Parallel Threads for Summing a Large Array

# Introduction

Often, threads are used in many cases for achieving “better performance.” When we run code in a serial fashion as we have done for most of our projects in this class, we only ever take advantage of one of the cores of a processor. This means that running a serial program today runs in about the same amount of time that the same program could have ten years ago! So, how can we take advantage of all these cores and achieve better performance? One way is simply to use more threads to accomplish a single task. In this project, we will start with getting the time it takes to sum up all the elements in a large array without any threads. The next portion is then to split up the task so that each thread can compute a smaller portion of the sum. By the associative law of addition, we know that this will still provide the same sum, but hopefully quicker.

The following steps will walk you through how to do just that. The steps are designed to be very incremental. If there are steps that you feel comfortable skipping to get to the final result, feel free to do so. *(Take note that Java is limited in its ability to schedule threads for better performance. We might observe better performance using other, more suitable languages.)*

# Steps

1. Start by running the current program and change the size of the array. (A good number of elements to use is about 200,000,000.)
2. Just under the line that n is declared, uncomment the line int numThreads = 4;
3. The following steps may best help in creating threads that will give the correct result. If there are any steps that you feel comfortable in skipping to get to the final portion, you may do so.
   1. Start by creating a class called SumThread that does NOT implement Runnable (we will do this in a subsequent step).
   2. In this class, we need to store 4 member variables:
      * + - The data array (containing the data to be summed)
          - A starting index for data — This is where the current instance of the SumThread will start summing elements in data
          - An ending index for data — This is where the current instance of the SumThread will stop summing elements in data
          - The sum for the current thread
   3. Of course there will be a constructor that takes all the above data (except sum) and sets each member variable (you should initialize sum, but you don’t need to pass it in)
   4. We need a method that will loop through data from the start index to the stop index and store the partial sum in sum
   5. In Main.cs, create a method called sumParallel(…) that takes the array to sum and the number of threads that returns a long value result. This method should create X instances of SumThread where X is the total number of threads. Then, for each instance that is created, pass it the correct values for starting and ending position, etc., call the method that stores a partial sum in the SumThread. When these are finished, sum up all of the partial sums in each thread and return the overall sum.
   6. When you have finished steps A — E, you may then simply call the sumParallel method from main(). When doing this, calculate the starting and ending time, as was done for sumSerial, and print the time that sumParallel also takes. DO NOT CONTINUE to the next step until the result from these steps is the same result as the serial version. You can check by comparing the result returned from sumSerial and the result from sumParallel. They should be the same result since they both calculate the sum of data.
4. Once you have step 4 finished, using threads is now very straightforward. First, alter SumThread so that it implements Runnable.
   * + The code that sums up the partial sum in data is now the code that should be in the required run() method.
5. Now, all that must be altered is sumParallel. Rather than calling the run() method directly for each of the SumThread objects created in step 4, now create a new Thread for each instance, passing the corresponding instance of the SumThread object and then call start() on the thread object. It would be best to have an array of Thread objects for the next step.
6. Once step 6 is finished, we must now “wait” for all the threads to finish. To do this, place the following code below the portion that starts all the threads:

//now just wait for them to finish...

for(int i = 0; i < numThreads; i++) {

try {

threads[i].join();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

1. This will “wait” for all threads to finish before continuing to the next step
2. Once step 8 is finished, all that is left is to sum up all the partial sums in data.threadData and return that result.
3. Change the number of threads to see which performs the fastest summation.

# Bonus Challenge

If you note, one of the steps that actually takes the longest is creating the array data. Use what you learned to create the array data in a parallel version to speed up the initial process