Large-Scale Merging of Histograms using Distributed In-Memory Computing

Jakob Blomer and Gerardo Ganis

CERN, Geneva, Switzerland

E-mail: `jblomer@cern.ch`

Abstract. Most high-energy physics analysis jobs are embarrassingly parallel except for the final merging of the output objects, which are typically histograms. Currently, the merging of output histograms scales badly. The running time for distributed merging depends not only on the overall number of bins but also on the number partial histogram output files. That means, while the time to analyze data decreases linearly with the number of worker nodes, the time to merge the histograms in fact increases with the number of worker nodes. On the grid, merging jobs that take a few hours are not unusual. In order to improve the situation, we present a distributed and decentral merging algorithm whose running time is independent of the number of worker nodes. We exploit full bisection bandwidth of local networks and we keep all intermediate results in memory. We present benchmarks from an implementation using the parallel ROOT facility (PROOF) and RAMCloud, a distributed key-value store that keeps all data in DRAM.

1. Introduction

While computer hardware systems and micro chips provide ever-growing degrees of parallelism, it is increasingly difficult to develop software systems that exploit the available peak performance from the beginning to the end of a computing workflow. The problem that even small sequential parts can dominate the running time of an otherwise perfectly parallel workflow is, for instance, described by the well-known *Amdahl law*. We often associate Amdahl's law with the programming of multi-threaded applications but similar effects also arise when we scale up the number of worker nodes that participate in the execution of a distributed workflow.

The merging of output histograms at the end of a data analysis job can be a particular tough case. Listing 1 shows pseudo code for a typical analysis workflow. The analysis itself is usually very well parallelizable through independent tasks that work on different subsets of the input events; thus the analysis step often shows a linear speedup even up to a large number of thousands of worker nodes. The merging of the partial output histograms created on every worker node, on the other hand, shows the reverse scaling behavior. The more worker nodes we add, the more data we need to merge into the final set of output histograms. Unlike the analysis itself, the size of the merging problem is not a function of the amount of input data but it is given by the overall number of bins that get filled multiplied by the number of worker nodes.

In extreme (but real) cases the analysis might be performed in a few minutes while the merging of the output histograms takes hours. In such cases, the size of the output histograms can be on the order of gigabytes, which every worker nodes sends to a single or a few nodes responsible for the merging. Large turn-around times prevent users from previewing analysis results, for instance in a phase when the analysis task is still being developed. They also inhibit use cases that require fast response times, for instance the detector calibration necessary for a first reconstruction pass.

In this contribution, we will present an approach that is symmetric with respect to sending data and retrieving data: the merging phase will involve *each and every* worker node in the task of merging. Thus the time to merge histograms becomes largely independent of the number of worker nodes. The more worker nodes, the more resources contribute to the final merging phase. This assumes that the network itself does not become a bottleneck but that all nodes can send and receive data at high speed at the same time (the network has a high *bisection bandwidth*). As a means to store and merge histograms in a distributed manner, we will use a slightly modified version of RAMCloud [1, 2], an open source, fast, and distributed key-value store that keeps all data in DRAM.

Listing 1. A typical data analysis workflow.

```
for all events do
  for all jets do
    for all particles with properties do
      value = Correlation(...)
      FillHistogram(histogram, value)
      ...

for all histograms
  Merge(histogram)
```

The remainder of the paper is structured as follows: Section 2 describes our distributed merging algorithm. Section 3 describes the implementation details. Section 4 presents the benchmark results based on a large, real-world use case. Sections 5 and 6 discuss related work and summarize the contribution.

2. A peer-to-peer approach to merging

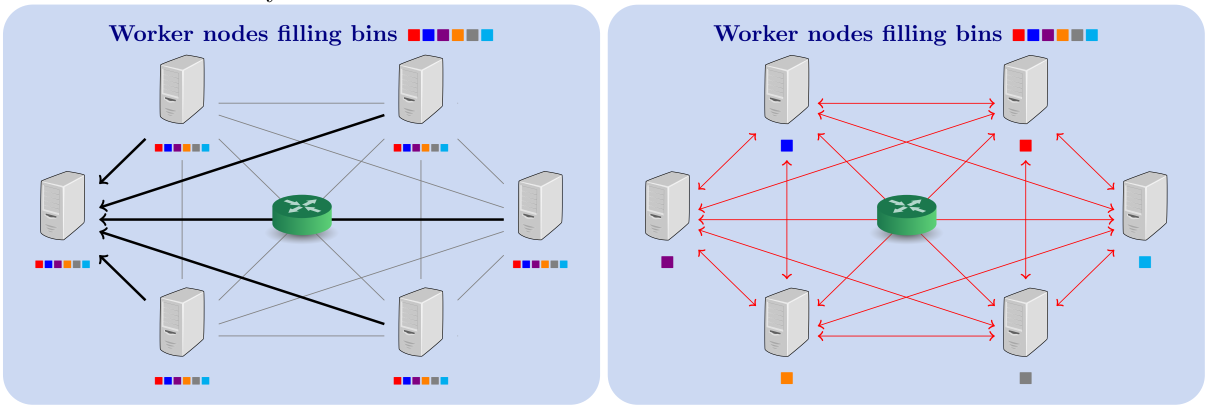
At the end of the data analysis step, we are in a situation in which every worker node has created partial sums for the bins of all the output histograms. With a single or only a few mergers, the merging is asymmetric and the merger(s) become a bottleneck as the number of worker nodes increases. Instead, we can make every worker node responsible for a small subset of the final set of output histograms.

To do so, we represent every histogram as key-value pairs in a hash map. This can be done for all histograms, even multi-dimensional and sparse ones, provided that the binning is known upfront. With a known binning, every bin can be assigned a *global bin number*, simply by unrolling the dimensions of the histogram. A unique key for every bin in a set of output histograms can then be constructed by concatenating a histogram identifier and the global bin number. The value of the key-value pair consists of two floating point numbers representing the bin value and the bin error.

Having the output histograms represented as a hash map makes it easy to distribute them uniformly to worker nodes using a distributed hash table (DHT). Thus, at the end of the data analysis, we span a distributed hash table among the worker nodes. All the n worker nodes send the bin contents of their partially filled histograms to the other worker nodes, using the bin partitioning imposed by the DHT. At the same time, worker nodes receive bin contents from all other worker nodes for $\approx 1/n$ of all the available bins. Figure 1 compares the naive approach and the peer-to-peer approach to merging.

For performance reasons, the DHT implementation not only has to be capable of storing and retrieving key-value pairs, but also of the summation of bin contents. That allows for sending batches of bin contents to other worker nodes. Otherwise the addition of every bin would require an atomic `get value` − `modify value` − `store value` cycle between worker nodes. For example,

Figure 1. Naive merging with a single merger compared to a peer-to-peer approach to merging. Bins are visualized by colored boxes.



worker node α might send $(42, (v, w))$—the value v and the sum of squares of weights w (which is used to calculate the bin error) for bin number 42 to worker node β for merging. At worker node β, there is already an entry $(42, (v', w'))$. Worker node β will then store $(42, (v' + v, w' + w))$.

In a last step, all the merged bins on all the worker nodes are enumerated to read out the histograms into a single file. This is different from having just a single merger node in the first place, because the readout size after merging is already reduced. For instance, if 10 worker nodes produce a 100 MB set of output histograms each, the final readout needs to read some 10 MB from each worker node and not 100 MB as in the case of a single merger node.

Note that this merging algorithm would represent the "shuffle-exchange" step in a MapReduce formulation of the problem [3], provided that every worker node would run a so-called *reduce task*.

3. Implementation

As a distributed key-value store used to merge the bins on the worker nodes, we use a slightly modified version of the RAMCloud open source system. RAMCloud keeps all data in memory and provides exceptionally small latencies down to 5 μs per request over InfiniBand. If requests are batched, RAMCloud's remote procedure call implementation sustains hundreds of thousands of requests per second and node over 1 GbE TCP/IP networks.

RAMCloud stores *objects*, which are key-value pairs. Keys typically are a few bytes in size, and values are typically a few hundred bytes in size. Objects are grouped in *tables*. A table can act as a DHT, i. e. the key-value pairs of a table can be uniformly distributed over an arbitrary number of nodes. We use a single table to store all the bins of all the output histograms of an analysis workflow, as described in Section 2.

From the point of view of deployment, RAMCloud compiles from C++ sources into a few binaries. In order to run a cluster, there needs to be a *coordinator* process that maintains the cluster membership and the table definitions. All the nodes run a *master* process that manages a share of DRAM on the node and that processes storage and retrieval requests from other nodes. In a fault-tolerant setup, every node also runs a *backup* process for data replication and recovery tasks. In our use case we omit the backup processes.

RAMCloud comes with a client library and several language bindings (C, C++, Python, Java). The client library can be used to connect to a cluster, create and delete tables, and to store and retrieve objects. The internals of the network processing, selection of the master node according to the object's key, batching of requests, and so on are hidden from the user.

3.1. Customizations

Our own additions and modifications needed for a proof-of-concept are modest. In addition to the standard requests (put, get, enumerate, . . .), we added a type of request specifically for the merging of the bin content and the bin error. Instead of just storing the value of an object, this request type interprets the object's value as floating point numbers and, if an object with the same key already exists, atomically sums up the values. For better performance, we increased maximum number of objects in a batched request from the default value to 2000. Using the client library, we implemented two small utilities: the first utility reads a ROOT file [4, 5], extracts all the histograms, and merges their content on RAMCloud. The second utility reads out the merged bins by enumerating all the objects in a table.

4. Experimental Setup and Benchmarks

As a real-world benchmark, we were kindly provided with a 175 MB ROOT file which contains the output histograms from an ALICE data quality assurance task. This (merged) ROOT file comprises some 12 000 one dimensional, two dimensional, and three dimensional histograms with around 19 million filled (non-zero) bins. Creating this file on the grid with a few thousand job slots involves multiple merge phases, and depending on the usage of the grid these merge phase can sum up to a couple of hours.

Here, we measure the merging time as a function of the number of worker nodes. To do so, we pre-place the ROOT file in memory on all the benchmark nodes[1]. We compare the time it takes to read the ROOT file, merge the results in RAMCloud, and read-out the merged bins with the time it takes to merge the histograms with PROOF, the parallel ROOT facility [6, 7]. In both cases, we are only interested in the difference when adding whole nodes; we assume that for nodes providing multiple analysis slots, a local merging step on the results has already been done. Such a local merging step can be done in less than a minute.

4.1. Benchmark Cluster

The benchmark cluster comprises 29 machines that are connected by a 1 GbE TCP/IP network. The switching capacity of the Ethernet switch connecting the nodes is 10 Gbit. The machines have 8 core Xeon E5450 CPUs at 3 GHz and 16 GB RAM each. They run Scientific Linux 6. We used ROOT/PROOF version 5.34/26 and our slightly modified version of RAMCloud as available in the source code repository by the end of March 2015.

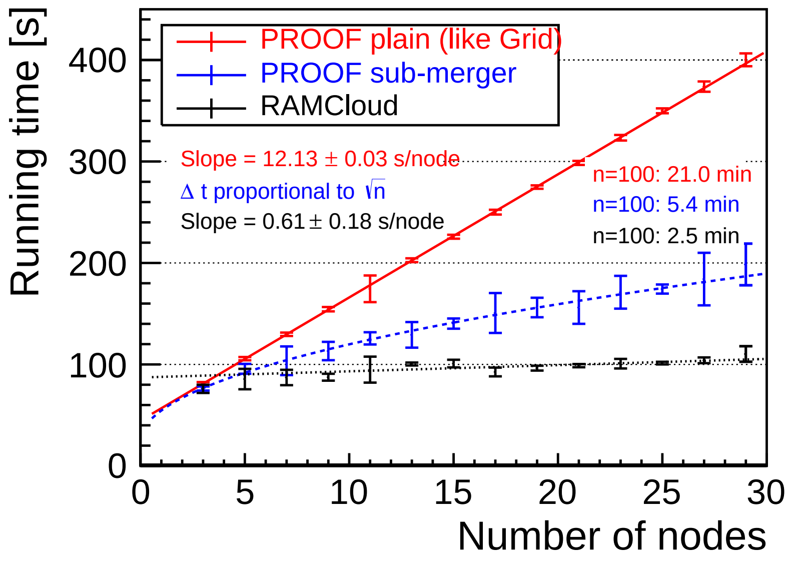
4.2. Results

Figure 2 shows the scaling of the merging task as a function of the number of nodes. PROOF is benchmarked in two modes. A "plain" mode with a single merger represents many practical setups, including the grid scenario. The second mode uses a feature called "sub mergers" that uses two phases for merging with $\sqrt{n}$ worker nodes as mergers in the first phase. Sub mergers were introduced in PROOF in 2011 with the aim to speed-up merging when the output is composed by a large number of objects whose size does not depend on the number of entries or processing time (e. g. for histograms).

As expected, the scaling with a single merger is linear in the number of worker nodes. One can extrapolate that the merging time reaches the order of hours as the number of worker nodes reaches the order of thousands. In the smarter merging mode with sub mergers, the merging times scales proportionally to $\sqrt{n}$. The merging time for a large number of worker nodes is largely reduced and extrapolates to some 15 min for 1 000 worker nodes.

[1] Merging these identical files which are already the final result of a merging process does not make sense from the physics point of view, but for the sake of the benchmark this file represents an upper bound for the size (in number of non-zero bins) of the output histograms that could possibly be produced during the real quality assurance task.

Figure 2. Scaling of the merging task as a function of the number of nodes. All PROOF numbers include 15 s transfer time of results from the master node to the client.



The scaling behavior of the merging with RAMCloud is almost flat. Running times are shorter than in the two other cases. Extrapolated to 1 000 nodes the merging time is around 10 min. It should be noted that at this scale scaling issues of the underlying Ethernet and TCP/IP stack could become relevant.

Figure 3 explains in part the remaining small dependence on the number of worker nodes in the RAMCloud results. The overall merging time is broken down into the actual merging time and the time to readout the bins from all the worker nodes. It turns out that most of the linear contribution comes from the read-out. This is due to the fact that the RAMCloud table enumeration queries master nodes one after the other. This does not seem to be a fundamental issue: one could imagine full table enumeration that queries all master nodes in parallel. Assuming that the linear contribution to the scaling coming from the read-out phase could be removed, an overall merging time with 1 000 worker nodes of under 2 minutes seems reachable.

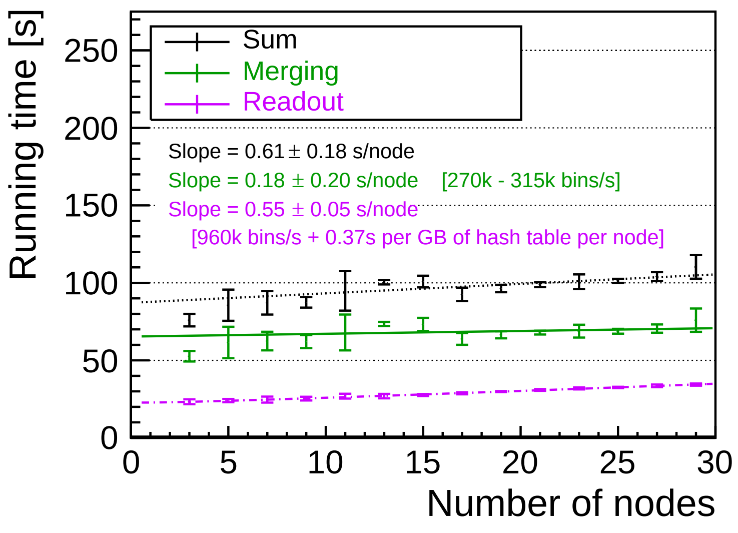
5. Related Work

While histograms are in general widely used to analyze large data sets ("BigData"), in high energy physics histograms are specifically ubiquitous. The HEP community has high demands on the features provided by a histogram implementation and on the ability to handle a large number of histograms with many bins.

The Hadoop [8] MapReduce implementation has a simple implementation of histograms but is lacking support for multi-dimensional histograms or errors of bins. While the merging of histograms is already available, it would require to port the entire computing workflow on the Hadoop platform.

The HistogramTools package for R [9] supports serialization of histograms which facilitates using them in a distributed environment such as MapReduce. However, also the HistogramTools package supports only one dimensional histograms with integer values and without bin errors.

Figure 3. Detailed view on the RAMCloud scaling.



6. Conclusion
We have shown a proof-of-concept implementation of histogram merging whose merging time is almost constant with a growing number of worker nodes. Our benchmark results suggest that there is significant room for speed-up when merging large histograms from many worker nodes. As a follow-up to this work, we will investigate how this approach can be integrated in the existing HEP tool chain, for instance in PROOF. It would be interesting to see if a similar method can be used for merging ROOT trees, for instance by distributing branches to worker nodes.

Acknowledgments
We would like to thank John Ousterhout and the RAMCloud team at Stanford University for hosting one of the authors and for numerous suggestions and stimulating discussions. We would like to thank Mihaela Gheata from the ALICE experiment for kindly providing us the output histograms of a data quality assurance task.

References
[1] Ousterhout J, Agrawal P, Erickson D, Kozyrakis C, Leverich J, Mazières D, Mitra S, Narayanan A, Ongaro D, Parulkar G, Rosenblum M, Rumble S M, Stratmann E and Stutsman R 2011 *Communications of the ACM* **54** 121–130
[2] RAMCloud `https://ramcloud.stanford.edu/`
[3] Dean J and Ghemawa S 2008 *Communications of the ACM* **51** 107–114
[4] Brun R and Rademakers F 1997 *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment A* **389** 81–86
[5] Naumann A, Brun R *et al.* 2009 *Computer Physics Communications* **180** 2499–2512
[6] ROOT: A C++ framework for petabyte data storage s a, visualization I Antcheva, Ballintijn M, Bellenot B, Biskup M, Brun R, Buncic N, Canal P, Casadei D, Couet O, Fine V, Franco L, Ganis G, Gheata A, Maline D G, Goto M, Iwaszkiewicz J, Kreshuk A, Segura D M, Maunder R, Moneta L, Naumann A, Offermann E, Onuchin V, Panacek S, Rademakers F, Russo P and Tadel M 2011 *Computer Physics Communications* **182** 1384–1385
[7] PROOF `https://root.cern.ch/drupal/content/proof`
[8] White T 2009 *Hadoop: The Definitive Guide.* (O'Reilly)
[9] 2014 R HistogramTools package `http://cran.r-project.org/web/packages/HistogramTools/`