Parallel Min and Max

Course Level:

CS1

PDC Concepts Covered:

PDC Concept	Bloom Level
Concurrency	C
Sequential Dependency	C
Data Parallel	A

Programming Knowledge Prerequisites:

Basic programming knowledge in Java is required for this lab. More specifically, the below skills are sufficient to complete the coding assignment.

1. Variable declaration
2. If-else
3. For loop
4. Functions/methods

Tools Required:

An editor
JDK 8+ and Pyjama

Problem Description

In this lab you will be writing a program to find the smallest and the largest numbers in an array. Consider Table 1 below which shows the steps for finding the smallest number in an array. The algorithm uses a loop that indexes into the array using the loop variable. It also uses a variable that holds the smallest value found in the array so far. This variable called *smallest* begins with the value in the first item of the array. Then, at each iteration, the value of the array item at the current index is compared to the value in *smallest*. If the array value is smaller, then the *smallest* is updated to have the new smaller value. Once all iterations are finished, the smallest variable will hold the overall smallest value in the array. Note that the loop only iterates nine times. The algorithm starts iterating at index 1, not 0. It does not need to compare the first number in the array because it is initially considered the smallest. Implementing an algorithm to find the largest is similar.

Convince yourself that the table below is correct. Can you create an algorithm that does what the table depicts?

array:

5.0	10.3	8.7	5.0	52.9	18.0	13.0	2.3	82.7	68.2

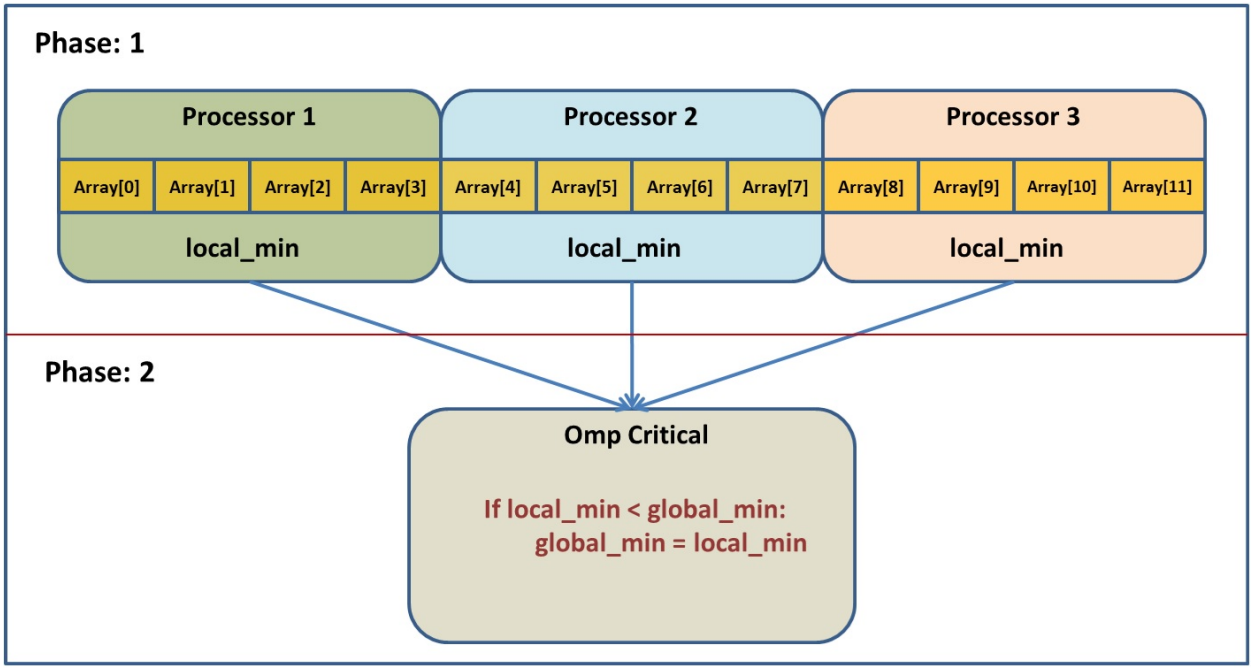
Loop step	Index	array[index]	Smallest
initialize			5.0 ← array[0]
first	1	10.3	5.0
second	2	8.7	5.0
third	3	5.0	5.0
fourth	4	52.9	5.0
fifth	5	18.0	5.0
sixth	6	13.0	5.0
seventh	7	2.3	2.3
eighth	8	82.7	2.3
ninth	9	68.2	2.3

Table 1: Summing integers in an array.

Methodology

To implement a parallel version, you will use domain decomposition, also sometimes called data decomposition, to find the smallest (min) and largest (max) value of the array in parallel. Domain decomposition requires dividing the array into equal parts and assigning each part to a processor. Consider the simple example of 12 numbers and three processors. The computation occurs in two phases. First, the work is divided equally among the three processors. In this case, each processor finds the min and max of four numbers in the array. The first processor finds the min/max value among the elements starting at index 0 and ending at index 3, the second processor finds the min/max value among the elements starting at index 4 and ending at index 7, and the third processor finds the min/max value among the elements starting at index 8 and ending at index 11.

The second phase occurs after all of the processor are finished with the first phase. In this phase, each processor compares its min/max to the global min/max and updates the global values if needed.



Implementation

The first thing you will need is the following:

```
// define gmax and gmin, the global min and max variables, here…
//#omp parallel num_threads(THREAD_NUMBER)
{
// declare local min and max here…
…
}
```

This creates a number of threads that will each run what is in the braces.
Each thread will need a local min and max variable. If you define the variable inside of the braces then each thread gets its own copy. If it is defined before the parallel section then by default there is only one variable and all the threads can access it. We want the first option.

Now we want to split up the array so that each thread gets a portion to search through.
This can be done using:

```
//#omp for
for(int i = 0; i < …) {
  // put code to update local min and max here…
}
```

Putting this pragma before a loop will assign each thread an equal portion of the indices looped over.

Now, inside the FOR loop, we can read the values and update the local min and max as needed.

The final thing to do is to combine the local mins and maxes into a single total min and max. The potential problem here is that there is only one global min (and max) variable. If multiple threads try to change the value then they will create a *race condition*. In other words, the final value depends on the order of the writes. In fact, if you run the program multiple times, you can get different answers each time! The solution to this is to use a critical section:

```
#//omp critical
{
  // put code here to update gmax and gmin
}
```

The code inside the braces is executed by all the threads, but only by one thread at a time. We can compare the local min/max to the global min/max and update the global values if needed inside this section to avoid the issues mentioned.

To see if your parallel version of this program is any faster, add some timing code.

Put a call to this function before and after your parallel code, then print the difference, i.e.

```
double start = System.currentTimeMillis();
// Do all the parallel stuff above right here…
double end = System.currentTimeMillis();
double compute_time = (end-start)/1000;
System.out.println("Computation time = "+compute_time+" seconds");
```

Post Lab Questions

1. What is the average time for twenty runs of the serial version of the code?
2. What is the average time for twenty runs of the parallel version of the code?
3. Calculate the speedup of the parallel version. Is the parallel code significantly faster?
4. The *Methodology* section above described how you decompose the summation routine to parallelize it. Obviously, Pyjama did all the work for you. How many elements of the array do you think OpenMP assigned to each processor? Hint: have your code print the number of threads in the computation (the function Pyjama.omp_get_thread_num() returns the number of threads).

Turn In

Submit a file called README, as a text file, that has the post lab questions and the answers to those questions. Also include your source code.