UNIVERSITY OF WISCONSIN PARKSIDE

Analysis of Thread Synchronization

Homework 2

CSCI 370: Operating Systems

Author: *Tanner Smith*

Instructor: *Susan Lincke*
Department: Computer Science
Date: March 27, 2023

Contents

1 Introduction

In this lab we explore some of the key concepts behind threaded programming synchronization. Whenever working with multiple threads that share resources programmers need to be extra careful to ensure the threads do not interfere with each other. If a programmer is not careful the threads may overwrite values, encounter race conditions, or other types of data dependencies issues. In this lab we will solve a problem using one, and many synchronized and unsanctioned threads then compare their results. The end goal is to compare and contrast accuracy, speed, and fairness among threads depending on whether they were synchronized or not.

2 Method

2.1 The Test Problem

To explore this concept we will be using a program that calculates and propagates the average value of a 2 dimensional array, which represents a heat plate. The matrix/heat place will start with constant heat values on the sides and every other value starts at zero. Each time a value gets ”updated” it gets recalculated to average of its 4 adjacent values. The difference between the old value and the new value is considered as ”error”. Initially the values will change drastically but over enough iterations the error will get very low as the updated values are very similar to their old values. The plate is ”done” once the total error of the grid is about 5. We will use this heat plate pseudo problem to explore thread synchronization by attempting to solve it using one, many threads with no synchronization measure, and many threads with synchronization measures.

2.2 System Configuration

CPU: AMD Ryzen 7 2700x 8 core 16 thread Processor, 3700Mhz.
Memory: 32GB of DDR4 3000Mhz
OS: Windows 11 Pro ver 10.0.22000 Build 22000

2.3 Test Runs and Configuration

For all runs the configuration of the heat values for each side of the plate was:
Left Side = 20, Top Side = 80, Right Side = 30, Bottom Side = 40. In total they sum up to 160. Tests we conducted on multiple plate sizes, different thread counts, barrier on/off, and different data types float/double. For each plate size, threads of size [1, 4, 8, 16] were tested, and each of these tests were ran twice, one with the barrier on and off. Then, those tests were repeated twice once again for the float and double version. With one exception... tests of thread size 1 were only ran once for each plate size and data type used once and are not different for each barrier type. Although the tables show thread size of 1 in them, note that the barrier on and barrier off version are the same. In total 42 tests were ran.

3 Results

Below are several tables and charts of the run results. These tables and charts provide useful information on the run times, thread iterations, and differences between the different run configurations.

(note that all time listed is in seconds)

3.1 Overall Run Time and Error

This sub section contains tables of the general results of all the test runs

Table 1: Double Version — No Barrier

Plate Size	Num Threads	Time	Final Avg	Final Err
250	1	8.825	39.5	5
	4	2.433	39.5	4.99
	8	1.583	39.5	4.99
	16	1.433	39.28	5.3
500	1	162.757	39.5	5
	4	38.696	39.49	5
	8	22.503	39.49	5
	16	16.413	39.44	5.17
750	1	869.469	39.49	5
	4	201.224	39.49	5
	8	104.191	39.49	4.99
	16	82.036	39.48	5.06

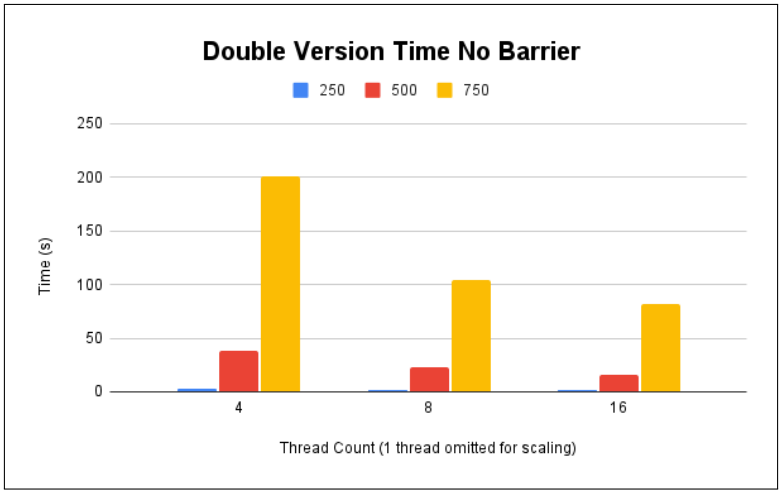


Table 2: Double Version - With Barrier

Plate Size	Num Threads	Time	Final Avg	Final Err
250	1	8.825	39.5	5
	4	3.017	39.5	5
	8	2.634	39.5	5
	16	2.869	39.5	5
500	1	162.757	39.5	5
	4	46.577	39.5	5
	8	33.494	39.5	5
	16	30.601	39.5	5
750	1	869.469	39.49	5
	4	243.04	39.49	5
	8	146.966	39.49	5
	16	134.438	39.49	5

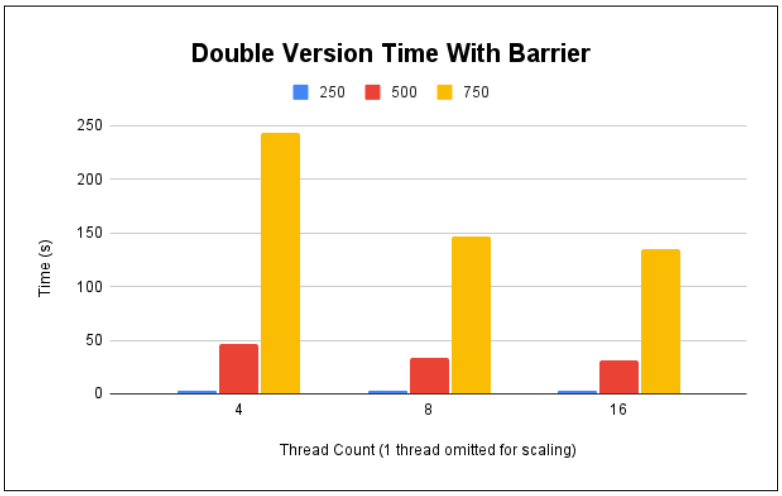


Table 3: Float Version — No Barrier

Plate Size	Num Threads	Time	Final Avg	Final Err
250	1	8.016	39.5	5
	4	2.451	39.5	5
	8	1.767	39.5	4.99
	16	1.726	39.44	5.3
500	1	131.566	39.5	5
	4	35.708	39.5	5
	8	21.324	39.49	5
	16	17.46	39.44	5.08
750	1	788.309	39.49	5
	4	186.423	39.49	5
	8	104.349	39.49	5
	16	89.081	39.46	5.14

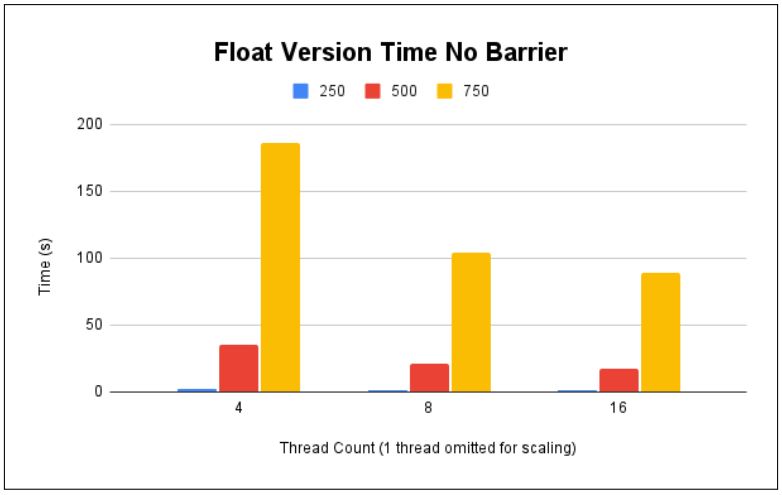
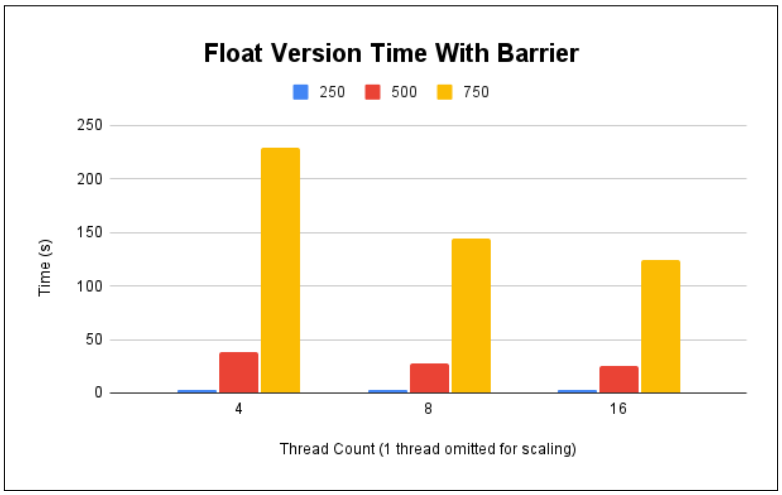


Table 4: Float Version - With Barrier

Plate Size	Num Threads	Time	Final Avg	Final Err
250	1	8.016	39.5	5
	4	2.874	39.5	5
	8	2.845	39.5	5
	16	2.974	39.5	5
500	1	131.566	39.5	5
	4	38.006	39.5	5
	8	27.184	39.5	5
	16	25.207	39.49	5
750	1	788.309	39.49	5
	4	228.893	39.49	5
	8	143.837	39.49	5
	16	123.8	39.49	5



3.2 Double vs Float Time Difference

These tables show the time for the double version minus the time of the float version. If the cell is green then the float version was faster, and if the cell is red then the double version was faster.

Table 5: No Barrier - Difference between Double and Float

Plate Size	Num Threads	Time
250	1	0.809
	4	-0.018
	8	-0.184
	16	-0.293
500	1	31.191
	4	2.988
	8	1.179
	16	-1.047
750	1	81.16
	4	14.801
	8	-0.158
	16	-7.045

Table 6: With Barrier - Difference between Double and Float

Plate Size	Num Threads	Time
250	1	0.809
	4	0.143
	8	-0.211
	16	-0.105
500	1	31.191
	4	8.571
	8	6.31
	16	5.394
750	1	81.16
	4	14.147
	8	3.129
	16	10.638

3.3 Barrier vs No Barrier

The tables below show the difference in time it took for the version with the barrier enabled and the version with the barrier disabled. The darker the shade of green represents how much faster the non barrier version ran compared to the barrier version.

Table 7: Double Version - Difference between Barrier on minus off

Plate Size	Num Threads	Time
250	1	0
	4	0.584
	8	1.051
	16	1.436
500	1	0
	4	7.881
	8	10.991
	16	14.188
750	1	0
	4	41.816
	8	42.775
	16	52.402

Table 8: Float Version - Difference between Barrier on minus off

Plate Size	Num Threads	Time
250	1	0
	4	0.423
	8	1.078
	16	1.248
500	1	0
	4	2.298
	8	5.86
	16	7.747
750	1	0
	4	42.47
	8	39.488
	16	34.719

3.4 Thread Iterations

The tables below show the number of times each thread ran during the tests. Blacks cells represent when the thread was not used since tests were conducted for [1, 4, 8, 16] threads. The cells are color scaled such that green is 0 and red is the highest thread iteration for that table.

Table 9: Float Version - No Barrier - Thread Iterations

Thread	Plate Size 250				Plate Size 500				Plate Size 750			
0	24800	24312	25693	23419	99587	99568	99046	132347	224292	224844	223994	148225
1		25099	26112	37355		99039	98263	75277		223108	222882	147914
2		24440	24737	34725		99948	98690	75957		224780	223651	193111
3		25569	23924	41719		99698	98478	87450		224484	223176	191876
4			24827	39018			100599	87399			225293	359845
5			24996	36106			99836	146410			224576	361866
6			26034	25206			102542	145605			227306	326943
7			26586	24167			100587	137726			225708	329142
8				23590				134093				153729
9				35784				78731				155091
10				32851				80057				201915
11				39848				90175				199470
12				38708				86019				362594
13				23791				147960				362320
14				26202				147354				327535
15				23366				136257				328793

Table 10: Double Version - No Barrier - Thread Iterations

Thread	Plate Size 250				Plate Size 500				Plate Size 750			
0	24800	24939	24996	30631	99612	99607	100370	108378	224436	224441	223522	274475
1		24677	25017	27566		99241	98726	111526		223986	223733	274503
2		24915	24879	28989		100283	98474	112099		224611	224332	232685
3		24772	24735	33868		99316	99056	105740		224773	224639	234854
4			24748	33246			99869	110699			224423	247611
5			24969	31491			100721	89569			224719	246403
6			25141	27607			101999	91851			225412	177527
7			25204	30423			101603	101320			225304	183461
8				33717				99751				276553
9				24573				110687				278971
10				28617				113765				234950
11				33921				108493				236442
12				34252				109703				247698
13				28769				91160				245152
14				27232				93621				178935
15				33553				102404				178194

Table 11: Float Version - With Barrier - Thread Iterations

Thread	Plate Size 250				Plate Size 500				Plate Size 750			
0	24800	24805	24808	24802	99587	99592	99596	99576	224292	224313	224302	224246
1		24805	24808	24802		99592	99596	99576		224313	224302	224246
2		24805	24808	24802		99592	99596	99576		224313	224302	224246
3		24805	24808	24803		99592	99596	99576		224313	224302	224245
4			24808	24802			99596	99576			224302	224245
5			24808	24802			99596	99576			224302	224246
6			24808	24803			99596	99576			224302	224246
7			24808	24802			99596	99575			224302	224246
8				24802				99576				224246
9				24802				99576				224246
10				24803				99576				224246
11				24802				99575				224246
12				24803				99575				224246
13				24802				99575				224245
14				24802				99576				224245
15				24802				99576				224246

Table 12: Double Version - With Barrier - Thread Iterations

Thread	Plate Size 250				Plate Size 500				Plate Size 750			
0	24800	24807	24803	24796	99612	99618	99625	99606	224436	224450	224449	224428
1		24807	24803	24796		99618	99625	99605		224450	224449	224428
2		24807	24803	24796		99618	99625	99606		224450	224449	224429
3		24807	24803	24796		99618	99625	99606		224450	224449	224429
4			24803	24796			99625	99605			224449	224429
5			24803	24797			99625	99606			224449	224429
6			24803	24796			99625	99605			224449	224429
7			24803	24797			99625	99606			224449	224429
8				24797				99606				224429
9				24796				99605				224429
10				24796				99605				224428
11				24796				99606				224429
12				24796				99605				224429
13				24796				99606				224429
14				24796				99605				224429
15				24796				99606				224429

4 Analysis

4.1 Single Thread Solution and Asymptotic Analysis

While the single thread solution by itself is not very interesting we can use its run time to preform an asymptotic analysis on the overall run time of the algorithm. It is very obvious every iteration needs to run n^2 times assuming n is the size of the heat plate. Then the outer loop runs based off a function of n therefore the the asymptotic time complexity of the algorithm must be $\mathcal{O}(n^3)$ right? Well while that may seem like the simple and correct answer it is ultimately wrong and we come to the conclusion why it is wrong. We know for sure the inter loops run n^2 times to iterate over the heat plate, but how many times does the outer loop actually run? We know that the outer loop will run until the total error is less than 5. Looking at the data we see the total iterations increase as n increases, but how exactly? Is it scaling logarithmic, linearly, or exponentially? To find out let's take the number of iterations ran for the single thread runs and see if we can find how they relate to n. For $n = 250$ the outer loop ran 24800 times and for $n = 500$, 99612, and for $n = 750$, 224436. If we divide the iterations by the n value...

$$\frac{24800}{250} = 99.2 \quad \frac{99612}{500} = 199.224 \quad \frac{224436}{750} = 299.248$$

Looking at the values calculated above we can see that the iterations are not scaling linearly. If it was, then all the ratios of iterations divided by n would be the same value. Instead we see each time we increase n the number of iterations increases by about 100. With this information we can calculate a formula of how many times the outer loop is running in relation to n. For simplicity we are going to assume the scaling is rounded to intervals of 100. We can then define the estimate of times the outer loop will run with a function $f(n)$.

$$f(n) = \frac{2}{5} * n^2$$

We can then validate our equation for the outer loop by plugging in n and comparing it to the actual iterations.

$$f(250) = \frac{2}{5} * 250^2 = 25000 \approx 24800$$

$$f(500) = \frac{2}{5} * 500^2 = 100000 \approx 99612$$

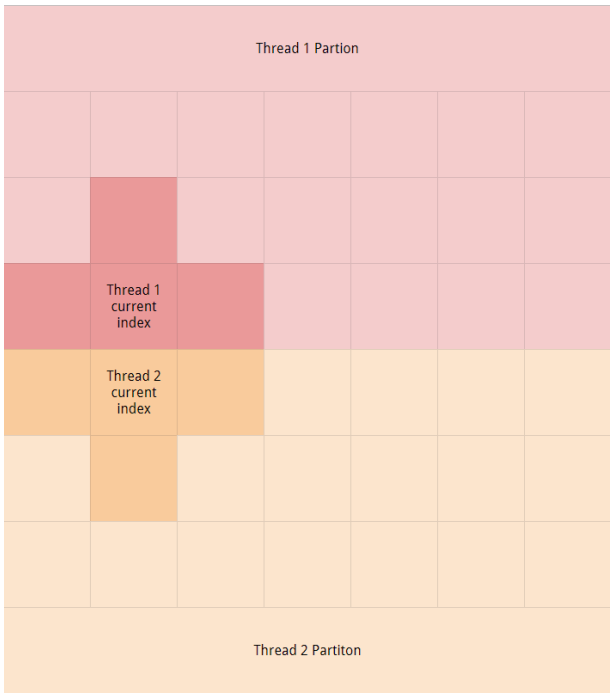
$$f(750) = \frac{2}{5} * 750^2 = 225000 \approx 224436$$

Therefore we can confidently say the outer loop also runs at a time complexity of $\frac{2}{5} * n^2$. If we then take that result and multiply it by the times the inner loops run we get $\frac{2}{5} * n^4$, with Big-O notation $\mathcal{O}(n^4)$. I also think it's important to note this type of analysis would be much more difficult to perform using data from the multiple threaded versions. First of all all the threads would need to have run equally, and the presumed thread speedup would need to be factored out of the time value used.

4.2 Multi-Thread Solution/Thread Scaling

Instead of just having one thread iterate through and update the heat plate we can use multiple threads. Each thread will receive equal sized sub partitions of the total heat plate for them to individually update. Using multiple threads we should see a k speedup for k threads. Looking at the tables and charts in section 3.1 we can see that we are getting close to the theoretically speedup except jumping from 8 threads to 16 threads. The relative speedup is more pronounced with the larger heat plate sizes. This this speedup in both the no barrier and with barrier versions. However the parentage of speedup scales slightly differently between versions which we will elaborate more on in their individual subsections.

Another thing to mention, although it happens rarely, is the potential race condition between threads at the borders of their partitions. If two threads happen to be updating values adjacent of each other along the border of their partitions at the same time then a race condition occurs. Both threads are reading values that the other thread is in the process of changing. See the image below for a clear visual.



This was a known problem that may occur but was left in the final code for two reasons. The first reason is the likely hood of this occurring is not very often. Secondly if this were to occur it wouldn't really matter that much since the values are going to converge the data being one iteration older or newer won't really matter.

4.3 No Barrier

The no barrier version was able to finish iterating through the heat plate but with some notable issues. Looking at the tables in section 3.1 only in the non barrier versions we see the final error sometimes be greater than 5. Why is that? The code is supposed to run until the total error is less than 5. This issue arises because of how each thread checks to see if the program should stop and because the threads are not synchronized. Every time a thread completes an iteration it updates it's index in the errors array. (The errors array is an array of size n threads where each thread reports the error between its iterations). Then the thread sums the errors in the errors array to check and see if the total error of all the threads is less than 5. If the error is less than 5 then it signals every other thread to stop. However because the threads are not synchronized with a barrier while one thread may check and find the errors array sum is less than or equal to 5. Then at the same time another thread just updated is local error in the errors array such that the sum is no longer less than or equal to 5. But the first thread already checked and found the global error to be less than 5 and signals the threads to all stop. This results in the global error after the threads stop to be greater than 5. Although it sometimes is stopping earlier than it should, the global average value at the end is still very similar to consistent single thread version of 39.5.

4.4 Barrier

The barrier version ran almost flawless, with one very minor issue. All the final errors were less than or equal to 5 as expected, and the final grid average was extremely close to the expected average. The only issue with the barrier solution is that sometimes the threads ran 1 more or 1 less times than the other threads. Ideally we would expect them to be all the exact same number instead of one off. We already described in the no barrier section how each thread checks to see if the program should stop. The major difference in the barrier version is that all the threads wait at the barrier after it updates its local error in the errors array. I believe the issue happens when the threads are finally released from the barrier and the total error is not less than or equal to 5 yet. The first thread checks if it should stop and sees no it should not. Then it starts it's next iteration and updates its local error making the sum now be less than or equal to 5. A slower thread that may have been delayed by the operating system finally goes to check the error and sees it is less than 5 and signals the all the threads to stop. This is likely why the barrier version has thread iterations in the range of +/- 1. This issue could likely be fixed with another barrier or better barrier placement. However, the issue for this problem is so minuscule that it doesn't really matter. If you were programming something critical that required all threads to run the exact same number of times then this would not be a minor issue anymore.

4.5 Comparing Barrier vs No Barrier

In the previous two sections we talked about the accuracy of the two version of the multi threaded. The no barrier version was less accurate because it was stopping to early and the barrier version was much more accurate and all threads stopped at the same time +/- 1 iteration. We have yet to talk about how using a barrier affected the overall run time between the two programs and here is where we are going to do that. The tables in section 3.3 lists the difference in run time between the barrier version and non barrier version. This is an expected result because in the non barrier versions the threads are always moving on to their next iteration while the version with the barrier enabled threads must wait until all other threads complete an iteration to start their next. The time threads spend waiting may vary depending on the scheduler of the operating system. One thread might get done fast because it is constantly being scheduled while another thread may get done slower as it has to wait more time to be scheduled to run. We can see this behavior in section 3.4 in the tables where the barrier is off. In the double version of plate size 750 with 16 threads, thread 6 ran 177527 times while thread 0 ran 274475 times. We can see clearly that thread 0 was scheduled to run much more often than thread 6.

Remember earlier in section 4.2 where we talked about multi-thread scaling? We briefly mentioned that the two version of barrier and no barrier didn't see the same ratio in thread scaling. While both saw a speedup as k threads increased, the speedup of the barrier version slowed down as we increased threads. The speedup of the non barrier version did not slow down as much as we increased threads. The explanation is related to what we just described. The threads must wait in the barrier version each iteration until all other threads run. And we just talked about how the operating system is inconsistent with scheduling threads. In the barrier as we increase threads we get the parallel thread compute speedup, but we also are increasing the average wait time at the barrier as more threads need to be scheduled and completed. This is the most likely reason as to why the barrier version gets a lower speedup as threads increase compared to the no barrier version.

4.6 Comparing Double vs Float version

In an attempt to try and speedup the run time of the program I created a duplicate version but replaced all instances of type double with type float. In java a double, more formally called double precision float, is 64 bits long, and a float is 32 bits long. A double is really just a float with double the number of bits, hence the name. The ideas behind trying this out was... One, we really don't need a double precision float since we are rounding our answers at the end to the point of where the extra precision doesn't matter. Two, with less bits being used per calculation we should see a speedup in run time. That was my initial thought process to test this method. Other than trying to speedup run time we also half the memory use of the program. In the event memory usage is critical and precision is not, then opting to use floats over doubles is a good idea. Looking at the tables in section 3.2 we can see that the float version typically ran faster than the double version, with some variations. Sometimes the double version ran faster but when it did run faster than the float version its time difference was very small. Compare that to the time save we see in the float version is much greater on average. Clearly the float version is running faster than the double version. But something weird is happening. Looking at section 3.4 of the thread iterations on tables 11 and 12, (ignoring single thread runs since only 1 was

ran for all tables), we can see that the float and double version seems to be running almost the same number of iterations. Originally I thought the float version would run a lot less iterations because it would end faster, but it is running the same iterations but each iteration takes less time. After seeing these results I did some more thinking. Even though the floats are 32 bits long and the doubles are 64 bits long, the registers on my x86 based processor are 64 bits long. So the processor is just as capable of calculating a double at the same speed of a float since it can fit them both all within one register. If we were to use something like a Big Decimal that used more than 64 bits we would need to use multiple registers to do partial sums and then put it back together. So where is this speedup coming from? My best theory of this seen speedup is due to memory and caching. A processor only has small finite amount of cache. My processor specifically having and L1 cache size of 768KB, L2 cache size of 4MB, and L3 cache size of 16MB. Because floats take half the memory as a double we can fit twice as many floats into the cache than doubles. It is very likely that the speedup seen in the float version of the program comes from the data more likely being stored in the faster memory cache. This would sort of explain the faster run time, slightly random variation in results, and the similar number of iterations. However this is just a theory and I can not 100 percent prove that this is why we see a speedup in the float version, but it what makes the most sense to me.

5 Conclusion

In conclusion, this lab explored a lot about our main goal of multi threaded programming, synchronization, the issues with no synchronization. In addition to that we also learned a bit about determine the time complexity of a conditional loop and potentially memory reduction speedup in terms of better caching. Synchronized threads ran slower than asynchronous threads but guaranteed better accuracy. On the other hand the asynchronous threads ran faster but had issues with properly stopping and was less accurate as a result of that. However the amount that it was less accurate wasn't that much different than the synchronized version. Combining everything that we learned, if speed, low memory, and little accuracy is desired use an asynchronous float implementation. If precision is valued above all else such as memory and speed use the synchronized double version. Lastly, another implementation that I wanted to try but cut out due to not having enough time to research and implement was to attempt to use the power of parallel computing with a graphics compute unit to parallel compute thousands of cells at a time. Graphics cards have thousands of individual simple compute units that could potentially tackle this heat plate problem efficiently. It would have created another synchronization challenge, how do you synchronize offloading work between the CPU and GPU properly.

6 Code Used

Note that the float version is not listed but is the same as the HeatPlate class but with all instances of doubles replaced with float

```
public class Main
{

    public static void main(String[] args)
    {
//        HeatPlateFloatVersion plate = new HeatPlateFloatVersion(250, 250,
    10, 80, 30, 40, 16);
        HeatPlate plate = new HeatPlate(250, 250, 10, 80, 30, 40, 4);

        plate.setBarrier(true);

        long start = System.currentTimeMillis();
        plate.beginSimulation();
        long stop = System.currentTimeMillis();

        double runtime = stop - start;

        System.out.println("Total Run Time: " + runtime / 1000 + "s");
//        System.out.println(plate);

    }
}


/**
 * @Author Tanner Smith
 * @Date 11/13/2022
 */

import java.util.LinkedList;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;


public class HeatPlate implements Runnable
{

    private double[][] matrix; //Shared Heat Plate
    private int width, height; //Width and Height
    private double totalError; //Total Error Sum

    private Thread[] threads; //Tread array
    private int[] iterations; //Iteration count array for threads
    private double[] errors; //Individual error sum per thread

    private LinkedList<int[]> ranges; //List of [x1, x2, y1, y2] ranges for
        threads

    private boolean usingBarrier; //Barrier toggle
    private boolean done; //Done flag
```

```
private ThreadGroup group;//Thread group
private CyclicBarrier barrier; //Barrier

/**
 *
 * @param w -width
 * @param h -height
 * @param leftHeat - static heat of the left side
 * @param topHeat - static heat of the top side
 * @param rightHeat - static heat of the right side
 * @param bottomHeat - static heat of the bottom side
 * @param num_threads - number of threads to use
 */
public HeatPlate(int w, int h, double leftHeat, double topHeat, double
    rightHeat, double bottomHeat, int num_threads)
{
    //Link globals
    matrix = new double[h][w];
    width = w;
    height = h;

    //Array creation
    iterations = new int[num_threads];
    threads = new Thread[num_threads];
    errors = new double[num_threads];

    group = new ThreadGroup("Heat Plate Threads");
    barrier = new CyclicBarrier(num_threads);
    done = false;
    usingBarrier = false; //False by default

    ranges = getPartitions(num_threads);

    //Fill the array with the starting values
    fillHorizontal(0, 1, width - 1, topHeat);
    fillHorizontal(height - 1, 1, width - 1, bottomHeat);
    fillVertical(0, 1, height - 1, leftHeat);
    fillVertical(width - 1, 1, height - 1, rightHeat);

    //Corner values can be pre-calculated
    preCalculateCorners();


}

/**
 * Call this method on the object to begin the simulation
 */
public void beginSimulation()
{
    //Initialize the threads and start them
    for (int i = 0; i < threads.length; i++)
    {
        Thread t = new Thread(group, this);
        t.setName(i + "");
```

```
            threads[i] = t;
            t.start();
        }

        //Wait on all threads to finish
        for (Thread t : threads)
        {
            try
            {
                t.join();
            } catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }

        //Calculate final error
        for (double err : errors)
            totalError += err;

        //Print stats
        printStatistics();

    }

    /**
     * Calculates corner values
     */
    private void preCalculateCorners()
    {
        //Top Left
        matrix[0][0] = (matrix[1][0] + matrix[0][1]) / 2;
        //Top Right
        matrix[0][width - 1] = (matrix[1][width - 1] + matrix[0][width -
            2]) / 2;
        //Bottom Left
        matrix[height - 1][0] = (matrix[height - 2][0] + matrix[height -
            1][1]) / 2;
        //Bottom Right
        matrix[height - 1][width - 1] = (matrix[height - 1][width - 2] +
            matrix[height - 2][width - 1]) / 2;
    }

    /**
     *
     * @param row - row to fill
     * @param start - start of the row
     * @param end - end of the row
     * @param value - value to be filled in
     */
    private void fillHorizontal(int row, int start, int end, double value)
    {
        for (int i = start; i < end; i++)
        {
            //System.out.println(Arrays.toString(matrix[i]));
```

```
            matrix[row][i] = value;
        }
    }
    /**
     *
     * @param col - column to fill
     * @param start - start of the column
     * @param end - end of the column
     * @param value - value to be filled in
     */
    private void fillVertical(int col, int start, int end, double value)
    {
        for (int i = start; i < end; i++)
        {
            matrix[i][col] = value;
        }
    }

    /**
     * Calculates the new value at a given index
     * @param row - Row index
     * @param col - Column index
     * @return new value
     */
    private double calculateValueAt(int row, int col)
    {
        //(Left + Top + Right + Bottom)/4
        return (matrix[row][col - 1] + matrix[row - 1][col] + matrix[row][
            col + 1] + matrix[row + 1][col]) / 4;
    }

    /**
     * Assumes x1 < x2 and y1 < y2
     * Inclusive
     * @param x1
     * @param y1
     * @param x2
     * @param y2
     * @return the error of the new values applied compared to their
         previous values
     */
    private double calculateRange(int x1, int y1, int x2, int y2)
    {
        double localErrorSum = 0;
        for (int c = x1; c <= x2; c++)
        {
            for (int r = y1; r <= y2; r++)
            {
                double newValue = calculateValueAt(r, c);
                double error = Math.abs(matrix[r][c] - newValue);
                localErrorSum += error;
                matrix[r][c] = newValue;
            }
        }
        return localErrorSum;
```

```
//          totalError = localErrorSum;
            //System.out.println(totalError);
        }

        /**
         * Figures out how to partition the grid among the desired thread count
         * I hardly understand how this works anymore, but it just works :)
         * each array is [x1, x2, y1, y2]
         * @param p - # of partitions needed
         * @return a linked list of arrays of size 4 containing points to
             partition for p number of threads
         */
        private LinkedList<int[]> getPartitions(int p)
        {
            //Return list
            LinkedList<int[]> partitions = new LinkedList<>();

            //Since the side values don't get calculated we need the grid one
                value inside
            int innerWidth = width - 2;
            int innerHeight = height - 2;

            //Figure out how to split the grid
            int rows = 0;
            int cols = 1;
            int powerOfTwo = 1;
            while (p % powerOfTwo == 0 && cols < Math.sqrt(p))
            {
                cols = powerOfTwo;
                powerOfTwo *= 2;
            }
            rows = p / cols;

            //Start creating partitions
            int widthOffset = (innerWidth / cols) - 1;
            int widthRemainder = innerWidth % cols;
            int x1 = 1;
            int x2 = 0;

            for (int r = 0; r < cols; r++)
            {
                x2 = x1 + widthOffset;
                if (widthRemainder > 0)
                {
                    x2++;
                    widthRemainder--;
                }

                int heightOffset = (innerHeight / rows) - 1;
                int heightRemainder = innerHeight % rows;
                int y1 = 1;
                int y2 = 0;
                for (int c = 0; c < rows; c++)
                {
                    y2 = y1 + heightOffset;
```

```
            if (heightRemainder > 0)
            {
                y2++;
                heightRemainder--;
            }

            int[] box = new int[4];
            box[0] = x1;
            box[1] = x2;
            box[2] = y1;
            box[3] = y2;
            partitions.add(box);
            y1 = y2 + 1;
        }
        x1 = x2 + 1;
    }
    return partitions;
}


/**
 * Print the statistics of the simulation
 */
public void printStatistics()
{
    StringBuilder sb = new StringBuilder();
    sb.append("Thread count: ");
    sb.append(threads.length);
    sb.append(",\t");
    sb.append("Final Error: ");
    sb.append(String.format("%.2f", totalError));
    sb.append(",\t");
    sb.append("Global Average Value: ");
    sb.append(String.format("%.2f", totalGridAverage()));
    sb.append("\n");

    for (int i = 0; i < iterations.length; i++)
    {
        sb.append("Thread ");
        sb.append(i);
        sb.append(" ran ");
        sb.append(iterations[i]);
        sb.append(" times");
        sb.append("\n");
    }

    System.out.println(sb.toString());
}

/**
 * Set wheather the barrier is enabled or not
 * @param b
 */
public void setBarrier(boolean b)
{
```

```
        usingBarrier = b;
    }

    /**
     *
     * @return sum of grid values/total values
     */
    public double totalGridAverage()
    {
        double sum = 0;
        for (double[] row : matrix)
        {
            for (double col : row)
            {
                sum += col;
            }
        }
        return sum / (width * height);
    }

    /**
     * Prints the matrix and its values
     * @return string representation of the array
     */
    public String toString()
    {
        StringBuilder sb = new StringBuilder();
        for (double[] row : matrix)
        {
            for (double col : row)
            {
                String val = String.format("%.6f", col);
                sb.append(val);
                sb.append("\t");
            }
            sb.append("\n\n");
        }
        return sb.toString();
    }


    /**
     * Individual Threads Run
     */
    @Override
    public void run()
    {
        int threadNumber = Integer.parseInt(Thread.currentThread().getName
            ());
        int[] range = ranges.get(threadNumber);
        double localTotalError = 0;

        do
        {
```

```
            double localError = calculateRange(range[0], range[2], range
                [1], range[3]);
            errors[threadNumber] = localError;
            iterations[threadNumber]++;
            localTotalError = 0;
            if (usingBarrier)
            {
                try
                {
                    barrier.await();
                } catch (InterruptedException e)
                {
                    //System.out.println("Interrupted");
                } catch (BrokenBarrierException e)
                {
                    //System.out.println("Broken Barrier");
                }
            }
            for (double err : errors)
            {
                localTotalError += err;
            }
        } while (localTotalError > 5 && !done);
        done = true;
        Thread.currentThread().getThreadGroup().interrupt();
//        System.out.println("Thread " + threadNumber + " finished");
    }
}
```