PA3 – Bucket Sort Documentation

Brian Conway

6 Apr. 2017

CS415

Note: This report is currently incomplete, as I plan to submit this project late for reduced credit.

Bucket Sort - Sequential

This sequential implementation of bucket sort involves taking an unsorted collection of numbers and partitioning them into their corresponding buckets, with their buckets being determined by the specific interval they fall into. Bucket sort requires the values in the collection to be uniformly distributed. After partitioning into the buckets, the buckets are sorted using bubble sort.

Bucket Sort - Parallel

This parallel implementation of bucket sort sort involves the master node taking an unsorted collection of numbers and partitioning them into their corresponding regions based on the number of processes specified. Starting from process 1 and with total number of items being *n*, process *k*'s region will be the first *n / k* numbers. The master will take the last region, and depending on input size and number of processes may have a slightly larger region than the rest. After distributing the regions, each process will put the numbers in their region into one of *n* buckets, with the buckets being determined by which interval the number falls into. Once the numbers in the region are in their proper buckets, each process will send bucket *k* to process *k*. When a process receives one of these buckets, they empty the contents into one big bucket. After all buckets are exchanged, each process sorts their big bucket with bubble sort.

Methodology

The file sequential.cpp is responsible for generating a specified number of random values, bucket sorting them, and outputting statistics about the timing as well as optionally writing the sorted numbers to a file.

The program has one mandatory command line parameter for the number of items to generate, and an optional parameter for if you want the sorted numbers output to a file. If the second command line parameter is "y", then the results are output to the file "sortedSeq.txt".

The program first calls generateNumbers(), a function which c++11 random devices to generate uniformly distributed values from 0 to 999,999. The generator is seeded appropriately so that this sequential version generates the same numbers as the parallel one. Afterwards, the program calls the bucketSort() function, which splits the data into the number of buckets specified by the NUM_BUCKETS constant, then uses the bubbleSort() function to sort each bucket. It times this using the Timer class, and it repeats this process the amount of times specified by the NUM_MEASUREMENTS constant. Afterwards, it calls the calcStatistics() function to get an average of the time taken to bucket sort the numbers and outputs it to the console.

The file parallel.cpp is responsible for generating a specified number of random values, bucket sorting them in parallel, and outputting statistics about the timing as well as optionally writing the sorted numbers to a file.

The program has one mandatory command line parameter for the number of items to generate, and an optional parameter for if you want the sorted numbers output to a file. If the second command line parameter is "y", then the results are output to the file "sortedPar.txt".

The program's master process first calls generateNumbers(), a function which c++11 random devices to generate uniformly distributed values from 0 to 999,999. The generator is seeded appropriately so that this parallel version generates the same numbers as the sequential one.

Afterwards, the master node divides the numbers into their corresponding regions based on the number of processes specified. Starting from process 1 and with total number of items being *n*, process *k*'s region will be the first *n / k* numbers. The master will take the last region, and depending on input size and number of processes may have a slightly larger region than the rest. The master sends the region size as well as the numbers in each region to their corresponding process using MPI_Send(). After everyone has their regions, a barrier is called to ensure all processes are at the same spot. After the barrier, the master starts the timer and everyone takes their region and puts the numbers into one of *n* small buckets, with the buckets being determined by which interval the number falls into. Once the numbers in the region are in their proper buckets, each process will send bucket *k* to process *k*. When a process receives one of these buckets, they empty the contents into one big bucket. After all buckets are exchanged, each process sorts their big bucket with bubble sort. A barrier is called after this to ensure all processes have finished sorting.

Afterwards, the master stops the timer and calls the calcStatistics() function to get an average of the time taken to bucket sort the numbers and outputs it to the console.

Timing is done using the custom Timer class, which has start, stop, and resume functions similar to a stopwatch. Timer makes use of timevals and the gettimeofday() function in order to get an accurate reading of seconds and milliseconds at certain instants. When the elapsed time is to be given, the Timer takes the difference in seconds and milliseconds between when the timer was started and when the timer was stopped. It then combines the seconds and milliseconds reading into a double precision floating point variable.

Results

Runtimes

Speedup

Efficiency

Analysis