces to an essentially sequential computation! Consider a 1×1 tile size (a scenario that most students like to suggest as an option). In this scenario, a large amount of computational overhead is spent in managing thread access to the tiler since threads are constantly blocking on the nextTile method. This suggests selection of an optimal tile size that is dependent on the number of available processors.

Figure 4 illustrates the interplay between tile size and number of threads. The vertical axis denotes the computation time required to median filter a 4188×6570 grayscale image on a 2.33 GHz quad-core Xeon processor. The horizontal axis denotes the tile size where tiles are always square. Each line on the chart denotes the number of threads used to median filter the source image.

Note that a single thread provides the worst relative performance regardless of tile size since this obviously reduces the parallel solution into a sequential implementation. Also note that a 1×1 tile size also yielded poor performance due to the threads competing for tiles through the single tiler object. Tiles on the order of 16 through 1024 in dimension yielded relatively constant performance while the data series converge towards the right of the chart since larger tile sizes reduce the problem into sequential performance. Increasing the thread count yields an increase in computational efficiency until the thread count exceeds the available processors where note the eight-thread series gives no improvement over the

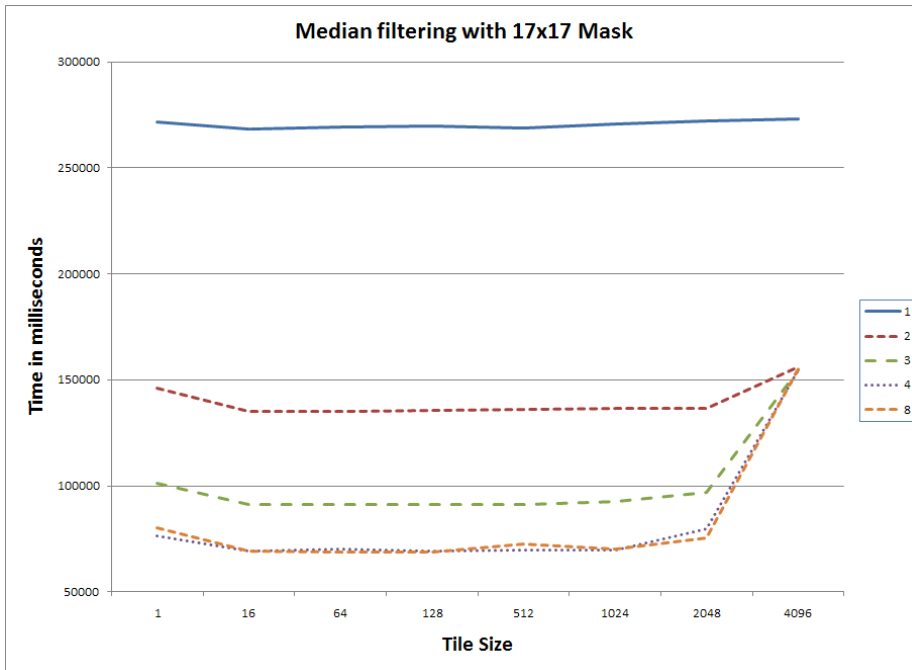


Figure 4: Effect of tile size and thread count.

four-thread series as we would expect since the tests were run on a quad-core system.

Students appeared to be interested in the project primarily as it provided an excellent venue for exploiting parallelism in an accessible and interesting application domain. The majority of students were able to produce reasonably accurate implementations in addition to written comments that indicated an understanding of the interplay between thread count, tile size and mask size. As an interesting side note, it became apparent that students generally struggled more with the concept of the `Tiler` class than with concepts of concurrency and hence the author recommends that the unsynchronized `Tiler` class be either explicitly provided to the students or that the `Tiler` class be discussed in some detail prior to assigning the project.

References

- [1] ARMONI, M., AND BEN-ARI, M. The concept of nondeterminism: its development and implications for teaching. *SIGCSE Bull.* 41, 2 (2009), 141–160.
- [2] ATAMAN, E., AATRE, V., AND WONG, K. A fast method for real-time median filtering. *Acoustics, Speech and Signal Processing, IEEE Transactions on* 28, 4 (1980), 415–421.
- [3] COLE, R., AND YAP, C. A parallel median algorithm. *Inf. Process. Lett.* 20 (1985), 137–139.

- [4] HUANG, T., YANG, G., AND TANG, G. A fast two-dimensional median filtering algorithm. *Acoustics, Speech and Signal Processing, IEEE Transactions on* 27, 1 (1979), 13–18.
- [5] PAPRZYCKI, M. Education: Integrating parallel and distributed computing in computer science curricula. *IEEE Distributed Systems Online* 7 (2006).
- [6] PATTERSON, D. A. Computer science education in the 21st century. *Commun. ACM* 49, 3 (2006), 27–30.