1 General Implementation Comments

For both implementations of this problem, the program requires 2 mandatory arguments. These mandatory arguments are the interior grid size and the number of threads that are being executed, in that order. In addition to these arguments, there are 4 optional arguments allowing you to supply new values for the left, top, right, and bottom exteriors (in that order). The default values for these optional arguments are 1.0, 1.0, 80.0, and 80.0 respectively.

In both implementations, there are some obvious performance bottlenecks that exist. For example, in both implementations I repeatedly copy values from my new grid into my old grid. Although this operation is threaded, it is still a performance impact that is unecessary. In the textbook readings, Andrews describes an implementation in which you would merely swap the roles of the old and new grids after each iteration, thus removing the need for copying the values.

2 Java Timing Results

Throughout the course of my testing the Java version, I was unable to receive any significant speed increases by increasing the number of workers assigned to the grid. In fact, I would frequently encounter a speed decrease. Here are some of my results from trying different grid sizes with different numbers of workers:

Grid Size	1	2	4	8	16
16	38285	17775	48946	44460	11755
32	47037	83092	73227	38341	136792
64	67673	76770	167441	274588	155512
128	90840	110598	187661	137327	514625
256	209224	259648	268598	400483	501207
512	905593	652143	386134	638053	1907518
1024	3029917	2015317	1218102	1780524	2776852

The timings presented are all presented in microseconds.

I tested the timings with the following bash script:

```
for gridSize in 16 32 64 128 256 512 1024
do
    for workers in 1 2 4 8 16
    do
        java Jacobi $gridSize $workers
    done
done
```

3 C Timing Results

I did not have the opportunity to get an implementation of the C version entirely working because of issues with the semaphores in the POSIX library. The version that I currently have working, after staying in Isabelle's office until 8PM, completes about half of the array and is approaching a level of correctness. However, if I was able to get a version of this working, the steps that I would have followed to test it would have been to perform the exact same tests on the C version that I had on the java version and then generate charts to compare their performance. When I was working solely on the sequential version, I noticed an order of magnitude performance increase with the C version over the java version. I would have expected similar results with the multithreaded version, possibly even giving a little bit higher of a speed advantage to C. The lack of the JVM's overhead in creating the threads probably would have yielded another small performance boost over the Java version.

With the understanding that I was unable to get a working C version, I will use Dr. Homer's binary to perform my tests. The code to execute the tests is the same as the code used to execute the Java tests, except that I will be executing "patrickJacobi" instead of "java Jacobi". Here are my timing results for Patrick's C implementation:

Grid Size	1	2	4	8	16
16	912	2857	13356	9218	16291
32	6636	8537	17460	19047	31604
64	32385	27264	24357	37363	84323
128	121163	48745	52577	61630	95762
256	481900	169912	115359	251331	229328
512	60707	686240	532937	563016	537420
1024	366069	960171	587908	167930	191562

From this data, we can notice that the addition of more workers increased the performance up until a point, and then the performance would either plateau or go back down. The performance detriment would likely result from the overhead of managing the barriers and threads. Compared to the Java version, the C version exhibited significant performance increase.