Hone: "Scaling Down" Hadoop on Shared-Memory Systems

K. Ashwin Kumar, Jonathan Gluck, Amol Deshpande, and Jimmy Lin
University of Maryland, College Park, USA
{ashwin, jdg, amol}@cs.umd.edu, jimmylin@umd.edu

ABSTRACT

The underlying assumption behind Hadoop and, more generally, the need for distributed processing is that the data to be analyzed cannot be held in memory on a single machine. Today, this assumption needs to be re-evaluated. Although petabyte-scale datastores are increasingly common, it is unclear whether "typical" analytics tasks require more than a single high-end server. Additionally, we are seeing increased sophistication in analytics, e.g., machine learning, which generally operates over smaller and more refined datasets. To address these trends, we propose "scaling down" Hadoop to run on shared-memory machines. This paper presents a prototype runtime called Hone, intended to be both API and binary compatible with standard (distributed) Hadoop. That is, Hone can take an existing Hadoop jar and efficiently execute it, without modification, on a multi-core shared memory machine. This allows us to take existing Hadoop algorithms and find the most suitable runtime environment for execution on datasets of varying sizes. Our experiments show that Hone can be an order of magnitude faster than Hadoop pseudo-distributed mode (PDM); on dataset sizes that fit into memory, Hone can outperform a fully-distributed 15-node Hadoop cluster in some cases as well.

1. INTRODUCTION

The Hadoop implementation of MapReduce [3] has become the tool of choice for "big data" processing (whether directly, or indirectly via higher-level tools such as Pig or Hive). Among its advantages are the ability to horizontally scale to petabytes of data on thousands of commodity servers, easy-to-understand programming semantics, and a high degree of fault tolerance. There has been a wealth of activity in applying Hadoop to problems in data management as well as data mining and machine learning; the community has learned much about how to recast algorithms in terms of the restrictive primitives *map* and *reduce*.

Computing environments have evolved substantially since the inception of Hadoop. For example, in 2008, a Hadoop node might have two single-core processors with a total of 4 GB of RAM. Today, a single high-end commodity server might have two hex-core processors and 256 GB of RAM—such a server can be purchased for less than $10000 USD. This means that a single server today has more cores and more memory than a small Hadoop cluster from a few years ago. The assumption behind Hadoop and the need for distributed processing is that the data to be analyzed cannot be held in memory on a single machine. Today, this assumption needs to be re-evaluated.

Although it is true that petabyte-scale datastores are becoming increasingly common, it is unclear whether datasets used in "typical" analytics tasks today are really too large to fit in RAM on a single server. Of course, organizations such as Yahoo, Facebook, and Twitter routinely run Pig or Hive jobs that scan terabytes of log data, but these organizations should be considered outliers—they are not representative of data analytics in most enterprise or academic settings. Even still, according to the analysis of Rowstron et al. [7], at least two analytics production clusters (at Microsoft and Yahoo) have median job input sizes under 14 GB, and 90% of jobs on a Facebook cluster have input sizes under 100 GB. Given these numbers, holding all data in memory doesn't seem too far-fetched. Furthermore, over the past several years, the sophistication of data analytics has grown substantially. Whereas yesterday the community was focused on relatively simple tasks such as natural joins and aggregations, there is an increasing trend toward data mining and machine learning. Such algorithms usually operate on more refined, and hence, smaller datasets (e.g., sparse feature vectors)—often in the range of tens of gigabytes.

These factors suggest that it is worthwhile to consider in-memory data analytics on modern servers—but it still leaves open the question of how we would orchestrate computations on large multi-core, shared-memory machines. Do we go back to multi-threaded programming? That seems like a bad idea because we embraced the simplicity of MapReduce for good reason—the complexity of concurrent programming with threads is well known. The solution, we propose, is to "scale down" Hadoop to run on shared-memory machines. In this paper, we present a prototype runtime called Hone ("Hadoop One") that is intended to be both API *and* binary compatible with Hadoop. That is, we can take an existing Hadoop jar and efficiently execute it, without modification, on a multi-core shared memory machine using Hone. This allows us to take an implemented algorithm and find the most suitable runtime environment for execution on datasets of varying sizes—if the data fit into memory, we can avoid network latency and significantly decrease execution time in a shared-memory environment.

2. RELATED WORK

API and binary compatibility with Hadoop is the central tenant in our design. Although there have previously been alternative MapReduce implementations for shared-memory machines [6, 10, 5, 2, 4, 9], taking advantage of them would require porting Hadoop

Proceedings of the VLDB Endowment, Vol. 6, No. 12

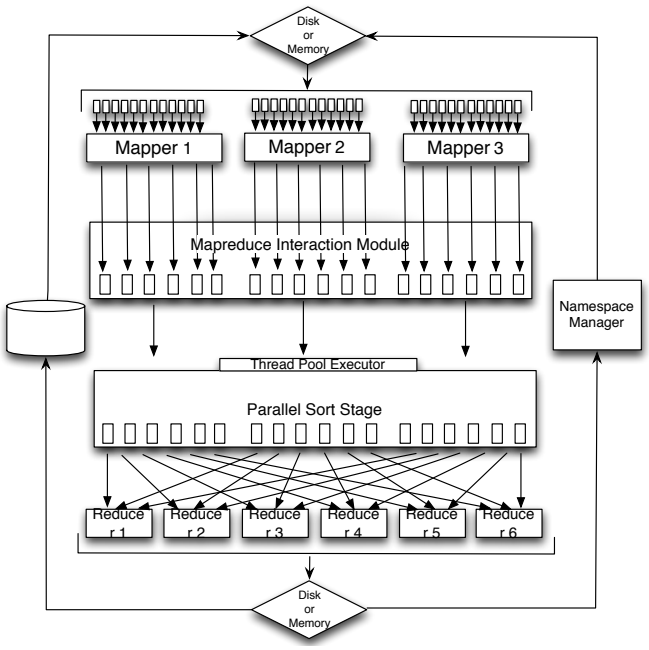

Figure 1: Hone system architecture

code over to another custom API. In contrast, Hone is able to leverage existing implementations—in this paper, we present experiments on a number of "standard" MapReduce algorithms (word count, PageRank, etc.) as well as a Hadoop-based implementation of Latent Dirichlet Allocation (LDA) [12], comparing the performance of Hone on a single shared-memory machine with a 15-node Hadoop cluster. This LDA implementation was itself a major research effort, and demonstrates API and binary compatibility on "non-toy" code.

Closest to our system is the work of Shinnar et al. [8] who proposed a system called M3R. However, they mainly focus on scale-out architectures, whereas we focus explicitly on scale-up on a single shared-memory machine. M3R is implemented in the X10 language: although X10 interoperates with Java, the M3R system introduces an additional layer of indirection that we feel is unnecessary. Similarly, Spark [11] is specifically optimized for scale-out architectures and provides limited multi-core scalability on a single machine. Moreover, it is not Hadoop API compatible and thus represents a different point in the design space.

3. SYSTEM ARCHITECTURE

We describe the Hone system architecture, shown in Figure 1:

Map stage: Analogous to Hadoop, this stage applies the map function on the input dataset to emit intermediate (key, value) pairs. Each mapper is handled by a separate thread, which consumes the supplied InputSplit and processes input records according to the user-specified InputFormat. As with Hadoop, the total number of mappers is determined by the number of input splits. This stage uses a standard thread-pooling technique to control the number of parallel mapper tasks. Mappers in Hone accept input either from disk or from a 'namespace' residing in memory (more below).

Sort stage: Hone sorts intermediate (key, value) pairs emitted by the mappers per the standard contract defined by the MapReduce model. Sorting is handled by a separate thread pool with a built-in load balancer. If the sort streams grow too large then an automatic splitter determines the optimal split size, efficiently splits the streams on the fly, and performs parallel sorting on the split streams. This splitting information is passed on to the reduce stage so that proper stream assignment is performed on the reducers. One can also specify the stream split size in the configuration.

Reduce stage: A reducer in Hone applies a reduce function on intermediate (key, value) pairs emitted by mappers. Depending on how mappers interact with reducers as discussed below, reducers may have to apply the partitioning function on the key to gather the appropriate (key, value) pairs. A reducer either writes output to disk or it can store output in memory for further iterative processing.

MapReduce interaction module: Running MapReduce on multi-core shared memory machines creates interesting possibilities in the way mappers interact with reducers. For example, one option is that each mapper emits intermediate (key, value) pairs in a corresponding output stream, and each reducer iterates through the stream corresponding to each mapper and ingests (key, value) pairs as determined by the partitioner. In another possibility, each mapper emits intermediate pairs into r (number of reducers) streams by applying the partitioner to every emitted (key, value) pair. In this case, each reducer only needs to access a single stream. Hone provides a flexible way of controlling these interactions through a user-specified option, by providing three different MapReduce interaction models: *pull-based*, *push-based* and *hybrid*. In the pull-based approach, each mapper emits keys into r streams, where r is the number of reducers. Each mapper applies the partitioning function to assign (key, value) pairs to one of the r corresponding streams. If m is the total number of mappers then there will be a total of $m \times r$ intermediate streams. In the sort stage, these $m \times r$ intermediate streams are sorted in parallel. In the reduce stage, each reducer thread "pulls" the appropriate (key, value) pairs from the appropriate r streams. In the push-based approach, there are only r intermediate streams, each corresponding to a reducer. Each mapper emits intermediate (key, value) pairs directly into these r streams (as determined by the partitioner)—in this way, the (key, value) pairs are "pushed" to the reducers. Because the r streams in this case are being updated by m mappers in parallel, the streams must be synchronized. In the *hybrid* approach, k ($1 < k < m$) streams are maintained for each reducer and (key, value) pairs emitted by mappers for a particular reducer are distributed among the streams corresponding to that reducer.

Namespace manager: This module manages memory assignment to enable data reading and writing for MapReduce jobs. It converts filesystem paths that are specified in the standard Hadoop API into an abstraction we call a 'namespace': output is directed to an appropriate namespace that resides in memory, and, similarly, input records are directly consumed from memory as appropriate.

4. CHALLENGES

This section discusses the key challenges in performing large-scale analytics on shared-memory, multi-core platforms on the Java Virtual Machine, and how we addressed them in Hone.

Runtime Memory Consumption: To seamlessly support Hadoop code on a multi-core shared memory system, we made a design decision to implement Hone completely in Java, which meant contending with the limitations of the language. In Hone, each mapper task thread competes for shared resources with other mapper threads running in parallel. Thus, we must be careful about the choice of data structures, the number of object allocations, dereferencing of objects for better garbage collection, and so on. Standard Java programming practices scale poorly to large datasets. For example, consider a naive implementation of MapReduce using

Java `TreeMap<String, Integer>` as the underlying data structure: on a server with 128GB RAM, running word count on input with size 10% of the total RAM results in memory exhaustion. Java objects are heavyweight and it is not uncommon for some data structures to have 95% memory overhead, i.e., only 5% of the memory consumed is used for actual data.

To combat this, we extensively use primitive data structures such as byte arrays to minimize JVM-related overheads as much as possible. In the map phase, (key, value) pairs are serialized to raw bytes and in the reduce phase, new object allocations are minimized by reading pairs from byte streams using bit operations and reusing existing container objects. To the extent possible, we try to avoid using standard Java containers in favor of more efficient custom implementations.

Sorting is Expensive: For large intermediate outputs (on the order of GBs), we found sorting to be a major bottleneck. Operations such as swapping elements in a collection can be expensive, especially if an inappropriate data structure is used.

We experimented with two approaches to sorting. In the first, each thread from the thread pool handles both mapper execution and sorting of intermediate pairs. In the second, mapper execution and sorting are handled by two separate thread pools. We ultimately adopted the second, decoupled approach (note that this is orthogonal to the push/pull/hybrid models described above). Our decoupled design was selected based on several factors:

- The optimal thread pool size mainly depends on: (a) number of available cores in the machine, (b) size of the data each thread will handle, and (c) skew in the data distribution among the threads. The decoupled approach helps Hone automatically configure thread pool size based on these considerations. This helps the system be robust enough to handle data skew at the intermediate stage across different applications.
- The decoupled approach helps the garbage collector clean up the memory used by the map stage before starting the sort stage. This reduces the variance in overall execution time.
- Assignment of mapper with sorting to a thread may cause increased memory consumption of that thread. Stressing a thread in terms of both processing and memory may affect other threads by starving them for thread-level shared resources. Hence, it is better to decouple processing- and memory-intensive elements into separate threads.

Hone implements a custom quick sort solution that works solely with byte arrays. The main idea is to store all the intermediate (key, value) pairs in a byte array, and to create an offset byte array on the fly that records offset information corresponding to data in the raw data byte array. Once mapper output is stored in these data and offset byte arrays, quick sort is applied. Offsets are read from the offset array and data is read using bit operations, depending on the data type (so as to avoid object materialization whenever possible). Data items are compared with each other, but only offset bytes are swapped. In most cases, the size of the offset byte array is much smaller than the size of the data byte array, and therefore it is much more efficient to perform swapping operations on the offset byte array instead of the data byte array. The cost of swapping offsets is further reduced by swapping only the non-zero bytes in the offset array, as opposed to the entire four-byte span that comprises the entire offset.

Disk-Based Readers and Writers: In developing a MapReduce implementation for shared-memory machines that is API and binary compatible with standard (distributed) Hadoop, one major challenge is that Hadoop application code typically makes extensive use of disk-based readers (RecordReader) and writers (RecordWriter). The simplest way to avoid disk-based reading and writing overhead is to provide a parallel set of APIs that read/write from memory and then change the Hadoop user application code to use these new APIs. This is not a desirable approach in our case, as we wish to maintain compatibility with the existing Hadoop API.

Instead, we introduce the notion of a *namespace*, which is a dedicated part of memory where data is stored. Application code can access namespaces through the Job object to read input and store output. To maintain compatibility of our API with Hadoop's API, we provide efficient in-memory alternatives for existing FileReader and FileWriter Hadoop classes (these lower-level abstractions do not require swapping RecordReader and RecordWriter classes in the user's application code). Via these implementations, filesystem paths in the user's application code are automatically and transparently converted to appropriate namespaces; all reads and writes are redirected to these namespaces, bypassing disk.

Support for Iteration: Iterative MapReduce algorithms, where a sequence of MapReduce jobs are chained together such that the output of the previous reducers serves as input to the next mappers, are a well-known weakness of Hadoop and have been previously studied [1]. Since many interesting algorithms are iterative in nature (e.g., PageRank, LDA), this is an important problem to address. The primary issue with Hadoop implementations is that reducer output at each iteration *must* be serialized to disk, only to be immediately read by mappers at the next iteration. Of course, serializing serves the role of checkpointing and provides fault tolerance, but since Hadoop algorithms are forced to do this *every* iteration, there is no way to trade off fault tolerance for performance.

In Hone, all of these issues go away, since intermediate data reside in memory at the end of each iteration. The choice to serialize data to disk can be made independently by the developer. Thus, Hone provides natural support for iterative MapReduce algorithms.

To provide a bit more detail: typically, in an iterative algorithm, there is a "driver program" that sets up the MapReduce job for each iteration, checks for convergence, and decides if another iteration is necessary. Convergence checks are usually performed by reading reducer output (i.e., files on HDFS). In Hone, this is transparently handled by our notion of namespaces, since HDFS paths map (without programmer intervention) to in-memory buffers. Thus, existing Hadoop code continues to run unmodified.

5. EXPERIMENTAL RESULTS

We developed Hone completely from scratch in Java. The results presented here used the pull-based approach in shuffling data from mappers to reducers, with the thread pool size set to 16. The Hone experiments were performed on a server with dual Intel Xeon quad-core processors (E5620 2.4 GHz) and 128 GB RAM. We compared Hone against two setups: 1) a standard distributed Hadoop cluster and 2) Hadoop running in pseudo-distributed mode (PDM). The cluster configuration is as follows: 15 compute nodes, each of which has two quad core Xeon E5520 processors, 24 GB RAM, and three 2 TB disks. The PDM experiments were run on exactly the same machine as Hone. In PDM, Hadoop runs all daemons in separate processes; we set the configuration parameters for the maximum allowable in-memory buffer sizes to ensure as fair a comparison as possible, but note that Hadoop buffer sizes are limited to 32-bit integers. In both the distributed and PDM cases, we ran Cloudera's distribution of Hadoop (CDH4).

Here we present comparison results on five applications, each with three different dataset sizes. For the word count (WC) and inverted indexing (II) applications, we used subsets of English Wiki-

	Small					Medium					Large				
System	WC	KM	II	PR	LDA	WC	KM	II	PR	LDA	WC	KM	II	PR	LDA
Hone	5.3	4.5	7.4	1.4	138	29	16.4	56.1	4.3	500	268	66	341	11.9	957
PDM	53 (−10×)	59 (−13×)	67 (−9×)	59 (−42×)	854 (−6×)	139 (−5×)	54 (−3×)	255 (−5×)	53 (−12×)	3790 (−8×)	875 (−3×)	215 (−3×)	1183 (−3×)	66 (−6×)	8002 (−8×)
Cluster	67 (−13×)	42 (−9×)	42 (−6×)	30 (−21×)	285 (−2×)	90 (−3×)	57 (−3×)	150 (−3×)	34 (−8×)	1103 (−2×)	92 (+3×)	62 (∼)	380 (∼)	37 (−3×)	1970 (−2×)

Table 1: Performance comparison of Hone with Hadoop PDM and a 15-node Hadoop cluster for five different applications on small, medium, and large datasets. Reported numbers are in seconds. Parentheses show relative speedup compared to Hone.

pedia with sizes: {small: 125MB, medium: 1GB, large: 8GB}. For PageRank (PR), we used a Wikipedia graph dataset: {small: (0.4m nodes, 0.2m edges), medium: (1.8m nodes, 1.1m edges), large: (7.2m nodes, 4m edges)}. For k-means (KM), we used a randomly generated three-dimensional dataset with 10 centers, where the small dataset has 12m points, and the medium and large datasets have 51m and 398m points, respectively. Finally, we explored Latent Dirichlet Allocation (LDA) [12] on a TREC document collection; the small dataset contains documents totaling 125MB, and the medium and large datasets total 512MB and 1GB, respectively. Note that PageRank, k-means, and LDA are iterative algorithms, and we report the first iteration execution times.

Table 1 provides the results of our experiments. For the small datasets Hone can be more than an order of magnitude faster than Hadoop PDM and the fully-distributed Hadoop cluster. For the medium datasets, Hone outperforms both Hadoop PDM and the Hadoop cluster as well. For the large datasets, Hone still outperforms PDM; on word count Hone is slower than fully-distributed Hadoop, on k-means and inverted indexing, performance is about equal, and for PageRank and LDA, Hone remains faster.

We also compared Hone with the Phoenix system for the word count application, since that is the only system (among multi-core MapReduce implementations) available to compare against. We note that Phoenix is implemented in C++ whereas Hone is a Java-based system. For the word count application on the Wikipedia datasets of size 2GB, Hone was still 2× faster than Phoenix. Furthermore, Phoenix does not scale beyond 2GB datasets because of limitations imposed by `mmap()` that is used as an optimization, whereas Hone scales well over 8GB datasets.

6. DEMONSTRATION

Our demonstration will follow the experiments described above, comparing Hone, Hadoop running in pseudo-distributed mode, and fully-distributed Hadoop. The primary aim is to illustrate the advantages of Hone for running complex analytics on datasets that fit into memory on a single machine. Users will be able to experiment with different algorithm under different settings and observe the effects in real-time.

As part of the demonstration, we have developed a workload simulator that we believe is of independent interest. During our initial experimental evaluation of Hone, we found the use of a fixed set of applications to be too limited in providing sufficient control over the interactions between mappers and reducers. Thus, we designed a workload simulator that provides fine-grained control over the behavior of the mappers and the reducers through a set of parameters. One can control the communication pattern between mappers and reducers (one-to-one, one-to-many, many-to-many), skew in the key distribution, payload size, whether the overall computation is CPU-bound, memory-bound, or IO-bound, and so on. The workload simulator will allow us to compare Hone, Hadoop PDM, and fully-distributed Hadoop in a fine-grained manner under different execution scenarios.

7. CONCLUSION

Hadoop has already emerged as the *de facto* standard analytics platform for a variety of organizations. However, in cases where datasets can fit into memory, running fully-distributed Hadoop is inefficient. This is where Hone comes in: it allows us to "scale down" Hadoop algorithms to run efficiently on shared-memory machines, without breaking API and binary compatibility with existing Hadoop code. Independently, Hone creates many interesting implementation challenges regarding the application of data management techniques on the Java Virtual Machine.

8. ACKNOWLEDGMENTS

This work has been supported by the NSF under awards 0916043, 1144034, and 1218043. Any opinions, findings, or conclusions are the authors' and do not necessarily reflect those of the sponsor.

9. REFERENCES

[1] Y. Bu, B. Howe, M. Balazinska, and M. Ernst. HaLoop: Efficient iterative data processing on large clusters. *VLDB*, 2010.

[2] R. Chen, H. Chen, and B. Zang. Tiled-MapReduce: optimizing resource usages of data-parallel applications on multicore with tiling. *PACT*, 2010.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *OSDI*, 2004.

[4] W. Jiang, V. Ravi, and G. Agrawal. A Map-Reduce system with an alternate API for multi-core environments. *CCGRID*, 2010.

[5] Y. Mao, R. Morris, and M. Kaashoek. Optimizing MapReduce for multicore architectures. Technical Report MIT-CSAIL-TR-2010-020, MIT, 2010.

[6] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. *HPCA*, 2007.

[7] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas. Nobody ever got fired for using Hadoop on a cluster. *HotCloud*, 2012.

[8] A. Shinnar, D. Cunningham, V. Saraswat, and B. Herta. M3R: increased performance for in-memory Hadoop jobs. *VLDB*, 5(12), Aug. 2012.

[9] J. Talbot, R. Yoo, and C. Kozyrakis. Phoenix++: modular MapReduce for shared-memory systems. *MapReduce*, 2011.

[10] R. Yoo, A. Romano, and C. Kozyrakis. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. *IISWC*, 2009.

[11] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. *HotCloud*, 2010.

[12] K. Zhai, J. Boyd-Graber, N. Asadi, and M. Alkhouja. Mr. LDA: A flexible large scale topic modeling package using variational inference in MapReduce. *WWW*, 2012.