Assignment 4 description

3 Description

Assignment 4
CSCI 335 for Fall 2015

What Will Your Code Do?

Your code will do a very simple operation. It will find the sum of values stored in an array. To make it interesting you will do it several ways, both serially and in parallel.

To understand why you are doing it several ways, here is some background on numerical issues. (This may be review for some.) If you have two variables x & z and want the sum, you expect x + z and z + x to give the same value. Now if you have three variable, w, x & z and want the sum, you expect (w + x) + z to give the same value as x + (w + z). If they are integers this is generally true (may not be if you get an overflow in an intermediate result). If the variables are floating point numbers this may not be true. This is due to round off. Floating point numbers are only approximated in computers and the error incurred for this approximation varies by the order of the operations. For the code you will produce this means that adding an array of floating point values can give different results depending on the order you do it in.

Serial code

To start things out in an area you should be comfortable, you will first do the code serially. Afterward, you will then try the parallel equivalent. Your serial code should do the following steps:

1. input the number of values and random seed to use
2. create an array with the specified values
3. sum the array in each of the three ways specified and output the result

Let's expand each of these steps.

Step 1. First prompt the user for the number of values to sum and read in the value from the console (System.in). The input will be a single integer and you should make sure it is greater than 0. If they give you too big a value then it is their problem (you don't need to tell them now). Next, prompt the user for the random seed to use to for the permutation (used in step 3) and read in the value. The input will be a single integer.

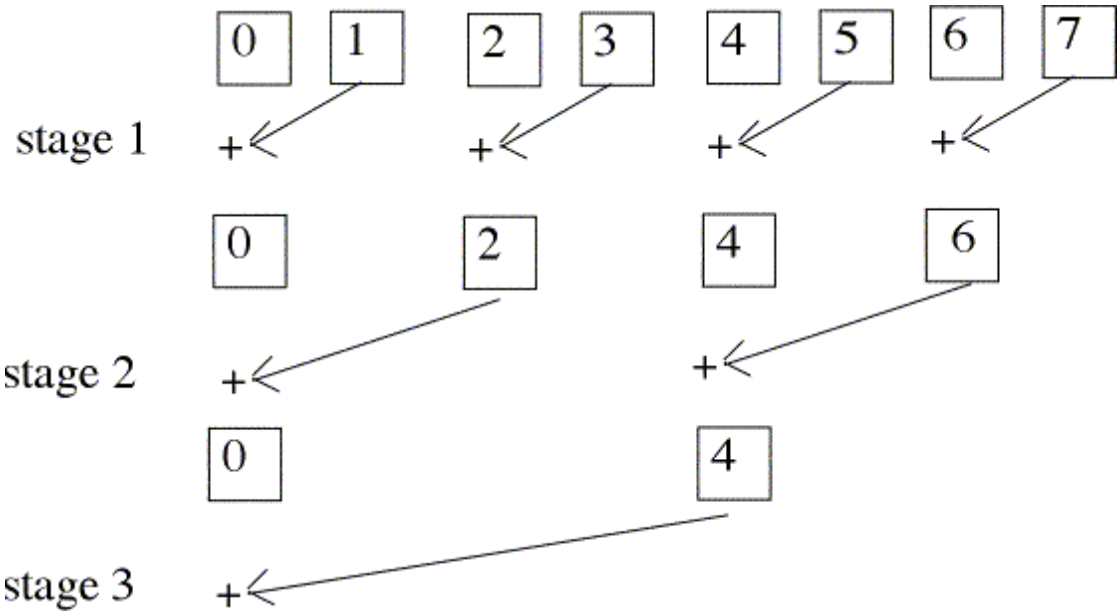
Step 2. Using the number of values to sum that was input in step 1, create an array of this size of type double. You then need to place a value in each index in the array. In the class **Helper** (package **a4Helper**) there is the static method procValue (for process value - referring to the parallel code). It expects one argument of type int. It returns one value of type double. Thus, its signature is
public static double procValue(int proc)
For index k in the array, you can use k as the input argument to procValue and must use the returned double as the value to store in the array at index k. Here is a link to the byte code (jar file) for the Helper class as well as the Javadoc. Here are two ways to easily include this package:

1. When you create the new project, on the first screen click next (after you input a name). On the next screen click the libraries tab. Then click Add External JARs. Browse to choose the a4Helper.jar file. It now shows up in the list. Click finish. Next add in the rest of your source in your own package.
2. If you already have a project you can add the a4Helper.jar after the fact. Right click on the project in the Package Explorer. Choose Properties from the popup menu. In the popup, click Java Build Path on the left side. Follow the directions in 1. by starting to click on the library tab.

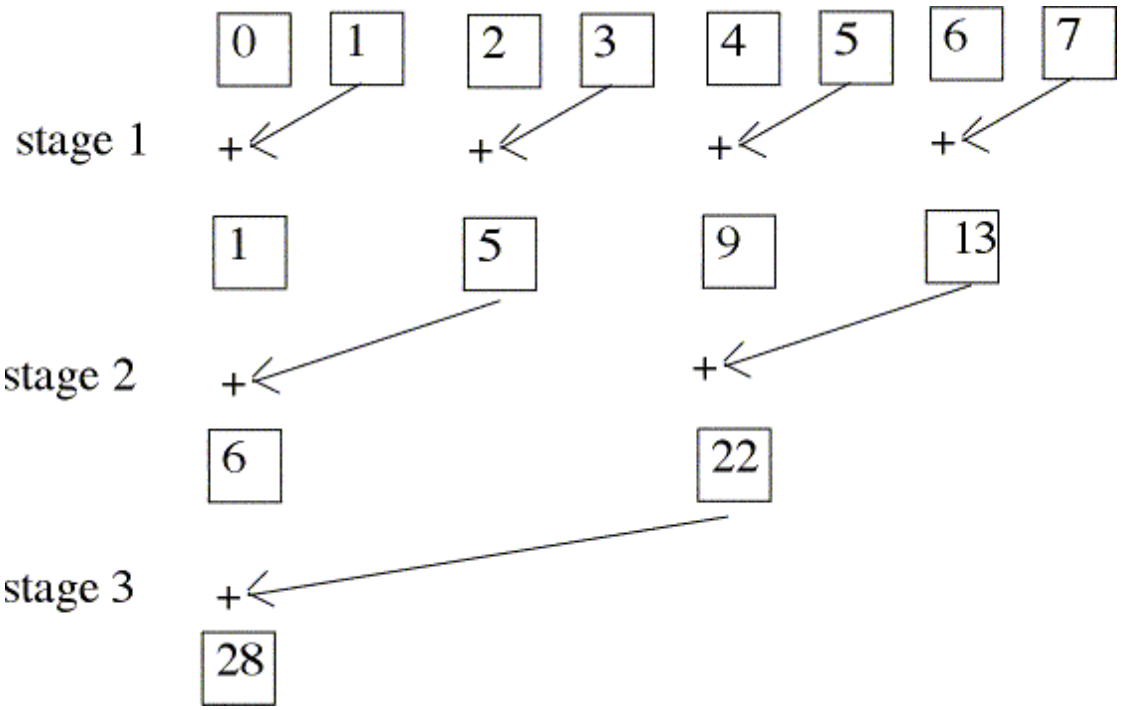
Once you have done all this, you need to make sure you import the a4Helper.Helper into your program.

By using these values you can compare your results to mine and they should agree. Note if you set testing to true then it generates simple values (not random) so you can easily check the result. This is good for initial work but you should switch to false to fully test your code.

Step 3. You will perform three sums.
Sum I) Here you simply sum the values in the array starting at index 0. This means they are added in exactly the order they appear in the array. This is considered the standard sum in the following discussion. Output the sum you obtain.
Sum II) Here you will sum the values in a random order. This random order will be determined by the method permute in the **Helper** class. This method has the signature
public static int [] permute(int numDo, int seed)
numDo is how many values you want to permute and seed is the seed to use in generating the random permutation. The seed should be set to the value read in step 1. The method returns an int array of size numDo. The values in the array are 0 to (numDo-1), inclusive. These values will be in random indexes in the array. The first value you should sum is in index 0 of the returned array. The second value to sum is in index 1, etc. If you think about it, it should become clear that it is easy to loop from 0 to (numDo - 1) and use the returned array index to sum the values. If it does not become clear then contact me. Once you have the sum you should output it and also output the relative error between this sum and the standard sum. This is defined as
(standardSum - sum) / standardSum
What this does is normalize the error to the value expected. If this value is zero then the sums are the same. It might be wise to check that the relative error is very small (say less than 1.0e-13) and print out a message if it is not. This can help catch errors.
Why do we input a seed and use it? If you give the same seed you will get the same permutation returned. This makes debugging much easier. However, you can also change the seed to make sure your code works for different cases and to see the impact of adding values in a different order.
Sum III) You will do this sum in the same order as done in a parallel prefix. This is a way to do parallel computation that can be faster. This was discussed in class. To be faster you need multiple processors doing the sum. We will not have this but will imitate what happens when we do. The basic idea is values next to each other get paired and summed. This eliminates half the locations. The remaining values are paired and summed and this continues until the sum is done. An example is given for 8 values in the figure below:



This figure indicates that in stage 1, index 1 is added to index 0 (so the result of the sum is stored in index 0), index 3 is added to index 2, etc. In stage 2, index 2 is added to index 0 and index 6 is added to index 4. In stage 3, index 4 is added to index 0. At this point, index 0 has the complete sum. In the example below, the value at the start is the same as the index and they are added to show the running values.



The trick to doing this step is to figure out who adds to whom at each stage. It isn't super hard but takes a little thought. Keep in mind your technique has to work even if there is not a power-of-two number of values. In this case, the index does not enter the computation until it needs to do an add. For example, if you have 5 indexes, then index 4 does not enter until stage 3. If you can't come up with an algorithm after a little thought then talk with me. Once you have the sum you should output it just as you did in sum II where the relative error is for this sum.

You may notice that the pictures look like an inverted binary tree. This also says something about how the order of operations is related to binary search. The height of the tree is O(log N) and if you have enough processors the algorithm can run in that amount of time.

Parallel code

The parallel code will do the same sum in the same three ways except each thread will hold one value. You will implement the first two and try to design a solution for the third (no code).

Here are a few details of what to do:

- You will need to initialize the array with the same values you used in the serial case. You can make the array static so all threads will see it (there are other techniques). **You may not use the constructor to set the initial values of the array or use a static method to do this**. Each thread must set its own value when it is started. An issue is how does the thread know which number it is to set/use in the array. You can set a field in the class before you start the thread (there are other techniques).
  A last detail is you will need to wait until all the threads have initialized the array. You could roll your own code to have them wait. However, Java provides its own solution. Take a look at java.util.concurrent.CyclicBarrier. It allows you to make every thread wait until they all reach the barrier.
- Surprisingly, the easiest one to do (I think) is the random case (sum II). Here, you can have the threads run in any order, add their value, and then finish. Note you could use the random permutation provided in the Helper class to start the threads in a random order.
- Doing the ordered sum (index/thread 0 first, then 1, etc.) is a little harder (sum I). One technique to think about is to have a static field that records whose turn it is to sum next. A thread that is woken up can check if it is its turn. **You may not spin wait to wait for a threads turn to add - you must you some form of wait/notify.**
- The parallel prefix sum is the hardest (sum III). **You will not implement this now.** What you will do is design a solution and turn that in. We will discuss the different ideas in class. (**You may not discuss across teams during the assignment.)** What I want to see is a good effort that comes up with a way to deal with potential race conditions. If you are having a lot of trouble then come and see me. A few notes:

    - If the number of values/threads is a power-of-two then it should be easier. Work on this first. Try the other cases once you have this done. Consider an algorithm that works in the general case to be extra credit.
    - I was able to come up with a way to do this where you don't notifyAll but only notify the thread that needs to know. This is trickier. Again, you don't have to do this but it is interesting and will teach you something.
    - It turned out to be surprisingly tricky to get everything to work. What I am looking for is your careful attempt to solve this. It might have mistakes but not ones you can see or should have seen.
- I created three separate classes (you only do two classes at this point) to do each of the three types of sums. This may not be required but I found it easiest. It means that threads are created and destroyed for each of the sums. The setting of the array values is the same in each but the sum details varies. It is up to you how you choose to do this.
- You will output both results along with the relative error between the parallel and sequential sum. You use the sequential value for the denominator. For each you do
  (sequentialSum - parallelSum) / sequentialSum
  where the same type of sum (I or II) is used for both parallel & sequential. In my sample output I say that the random case is a difference and the ordered case is an error. This is because adding in a random parallel order will yield a different order than the sequential random.
- One trick I used was to start the threads in a random order to see if that caused problems. I used the permute method in the Helper class to decide the order. This is very useful to catch mistakes in your code.