Prof. Jingke Li (FAB 120-06, lij@pdx.edu); Classes: M 16:40-18:30, W 16:40-17:20, Labs: W 17:25-18:55, all @ FAB 88-10.

Assignment 4: Programming with MPI

(Due Wednesday, 3/11/20)

This assignment is to practice message-passing programming with the MPI interface. You'll implement the sample sort algorithm discussed in class. For this assignment, the requirements are the same for all students.

1 The Algorithm

The sample sort algorithm consists of the following steps:

1. Divide the unsorted data (of size N) evenly among processes.
2. Every process sorts its data with a fast sequential algorithm.
3. Every process collects $P-1$ samples at equal-interval points.
4. All the samples are collected to a single process and are sorted there.
5. $P-1$ values are selected from equal-interval points in the sorted list, and are broadcast to all processes.
6. Every process divides its data to P partitions using the $P-1$ "splitter" values it receives; and send partition j to process j.
7. Every process merges P copies of partition.

2 Implementation Details

Your program should be written in C and be named `sampleSort.c`. It should follow the standard SPMD-style, *i.e.* a single program to be replicated over the participating processes. The program's interface is as follows:

```
linux> mpirun -n P ./sampleSort <infile> <outfile>
```

It reads data from `<infile>` and outputs result to `<outfile>`. The data are integers and are encoded as 32-bit (four-byte) values in the input and output files. (*Note:* Their values happen to be in the range `[0, 8191]`, but this has no impact on this program.) The data size N is to be derived from `<infile>`'s size. There is no restriction on the values of N and P.

The following are step-by-step implementation details:

1. **Concurrent File Input** All processes perform the following:
   (a) Open the input file for concurrent reads;
   (b) Use the routine `MPI_File_get_size()` to get file size and then calculate data size `N` from it;
   (c) Allocate a data buffer of size `max(64,2N/P)`;
   (d) Compute a proper offset to read an equal-sized section of values from the input file;
   (e) When done, close the file.

   Different processes will read different sections of the input file, but all sections are about the same size, *i.e.* `N/P`. If `P` does not divide `N`, the left-over items can be assigned to any process(es).

   The buffer size of `max(64,2N/P)` is on the safe side; it should be enough to accommodate all cases.

2. **Sorting the local data** It is fine to use bubble sort (or any other simple sorting routine) to sort the local data.

3. **Finding the "splitter" values** This step requires the use of collective communication routines, for gathering samples and for distributing the final set of "splitter" values. It is a common practice to use process 0 as the single "leader" process.

   Note: The following alternative to Step 4 of the Algorithm is allowed: Instead of sorting all the samples and then selecting the final set of "splitter" values from the sorted result, we can compute the average value for each of the $P - 1$ sample-point groups, and use these average values as the final "splitter" values.

   The advantage of this alternative is that it does not require assembling groups of data and sorting them, and the average computation can be achieved with a collective reduce routine (plus some additional operations).

4. **Shuffling sections** This is an all-to-all data shuffle step, and is the most challenging part of this program. You have to think carefully how to achieve this. Multiple communication routines and additional helper arrays might be needed. You should look into variable-sized collective communication routines for this part.

5. **Concurrent File Output** Arrange for all the processes to write their sorted results to the output file. You are required to use concurrent writes for this part. Figure out a way to compute output offsets for all processes. Open the output file (for concurrent writes); write the sorted array to the proper location using the offset information. Close the file afterwards. Since the offset value cannot be computed with local information alone, some form of communication is needed.

Your program should only use collective routines for communication needs, no individual sends and receives. It should also not allocate any buffer with a size larger than `max(64,2N/P)`.

Include a pair of `printf()` statements in the `main` function of the program, one at the beginning and one at the end, to indicate the start and the end of the current process. Include process' rank and host node's name in the message. You may insert additional debugging printing statements in your program, but they should all be placed within brackets `#ifdef DEBUG` and `#endif`.

3 Additional Information

- The program `datagen.c` can be used to generate new input files. It takes an integer argument, `N`, and generates `N` random integers with value in the range [0, 8191].

```
linux> ./datagen 1024 > data1k
```

- The program `verify.c` can be used to verify that the data in a file are sorted in an ascending order:

```
linux> ./verify out1k
Data in out1k are sorted.
```

- To see the content of an input or output file, use the Linux utility, `od`:

```
linux> od -i in32   -- display binary content of file in32 as integers
```

Summary and Submission

Write a short (one-page) summary (in plain text or pdf) covering your experience with this assignment. Make a zip file containing all your programs and your write-up. Submit it through the "Assignment 4" dropbox on D2L.