Distributed Histogram Creation with MapReduce and MPI

CSC 569

Ian Dunn
Computer Science Department
Cal Poly
San Luis Obispo, California
idunn01@calpoly.edu

Toshi Kuboi
Computer Science Department
Cal Poly
San Luis Obispo, California
tkuboi@calpoly.edu

Mitchell Rosen
Computer Science Department
Cal Poly
San Luis Obispo, California
mwrosen@calpoly.edu

Austin Wylie
Computer Science Department
Cal Poly
San Luis Obispo, California
awylie@calpoly.edu

1. INTRODUCTION

This project familiarized us with MapReduce, MPI, and distributed computing performance tradeoffs by calculating vector sums and histograms on a cluster of five Raspberry Pis. We wrote two programs that, given a pair of input files containing vectors of floating-point numbers, summed those vectors and created histograms for each of them as well as the vector sum.

One program's aim was to distribute the work among the five Pis and compute the results as quickly as possible. The other program made this process fault-tolerant (albeit less efficient) with a different method that allowed one node to be disconnected during its run.

2. PART A: DISTRIBUTED PERFORMANCE

This section describes our non-fault-tolerant programs that distributes histogram creation among the five nodes.

2.1 Program Setup

2.1.1 MR-MPI

Using MPI, this first MR-MPI-based program sends the input vector files residing on the master to the other four nodes as character arrays. Each node then breaks the input vector character arrays into large chunks and maps them.

The map callback tokenizes each large string on whitespace, converts each token into a float, and creates KeyValues with the float's index as the key and the float as the value. The addition happens after the two vectors' MapReduce objects are combined (with MapReduce's add and collate functions), and the results are reduced by summing the values for each index and creating new KeyValues with the index as key and the summed result as value.

Now that each of the three vectors (the third being the summed vector) are available, each one's maximum is found by performing a built-in MapReduce sort on the values and returning the first element of the vector.

Next, histogram counts are calculated with a map function that creates KeyValues with a bin number as key, a collate that combines them, and a reduce that counts the number of (empty or null) values per bin number. Those counts are then scanned into arrays on the master node, and are written to result files (in addition to the summed vector).

2.1.2 MPI

We ran into an issue where our MR-MPI setup wouldn't calculate histograms for input vector sizes 10,000, 100,000, or 10,000,000, so we added another implementation based solely on MPI (without MapReduce) that performs the same task.

This version of the program has the master break the input vectors into equal-sized chunks and send them to slaves, which convert their two vector portions into floats and sum them. The slaves then return those sums to the master, which creates the histogram.

Table 1: MR-MPI Histogram Creation Performance

Input Size	Time (s)
10	2.157
1,000,000	95.231

Table 2: MPI Histogram Creation Performance

Input Size	Time (s)
10,000	1.990
100,000	3.525
1,000,000	17.494
10,000,000	170.553

2.2 Performance Analysis

This section includes the performance information for our three implementations as well as the outputted program results (shown in Figures 3 through 10).

2.2.1 MR-MPI

The MR-MPI implementation runtimes are shown in Table 1 and Figure 1.

2.2.2 MPI

The MPI implementation runtimes are shown in Table 2 and Figure 2.

3. PART B: FAULT-TOLERANCE

This section describes our fault-tolerant program that distributes histogram creation among the five nodes.

3.1 Program Setup

The fault-tolerant program using MPI first has the master read and parse the input files. The master forks one process and opens one pipe for every slave node. Each extra process sends a single pair of floats to its respective node, and the slave node adds the floats and sends back the sum. The extra process waits for the node response indefinitely and sends back the result through the pipe (if such response comes); it then calls exit(0) to avoid MPI_Finalize.

The master computes the sum for all vector elements and waits five seconds for extra processes to catch up. This wait is performed using select() and may be interrupted by an MPI_Recv returning. Master continually retries to wait until either the wait times equals the number of processors or the wait is successful.

The master then performs a 0.5-second select on each child's pipe to see if anything is ready to read. If a read is possible, the result is read and compared to the master's calculation. In the event of a mismatch, an error is reported and the slave's version is accepted.

Last, master sends SIGKILL to all children, makes histogram and writes output files, sends SIGKTERM to its parent, and sends SIGKILL to itself.

3.2 Performance Analysis

Just as a point of reference, on average, this fault-tolerant implementation could complete the 1,000,000-element test in 42 seconds (most of which is spent in file reading and writing). A node death adds approximately 0.5 seconds.

Figure 1: Performance with MR-MPI

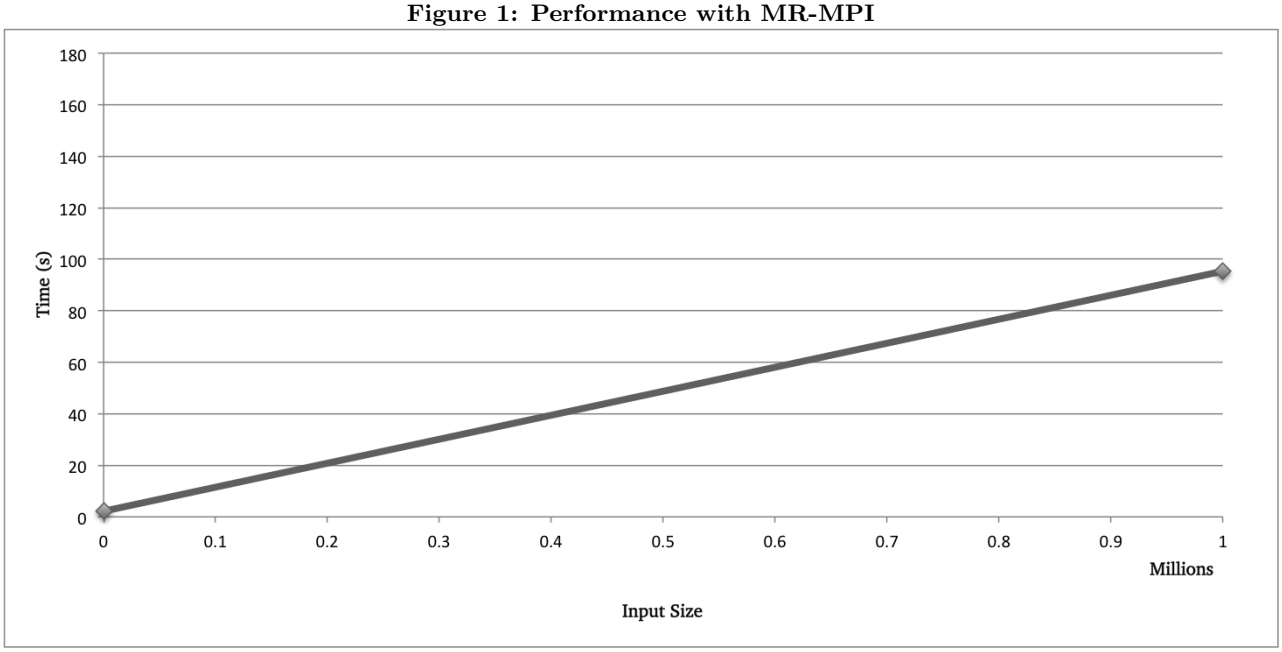

Figure 2: Performance with MPI

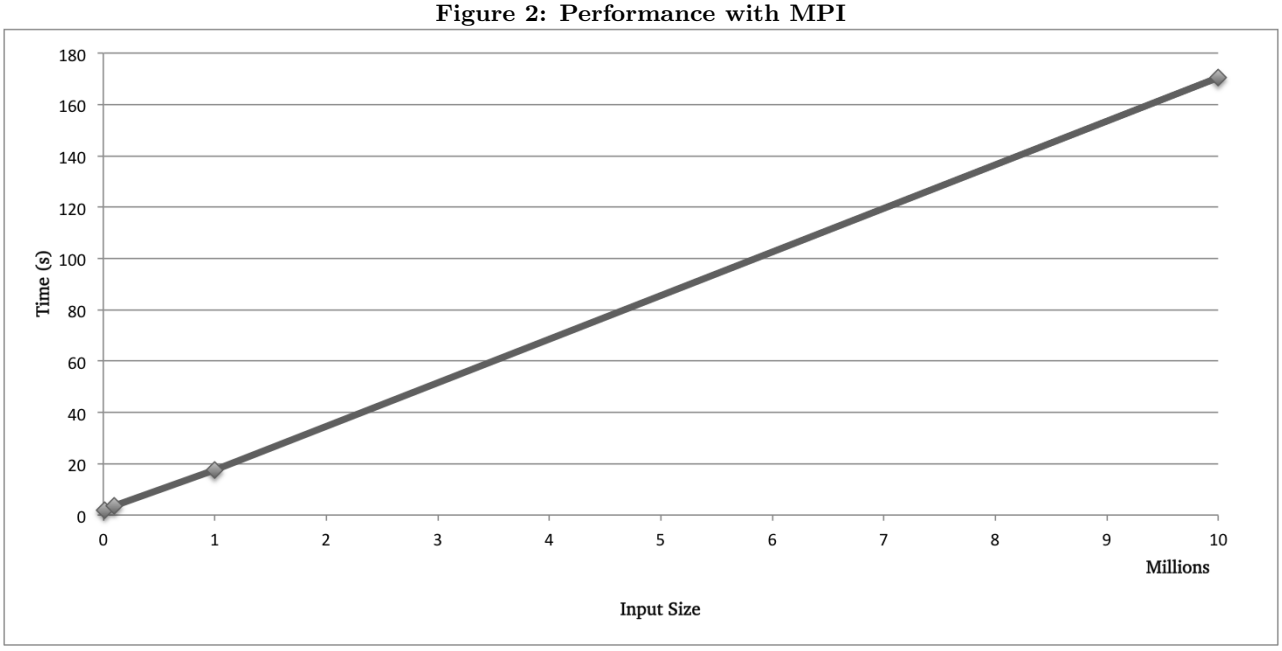

Figure 3: Histogram for 10,000-Count Vectors

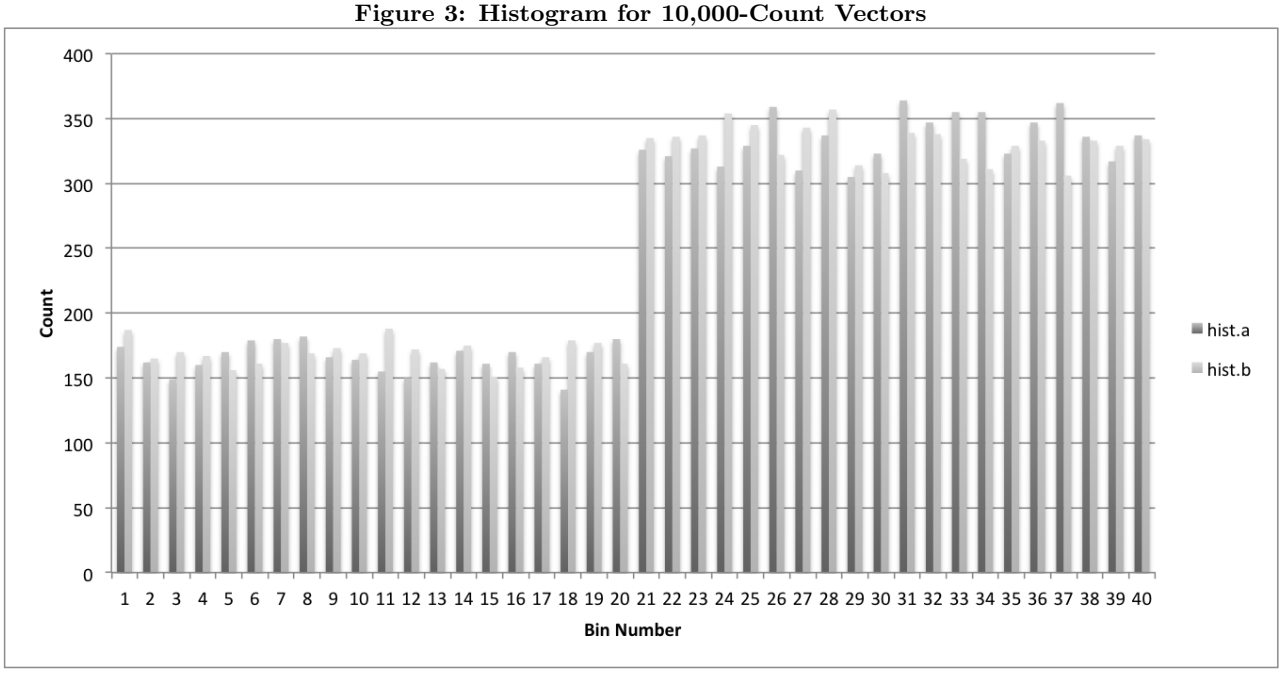

Figure 4: Histogram for 100,000-Count Vectors

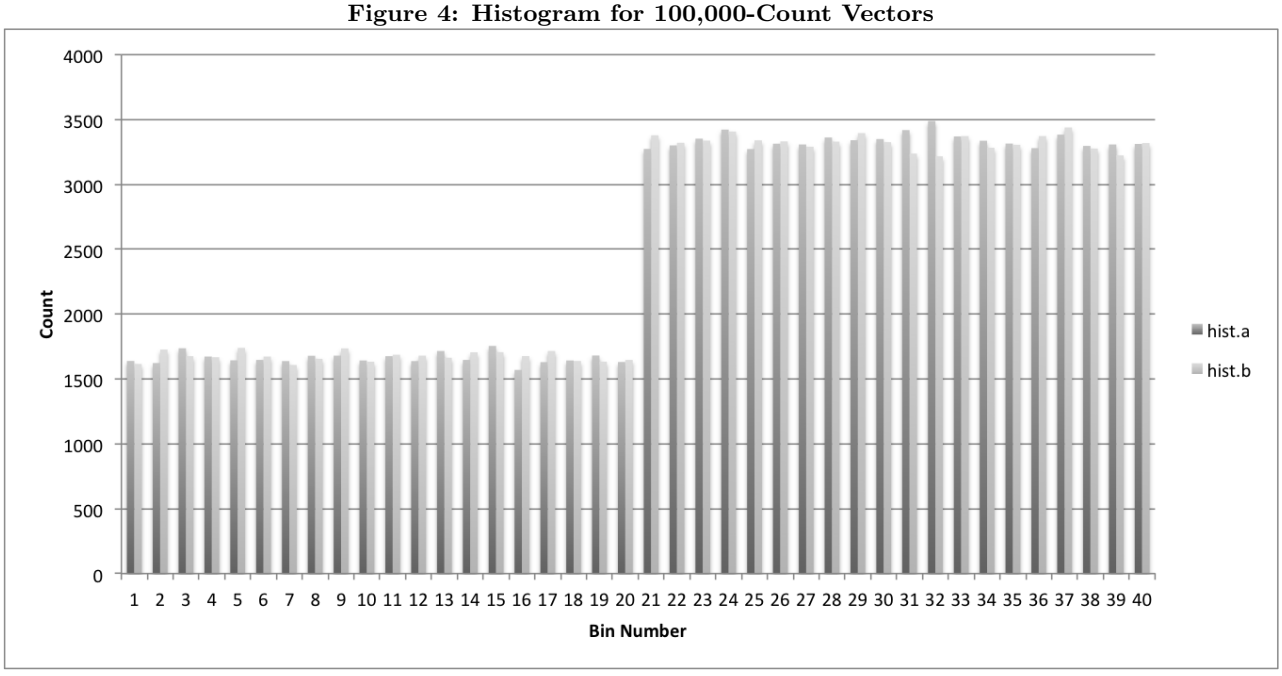

Figure 5: Histogram for 1,000,000-Count Vectors

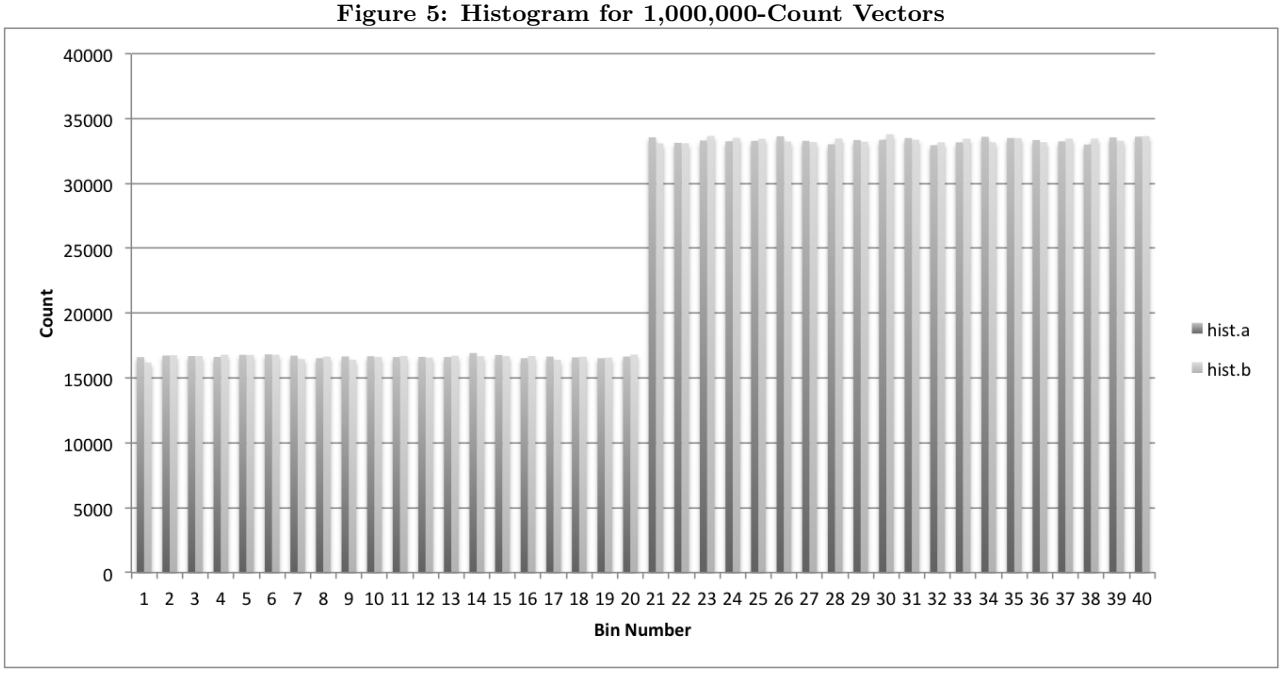

Figure 6: Histogram for 10,000,000-Count Vectors

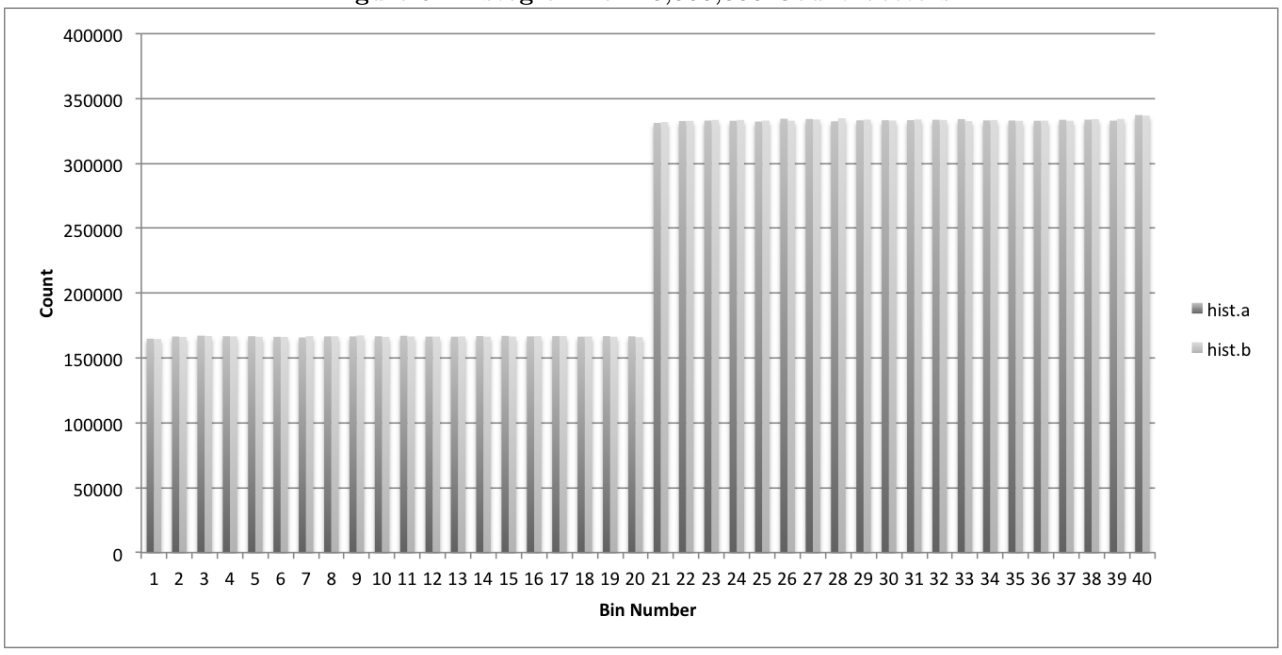

Figure 7: Histogram for 10,000-Count Vector Sum

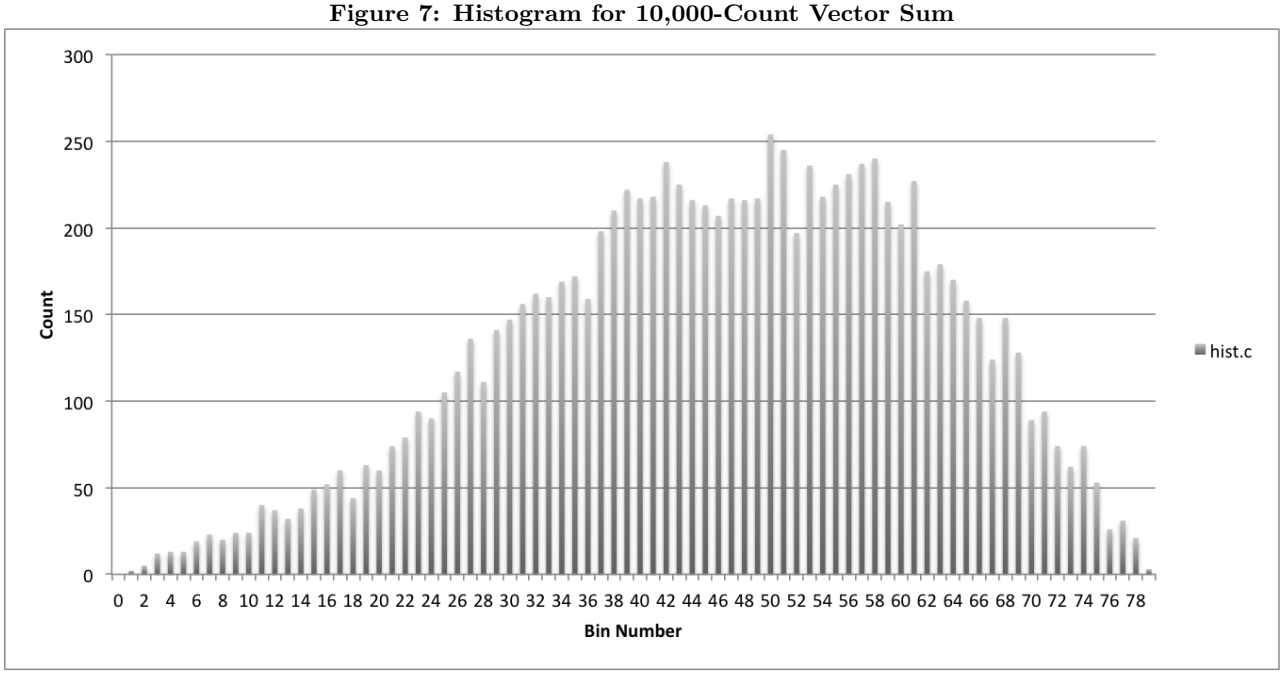

Figure 8: Histogram for 100,000-Count Vector Sum

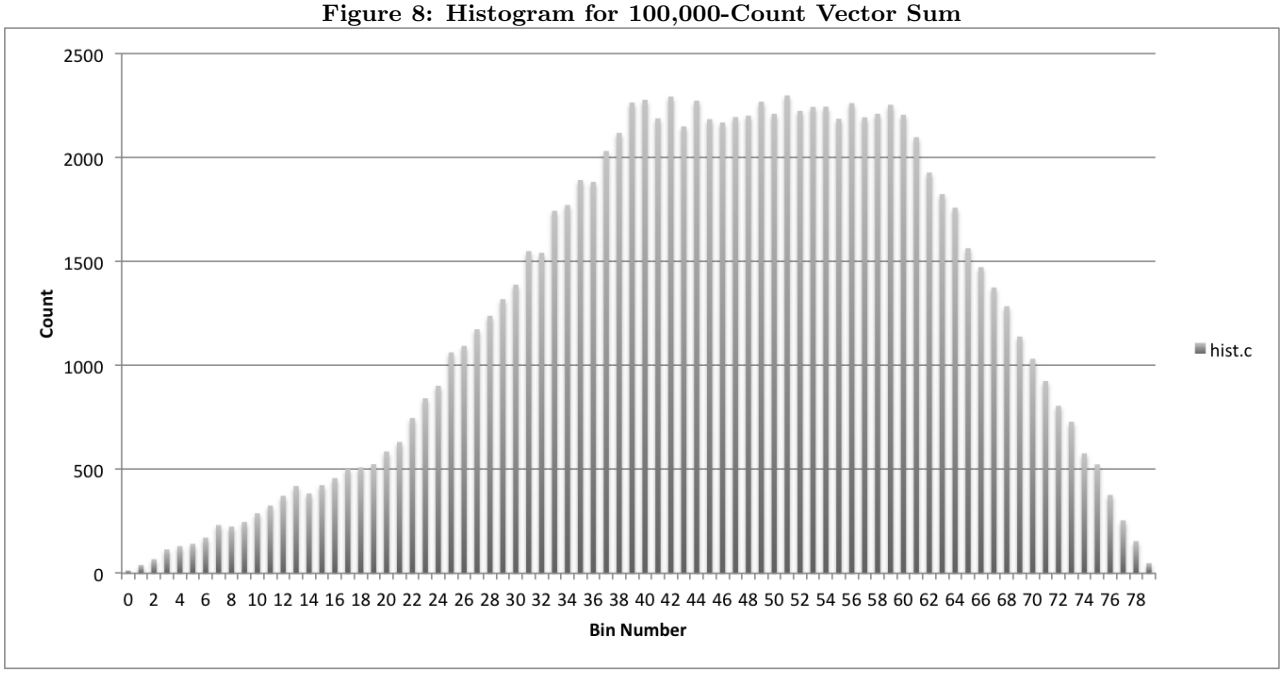

Figure 9: Histogram for 1,000,000-Count Vector Sum

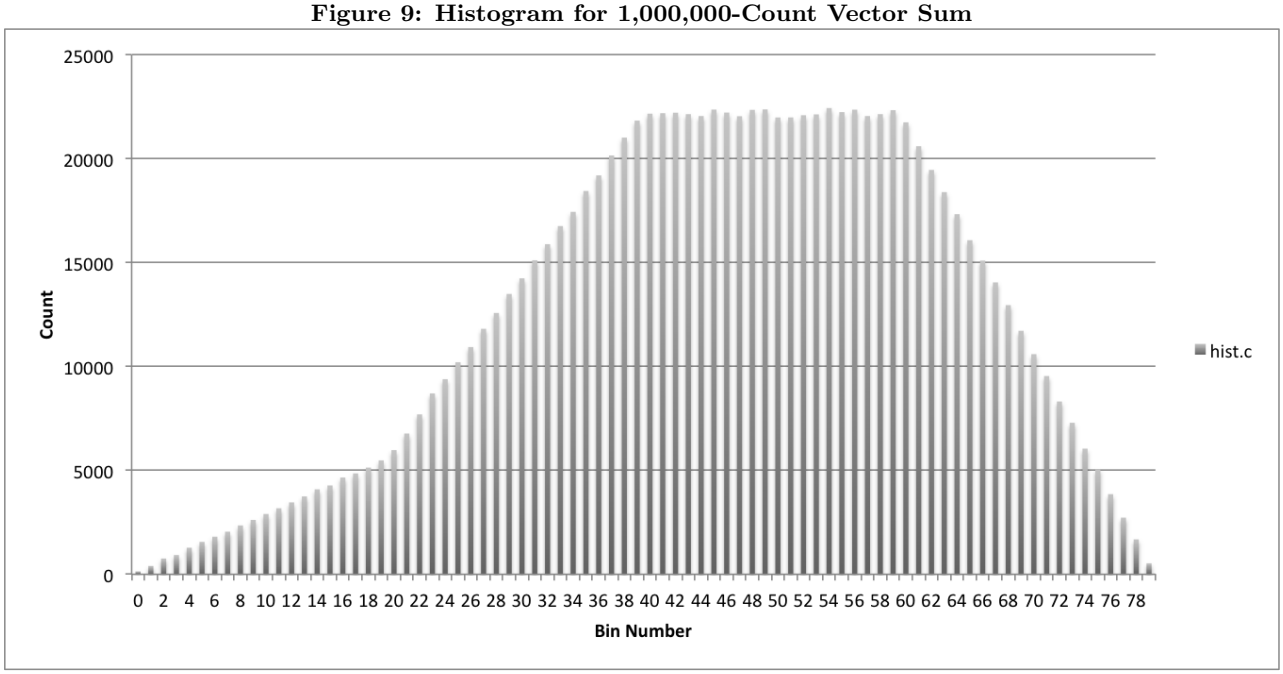

Figure 10: Histogram for 10,000,000-Count Vector Sum