CS 420: Introduction to Parallel Computing for Scientists and Engineers

Fall 2013

Machine Problem 3 – MPI Sorting

Due date: November 11, 5:00 PM

Background

For this MP you are given serial code for performing a sample sort, which you will parallelize using MPI. Your code will be graded on both correctness and speed. There is an extra part to the assignment for students enrolled in 4 credit hours.

System

You can use any computer with a multi-core processor and the MPI software installed to develop the code for this assignment, but you will need to use Taub to run your code at higher core counts and get running times for this assignment. Both EWS and Taub have the gcc compiler and MPI available.

You can use the provided Makefile to automate your compilation. Use the optimization level -O0 when compiling the code. Taub nodes contain dual 6-core processors. You will be asked to run tests using only one node (12 cores) when timing the code to test performance (see **Submission** for details).

Timing code is provided. As always, do not time the setup and cleanup portions of your code. Also be sure to put a barrier before beginning to time the code. You will also want to optimize the code as much as possible to get the best performance.

The program will by default output an `xor` value, which will be `0` if the sort worked correctly. It will also output a boolean value which notifies you whether the maximum and minimum values of neighboring buckets are as we would expect. The output should be `true` if the program runs correctly. This is not a complete check, but it is a quick and simple way to tell whether something has gone wrong. It also does not require all of the values to be on the same process in the parallel case, which is more realistic given that you could be expected to sort huge amounts of numbers.

The program also has the option to output a sorted list to standard output. You can enable the output by changing the `OUTPUT` constant at the top of the program. For testing, you can redirect the output to a file using "`mpirun -n 2 ./sampleSort [arg] > my_filename`". If you keep a copy of the sequential output in a separate file, you can then use the `diff` utility to check your results. Please do not use OUTPUT for large runs.

You will need to use batch scripts to run your code on the Taub's compute nodes when testing the performance of your programs. Sample batch scripts are included on the wiki.

The batch scripts and Makefile are set up so that you can have a parallel version named `psampleSort` and a sequential version named `sampleSort`. If you intend to keep your parallel code in `sampleSort`, you will need to edit the Makefile and the batch scripts.

Assignment

You will code a parallel MPI implementation of a sample sort. For those of you with some algorithms experience, a sample sort is related to a bucket sort.

A sample sort splits an array into `p` "buckets", then sorts the buckets. For this assignment, `p` should equal the number of processors. To split the items into buckets, the sort first chooses `p-1` "splitter" values, sorted from smallest to largest. Then, each item in the array is put in a bucket so that all splitters to its left are smaller (or equal), and all splitters to its right are strictly greater than the item.

The algorithm is called a sample sort because of the way we choose the splitters. We cut the array into `p` pieces. We then sort the pieces individually, using quicksort. After this, `p-1` evenly spaced elements are taken from each piece, for a total of `p(p-1)` elements. These constitute the sample. To get the splitters, we sort the sample using quicksort, then select `p-1` evenly spaced elements.

So, to put the steps in order:

1. Split the array into evenly sized, unsorted pieces
2. Sort the pieces.
3. Select evenly spaced elements of the pieces. This is your sample.
4. Sort the sample, and select evenly spaced elements of it. These are your splitters.
5. Put the data into buckets. Each item goes between a smaller splitter and a larger splitter than itself.
6. Sort the buckets individually.

The code uses two checks to determine whether the sort has run correctly. The first is an `xor` of all the elements before the sort, followed by an `xor` of this same value after the sort. If all of the same values are present before and after the sort, then the `xor` should evaluate to `0`. If not, then it will most likely *not* evaluate to 0. So, this check will tell you whether a value was "lost" when sorting. The second check is to determine whether the last value of a bucket is greater than the first value of the next. This should obviously not be the case after sorting. If the progam outputs `true`, then the sort has worked correctly in this sense. Given that sorting within a bucket is done by a library routine, it is likely that these two checks will catch almost all possible errors in your sorting output.

You must implement both of these checks in the parallel version of the code, and do so without bringing all of the values into a single node. You should not time this portion. You do not need to implement the OUTPUT functionality described in the introduction. If you find it useful to do so for debugging, please be sure that the printing code is controlled by a global variable and that this variable is set to 0 when you submit, as it is in the sequential version.

Input

The program should take one command line parameter:

```
./sampleSort N
```

Where `N` is the size of the list to be sorted. Note that N should be evenly divisible by the number of processes (or buckets), and should be greater than the square of the number of processes.

Output

You should output three values:

1. The `xor` of the elements calculated before and after sorting (this is one value), and

2. A boolean value corresponding to the second check described above, and

3. The running time for the useful portion of code.

4 Credit Hours

If you are enrolled in 4 credit hours, you will also need to parallelize a histogram sort, which is very similar to sample sort. The primary difference between histogram sort and sample sort is the way the splitters are selected. *Note that the algorithm given here is different than the one given by NIST.*

A histogram sort first splits the data into `p` evenly sized, unsorted pieces, where `p` is the number of buckets. Then it sorts the first bucket and chooses `p-1` evenly spaced values. These values are called a "probe". They represent a potential set of splitter values, as we used in sample sort. However, in histogram sort we will iteratively update our probe until the splitter values produce a nearly ideal breakup of the data.

To do so, we next sort each bucket individually using quicksort. Then a "histogram" is created for each bucket. A histogram is a count of the number of elements that fall between values of the probe. It represents a count of the way the data would be split up if we used the probe values as splitters. For example:

probe:	1	3	8				
bucket[0] data:	0	0	2	3	7	9	10
bucket[0] histogram:	2	1	2	2			
bucket[0] cumulative:	2	3	5	7			

The histograms from each bucket are then combined into a master histogram. Then, a cumulative sum of the elements of the master histogram is recorded. An example is shown above, although we do not actually get a cumulative sum within each bucket. The result tells us the effectiveness of the probe in splitting up the data into even buckets. The *i*th entry in the cumulative histogram should be $\frac{(i+1)*n}{b}$, where *n* is the size of the data and *b* is the number of buckets. This exact split will almost never be achieved, so we allow for a tolerance factor, denoted *tol*. If all of the entries of the histogram are within *tol* of the ideal values**,** then we continue on. Otherwise, we adjust our probe accordingly and repeat the process until the resulting histogram is satisfactory. To adjust the probe, we use scaled linear interpolation.

With the splitters determined, the program proceeds in the same way as sample sort. The data are moved into the correct buckets relative to the splitter values, then sorted within the buckets.

Parallelize the provided serial code, then run the same analysis on it that you run on the sample sort. The histogram sort should be run on with values greater than or equal to $nbuckets^3 * 2$.

Submission

You should run the MPI sampleSort program and record the running times for the following list sizes:

N = 1200000, 12000000, 120000000 (1.2 million to 12 million)

For each size, you should use the following numbers of cores, using only one node on Taub:

P = 1,2,6,12

So, in total, you should have 12 timed runs. The list sizes to test are a multiple of all of the processor sizes.

We are providing two scripts, one to run the parallel code on 1 compute node (maximum 12 cores) on Taub, and one to run the serial code on a single Taub compute node. The walltime limits in the scripts should give you enough time to run your code, assuming your MPI programs are well designed and are at least as fast as the serial code.

Submit answers to the following items in a pdf file.

1. Create a table containing the running time for each combination of P and N above.

2. Plot the running time as a function of P for different values of N. Plot the number of cores (P) as the x-axis , and the running time as the y-axis, with a different line for each N.

3. Create a speedup plot. With the number of processors as the x-axis, plot the speedup $\frac{Time\ on\ 1\ processor}{Time\ on\ p\ processors}$ as the y-axis. Make different lines for each value of N.

4. Discuss the results you obtained. Were you able to gain a linear or near-linear speedup? Why or why not?

5. A bucket sort is similar to a sample sort, except the endpoints of the buckets are predetermined as evenly spaced along the interval from the minimum to the maximum value. Therefore, no sample is taken. Instead, the algorithm immediately splits the data into the buckets and then sorts the buckets. When could a bucket sort be better than a sample sort? When would it be worse?

Name your discussion file `netid_mp3.pdf`. It is important that you strictly follow the file naming guidelines for your submission since we will be using automated scripts to help with the grading.

Submit your fastest parallel code in the file `sampleSort.c`.

If you are a 4 credit hour student, repeat items 1 through 4 with the histogram sort. Name your C file `histogramSort.c`.

Copy your report, all of your parallel source code, and your Makefile into a folder named mp3. Either zip the folder (Windows) or `tar` and `gzip` it (Linux). Be careful when using `tar` as you can accidentally overwrite your files if you put the arguments in the wrong order. Submit your .zip or .tar.gz file to the Subversion repository. The timestamp given by Subversion will be used to determine whether an assignment is on time, so be sure to upload it by the deadline.