CSE 491-Section 2
Parallel Processing

Homework 3
Due in class, November 9, 2015

Important Notes:

- *For your questions about the homework, please use the D2L discussion forum for the class so that answers by the TA, myself (or one of your classmates) can be seen by others.*
- *The performance data for all problems must be collected on HPCC's intel14 cluster.*
- *Note that jobs may wait in the queue to be executed for a few hours on a busy day, thus plan accordingly and start well before the deadline.*
- ***Late submission policy:** You can submit one late homework during the semester. However, there will be a 25% penalty for each late day.*

Parallel Bucket sort with MPI

In this problem, you will be writing code to sort an array of doubles using distributed memory parallelism. Use the bucket sort algorithm and MPI. `N` is the length of array `A` to be sorted, `P` is the number of processes used in the parallel program. The goal of the project is to become acquainted with some of the challenges of writing efficient and deadlock-free message-passing programs.

In the bucket sort algorithm, you will need a fast sequential function for local sorting. For this purpose, you can use the quicksort library function in C, called `qsort` which only requires a comparison function to determine the ordering of elements. You can use the code below to sort doubles in ascending order:

```
// Comparison function used by qsort
int compare_dbls(const void* arg1, const void* arg2)
{
  double a1 = *(double *) arg1;
  double a2 = *(double *) arg2;
  if (a1 < a2) return -1;
  else if (a1 == a2) return 0;
  else return 1;
}

// Sort the array in place
void qsort_dbls(double *array, int array_len)
{
    qsort(array, (size_t)array_len, sizeof(double),
compare_dbls);
}
```

Bucket sort needs to know the range of the data values. To make things simple, we will generate numbers between 0 and 1:

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

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

Part 1

1. The crucial step in bucket sort is binning elements into their corresponding buckets. With p processors, 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.

   You will first implement two different versions of parallel bucket sort, both using MPI.

a) Write bucket_sort_v1.c:
    - **generate:** create the array at the root process (as described above),
    - **bin:** root process determines buckets -- which data belongs on which processor,
    - **distribute:** root sends buckets to appropriate processes,
    - **local sort:** each process sorts the data locally using qsort(),
    - **gather:** finally, results are gathered at the root process.
    - root process should verify the correctness of the result – scan the list from start to end and make sure elements are in increasing order.

b) Write bucket_sort_v2.c:
    - **generate:** create N/P elements randomly on each process (as described above),
    - **bin:** each process determines buckets - which of their data belongs on which processor,
    - **distribute:** each process sends buckets to appropriate processes,
    - **local sort:** each process sorts the data locally using qsort(),
    - **gather:** finally, results are gathered at the root process.
    - root process should verify the correctness of the result.

2. You can compile your code with mpicc:
```
mpicc bucket_sort_v1.c -o bucket_sort_v1
```
3. Report the total execution time (excluding the verification of correctness part), speedup and efficiency of your codes using p = 1, 2, 4, 8, 16 and 32 processors on the intel14 cluster at HPCC. Use problem sizes of N = 10 million, 100 million and 1 billion. The problem size must be read as an argument (see skeleton codes in homework 2 as an example).
4. You can run your executables with mpirun:
```
mpirun -n 32 bucket_sort_v1 10000000
```
5. Time the different phases of your codes (generate, bin, distribute, local sort, gather) using MPI_Wtime:
```
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 parts take the longest? Be sure to vary your problem size so that you can detect any trends.

IMPORTANT: Make sure that you request enough memory in your job scripts! Note that default total memory per job is 750 MBs, 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 they will have in their buckets. To ensure timely completion of your jobs, do not specify unnecessarily long times in your job scripts. A good implementation should be able to sort 1 billion numbers in under 10 minutes even using a single processor. Note that with larger processors, you will ideally need shorter times, i.e. a couple minutes should be more than enough for starting the job, sorting and checking with p = 32.

Part 2

1. In this part, you will investigate what happens if we use data with different distributions. Recall that rand() function produces numbers with a uniform distribution. Now, use the square of each number pulled from a uniform distribution, i.e.

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

2. This new distribution is expected to make the load imbalanced and slow down the sort. Investigate whether this is the case. Set N = 100 million, and vary the number of processors, p = 2, 4, 8, 16, 32. Use your bucket_sort_v2 code from the first part and compare the total execution times with uniform and squared distributions. Report the min, max and average running times per processor in each case in a table such as the one below:

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

3. 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 with sampling. Your code should randomly select S samples from the entire array A, and then choose P-1 pivots from the selection using the following process:
   a. Each process selects S/P elements randomly within its part of array A
   b. S samples are gathered at the root process
   c. root process sorts samples locally using qsort()
   d. root selects and broadcasts pivots = [S_sorted[S/P], S_sorted[2S/P], S_sorted[3S/P] ...]

   Note that the result `pivots` needs to be an array of `P+1` elements – 0 is 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. Report the total execution time (excluding the verification of correctness part), speedup and efficiency of your new code, bucket_sort_v3.c, using p = 1, 2, 4, 8, 16 and 32 processors on the intel14 cluster at HPCC. Use problem sizes of 100 million, and 1 billion.
5. Indicate how much time your new code spends in selecting pivots vs. doing the actual bucket sort. Compare the performance and load balance of bucket_sort_v3 with bucket_sort_v2.

Turning in Your Homework, Write-up and Grading

You will turn in your source codes and output files by emailing them to the TA.

You will also turn in a hardcopy write-up providing clear explanations to the questions in the assignment. Your write-up should include a detailed analysis and discussion of observed performance trends, bottlenecks, your optimizations, etc. Your overall grade will be a combination of the quality of your codes, obtained performance as well as the quality of the write-up.