Multithreaded Adder analysis by Anirudh Lakra

1) This bug is caused by getSum() being called before run() finishes executing so the wrong value is returned. To solve this issue, we need to add conditional synchronization to run() and getSum(). Firstly, we add wait() to getSum() if ready == false and a notifyAll() to the end of run(). However, this isn't enough and we also need to add synchronized to both run() and getSum() because what happens is that there is no object lock yet and by adding synchronized we make one. What happens is that getSum() obtains the object lock first and sees that ready is false and gives up the lock. Run() obtains the lock afterwards and after finishing it calls notify to getSum() which wakes it up. Finally, getSum() gets the object lock again and returns the correct sum. To safely publish SerialAdder we add volatile to sum and ready.

2) The IEEE format for doubles uses 64 bits where 1 bit is for the sign, 11 bits are for the exponent and 52 bits for the mantissa. The problem that occurs is that the mantissa (which has a range of 2^52 - 1) is unable to represent 100000000000000000003.0 and so it just throws up away 3.0. This leads to a rounding error and results in the wrong result of 0.0. In a multithreaded system, the various threads may complete their calculations in a different order each time which means that it could give a different result depending on what order they were completed. As you can see in this example, changing the order of the calculation led to a different answer which could occur in a multithreaded system.

3) The large time difference of the code executing, and the timer is due to the static initializer. The static initialization occurs before running the main part and we are initialising a very large array (2^24 elements) which ends up taking up a lot of time. Finally, once all these elements are initialized does the timer start and threads run but by this time the code has already been running for a noticeable amount of time. The timer is accurate for the time it takes for the threads to finish but it doesn't include the time used to initialise the static part.

4)

Trial	**Timer**	**Actual Time**
1	3.04	19.5
2	2.84	18.9
3	2.72	17.2
4	2.69	17.5
5	3.17	19.5
Average	2.89	18.5

5) I ran the multithreaded adder version with threads 2, 4, 8, 16 5 times each and timed them. The averages are:

Threads	**Timer**	**Actual Time**
2	2.52	18.15
4	2.90	21.00
8	2.21	19.16
16	2.29	19.51

Both the actual time and timer seem to stay roughly the same as we increase the number of threads. We would expect both timer and actual time to decrease as we increase the number of threads since each thread will have less computations. However, it is possible that the time reduced by having more threads is offset by the time needed to create and delete the new threads. If the time reduced by allowing threads to do less computations and time gained by the creation and deletion of threads are similar, the overall impact of adding new threads is negligible and the timings stay the same.

Another cause may be that my processor has only 2 cores so it can only execute 2 threads at one time. This means adding more than 2 threads has no effect on performance since they are waiting for my cores to finish running their threads. This accounts for the no overall change in running time after 2 threads.