HJ-Hadoop: An Optimized MapReduce Runtime for Multi-core Systems

Yunming Zhang and Alan Cox and Vivek Sarkar
Rice University
{*yunming.zhang, alc, vsarkar*}*@rice.edu*

Abstract

We introduce HabaneroJava-Hadoop (HJ-Hadoop), an extension to the popular Hadoop MapReduce runtime system that is optimized for multi-core machines. The current Hadoop MapReduce implementation utilizes multiple cores in a machine by exploiting parallelism among map tasks and among reduce tasks. Each task is executed in a separate Java Virtual Machine (JVM). Unfortunately, in some applications, this design leads to poor memory utilization because some data structures used by the application are duplicated in their entirety across multiple JVMs running on the same machine. To address this problem, HJ-Hadoop implements an alternative way to utilize multi-core systems. Our approach reduces the number of map tasks created by exploiting multi-core parallelism within each map task. Adapting Hadoop MapReduce applications to use HJ-Hadoop is easy. Users just need to extend the HJMapper class in the HJ-Hadoop package instead of the Mapper class in the Hadoop package. The HJ-Hadoop runtime will automatically divide each map task into fine-grained HJ asynchronous tasks.

For some applications, HJ-Hadoop significantly reduces duplication of large static data structures. Specifically, our results for the memory-intensive KNN Join application show that the maximum input data sizes of the Hadoop Multithreaded Mapper, standard Hadoop, and HJ-Hadoop are approximately 50MB, 220MB and 400MB respectively, thereby demonstrating HJ-Hadoop's ability to improve memory utilization. At 250MB, HJ-Hadoop shows a 3× running time improvement relative to standard Hadoop. For non-memory intensive benchmarks taken from the Apache Mahout package, the relative improvement is in the 8% – 16% range.

1 Introduction

MapReduce [6] is a programming model that allows programmers to write data parallel programs that can run on thousands of machines. Moreover, it supports automatic concurrency management, locality-aware scheduling and fault tolerance. The Apache Hadoop implementation of MapReduce has been widely adopted due to its scalability, reliability and support from the open source community [2]. Going forward, it will become even more challenging for Hadoop MapReduce to utilize memory and CPU resources efficiently, as the number of cores in future processors continues to increase, and the available memory per core starts to decrease.

The current Hadoop MapReduce implementation uses multi-core systems by decomposing a MapReduce job into multiple map/reduce tasks that can execute in parallel. Each map/reduce task is executed in a separate JVM instance. The number of JVMs created in a single node (machine) can have a significant impact on performance due to their aggregate effects on CPU and memory utilization. There is a tendency to spawn a large number of tasks, and thus JVMs, to improve CPU utilization in multicore systems. For example, it is not uncommon to create 24 map tasks on an 8-core machine. However, this approach is only effective for *non-memory-intensive applications*.

For *memory-intensive applications*, a significant drawback of the current design is that some data structures are duplicated across JVMs, including static data and in-memory data structures used by map/reduce tasks. For example, a typical hash-join application requires each map task to store a copy of the lookup table in memory [1]. To make sufficient memory available to each map task, memory intensive applications are often forced to restrict the number of JVMs created to be smaller than the number of cores in a node at the expense of reducing CPU utilization.

In this paper, we propose a solution to this memory bottleneck by exploiting multicore parallelism at the intra-JVM level, while limiting the number of JVMs created on each node. At the same time, we don't want to sacrifice the fault-tolerance and reliability of the system. Thus, in addition to parallelizing multiple map tasks[1], we also parallelize the execution of a single map task to exploit intra-task parallelism. Our runtime system is called *HJ-Hadoop*, since it leverages the Habanero-Java (HJ) runtime model [5] for multicore parallelism.

Previous work in the Hadoop community to create multiple threads within a mapper JVM led to the Multithreaded Mapper [12]. In the Multithreaded Mapper, each thread runs as if it is a separate map task that reads

[1]The implementation described in this paper focuses on intra-JVM parallelism within map tasks, but can also be extended to reduce tasks.

from a common input split. However, the Multithreaded Mapper is rarely used in practice because its implementation does not solve the memory utilization problem. It replicates in-memory data structures across threads, thereby leading to a larger memory footprint than the default sequential Mapper. In addition, the parallelization is not very efficient as it synchronizes on reading (key,val) pairs from the common input split to ensure that the same (key,val) pair is not processed by two threads in parallel.

HJ-Hadoop delivers significant benefits for memory-intensive applications by reducing the number of JVM instances created per node, while still utilizing all available cores. At the same time, we maintain the reliability of Hadoop applications – the HJ-Hadoop runtime system is unaffected if a threaded map task crashes, since each task runs in a separate JVM [12].

Our performance results for the memory-intensive KNN Join application (Figure 1) show that the performance of the Hadoop Multithreaded Mapper, standard Hadoop, and HJ-Hadoop peak at input sizes of approximately 50MB, 220MB and 400MB respectively (and degrade thereafter), thereby demonstrating HJ-Hadoop's ability to improve memory utilization. At 250MB, HJ-Hadoop shows a $3\times$ performance improvement relative to standard Hadoop. For non-memory intensive benchmarks, the relative improvement is in the 8% – 16% range, primarily due to HJ-Hadoop's ability to achieve better load balance across cores. In particular, standard Hadoop often results in *straggling tasks* that take longer to run and are left running after other tasks have completed. HJ-Hadoop enables straggling tasks to use multiple cores to speed up their executions.

2 Motivating Examples

In this section, we describe the MapReduce implementations of K Nearest Neighbor Join and Fuzzy Kmeans [4] to motivate the memory and load balance optimizations.

2.1 K Nearest Neighbor Join

K Nearest Neighbor (KNN) Join takes in two data sets R and S. It compares every point in R against every point in S, and outputs results based on all pairwise comparisons, which leads to a complexity of $O(|R| \times |S|)$. KNN Join is representative of many large scale data analytics applications that examine interactions among different large data sets such as in-memory hash-joins, spatial range joins, and similarity-based search in databases. For example, the KNN Join is like the Fragment Replicated Join in PIG [7] and Map Side Join in Hive [11].

Since the two data sets are too large to fit in one node, the MapReduce KNN Join application has to divide up the data sets to process them across machines in parallel. It is often true that the size of one table S is much larger than the other data set R. [1] The current MapReduce implementation of KNN Join broadcasts the smaller R to every map task and divides up the larger S to stream each datum through the mappers.

In the map stage, the application loads the smaller data set R into the memory of each map task. The Hadoop MapReduce system then uses the larger data set S as input to the MapReduce job. The Hadoop MapReduce runtime splits up S into small pieces and uses them as input to each map task. The map function calculates the distance between two data points in R and S. The memory footprint of the mapper JVM is large because the data structure containing R takes up a lot of memory [3]. R can't be garbage collected because every (key,val) pair needs to compare against it. In the reduce stage, the application goes through each data point in R and chooses the closest K data points in S. The optimized algorithm uses very little time during the reduce phase.

Because R has to be read into every mapper JVM, it will be duplicated across tasks, incurring significant memory inefficiency. Furthermore, the limited memory in each JVM makes it hard to process large R data sets efficiently.

In KNN Join, the Query Set, which contains the documents we are trying to classify, is R. The Training Set, which contains documents that are already classified, is S.

2.2 FuzzyKMeans

FuzzyKMeans is an extension to the popular iterative clustering algorithm KMeans. While KMeans assigns each point to a cluster, FuzzyKMeans records the probability that each point belongs to a certain cluster.

In each map phase, we calculate the probability that the points belong to each cluster based on the distances to the centroids. In the reduce phase, the new means of the centroids are calculated. The algorithm terminates when it reaches a maximum number of iterations or when the centroids have stabilized. A typical implementation chains together many MapReduce jobs to refine the coordinates of the centroids.

FuzzyKMeans is representative of many popular machine learning algorithms. The application doesn't have a large memory footprint. We use it to show the performance benefit from the load balance optimizations. We have also included results for KMeans and Dirchlet clustering [8].

3 Programming Model

We aim to make it easy for users to adapt their Hadoop MapReduce applications to run on HJ-Hadoop. To achieve this goal, we created an HJMapper class that can be extended by the user, just like a regular Mapper class. HJMapper uses HJ parallel constructs to create and manage HJ async tasks. However, the user doesn't have to write any Habanero Java code to take advantage of the HJ async task parallelism. Users simply need to extend the HJMapper class instead of the Hadoop Mapper class. It does require user-provided map functions to be thread-safe so that multiple input (key,val) pairs can be processed in parallel.

```
public class MyMapper extends
    HJMapper<LongWritable,Text,
    Text, LongWritable> {

  public void map(LongWritable key,
            Text value,
            Contextcontext)  {
               ...
  }
}
```

HJ-Hadoop also allows users to use HJ parallel constructs in map and reduce codes. This could be useful in more compute intensive Map or Reduce functions and is the focus of future work.

4 HJ-Hadoop Implementation

The computation in Hadoop MapReduce jobs is performed by the mapper and reducer implementations provided by the user. The user defines how to process the input (key,val) pairs in the map and reduce functions. We focused on improving the efficiency of the Map phase. It is often more compute intensive and takes significantly longer than the Reduce phase in data analytics applications such as KMeans and KNearestNeighbor.

Once the user submits a Hadoop MapReduce job, the runtime splits the input of the job based on the user specified split size. Each mapper then reads in its own input split. By default, mappers sequentially generate (key,val) pairs from their input split. Every time a (key,val) pair is generated, the Mapper JVM immediately processes it using the user defined map function to produce zero or more intermediate (key, value) pairs. The intermediate pairs are later processed in the Reduce phase. This design is inherently sequential as the mapper JVM has to finish processing a (key,val) pair before moving on to process the next one.

The key idea behind the HJMapper implementation is simple and straightforward. Instead of reading in and processing one (key,val) pair at a time, we read in a certain number of (key, val) pairs to create chunks of (key,val) pairs and process these different chunks in parallel.

Automatic parallelization of map tasks must rely on an efficient parallel runtime to execute them. We chose Habanero-Java [5], a parallel programming language developed at Rice University, to create our parallel runtime with lightweight async parallel tasks. We used the Habanero Java work sharing runtime for managing the creation, scheduling, execution and termination of tasks. The runtime uses a single task queue and has worker threads pick work from the queue to achieve load balance across cores.

Our experiments show that it takes a non-negligible amount of time to read in the (key, val) pairs. For example, the KMeans application takes up to 8 seconds to load 128 MB of input within one JVM, whereas the overall execution takes 20 seconds. To improve the performance, HJ-Hadoop overlaps I/O with computation by dedicating one worker thread to prefetch (key,val) pairs into a buffer while other worker threads are executing map tasks. To do this, the runtime allocates a new buffer for each async task. Once the buffer is full, the I/O thread starts an async task to process it.

Another advantage of our design that uses a separate buffer for each HJ async task is that it allows the JVM to free up buffers in completed tasks. This further reduces the memory footprint of the Mapper.

Our experiments also show that chunk sizes have a non-trivial impact on the running time of the program. Since the execution time for each call to a map function for each pair is different from application to application, there isn't a fixed chunk size that is optimal for all applications. The challenge becomes how to dynamically select a chunk size that will achieve good performance. To tackle this issue, we implemented a chunking of the (key, val) pairs that adapts to the execution time of the map function. The main thread reads in a small number of input (key, val) pairs as a sample chunk. It records the time it took to process the sample chunk. Based on an empirically chosen desired running time for each chunk, the runtime calculates the optimal chunk size knowing the running time of the sample chunk. In the results section, we show that dynamic chunking works better than fixed chunk sizes.

5 Experimental Results

This section presents our experimental evaluation of the HJ-Hadoop runtime. We used KNN Join to demonstrate the benefits of HJ-Hadoop's improved memory efficiency. To demonstrate the effects of HJ-Hadoop's improved load balancing, we benchmarked some widely used clustering algorithms from Apache Mahout. In each

case, we compared the performance of HJ-Hadoop to Hadoop with the standard sequential Mapper and with the multithreaded Mapper.

5.1 Experimental Setup

Each worker node had two quad core 2.4 GHz Intel Xeon CPUs with an 8 MB last-level cache. There was 32 GB of memory per node shared by all cores. We used Habanero Java 1.3.1, Java 1.6.0 and Hadoop 1.0.3. 8 mappers were used on Hadoop with the standard sequential mapper to fully utilize the 8 cores in each node. The heap size limit was set to 1.5 GB for each JVM to simulate about 12 GB available RAM in a machine. We used 4 mappers for HJ-Hadoop and Hadoop with the multithreaded mapper, and we set the maximum heap size of each JVM to 2.5 GB. Our experiments showed that it is hard to achieve a good I/O utilization rate with fewer than 4 JVMs. Using 4 mappers achieves a good balance between I/O utilization and memory efficiency for HJ-Hadoop. For the multithreaded mapper, we used 8 threads for each mapper to ensure full utilization of the available cores. We always used the same block size for all test runs for the same application to rule out the impact of block size.

5.2 KNN Join

For KNN Join, we benchmarked the performance on a single worker node, since we found that the performance is most strongly impacted by the number of times garbage collection is performed within each JVM. Increasing the number of worker nodes will have no impact on the execution time of individual map or reduce tasks.

The results are shown in Figure 1. The x axis represents the size of the Query Set for KNN Join. The Query Set is loaded into every map task. The input to the MapReduce job is 64 MB of Training Set. The block size is 128 MB and the split size is 8 MB.

As we can see in Figure 1, as the Query Set size increases, the multithreaded mapper could only process up to 50 MB of Query Set data efficiently. This behavior is similar to that of the "breakdown regions" in the model presented in [10]. The performance of standard Hadoop peaks at approximately 220 MB of Query Set data. We have logged a 3-fold increase in the number of garbage collection calls within each JVM between the two runs of Hadoop with 220 MB and 230 MB of Query Set data. The Hadoop JVMs with sequential mappers cannot process Query Sets larger than 250 MB. In contrast, HJ-Hadoop's running time increases linearly all the way to 400 MB due to the larger heap size available in a single HJ-Hadoop JVM. This allows each HJ-Hadoop KNN job to process almost twice as much Query Set data as a Hadoop KNN job.

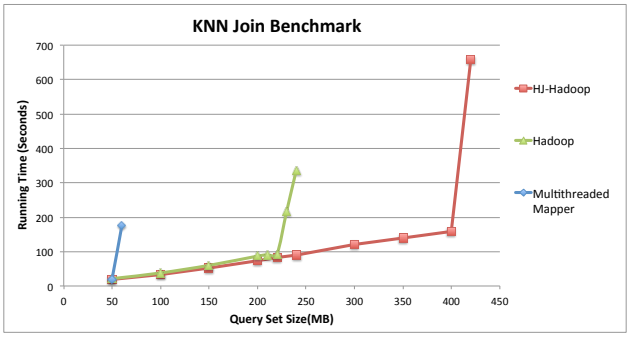

Figure 1: Running time of HJ-Hadoop, Hadoop and Hadoop with multithreaded mapper using KNN Join benchmark on a single node with fixed 64MB Training Set and varying Query Set size.

To avoid significant performance degradation due to insufficient memory, one could create a chain of MapReduce jobs and process smaller parts of the Query Set in each job [8]. However, a large overhead is incurred on loading the Query Set and Training Set data and starting a large number of map/reduce tasks for each job. The ability to process as much Query Set data as possible in a MapReduce job is critical to the performance of memory intensive data analytics applications.

5.3 Clustering Algorithms

For the FuzzyKMeans, KMeans and Dirichlet clustering algorithms, we used 8 nodes with 12 GB input data each. All three algorithms were taken from the widely used Apache Mahout package, a collection of optimized Hadoop map reduce machine learning algorithms [4]. We made slight modifications to make data structures thread safe in the package. We chose to use Apache Mahout as our benchmarks because the way applications are implemented can have a non-trivial impact on the performance. We want to ensure that the implementations of clustering algorithms used in our evaluation are representative of real applications used in industry. To demonstrate the advantage of improved load balancing across cores, we chose applications that are not memory intensive, unlike KNN. The results are presented in Table 1.

From row 2 in Table 1, we can see that Hadoop with the multithreaded mapper using 4 JVMs per worker node actually results in a slowdown compared to Hadoop with the sequential mapper using 8 JVMs per worker node. The slowdown could be the result of inefficiencies in its implementation as we described in the introduction. On the other hand, HJ-Hadoop achieved 8% – 16% speedup with 4 JVMs.

In our experiments, we noticed that the size of each HJ asynchronous task has an impact on the overall perfor-

Applications	FuzzyKMeans	KMeans	Dirichlet
Hadoop	560s	625s	466s
Multithreaded Mapper	614s(0.91)	625s(1.00)	511s(0.91)
HJ-Hadoop	483s(1.16)	559(1.11)	431s(1.08)

Table 1: Comparing the results of HJ-Hadoop, Hadoop and Hadoop with the multithreaded mapper to demonstrate the speedup from improved load balance across cores. The tests were conducted on 8 worker nodes with a 12 GB input. For Hadoop, we uses 8 mapper JVMs per node. For HJ-Hadoop and Hadoop with the multithreaded mapper, we used 4 mappers. Execution times are reported in seconds. Speedup relative to standard Hadoop is reported in parenthesis.

Benchmark	10	100	500	5000	adaptive
KMeans	575s	560s	556s	561s	559s
FuzzyKMeans	489s	495s	505s	520s	483s
Dirichlet	450s	445s	425s	448s	431s

Table 2: HJ-Hadoop execution times for 12 GB input on 8 worker nodes with different chunk sizes for input (key,val) pairs

mance of the applications. Table 2 presents the running time of the three applications using the HJ-Hadoop runtime with different chunk sizes. For the more compute intensive FuzzyKMeans application, the optimal chunk size is around 10 - 100 (key, val) pairs. For the KMeans and Dirichlet Clustering applications, the optimal chunk size is around 500 (key, val) pairs. We can see that the adaptive chunking can achieve close or better running time results for each application without resorting to a fixed chunk size.

To demonstrate improved CPU utilization in HJ-Hadoop, we recorded the CPU utilization over time of the FuzzyKMeans application for both the Hadoop and HJ-Hadoop systems. In our job configuration, FuzzyKMeans chains together 3 MapReduce jobs to perform the iterative improvement algorithm. We present the first 180 seconds out of 560 seconds of the CPU utilization data in figure 2. Both HJ-Hadoop and Hadoop achieved full utilization from 0 to 130 seconds (800 percent for 8 cores). From 130 seconds to 180 seconds, HJ-Hadoop achieved better utilization of the cores because there were straggling map tasks towards the end of the map phase. A similar trend was seen in the later MapReduce jobs.

6 Related Work

Phoenix is a shared memory MapReduce framework optimized for multi-core systems [9]. It focuses on the performance of a single multi-core machine whereas HJ-Hadoop can easily scale to hundreds of machines by taking advantage of the scalability of Hadoop. Phoenix uses a new thread instead of a new JVM to execute each map

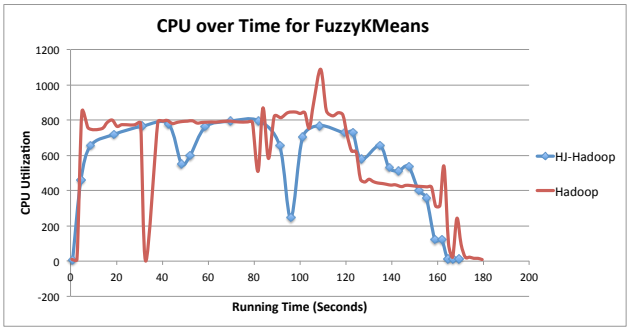

Figure 2: CPU over time for HJ-Hadoop and Hadoop in FuzzyKMeans benchmark

or reduce task. The design was not concerned with duplicating static data structures across map tasks. No optimization for static data structures used in map tasks was mentioned. The paper discussed the optimization of a dynamic framework that discovers the best unit size for each program as future work but never implemented one. We have explored an approach to dynamically setting HJ async task sizes based on sampling the execution time for chunks.

Spark is a MapReduce system that is built using Resilient Distributed Datasets (RDDs) [13]. Spark keeps data structures in memory for successive MapReduce jobs to avoid redundant disk I/O operations. Our work, however, tries to improve memory efficiency by minimizing duplication of in-memory data structures within a single MapReduce job.

7 Conclusion

We identified some key performance bottlenecks in the execution of the Hadoop MapReduce framework on multi-core systems. We implemented an alternative approach to utilize multi-core systems in HJ-Hadoop to address the poor memory utilization issue. The HJ-Hadoop runtime subdivides Hadoop map tasks into light-weight HJ async tasks for intra-JVM parallelism. HJ-Hadoop uses dynamically set chunk sizes for HJ async tasks to overlap computation and I/O. The programming interface to HJ-Hadoop is compatible with Hadoop MapReduce, making it easy to adapt Hadoop MapReduce applications to HJ-Hadoop. Experimental results show that HJ-Hadoop outperforms standard Hadoop on memory intensive applications by as much as 3× for the KNN application with large duplicated data and can process twice as much data in each job. For non-memory-intensive applications, HJ-Hadoop achieved an 8% – 16% speedup relative to standard Hadoop due to improved load balance across cores.

Acknowledgments The authors would like to thank Prof. Chris Jermaine for his feedback on the research, Vincent Cave for extending the Habanero Java implementation to make this work possible, Max Grossman for his suggestions on optimizations, and Adrien Pellerin for helping develop the benchmark scripts.

References

[1] Fragment Replicated Join in Pig. `http://wiki.apache.org/pig/PigFRJoin`.

[2] Hadoop. `http://hadoop.apache.org`.

[3] Hive Join Optimization. `https://cwiki.apache.org/Hive/joinoptimization.html`.

[4] Mahout. `http://mahout.apache.org`.

[5] CAVÉ, V., ZHAO, J., SHIRAKO, J., AND SARKAR, V. Habanero-java: the new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java* (New York, NY, USA, 2011), PPPJ '11, ACM, pp. 51–61.

[6] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[7] GATES, A. F., NATKOVICH, O., CHOPRA, S., KAMATH, P., NARAYANAMURTHY, S. M., OLSTON, C., REED, B., SRINIVASAN, S., AND SRIVASTAVA, U. Building a high-level dataflow system on top of map-reduce: the pig experience. *Proc. VLDB Endow. 2*, 2 (Aug. 2009), 1414–1425.

[8] GILLICK, D., FARIA, A., AND DENERO, J. Mapreduce: Distributed computing for machine learning, 2006.

[9] RANGER, C., RAGHURAMAN, R., PENMETSA, A., BRADSKI, G., AND KOZYRAKIS, C. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture* (Washington, DC, USA, 2007), HPCA '07, IEEE Computer Society, pp. 13–24.

[10] SARKAR, V., AND DOLBY, J. High-performance scalable java virtual machines. In *HiPC* (2001), pp. 151–166.

[11] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. Hive- a warehousing solution over a map-reduce framework. In *IN VLDB '09: PROCEEDINGS OF THE VLDB ENDOWMENT* (2009), pp. 1626–1629.

[12] WHITE, T. *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.

[13] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* (Berkeley, CA, USA, 2010), HotCloud'10, USENIX Association, pp. 10–10.