A Hierarchical Approach to Maximizing MapReduce Efficiency

Zhiwei Xiao, Haibo Chen, Binyu Zang
Parallel Processing Institute, Fudan University
{*zwxiao, hbchen, byzang*}*@fudan.edu.cn*

I. Introduction

MapReduce [1] has been widely recognized for its elastic scalability and fault tolerance, with the efficiency being relatively disregarded, which, however, is equally important in "pay-as-you-go" cloud systems such as Amazon's Elastic MapReduce. This paper argues that there are multiple levels of data locality and parallelism in typical multicore clusters that could affect performance.

By characterizing the performance limitation of typical MapReduce applications on multi-core based Hadoop clusters, we show that current JVM-based runtime (i.e., TaskWorker) fails to exploit data locality and task parallelism at single-node level.

Specifically, the open-source implementation of MapReduce, Hadoop [2], employs a JVM runtime to run the actual MapReduce tasks, which is suboptimal to explore the cache hierarchy and task parallelism existing in many multi-core based commodity clusters. Hadoop requires both key and value objects to implement the Hadoop Writable interface to support serialization and deserialization, causing extra objects creation and destroy overhead as well as memory footprint.

Moreover, some applications require processing the same piece of data multiple times or iteratively to get the final results. Though Hadoop exploits data locality with a single iteration of jobs by moving computation to its data as much as possible, unfortunately, it does not consider data locality across multiple processing iterations, and thus requires the same data being loaded multiple times from the networking file systems to nodes that process the data.

Based on the above observations, we propose Azwraith, a hierarchical MapReduce approach aiming to maximize data locality and task parallelism of MapReduce applications on Hadoop. In the hierarchical MapReduce model of Azwraith, each Map or Reduce task assigned to a single node is treated as a separate MapReduce job and is further decomposed into a Map and a Reduce tasks, which are processed by a MapReduce runtime specially optimized on a single node. Specifically, Azwraith integrates an efficient MapReduce runtime (namely Ostrich [3]) for multi-core to Hadoop. To exploit data locality among nodes at networking level, Azwraith integrates an in-memory cache system that caches data in memory that will likely be reused again, to avoid unnecessary networking and disk traffics.

II. Azwraith Design

Instead of writing a new runtime from scratch, we reuse and adapt an efficient MapReduce implementation for shared memory multiprocessor to Hadoop, called Ostrich. Ostrich adopts the "tiling strategy" to MapReduce on multicore and aggressively exploits task parallelism and data locality on multicore. To minimize complexity, Azwraith follows exactly the workflow of Hadoop and leaves most components (e.g., TaskTracker, HDFS) untouched. Hadoop with the Azwraith extension can still run original Java-based jobs in the normal way.

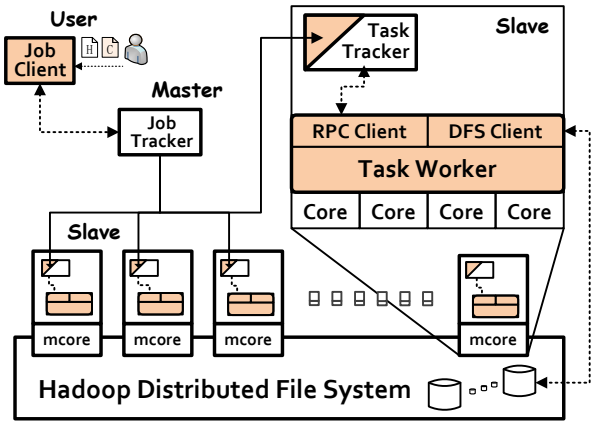

Figure 1. Azwraith architecture: the solid (colored) components are that need to be adjusted for Azwraith: Azwraith replaces the TaskWorker with Ostrich that exploits the data locality and parallelism in multi-core; The RPC Client makes the *TaskWorker* conform the Hadoop RPC protocol and provides RPC services to the *TaskWorker*; The DFS Client provides HDFS accesses for the *TaskWorker*. User can submit an Azwraith job or an original Hadoop job to the MapReduce system with JobClient.

Figure 1 shows the overall architecture of Azwraith, which resembles the original Hadoop. The execution of an Azwraith job also follows with the workflow of Hadoop. The client must specify the job type as Azwraith or Hadoop before submitting a job. The *TaskTracker* can fork a process (in case of an Azwraith job) or start a JVM instance (in case of a Hadoop job) to run the assigned task, according to the job type.

To adapt the Ostrich runtime as the *TaskWorker* of Hadoop, we modify Ostrich to conform to the workflow and communication protocols of Hadoop, including status reporting and output format. To communicate with the

TaskTracker, the new *TaskWorker* is embedded with an RPC (Remote Procedure Call) client. The RPC client is used to make the new *TaskWorker* conform to the Hadoop RPC protocol and provide RPC services to the *TaskWorker*. The *TaskWorker* accesses the HDFS (Hadoop Distributed File System) with the DFS client module, which directs accesses to HDFS.

We further enhance the Azwraith runtime with sevral optimizations. First, Azwraith overlaps the CPU burst and the I/O burst to hide the I/O blocking time as much as possible. Second, Azwraith can aggressively reuse the data in memory with pointer operations and thus avoid copying large amount of memory and enjoy a good cache locality. Thirdly, Azwraith requires applications to implement a set of aggregative (de)serialization interfaces to (de)serialize a set of data together, which saves tremendous amount of function calls and gain a better data-locality than Hadoop.

To enable data reuse among tasks, we design and implement a cache server as a daemon process on each slave to serve all HDFS read accesses. The cache server leverages the shared memory interfaces and the semaphore mechanism in an operating system to provide control and data information to cache clients (i.e., *TaskWorkers*), which access the shared memory managed by the cache server in the way of accessing the local memory. Using shared memory also gains a memory usage benefit, since all tasks in the same node can share one single copy of data and thus leave more memory available for caching and computing.

We also extend the affinity support of the Hadoop schedule, to schedule tasks to nodes where input data are cached. Azwraith maintains the cache locations information, with a mapping similar to Hadoop's disk-local task mapping. When receiving a task assignment request from a slave (node), the scheduler would first get the map-tasks list for the requesting slave (node), and assign a cache-local task if any. Otherwise, the scheduler works as usual. The extension to the scheduling system makes around 50 lines of code changes to Hadoop.

III. Evaluation

We conducted the experiments on a small-scale cluster with 1 master node and 6 slave nodes. Each machine was equipped with two AMD Opteron 12-core processors, 64 GB main memory and 4 SCSI hard drives. Each machine connected to the same switch through a 1Gb Ethernet link. We used Hadoop version 0.20.1 running on Java SE Runtime 1.6.0. Azwraith was also built on the same version of Hadoop.

As shown in Figure 2, Azwraith gains a considerable speedup over Hadoop with different input sizes, ranging from 1.4x to 3.5x. Computation-oriented tasks like WordCount and LinearRegression gain larger speedup than the I/O-intensive applications such as GigaSort.

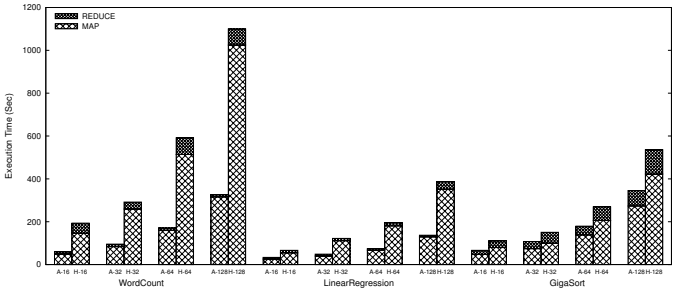

Figure 2. Overall performance and the breakdown of WordCount, LinearRegression and GigaSort. The symbol of A-x refers to Azwraith with xGB input, while H-x refers to Hadoop with xGB input.

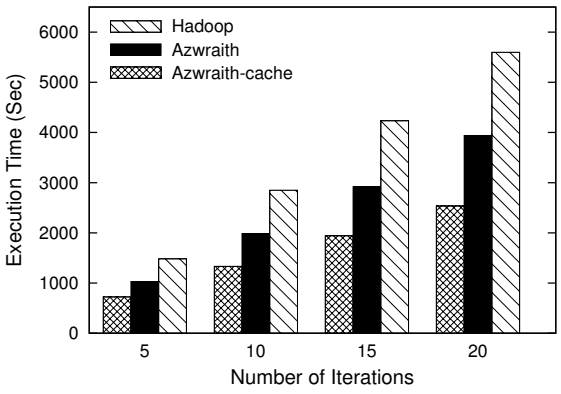

Figure 3. Performance of Azwraith Cache System on K-Means

As shown in Figure 3, with the support of the cache system, Azwraith gains a 1.43X to 1.55X speedup over Azwraith without the cache scheme, and 2.06X to 2.21X over Hadoop.

IV. Conclusion

In this paper, we argued that Hadoop has limitations in exploiting data locality and task parallelism for multi-core platforms. We then extended Hadoop with a hierarchical MapReduce scheme. An in-memory cache scheme is also seamlessly integrated to cache data that is likely to be accessed in memory. Evaluation showed that the hierarchical scheme outperforms Hadoop ranging from 1.4x to 3.5x.

References

[1] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] A. Bialecki, M. Cafarella, D. Cutting, and O. OMalley, "Hadoop: a framework for running applications on large clusters built of commodity hardware," http://lucene. apache. org/hadoop, 2005.

[3] R. Chen, H. Chen, and B. Zang, "Tiled mapreduce: Optimizing resource usages of data-parallel applications on multicore with tiling," in *Proc. PACT*, 2010.