Parallel Sort

Course Level:

CS2

PDC Concepts Covered

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

Programing Knowledge Prerequisites:

- Basic programming knowledge (arrays, looping, functions)
- Basic concept of parallel programming (in OpenMP)

Tools Required

- An editor.
- C++ compiler that is OpenMP capable (such as the gnu C++ compiler).

Prolog and Review

For this lab you should review your course instruction on sorting arrays. Most of the common sorting algorithms are implemented with nested loops. For example, selection sort, bubble sort, and insertion sort are all implemented with doubly nested loops. In computer science, such algorithms are termed as, asymptotically, n-squared algorithms. An n-squared algorithm that has n input items takes approximately n times n steps to sort the items. Such algorithms can take a very long time to sort large data sets. So, how can you reduce the sorting time? You can reduce the sorting time with parallelism, of course!

Problem Description

This lab you will be writing program to sort numbers stored in an array. You will use the bubble sort algorithm. The bubble sort algorithm is a pretty simple sort. Consider the Table 1 below. The table shows how the bubble sort algorithm make multiple passes over the array. At each pass, it iterates over the array and examines pairs of numbers. The numbers shown in white are numbers that did not have to be swapped because the numbers are already in the correct order relative to each other. For example, in the first pass and the first iteration, the algorithm compares the numbers 5.0 and 10.3. The algorithm does not swap the number because 5.0 is less than 10.3. However, in the second iteration, the algorithm swaps 10.3 and 8.7 because those

number are out of order. Notice that the algorithm stops when no more swaps are needed to properly order any pair of numbers.

	array:	5.0	10.3	8.7	5.0	68.2
Pass	Iteration					
1	1	5.0	10.3	8.7	5.0	68.2
1	2	5.0	8.7	10.3	5.0	68.2
1	3	5.0	8.7	5.0	10.3	68.2
1	4	5.0	8.7	5.0	10.3	68.2
2	1	5.0	8.7	5.0	10.3	68.2
2	2	5.0	5.0	8.7	10.3	68.2
2	3	5.0	5.0	8.7	10.3	68.2
2	4	5.0	5.0	8.7	10.3	68.2

Table 1: Sorting integers in an array.

Methodology

You will use domain decomposition, also sometimes called data decomposition, to bubble sort an 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 sorts four numbers in the array. The master processor sorts all elements starting at index 0 and ending at index 3, the second processor sorts all elements starting at index 4 and ending at index 7, and the third processor sorts 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. Note that sorting the pieces of the array does not result in a completely sorted array. In the second phase, the master must merge sorted pieces to produce a completely sorted result. However, the second phase must be done serially by a single process.

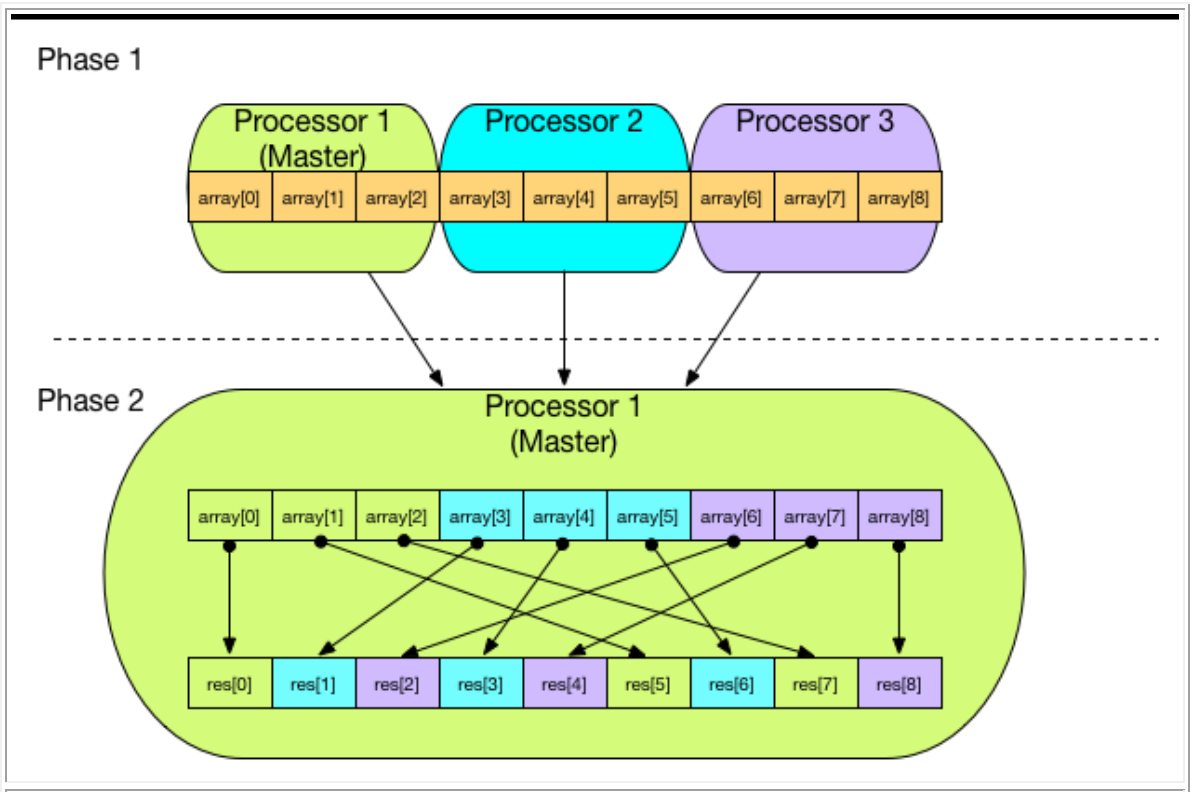

Figure 1: Three processors sorting in parallel

***Note to the instructor:** Instructor can illustrate **Data parallel** by pointing out that different processors are doing the same computation but on different pieces of data at phase 1.

Implementation

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

Generating Lots of Random Floating Point Numbers

You need lots of numbers 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. Initialize the random number generator by calling srand(time(NULL)). Then call rand() to get a new random number.

Sorting the numbers

An example of bubble sorting an array was shown in the *Methodology* section. You should be able to implement the bubble sort algorithm in C++. The simplest algorithm is the following:

```
bsort()
  input:
     array     - an array of integers
     num_items - the number of items in the array
  returns:
     nothing
  begin
    loop i from 0 to num_items - 1
      loop j from num_items-1 down to i + 1
        if the array item at j-1 is greater that the array item at j
          swap the array item at j-1 with the array item at j
        end if
      end for
    end for
  end
```

This is a very simple version of the bubble sort algorithm. See if you can implement a more efficient version.

After you write the bubble sort in C++, add code to your `main()` that calls the bubble sort algorithm on the array that after the code has read the numbers from the file. You should test your code to make sure it works, so you should call the bubble sort function on a small array and print the results.

Sorting in Parallel

You can you make the serial version faster in the previous step by sorting parts of the array in parallel and then merging the results, as describe in section *Methodology*.

Copy the `bsort_serial.cpp` file to a file named `bsort_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. For this lab, you will decompose the array into two approximately equal halves, and then have each half sorted by its own thread. In other words, each thread will call the bsort() function, but will pass the beginning of its half along with the number of items in its half. Something similar to the following should replace the call to bsort() in your code:

```
1   int size1 = how_many/2;
2   int size2 = how_many-size1;
3   #pragma omp parallel num_threads(2)
4   {
5
```

```
 6     if (omp_get_thread_num() == 0) {
 7       bsort(numbers, size1);
 8     } else if (omp_get_thread_num() == 1) {
 9       bsort(numbers + size1, size2);
10     }
11   }
```

In the code above, `how_many` is the total size of the `numbers` array (the array to be sorted). The code above stores the size of the first half of the array, i.e. the part that is to be sorted by the first thread, in the `size1` variable. Then, it computes the size of the rest of the array, which is the part that is to be sorted by the second thread, and stores that size into the `size2` variable. Note that `size2` may be one larger that size1. For example, in `how_many` is 11, then `size1` will be 5 and `size2` will be 6.

The pragma in the code above executes a *fork-join* with two threads at line 3. In other words, the program *forks* two threads, and each thread executes the code block beginning at line 4 and ending at line 11. Once both threads finish executing the code block, the threads *join* and only the main thread continues after line 11. What does the block of code do? Each thread, executes the `if` statement at line 6. The thread with the thread id 0 will call the `bsort()` function (at line 7) on the first half of the `numbers` array, and the thread with the thread id 1 will call the `bsort()` function (at line 9) on the second half of the `numbers` array.

Unfortunately, you are not finished. Each part has been sorted, but the entire array is not yet sorted. You must merge the two halve as explained in the *Methodology* section above. The following is a merge algorithm that combines two arrays into a finished result.

```
merge()
  input:
    a1    - first sorted array
    size1 - the size of the first array
    a2    - second sorted array
    size2 - the size of the second array
  output:
    results - the merged result
  begin
    set idx_a1 to 0
    set idx_a2 to 0
    set result_size to 0
    while idx_a1 is not equal to size1 ||
          idx_a2 is not equal to size2    // more items to merge
      while idx_a1 is not equal to size1  // more items a1 and ...
            and (idx_a2 is equal to size2  //  no more items in a2 ...
                or the item in a1 at idx_a1 is less than or equal to
```

```
                the item in a2 at idx_a2)
      set the item in results at index result_size equal to
          the item in a1 at index idx_a1
      add one to result_size
      add one to idx_a1
    end while
    while idx_a2 is not equal to size2 // more items in a2 and ...
          and (idx_a1 is equal idx1    // no more items in a1 ...
               or the item in a2 at idx_a2 is
                    less than or equal to the item in a1 at idx_a1)
      set the item in results at index result_size to
          the item in a2 at index idx_a2
      add one to result_size
      add one to idx_a2
    end while
  end while
end
```

Implement the merge function in your C++ program. Add a call to the merge() function after the parallel code block. The call should look like the following:

```
int results[how_many];
merge(numbers, size1, numbers + size1, size2, results);
```

Finally, test your parallel version of bubble sort 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 `bsort()` to your `main()` function in both your serial and parallel version (before the parallel block in `bsort_main_parallel.cpp`):

```
// 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 `bsort()` function in main() for both the serial and parallel version (after the parallel block in `bsort_main_parallel.cpp`):

```
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. In this lab, you used only two threads to sort the pieces of the array in parallel. However, you can use more threads if your computer's processor has more cores. If you use more threads, describe how the merge algorithm changes.

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.