Parallel Sum

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 C++ 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

A C++ compiler that is OpenMP capable (e.g. gnu gcc C++ compiler)

Prolog and Review

For this lab you should review your course instruction on single dimension arrays and iterating over single dimension arrays. Unlike scalar variable, such as variables of type int, double, and char, array variable can hold more than one value. Consider the following variable declarations.

```
1 int i = 5;
2 int numbers[3] = {10, 20, 30};
```

The variable declaration on line 1 of the above code declares a variable called i that holds the number 5, and that variable can only hold one number. However, the variable declared at line 2 can hold 3 integers, and is initialized to hold the integers 10, 20, and 30. Accessing the integers is as simple as indicating the correct index. In C++, arrays are zero based, so the first item in the array always starts as index 0. So, in the above example, numbers[0] holds the integer 10, numbers[1] holds the integer 20, and number[2] holds the integer 30.

Once you have item stored in an array, you can iterate over those items using a simple loop structure and indexing into the array using the loop variable. Consider the following code.

```
1  #include <iostream>
2
3  const int NSIZE = 10;
4
5  void gen_numbers(float numbers[], int how_many);
6  float gen_rand(int min, int max);
7
8  int main() {
9    float numbers[100];
10
11   gen_numbers(numbers, NSIZE);
12
13   std::cout << "Generated numbers are: " << std::endl;
14   for (int i = 0; i < NSIZE; i++) {
15     std::cout << numbers[i] << " " << std::endl;
16   }
17
18   return 0;
19 }
20
21 void gen_numbers(float numbers[], int how_many) {
22   for (int i = 0; i < how_many; i++) {
23     numbers[i] = gen_rand(0, 10);
24   }
25 }
26
27 float gen_rand(int min, int max) {
28         return (min + static_cast <float> (rand()) /
29                         ( static_cast <float> (RAND_MAX/(max-min))));
30 }
```

The above program defines a function called gen_numbers(). Note how gen_numbers generates an array of floating point numbers by using a for loop to place the random float, generated by calling gen_rand(), into the array. The for loop uses the loop variable i to index into the numbers array.

Problem Description

This lab you will be writing a program to sum numbers stored in an array. Summing numbers is straightforward. Consider the Table 1 below. All that you need to do is use a loop that indexes into the array using the loop variable. You will also need an accumulator variable. The

accumulator variable will hold the running sum for each iteration. Once all iterations are finished, the accumulator variable will hold the total sum.

array: 5.0 | 10.3 | 8.7 | 5.0 | 52.9 | 18.0 | 13.0 | 7.3 | 82.7 | 68.2

Loop step	Index	array[index]	Accumulator (sum)
initialize	0	----	0
first	0	5.0	0 + 5.0 = (5.0)
second	1	10.3	5.0 + 10.3 = 15.3
third	2	8.7	15.3 + 8.7 = 24.0
fourth	3	5.0	24.0 + 5.0 = 29.0
fifth	4	52.9	29.0 + 52.9 = 81.9
sixth	5	18.0	76.9+ 18.0 = 99.9
seventh	6	13.0	94.9 + 13.0 = 112.9
eighth	7	7.3	107.9 + 7.3 = 120.2
ninth	8	82.7	115.2 + 82.7 = 202.9
Tenth	9	68.2	197.9 + 68.2 = 271.1

Table 1: Summing integers in an array.

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

Methodology

You will use domain decomposition, also sometimes called data decomposition, to sum the array of numbers in parallel. Domain decomposition requires dividing the array into equal parts and assigning each part to a processor. Consider the simple example in Figure 1. The figure depicts three processors, including a processor designated as the master processor. The computation occurs in two phases. First, the work is divided equally among the three processors. In this case, each processor sums four numbers in the array. The master processor sums all elements starting at index 0 and ending at index 3, the second processor sums all elements starting at index 4 and ending at index 7, and the third processor sums all 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, the master adds all of the sums, stored by the variables accum1, accum2, and accum3 in the figure, to compute a final total.

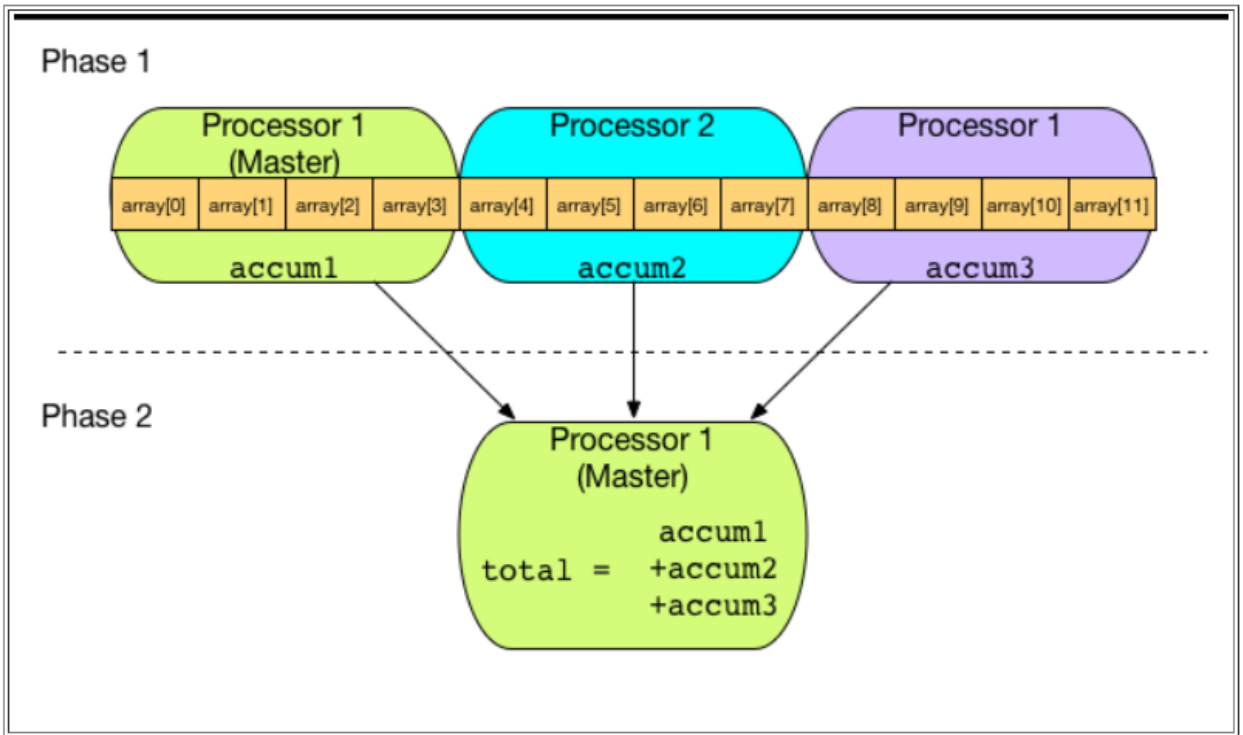

Figure 1: Three processors computing the sum in parallel

***Note to the instructor:** Instructor can illustrate **Sequential dependency** by pointing out that the global sum cannot be computed before the completion of computing the local sum by the individual processor. Instructor can illustrate **Data parallel** by pointing to that different processors are doing the same computation but on different pieces of data.

Implementation

For this lab, you will be implementing the code to sum numbers in serial and then in parallel.

Generating Lots of Random Floating Point Numbers

You need lots of number to store in the array so that you can sort them. Normally, you would have some important data, perhaps stored in a file or database, that you need to sort. However, for this lab, you will generate some data with which to work. The section Prolog and Review included some functions that generated floating point numbers. Use those function to complete your lab (`gen_numbers()` and `gen_rand()`).

Summing the numbers

Summing the numbers was summarized in the description section. You were challenged to think through an algorithm to sum the numbers. The following is such an algorithm:

```
sum()
   inputs:
       array       - the array of numbers
       num_elements - the number of elements in the array
   returns:
       sum         - the sum of all the numbers in the array

   begin
      set sum to 0
      loop index from 0 to num_elements
         set sum to sum + the number in the array at position
index
      end loop
      return sum
   end
```

Implement this algorithm as a C++ function. Now, write a `main()` that allocates an array of 1000000000 floating point numbers. If you allocate your array statically, you will need to make the array a global variable because an array that large may be bigger that the maximum size of the program stack for your particular operating system. Call the `gen_numbers()` function to generate numbers and put them into the array. Finally, in `main()`, call the `sum()` function to sum the numbers, and then print the result. Put all of you code into a file called `sum_serial.cpp`. Compile it and run it to make sure it works.

Summing in Parallel

Can you make the serial version faster? You can make it a bit faster by doing both the generating of the floating point numbers and the summing of the floating point numbers in parallel.

Copy the `sum_serial.cpp` file to a file named `sum_parallel.cpp`. Now, you will modify the new file to make a parallel version. The *Methodology* section above showed how the array should be decomposed for each processor in the system to compute a part of the array, and thus compute the parts in parallel. Fortunately for you, you will use OpenMP, and it will take care of doing all of the parallelization for you. All you need to do is annotate your C++ code with the proper compiler pragmas.

First, parallelize the number generation code. In the gen_numbers() function, add a line before

the `for` loop. The line should be the following.

```
#pragma omp parallel for
```

Yes, that's it. Now your `for` loop has been properly parallelized.

Now, parallelize your `sum()` function. You can parallelize the for loop in you sum() function just as you did the for loop in the gen_numbers() function. However, you will not get the correct answer. Review the diagram in Figure 1 of the *Methodology* section. Each processor must have its own accumulator, and then the master processor must sum the individual accumulators together to get a final total. Combining the results of the computations is called a *reduction* is parallel programming. Fortunately, OpenMP will handle this operation for you as well. You just need to tell OpenMP what variable to reduce and what operation needs to be applied. Therefore, add the following line to your sum function directly before the for loop.

```
#pragma omp parallel for reduction(+:accum)
```

Note that the above line of code is valid if you named your accumulator variable `accum`. If you named your variable something else, which you probably did, then change the line to use your variable name instead of `accum`. Compile your code and run it to make sure it works.

So, What's the Difference?

To see how your code with the added OpenMP code makes a difference, you can add the following code before you call `gen_numbers()` to your `main()` function in both your serial and parallel version:

```
// Note that the following code works for g++ on Linux, Mac OS, and Windows
(using MinGW)
struct timeval tv1;
struct timeval tv2;
gettimeofday(&tv1, NULL);
```

Next, add the following code immediately after your call the `sum()` function in main() for both the serial and parallel version:

```
gettimeofday(&tv2, NULL);
std::cout << "Total time: " << (double) (tv2.tv_usec - tv1.tv_usec) / 1000000
+
  (double) (tv2.tv_sec - tv1.tv_sec) << std::endl;
```

Now, run the serial version twenty times and calculate the average time. Next, run the parallel

version twenty times and record the average. Next, calculate the speedup based on the average times.

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, OpenMP 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 `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 for both the serial and parallel versions.