Varsha Singh
ENG EC527 Lab 4
Martin Herbordt

Lab 4: ENG EC527

Varsha Singh

Task 1:

I was able to use the generic pointer to print out the elements by adding a 0, 1, or 2 to ArrayA. This is possible because ArrayA is a pointer to the start of the array, which can also be accessed through indexing.

Task 2:

1st run:
```
Hello test_create.c
Hello World! from child thread 7f913b324700
Hello World! from child thread 7f913ab23700
Hello World! from child thread 7f913a322700
main() after creating the thread.  My id is 7f913bb31740
```

2nd run:
```
Hello test_create.c
Hello World! from child thread 7f21bce04700
Hello World! from child thread 7f21bc603700
Hello World! from child thread 7f21bb601700
Hello World! from child thread 7f21bbe02700
main() after creating the thread.  My id is 7f21bd611740
Hello World! from child thread 7f21bae00700
Hello World! from child thread 7f21bae00700
```

3rd run:
```
Hello World! from child thread 7f0f3ab7e700
main() after creating the thread.  My id is 7f0f3b38b740
Hello World! from child thread 7f0f39b7c700
Hello World! from child thread 7f0f38b7a700
Hello World! from child thread 7f0f3937b700
Hello World! from child thread 7f0f3a37d700
```

4th run:
```
Hello test_create.c
Hello World! from child thread 7fef689bf700
Hello World! from child thread 7fef681be700
Hello World! from child thread 7fef679bd700
Hello World! from child thread 7fef671bc700
main() after creating the thread.  My id is 7fef691cc740
```

5th run:

```
Hello test_create.c
Hello World! from child thread 7ff2141fd700
Hello World! from child thread 7ff2139fc700
Hello World! from child thread 7ff2129fa700
Hello World! from child thread 7ff2131fb700
main() after creating the thread.  My id is 7ff214a0a740
```

The output does change, it seems that usually the ‘main ()’ line is printed as the last statement, but there are a few times where it prints before the child threads are able to print.

Task 3:

I declared id as a pthread_t pointer, and casted the result of malloc as a pthread_t pointer, with the argument being NUM_THREADS * sizeof(pthread_t).

Task 4:

When I added the sleep(3) line before the print statement, I only received the output:

```
Hello test_create.c
main() after creating the thread.  My id is 7fb140e31740
```

Which leaves me to believe that this puts the threads to sleep for long enough for the main thread to finish and exit the main function before the other children threads wake up. When the main thread exits, the entire program is killed.

Task 5:

When I removed sleep(3) from the work function and put it right before the return from main, I received this as output:

```
Hello test_create.c
Hello World! from child thread 7f01c54fa700
Hello World! from child thread 7f01c44f8700
Hello World! from child thread 7f01c3cf7700
Hello World! from child thread 7f01c4cf9700
main() after creating the thread.  My id is 7f01c5d07740
Hello World! from child thread 7f01c34f6700
```

This allows all of the children threads to run until exiting the work function, because the main thread is put to sleep for much longer than the time it takes for the children threads to run. The children threads terminate before the main thread kills the program.

Task 6:

The only thing that changes between runs after adding sleep(3) is the thread IDs. Apart from that, the structure is as follows:

```
Hello test_join.c
main() -- After creating the thread.  My id is: 7efe1f531740
 Hello World!  It's me, thread #7efe1e523700 --
 Hello World!  It's me, thread #7efe1ed24700 --
 Hello World!  It's me, thread #7efe1cd20700 --
 Hello World!  It's me, thread #7efe1dd22700 --
 Hello World!  It's me, thread #7efe1d521700 --
After joining, Good Bye World!
```

This is different from the previous task in which we added sleep(3) to the child function because pthread_join requires blocking the currently executing thread until the thread passed as an argument has terminated. Putting all children threads to sleep for 3 seconds will block the main thread until they are all finished, which happens seemingly concurrently.

Task 7:

I added a join statement in the for loop as well as printing the value of threadid before it cast back to an int, and found that the printed value of threadid and and tid are the same.

When I set the value to -12, Both before and after it is interpreted to be a -12. I looked in a little bit on this and found that it might have to do with casting a pointer from integer of a different size, and the system will automatically sign extend to account for the negative values. This means that it will do the necessary extensions to keep the value interpreted as the same. This might be different if it was changed to an unsigned value in the second print.

Task 8:

I confirmed that the threads affect each others values by running the code multiple times and saw variable output, with *g being 10 each time and f being other values, with multiple occurrences of some values like 10 or 3.
I did f -= 1 and *g -=65 and now see often that f = -16 and *g goes from 5 to -40 in increments of 5. The order of g is consistent, but f is variable.

I believe that this has to do with the fact that the values are being passed by reference. Because it is passing the memory address, any changes to local variables in the function will persist. This means that when we set t to be a generic pointer and then look at it as a value, the value of t interates in each call, even though the same memory address is passed. This means that all threads are sharing the same value, and the value of t might have changed by the time a thread has read it. This shows that with shared memory, we might experience race conditions, which lead to uncertain output. If we used pthread_join in the main loop to block until execution, we would get all values in order.

Task 9:

I was able to reduce the number of threads with modifying main by creating a int pointer that has the same memory address as the value passed to the function, and I incremented it by NUM_THREADS. This consistently got the number of threads that executed to be less than 5.

Task 10:

I was able to access a unique element by adding a global variable that was incremented with each work call. I used this as an iterator in the array and modified the values by adding 5. I also used pthread_join after creating each thread so that it would modify as expected.

Task 11:

When I ran the compiled code, I got the following output:

```
Hello test_param3.c
In main:  creating thread 0
In main:  creating thread 1
in PrintHello(), thread #0 ; sum = 27, message = First Message
In main:  creating thread 2
in PrintHello(), thread #1 ; sum = 28, message = Second Message
In main:  creating thread 3
In main:  creating thread 4
in PrintHello(), thread #2 ; sum = 29, message = Third Message
in PrintHello(), thread #3 ; sum = 30, message = Fourth Message
in PrintHello(), thread #4 ; sum = 31, message = Fifth Message
```

This verifies that each of the different threads are working with unique data.

I added a sixth thread, and got the following output:

```
Hello test_param3.c
In main:  creating thread 0
In main:  creating thread 1
In main:  creating thread 2
in PrintHello(), thread #0 ; sum = 27, message = First Message
In main:  creating thread 3
in PrintHello(), thread #2 ; sum = 29, message = Third Message
in PrintHello(), thread #1 ; sum = 28, message = Second Message
In main:  creating thread 4
in PrintHello(), thread #3 ; sum = 30, message = Fourth Message
In main:  creating thread 5
in PrintHello(), thread #5 ; sum = 1000, message = Sixth Message
in PrintHello(), thread #4 ; sum = 31, message = Fifth Message
```

Task 12:

When I ran the code without any changes, I received the following output:

```
Hello sync1
In main: created thread 1, which is blocked
   Press any letter key (not space) then press enter:
w
in thread ID 0 (sum = 123 message = First Message), now unblocked!
After joining
GOODBYE WORLD!
```

After commenting out the trylock check:

```
Hello sync1
In main: created thread 1, which is blocked
   Press any letter key (not space) then press enter:
in thread ID 0 (sum = 123 message = First Message), now unblocked!
q
After joining
GOODBYE WORLD!
```

The difference is that the printed statement in PrintHello will print before user input because it is no longer trying to lock mutexA.

Task 13:

If I uncomment the trylock statement, the program no longer works as expected. It will print Hello sync1, and then no longer respond.

If I keep the trylock statement commented out, then the output is:

```
Hello sync1
in thread ID 0 (sum = 123 message = First Message), now unblocked!
```

But will no longer respond.

These happen because a mutex has a count value of 1, and enforces mutual exclusion, meaning that only one thread holds the lock at a time.

Task 14:

I compiled test_barrier and ran the program a couple of times:

1st run:

```
Hello test_barrier.c
In main:  creating thread 0
In main:  creating thread 1
Thread 0 printing before barrier 1 of 3
Thread 0 printing before barrier 2 of 3
Thread 0 printing before barrier 3 of 3
In main:  creating thread 2
Thread 1 printing before barrier 1 of 3
Thread 1 printing before barrier 2 of 3
Thread 1 printing before barrier 3 of 3
In main:  creating thread 3
Thread 2 printing before barrier 1 of 3
Thread 2 printing before barrier 2 of 3
Thread 2 printing before barrier 3 of 3
In main:  creating thread 4
After creating the threads; my id is 7ff78e9a7740
Thread 3 printing before barrier 1 of 3
Thread 3 printing before barrier 2 of 3
Thread 3 printing before barrier 3 of 3
Thread 4 printing before barrier 1 of 3
Thread 4 printing before barrier 2 of 3
Thread 4 printing before barrier 3 of 3
Thread 4 after barrier (print 1 of 2)
Thread 4 after barrier (print 2 of 2)
Thread 2 after barrier (print 1 of 2)
Thread 2 after barrier (print 2 of 2)
Thread 1 after barrier (print 1 of 2)
Thread 1 after barrier (print 2 of 2)
Thread 0 after barrier (print 1 of 2)
Thread 0 after barrier (print 2 of 2)
Thread 3 after barrier (print 1 of 2)
Thread 3 after barrier (print 2 of 2)
After joining
```

2nd run:

```
Hello test_barrier.c
In main:  creating thread 0
In main:  creating thread 1
Thread 0 printing before barrier 1 of 3
Thread 0 printing before barrier 2 of 3
Thread 0 printing before barrier 3 of 3
In main:  creating thread 2
```

```
Thread 1 printing before barrier 1 of 3
Thread 1 printing before barrier 2 of 3
Thread 1 printing before barrier 3 of 3
Thread 2 printing before barrier 1 of 3
Thread 2 printing before barrier 2 of 3
Thread 2 printing before barrier 3 of 3
In main:  creating thread 3
In main:  creating thread 4
Thread 3 printing before barrier 1 of 3
Thread 3 printing before barrier 2 of 3
Thread 3 printing before barrier 3 of 3
After creating the threads; my id is 7f3fbcb2f740
Thread 4 printing before barrier 1 of 3
Thread 4 printing before barrier 2 of 3
Thread 4 printing before barrier 3 of 3
Thread 4 after barrier (print 1 of 2)
Thread 4 after barrier (print 2 of 2)
Thread 1 after barrier (print 1 of 2)
Thread 1 after barrier (print 2 of 2)
Thread 2 after barrier (print 1 of 2)
Thread 2 after barrier (print 2 of 2)
Thread 3 after barrier (print 1 of 2)
Thread 3 after barrier (print 2 of 2)
Thread 0 after barrier (print 1 of 2)
Thread 0 after barrier (print 2 of 2)
After joining
```

3rd run:

```
Hello test_barrier.c
In main:  creating thread 0
In main:  creating thread 1
Thread 0 printing before barrier 1 of 3
Thread 0 printing before barrier 2 of 3
Thread 0 printing before barrier 3 of 3
In main:  creating thread 2
Thread 1 printing before barrier 1 of 3
Thread 1 printing before barrier 2 of 3
Thread 1 printing before barrier 3 of 3
Thread 2 printing before barrier 1 of 3
Thread 2 printing before barrier 2 of 3
Thread 2 printing before barrier 3 of 3
In main:  creating thread 3
In main:  creating thread 4
After creating the threads; my id is 7fb40cb38740
Thread 3 printing before barrier 1 of 3
Thread 3 printing before barrier 2 of 3
Thread 3 printing before barrier 3 of 3
Thread 4 printing before barrier 1 of 3
Thread 4 printing before barrier 2 of 3
Thread 4 printing before barrier 3 of 3
Thread 4 after barrier (print 1 of 2)
Thread 4 after barrier (print 2 of 2)
Thread 3 after barrier (print 1 of 2)
Thread 3 after barrier (print 2 of 2)
Thread 0 after barrier (print 1 of 2)
Thread 0 after barrier (print 2 of 2)
Thread 2 after barrier (print 1 of 2)
Thread 2 after barrier (print 2 of 2)
Thread 1 after barrier (print 1 of 2)
Thread 1 after barrier (print 2 of 2)
After joining
```

With these, all before barrier messages are printed before the after barrier messages.

After uncommenting the sleep(1) syscall, I get the following output:

```
In main:  creating thread 0
In main:  creating thread 1
In main:  creating thread 2
In main:  creating thread 3
In main:  creating thread 4
After creating the threads; my id is 7f09d27e9740
Thread 4 printing before barrier 1 of 3
Thread 0 printing before barrier 1 of 3
Thread 2 printing before barrier 1 of 3
Thread 3 printing before barrier 1 of 3
Thread 1 printing before barrier 1 of 3
Thread 4 printing before barrier 2 of 3
Thread 0 printing before barrier 2 of 3
Thread 2 printing before barrier 2 of 3
Thread 3 printing before barrier 2 of 3
Thread 1 printing before barrier 2 of 3
Thread 3 printing before barrier 3 of 3
Thread 0 printing before barrier 3 of 3
Thread 2 printing before barrier 3 of 3
Thread 4 printing before barrier 3 of 3
Thread 1 printing before barrier 3 of 3
Thread 1 after barrier (print 1 of 2)
Thread 1 after barrier (print 2 of 2)
Thread 0 after barrier (print 1 of 2)
Thread 0 after barrier (print 2 of 2)
Thread 3 after barrier (print 1 of 2)
Thread 3 after barrier (print 2 of 2)
Thread 4 after barrier (print 1 of 2)
Thread 4 after barrier (print 2 of 2)
Thread 2 after barrier (print 1 of 2)
Thread 2 after barrier (print 2 of 2)
After joining
```

The main thread is able to print the created threads message before the children threads print in their function, but the order of before barrier messages printing before all the after barrier messages is still respected.

After changing the sleep time to be a variable function of taskid, the barrier still works. The output is the following:

```
Hello test_barrier.c
In main:  creating thread 0
In main:  creating thread 1
In main:  creating thread 2
In main:  creating thread 3
In main:  creating thread 4
Thread 0 printing before barrier 1 of 3
After creating the threads; my id is 7f1ec5c69740
Thread 0 printing before barrier 2 of 3
Thread 0 printing before barrier 3 of 3
Thread 1 printing before barrier 1 of 3
Thread 1 printing before barrier 2 of 3
Thread 2 printing before barrier 1 of 3
Thread 3 printing before barrier 1 of 3
Thread 1 printing before barrier 3 of 3
Thread 4 printing before barrier 1 of 3
Thread 2 printing before barrier 2 of 3
Thread 2 printing before barrier 3 of 3
Thread 3 printing before barrier 2 of 3
Thread 4 printing before barrier 2 of 3
Thread 3 printing before barrier 3 of 3
Thread 4 printing before barrier 3 of 3
Thread 4 after barrier (print 1 of 2)
Thread 4 after barrier (print 2 of 2)
Thread 2 after barrier (print 1 of 2)
Thread 2 after barrier (print 2 of 2)
Thread 0 after barrier (print 1 of 2)
Thread 0 after barrier (print 2 of 2)
Thread 1 after barrier (print 1 of 2)
Thread 1 after barrier (print 2 of 2)
Thread 3 after barrier (print 1 of 2)
Thread 3 after barrier (print 2 of 2)
After joining
```

This is because the barrier waits for NUM_THREADS unique threads to reach the barrier before the program is allowed to continue. Thus, the barrier still works in all of these scenarios.

Task 15:

I ran test_sync.c and got the following results:

1st run:

```
Hello test_sync2.c
thread #2 waiting for 1 ...
thread #3 waiting for 1 ...
thread #4 waiting for 2 ...
Main: calling sleep(1)...
thread #5 waiting for 4 ...
thread #6 waiting for 3 and 4 ...
thread #7 waiting for 5 and 6 ...
Main: created threads 2-7, type a letter (not space) and <enter>
q
Main: waiting for thread 7 to finish, UNLOCK LOCK 1
Main: Done unlocking 1
It's me, thread #2! I'm unlocking 2...
It's me, thread #4! I'm unlocking 4...
It's me, thread #3! I'm unlocking 3...
It's me, thread #5! I'm unlocking 5...
```

```
It's me, thread #6! I'm unlocking 6...
It's me, thread #7! I'm unlocking 7...
Main: After joining
```

2nd run:

```
Hello test_sync2.c
thread #2 waiting for 1 ...
thread #3 waiting for 1 ...
thread #4 waiting for 2 ...
thread #5 waiting for 4 ...
Main: calling sleep(1)...
thread #6 waiting for 3 and 4 ...
thread #7 waiting for 5 and 6 ...
Main: created threads 2-7, type a letter (not space) and <enter>
q
Main: waiting for thread 7 to finish, UNLOCK LOCK 1
Main: Done unlocking 1
It's me, thread #2! I'm unlocking 2...
It's me, thread #3! I'm unlocking 3...
It's me, thread #4! I'm unlocking 4...
It's me, thread #5! I'm unlocking 5...
It's me, thread #6! I'm unlocking 6...
It's me, thread #7! I'm unlocking 7...
Main: After joining
```

These tests follow the pattern that is expected. Even though the join loop is executed after most threads have exited, it will still work because pthread_join collects the exit status of the thread it waits for before allowing the thread that called the function to continue. Some of the threads exited already, but this just means that it collects the status and continues, not needing any further blocking for that thread.

Task 16:

To add an eighth thread, I change NUM_THREADS to 8, added a case statement for 8, locking and unlocking 6 and 7, added an additional message, and had main lock 8. The output is the following:

```
Hello test_sync2.c
thread #2 waiting for 1 ...
thread #3 waiting for 1 ...
thread #4 waiting for 2 ...
thread #5 waiting for 4 ...
thread #6 waiting for 3 and 4 ...
thread #7 waiting for 5 and 6 ...
Main: calling sleep(1)...
thread #8 waiting for 6 and 7...
Main: created threads 2-8, type a letter (not space) and <enter>
1
Main: waiting for thread 8 to finish, UNLOCK LOCK 1
Main: Done unlocking 1
It's me, thread #2! I'm unlocking 2...
It's me, thread #3! I'm unlocking 3...
It's me, thread #4! I'm unlocking 4...
It's me, thread #5! I'm unlocking 5...
It's me, thread #6! I'm unlocking 6...
It's me, thread #7! I'm unlocking 7...
It's me, thread #8! I'm unlocking 8...
Main: After joining
```

Task 17:

I ran test_crit.c a few times and found that the results are inconsistent. I made the number of threads 10000 instead of 10, and got the following results:

1st run:

```
Hello test_crit.c
 MAIN --> final balance = 1002
        qr_total = 17452.753699
```

2nd run:

```
Hello test_crit.c
 MAIN --> final balance = 999
        qr_total = 17448.141102
```

3rd run:

```
Hello test_crit.c
 MAIN --> final balance = 997
        qr_total = 17449.600383
```

4th run:

```
Hello test_crit.c
 MAIN --> final balance = 1000
        qr_total = 17461.100124
```

5th run:

```
Hello test_crit.c
 MAIN --> final balance = 1015
         qr_total = 17446.948363
```

Neither the final balance or qr_total are the same each time.

Task 18:

I declared a global mutex and put it around what I considered to be the critical region. I determined the critical region to encompass reading from the balance and writing to the balance. This is because any thread that reads a balance may be 'outdated' because of a context switch, and then also a thread that writes to the balance might be outdated and overwrite some other threads.

The following outputs all yielded the same qr_total and final balance:

1st run:
```
Hello test_crit.c
 MAIN --> final balance = 1000
         qr_total = 17464.076758
```

2nd run:
```
Hello test_crit.c
 MAIN --> final balance = 1000
         qr_total = 17464.076758
```

3rd run:
```
Hello test_crit.c
 MAIN --> final balance = 1000
         qr_total = 17464.076758
```

4th run:
```
Hello test_crit.c
 MAIN --> final balance = 1000
         qr_total = 17464.076758
```

5th run:
```
Hello test_crit.c
 MAIN --> final balance = 1000
         qr_total = 17464.076758
```
6th run:
```
Hello test_crit.c
```

```
 MAIN --> final balance = 1000
        qr_total = 17464.076758
```

Task 19:

The original 1000 iterations was not close to the solution.

1000 iterations output:

```
test_sor setup...
In main:  creating thread 0
In main:  creating thread 1
In main:  creating thread 2
In main:  creating thread 3
In main:  creating thread 4
Thread for index 3 starting
In main:  creating thread 5
Thread for index 4 starting
Thread for index 1 starting
Thread for index 5 starting
Thread for index 2 starting
In main:  creating thread 6
Thread for index 6 starting
In main:  creating thread 7
Thread for index 7 starting
In main:  creating thread 8
Thread for index 8 starting
In main:  creating thread 9
Thread for index 9 starting
After creating the threads; my id is 7fcd47d08740
Thread for index 10 starting
After joining
  0.00 12.50 18.75 25.00 25.00 25.00 25.00 25.00 25.00 25.00 62.50 100.00

real	0m0.003s
```

10000 iterations seems to have stabilized further, but isn't close to the solution:

10000 iterations output:

```
test_sor setup...
In main:  creating thread 0
In main:  creating thread 1
In main:  creating thread 2
Thread for index 2 starting
Thread for index 1 starting
In main:  creating thread 3
Thread for index 3 starting
In main:  creating thread 4
Thread for index 4 starting
In main:  creating thread 5
Thread for index 5 starting
In main:  creating thread 6
In main:  creating thread 7
Thread for index 6 starting
Thread for index 7 starting
In main:  creating thread 8
In main:  creating thread 9
After creating the threads; my id is 7f05b6db7740
Thread for index 9 starting
Thread for index 10 starting
Thread for index 8 starting
After joining
  0.00  3.13  6.25  9.38 12.50 15.63 18.75 21.88 41.41 60.94 80.47 100.00

real	0m0.005s

15000 iterations converged more:
test_sor setup...
In main:  creating thread 0
In main:  creating thread 1
In main:  creating thread 2
Thread for index 2 starting
Thread for index 1 starting
In main:  creating thread 3
Thread for index 3 starting
In main:  creating thread 4
In main:  creating thread 5
Thread for index 4 starting
Thread for index 5 starting
In main:  creating thread 6
In main:  creating thread 7
In main:  creating thread 8
In main:  creating thread 9
After creating the threads; my id is 7fc90b105740
Thread for index 6 starting
Thread for index 10 starting
Thread for index 8 starting
Thread for index 7 starting
Thread for index 9 starting
After joining
  0.00  3.57  7.14 10.71 14.29 17.86 28.53 42.83 57.12 71.42 85.71 100.00

real      0m0.004s
```

50000 converged more:

50000 iterations output:

```
test_sor setup...
In main:  creating thread 0
In main:  creating thread 1
Thread for index 1 starting
In main:  creating thread 2
Thread for index 2 starting
In main:  creating thread 3
In main:  creating thread 4
Thread for index 3 starting
In main:  creating thread 5
Thread for index 4 starting
Thread for index 5 starting
Thread for index 6 starting
In main:  creating thread 6
In main:  creating thread 7
Thread for index 7 starting
In main:  creating thread 8
In main:  creating thread 9
Thread for index 8 starting
Thread for index 9 starting
After creating the threads; my id is 7fbd8c619740
Thread for index 10 starting
After joining
  0.00  3.57  7.14 10.71 14.29 17.86 21.43 37.14 52.86 68.57 84.29 100.00
```

```
real	0m0.009s
```

100000 iterations seems to somewhat converge more:
10000 iterations output:

```
test_sor setup...
In main:  creating thread 0
In main:  creating thread 1
In main:  creating thread 2
Thread for index 1 starting
Thread for index 2 starting
In main:  creating thread 3
Thread for index 3 starting
In main:  creating thread 4
Thread for index 4 starting
In main:  creating thread 5
In main:  creating thread 6
Thread for index 6 starting
In main:  creating thread 7
In main:  creating thread 8
In main:  creating thread 9
After creating the threads; my id is 7f0c70f80740
Thread for index 8 starting
Thread for index 9 starting
Thread for index 10 starting
Thread for index 7 starting
Thread for index 5 starting
After joining
  0.00  5.00 10.00 19.95 29.90 39.85 49.80 59.74 69.69 79.64 89.82 100.00

real	0m0.014s
```

I tried it a few times, but one of the outputs with 2000000 iterations converged.
2000000 iterations output:

```
test_sor setup...
In main:  creating thread 0
In main:  creating thread 1
In main:  creating thread 2
In main:  creating thread 3
Thread for index 2 starting
Thread for index 1 starting
Thread for index 3 starting
In main:  creating thread 4
In main:  creating thread 5
In main:  creating thread 6
Thread for index 6 starting
Thread for index 5 starting
In main:  creating thread 7
In main:  creating thread 8
In main:  creating thread 9
After creating the threads; my id is 7f0036723740
Thread for index 4 starting
Thread for index 7 starting
Thread for index 9 starting
Thread for index 10 starting
Thread for index 8 starting
After joining
  0.00  9.09 18.18 27.27 36.36 45.45 54.54 63.63 72.73 81.82 90.91 100.00

real	0m0.260s
```

Task 20:

When I set USE_BARRIERS to 1, I got the following outputs:

1000 iterations was much better compared to when USE_BARRIERS was set to 0.
1000 iterations output:

```
test_sor setup...
In main:  creating thread 0
In main:  creating thread 1
In main:  creating thread 2
In main:  creating thread 3
Thread for index 1 starting
Thread for index 2 starting
In main:  creating thread 4
Thread for index 3 starting
Thread for index 4 starting
Thread for index 5 starting
In main:  creating thread 5
In main:  creating thread 6
Thread for index 6 starting
In main:  creating thread 7
```

```
Thread for index 7 starting
In main:  creating thread 8
Thread for index 8 starting
In main:  creating thread 9
After creating the threads; my id is 7fdad0951740
Thread for index 9 starting
Thread for index 10 starting
After joining
  0.00  5.82 11.74 17.91 24.55 31.90 40.27 49.87 60.81 73.03 86.24 100.00

real	0m0.010s
```

5000 iterations was not quite stabilized, but it was very close, as compared to attempts with USE_BARRIER set for more iterations than this.
5000 iterations output:

```
test_sor setup...
In main:  creating thread 0
In main:  creating thread 1
In main:  creating thread 2
Thread for index 1 starting
Thread for index 2 starting
Thread for index 3 starting
In main:  creating thread 3
In main:  creating thread 4
In main:  creating thread 5
Thread for index 4 starting
Thread for index 5 starting
In main:  creating thread 6
Thread for index 6 starting
Thread for index 7 starting
In main:  creating thread 7
In main:  creating thread 8
Thread for index 8 starting
In main:  creating thread 9
Thread for index 9 starting
After creating the threads; my id is 7f58b558d740
Thread for index 10 starting
After joining
  0.00  8.94 17.88 26.86 35.86 44.91 54.00 63.14 72.31 81.52 90.75 100.00
```

```
real	0m0.050s
```

10000 iterations was much closer and is consistent as compared to before.
10000 iterations output:

```
test_sor setup...
In main:  creating thread 0
In main:  creating thread 1
In main:  creating thread 2
Thread for index 1 starting
In main:  creating thread 3
Thread for index 3 starting
Thread for index 2 starting
In main:  creating thread 4
Thread for index 4 starting
In main:  creating thread 5
Thread for index 5 starting
In main:  creating thread 6
Thread for index 6 starting
In main:  creating thread 7
Thread for index 7 starting
In main:  creating thread 8
Thread for index 8 starting
In main:  creating thread 9
Thread for index 9 starting
After creating the threads; my id is 7fa722a44740
Thread for index 10 starting
After joining
  0.00  9.09 18.18 27.27 36.35 45.45 54.54 63.63 72.72 81.81 90.91 100.00

real	0m0.081s
```

I converged by 15000 iterations.
15000 iterations output:

```
test_sor setup...
In main:  creating thread 0
In main:  creating thread 1
Thread for index 1 starting
In main:  creating thread 2
In main:  creating thread 3
Thread for index 2 starting
Thread for index 4 starting
Thread for index 3 starting
In main:  creating thread 4
In main:  creating thread 5
Thread for index 5 starting
In main:  creating thread 6
Thread for index 6 starting
In main:  creating thread 7
In main:  creating thread 8
Thread for index 7 starting
Thread for index 8 starting
In main:  creating thread 9
After creating the threads; my id is 7f77b73b7740
Thread for index 9 starting
Thread for index 10 starting
After joining
  0.00  9.09 18.18 27.27 36.36 45.45 54.55 63.64 72.73 81.82 90.91 100.00

real	0m0.154s
```

- Setting USE_BARRIERS to 1 was overall a faster method for converging. The real time of the program that converged was around 0m0.154s, whereas the USE_BARRIERS set to 0 converged with a real time of around 0m0.260s. For system time, setting USE_BARRIERS to 1 resulted in a faster convergence. This happens because there are less iterations to go through for the loop, so this will result in a faster runtime. However, it is about half of the runtime even though there are 133x more iterations because threads will block with the barrier and make sure that all are at the barrier until they can continue. This might result in more premature context switches and waiting until all threads reach the barrier might result in a longer runtime.
- It will take more iterations for the USE_BARRIERS = 0 solution to converge because it is unsafely editing shared data. If the data in the arrays were only accessible from each thread (hv_from; hatvals, hvals_2). Then it would not affect the results. However, because there is an i-1 and i+1 indexing from hv_from, a race condition is introduced. The race condition might result in previous or future time-step values being used for updates, which will incorrectly model the temperature of the pipe at timesteps.
- Setting USE_BARRIERS to 1 will consistently give the same answer. In this case, all threads have to wait at the barrier before they can continue their iterations. This makes it such that all values for the thermal modeling of the pipe will have to update for that timestep before they can continue. Because they pull values from the same time-step and

have the expected modeled temperatures, the process will be standardized. As explained before, the race condition introduced from threads not waiting before continuing the next time-step means that some threads may model their temperature at that time-step with other surrounding contextual values at a different time-step.