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 program that distributes histogram creation among the five nodes.

2.1 Program Setup

Using MPI, this first 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 vector's MapReduce objects are combined (with MaprReduce'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).

Table 1: Histogram Creation Performance

Input Size	Time
10	Time
10,000	Time
100,000	Time
1,000,000	Time
10,000,000	Time

2.2 Performance Analysis

Performance analysis information.

3. PART B: FAULT-TOLERANCE

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

3.1 Program Setup

Setup description.

3.2 Performance Analysis

Performance analysis information.

Figure 1: Histogram for 10-Count Vectors

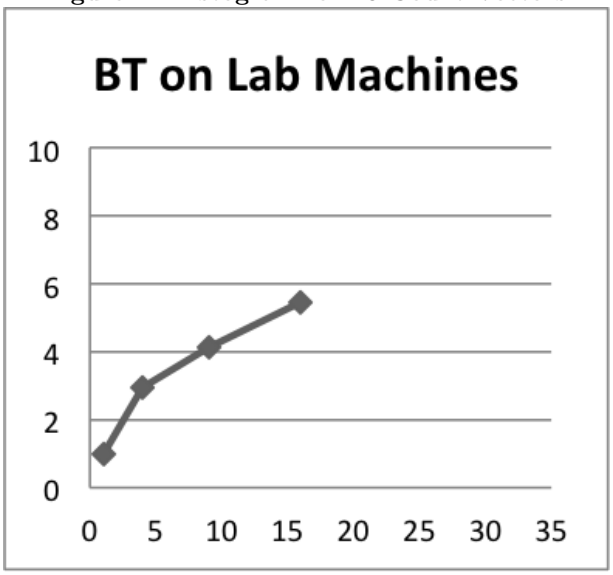

Figure 2: Histogram for 10,000-Count Vectors

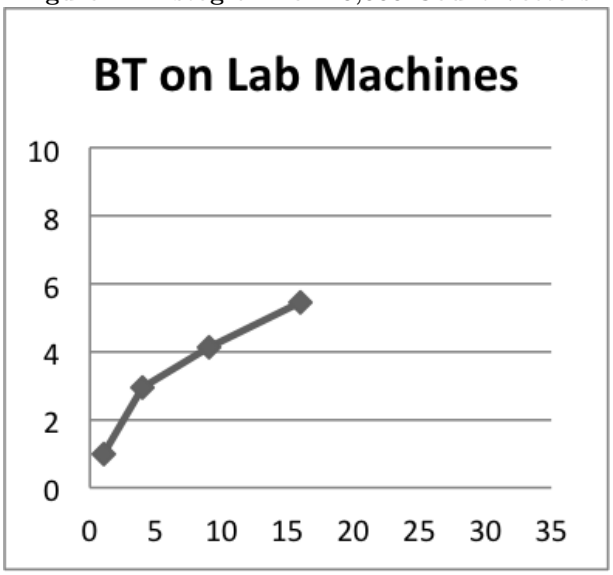

Figure 3: Histogram for 100,000-Count Vectors

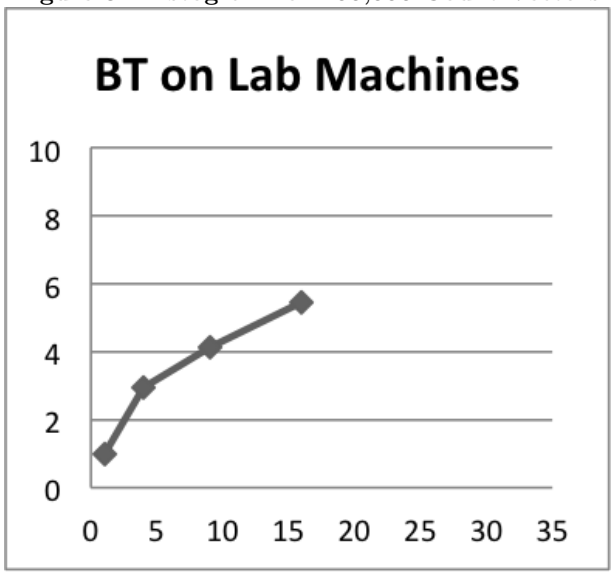

Figure 4: Histogram for 1,000,000-Count Vectors

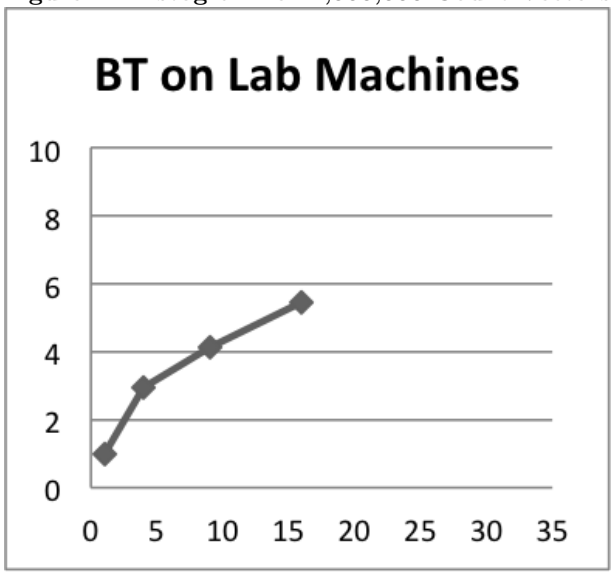

Figure 5: Histogram for 10,000,000-Count Vectors