CMSE/CSE 822: Parallel Computing
Fall 2019, Homework 6
Due 11:59 pm, Dec 1, 2019

Important note: *Please use a word processing software (e.g., MS Word, Mac Pages, Latex, etc.) to type your homework. Follow the submission instructions at the end to turn in an electronic copy of your work.*

Parallel Bucket sort with MPI

In this problem, you will implement a parallel bucket sort algorithm using MPI. For your convenience, a starter code for the bucket sort implementation is provided in bucket_sort_skeleton.c. You can learn more about the sequential bucket sort algorithm here. In bucket sort, the range of input numbers must be known in advance. To make things simple, we will be generating random numbers between 0 and 1:

```
double r = rand() / RAND_MAX;
```

Note that the above code piece will generate random numbers with a uniform distribution.

Part 1 [60 pts]

1. *Implementation.* You will first implement two different versions of MPI parallel bucket sort. The critical step in bucket sort is binning elements into their corresponding buckets. With p processors, let us define p buckets where bucket 0 contains numbers in the range [0, 1/p), bucket 1 contains numbers in [1/p, 2/p), so on and so fort. While both versions, use the same binning mechanism, they differ in terms of the source of the input values.

a) Single source bucket sort (bucket_sort_v1.c):
   - **generate:** create the array at the root process (using uniform distribution as above),
   - **bin:** root process determines buckets -- which data belongs to which processor,
   - **distribute:** root sends buckets to appropriate processes,
   - **local sort:** each process sorts the data locally using quicksort,
   - **gather:** finally, results are gathered at the root process.
   - **verification:** root process should verify the correctness of the result by scanning the list from start to end and make sure elements are in increasing order.

b) All source bucket source (bucket_sort_v2.c):
   - **generate:** create N/P elements randomly on each process (using uniform distribution),
   - **bin:** each process determines buckets - which of their data belongs to which processor,
   - **distribute:** each process sends its buckets to corresponding processes,
   - **local sort:** each process sorts their data locally using quicksort,
   - **gather:** finally, results are gathered back at the root process.
   - **verification:** root process should again verify the correctness of the result.

2. *Performance analysis.*

a) Report the total execution time (excluding the verification part), speedup and efficiency of your implementations starting with a single socket (14 cores) and using up to 5 nodes (140 cores) on the intel16 cluster. Use problem sizes of N = 100 million, 500 million and 2 billion which is provided as a command line argument to the program. You can take performance on a single socket (14 cores) as your base case.

b) In addition to measuring the total execution time, measure different phases of your codes (generate, bin, distribute, local sort, gather). For instance:

```
t1 = MPI_Wtime();
// code to bin numbers to buckets
t2 = MPI_Wtime();
if (my_rank == 0) printf("Binning took %.2f seconds\n", t2-t1);
```

Which phases in your code are the root causes of the performance difference that you observe between your bucket sort implementations? Make sure to vary your problem size so that you can detect any trends.

c) Part of the grading criteria here will be comparison of the performance of your best performing implementation against our reference code (which is actually not a well-tuned implementation).

Reminder: Make sure that you request enough memory in your job scripts! Note that default total memory per job is 750 MBs (across all nodes allocated for the job), but 1 billion doubles alone will require 8 GBs of memory! To prevent excessive memory usage, it may be a good idea to first count & send each processor the number of elements that they will have in their buckets.
Also, to ensure timely completion of your jobs, do not specify unnecessarily long times in your job scripts (definitely less than 4 hours of total execution time, but the shorter the better). A good implementation should be able to sort 2.5 billion numbers under 10 minutes even using a single socket. Note that with larger core counts, you will ideally need shorter times.

Measuring your execution time properly. The MPI_Wtime() command will allow you to measure the timing for a particular part of your program. *Make sure to collect about 3 (or ideally 5) measurements, eliminate the slowest one (or two) execution(s) (to cancel the network noise) and take the averages of the remaining runs while reporting a performance data point.*

Part 2 [40 points]

1. In this part, you will investigate the impact of data distribution on performance. Recall that rand() function produces numbers with a uniform distribution. Instead, now use the square of each number pulled from a uniform distribution, i.e.

```
double tmp = (rand() / RAND_MAX);
double r = tmp * tmp;
```

This new distribution may cause load imbalances and slow down bucket sort. Investigate whether this is the case. Use your bucket_sort_v2 code from the first part, and compare its total execution times with uniform and squared distributions. Make sure to experiment with different input sizes and core counts as in the first part.

Note: For reporting, you can give the min, max and average running times per processor in each case in a table such as the one below, or in a chart.

	Uniform Dist.			Squared Dist.		
#processors	Min	Max	Avg	Min	Max	Avg
2						
4						

2. The load imbalance problem can be remedied by selecting better pivots for a given input. Implement a new version of your bucket sort program, bucket_sort_v3.c. This new version will be identical to bucket_sort_v2, except now you will select pivots by sampling. Your code should randomly select S samples from the entire array A, and then choose p-1 pivots (where p is the number of cores) from the selection using the following process:

- Each process selects S/p elements randomly within its part of array A
- S samples are then gathered at the root process
- root process sorts samples locally into an array S_sorted
- root selects the pivots as [S_sorted[S/p], S_sorted[2S/p], S_sorted[3S/p] ...]

Note that there need to be p-1 new pivots – 0 is still the min for the first bucket and 1 is the max for the last bucket. You are free to select the value of S as you wish, but a good reference value is S=12*P*lg(N).

4. Analyze the performance of your new bucket sort implementation by reporting its total execution time (excluding the verification of correctness part), speedup and efficiency on a range of problem sizes and core counts. Is the load imbalance problem remedied?

5. Indicate how much time your new code spends in selecting pivots vs. doing the actual bucket sort. Compare the performance of bucket_sort_v3 with bucket_sort_v2. Is the time spent in selecting pivots worth it?

Instructions:

- **Cloning git repo.** You can clone the skeleton codes through the git repo. Please refer to *"Homework Instructions"* under the *"Reference Material"* section on D2L.

- **Discussions.** For your questions about the homework, please use the Slack group for the class so that answers by the TA, myself or your classmates can be seen by others, too.

- **Submission.** Your submission will include:
  - A pdf file named "HW6_*yourMSUNetID*.pdf", which contains your answers to the non-implementation questions, and report and interpretation of performance results.
  - All source files and makefiles used to generate the reported performance results. Make sure to use the exact files names listed below:
    - bucket_sort_v1.c
    - bucket_sort_v2.c
    - bucket_sort_v3.c
    - makefile

    These are the default names in the git repository. It is essential that you do not change the directory structure or file names.

  To submit your work, please follow the directions given in the "Homework Instructions" under the "Reference Material" section on D2L. Make sure to strictly follow these instructions; otherwise you may not receive proper credit.

- **Compilation and execution.** Your submission must include a Makefile, which will create three executables (with names same as the submission source files where .c is replaced with .x). If necessary, please consult the Makefiles provided in the previous assignments.

  Resulting MPI parallel binaries must be executable using srun (for example using 4 processes) as:

  srun –n 4 ./bucket_sort_v1.x 1000000

- **Executing your jobs.** You should develop, compile and test your programs on the dev nodes at HPCC. However, on the dev-nodes there will be several other programs running simultaneously, and your measurements will not be accurate. After you make sure that your program is bug-free and executes correctly on the dev-nodes, the way to get good performance data for different programs and various input sizes is to use the interactive or batch execution modes. Please consult HPCC's wiki pages for details (https://wiki.hpcc.msu.edu/display/ITH/Job+Scheduling+by+SLURM). *Note that jobs may wait in the queue to be executed for a few hours on a busy day, thus plan accordingly and do not wait until the last day.*

- **Batch job script.** Batch jobs are convenient, especially if you would like to collect data for several runs (this may still take a few hours to complete, but at least you do not have to sit in front of the computer). Note that you can execute several runs for your programs with different input values in the same job script – this way you can avoid submitting and tracking several jobs.