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

Yunming Zhang
Rice University
yz17@rice.edu

Categories and Subject Descriptors D.3.4 [*Processors*]: Runtime environments

Keywords Hadoop, Habanero Java

1. Introduction

This submission introduces HabaneroJava-Hadoop (HJ-Hadoop), an extension to the popular Hadoop MapReduce runtime system that is optimized for multi-core machines.

MapReduce allows programmers to write data parallel programs that can run on thousands of machines [1]. Going forward, it will become even more challenging for 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. It is important to exploit massive parallelism without sacrificing the memory efficiency of the machines.

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.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.

This submission proposes 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, we also parallelize the execution of a single map task to exploit intra-task parallelism. *HJ-Hadoop* leverages the Habanero-Java (HJ) runtime model [2] for multicore parallelism.

Previous work in the Hadoop community to create multiple threads within a mapper JVM led to the Multithreaded Mapper.However, Multithreaded Mapper replicates in-memory data structures across threads, thereby leading to a larger memory footprint than the default sequential Mapper.

SPLASH '13, October 26–31, 2013, Indianapolis, Indiana, USA.

ACM 978-1-4503-1995-9/13/10.
http://dx.doi.org/10.1145/2508075.2514875

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, thereby demonstrating HJ-Hadoop's ability to improve memory utilization. At 250MB, HJ-Hadoop shows a 3× performance improvement relative to standard Hadoop.For non-memory intensive clustering algorithms, the relative improvement is in the 8% – 16% range due to HJ-Hadoops ability to achieve better load balance across cores.

2. Motivating Examples

2.1 K Nearest Neighbor Join

K Nearest Neighbor (KNN) Join takes in two data sets R (Query Set)and S(Training Set). It compares every point in R against every point in S, and outputs results based on all pairwise comparisons. KNN Join is representative of many large scale data analytics applications such as Fragment Replicated Join in PIG. It is often true that S is much larger than R. In the map stage, the application loads the smaller data set R into the memory of each map task. 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. In the reduce stage, the application goes through each data point in R and chooses the closest K data points in S. 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.

2.2 FuzzyKMeans

FuzzyKMeans is representative of many popular machine learning algorithms. The application doesn't have a large memory footprint. I use it to show the performance benefit from the load balance optimizations. The results for KMeans and Dirchlet clustering algorithms are also included. In each map phase, the probability that the points belong to each cluster is calculated based on the distances to the centroids. In the reduce phase, the new means of the centroids are calculated. The algorithm chains together many MapReduce jobs to refine the coordinates of the centroids.

3. HJ-Hadoop Implementation

The computation in Hadoop MapReduce jobs is performed by user-defined mappers and reducers. 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. 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.

Instead of reading in and processing one (key,val) pair at a time, HJ-Hadoop reads 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. Habanero-Java parallel programming work sharing runtime was used to manage 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. 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.

Our experiments also show that chunk sizes have a non-trivial impact on the running time of the program and there isn't a fixed chunk size that is optimal for all applications. To find the right chunk size, I 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. Results show that dynamic chunking works better than fixed chunk sizes.

4. Experimental Results

4.1 Experimental Setup

Each worker node had two quad core 2.4 GHz Intel Xeon CPUs with an 8 MB last-level cache. 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. I used 4 mappers for HJ-Hadoop and Hadoop with the multithreaded mapper, and I set the maximum heap size of each JVM to 2.5 GB.

4.2 KNN Join

For KNN Join, I benchmarked the performance on a single worker node, since I 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. The performance of standard Hadoop peaks at approximately 220 MB of Query Set data. I 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.

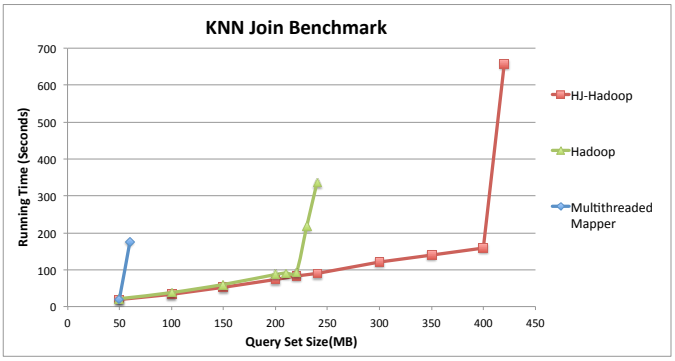

Figure 1: Running time of HJ-Hadoop, Hadoop and Hadoop with multi-threaded mapper using KNN Join benchmark on a single node with fixed 64MB Training Set and varying Query Set size.

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.

4.3 Clustering Algorithms

For non-memory-intensive applications, I chose FuzzyKMeans, KMeans and Dirichlet clustering algorithms to demonstrate the performance benefit of HJ-Hadoop.

From row 2 in Table 1, we can see that Hadoop with the multi-threaded 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 due to inefficient implementation. On the other hand, HJ-Hadoop achieved 8% – 16% speedup with 4 JVMs through improved load balance across cores.

5. Related Work

Phoenix is a shared memory MapReduce framework optimized for multi-core systems. It focuses on the performance of a single multi-core machine. 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). 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.

References

[1] Hadoop. `http://hadoop.apache.org`. URL `http://hadoop.apache.org`.

[2] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-java: the new adventures of old x10. PPPJ '11.