Mimir: Memory-Efficient and Scalable MapReduce for Large Supercomputing Systems

Tao Gao,[a,d] Yanfei Guo,[b] Boyu Zhang,[a] Pietro Cicotti,[c] Yutong Lu,[e,f,d]
Pavan Balaji,[b] and Michela Taufer[a]

[a]University of Delaware
[b]Argonne National Laboratory
[c]San Diego Supercomputer Center
[d]National University of Defense Technology
[e]National Supercomputing Center in Guangzhou
[f]Sun Yat-sen University

***Abstract*—In this paper we present *Mimir*, a new implementation of MapReduce over MPI. Mimir inherits the core principles of existing MapReduce frameworks, such as MR-MPI, while redesigning the execution model to incorporate a number of sophisticated optimization techniques that achieve similar or better performance with significant reduction in the amount of memory used. Consequently, Mimir allows significantly larger problems to be executed in memory, achieving large performance gains. We evaluate Mimir with three benchmarks on two high-end platforms to demonstrate its superiority compared with that of other frameworks.**

Keywords: High-performance computing; Data analytics; MapReduce; Memory efficiency; Performance and scalability

I. Introduction

With the growth of simulation and scientific data, data analytics and data-intensive workloads have become an integral part of large-scale scientific computing. Analyzing and understanding large volumes of data are becoming increasingly important in various scientific computing domains, often as a way to find anomalies in data, although other uses are being actively investigated as well. Big data analytics has recently grown into a popular catch-all phrase that encompasses various analytics models, methods, and tools applicable to large volumes of data. MapReduce is a programming paradigm within this broad domain that—loosely speaking—describes one methodology for analyzing such large volumes of data.

We note that big data analytics and MapReduce are not inventions of the scientific computing community, although several ad hoc tools with similar characteristics have existed for several decades in this community. These are generally considered borrowed concepts from the broader data analytics community [10] that has also been responsible for developing some of the most popular implementations of MapReduce, such as Hadoop [27] and Spark [30]. While these tools provide an excellent platform for analyzing various forms of data, the hardware/software architectures that they target (i.e., generally Linux-based workstation clusters) are often different from that which scientific computing applications target (i.e., large supercomputing facilities).

While commodity clusters and supercomputing platforms might seem similar, they have subtle differences that are important to understand. First, most large supercomputer installations do not provide on-node persistent storage (although this situation might change with chip-integrated NVRAM). Instead, storage is decoupled into a separate globally accessible parallel file system. Second, network architectures on many of the fastest machines in the world are proprietary. Thus, commodity-network-oriented protocols, such as TCP/IP or RDMA over Ethernet, do not work well (or work at all) on many of these networks. Third, system software stacks on these platforms, including the operating system and computational libraries, are specialized for scientific computing. For example, supercomputers such as the IBM Blue Gene/Q [3] use specialized lightweight operating systems that do not provide the same capabilities as what a traditional operating system such as Linux or Windows might.

Researchers have attempted to bridge the gap between the broader data analytics tools and scientific computing in a number of ways. These attempts can be divided into four categories: (1) deployment of popular big data processing frameworks on high-performance computers [24], [17], [26], [9]; (2) extension to the MPI [5] interface to support $\langle key, value \rangle$ communication [16]; (3) building of MapReduce-like libraries to support in situ data processing on supercomputing systems [25]; and (4) building of an implementation of MapReduce on top of MPI [21]. Of these, MapReduce implementations over MPI—particularly MR-MPI [21]—have gained the most traction for two reasons: they provide C/C++ interfaces that are more convenient to integrate with scientific applications compared with Java, Scala, or Python interfaces, which are often unsupported on some large supercomputers; and they do not require any extensions to the MPI interface.

MR-MPI has taken a significant first step in bridging the gap between data analytics and scientific computing. It embodies the core principles of MapReduce, including scalability to large systems, in-memory processing where possible, and spillover to the I/O subsystem for handling large datasets; and it does so while allowing scientific applications to easily and efficiently take advantage of the MapReduce paradigm [31], [22]. Yet despite its success, the original MR-MPI implementation still suffers from several shortcomings. One shortcoming is its inability to handle system faults: we addressed this shortcoming in our previous work [12]. Another significant shortcoming is its simple memory management. Specifically,

1530-2075/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPS.2017.31

MR-MPI uses a model based on fixed-size "pages": MR-MPI pages are static memory buffers that are allocated at the start of each MapReduce phase and used throughout this phase. As long as the application dataset can fit in these pages, the data processing is in memory. But as soon as the application dataset is larger than what fits in these pages, MR-MPI spills over the data into the I/O subsystem. While this model is functionally correct, it leads to a tremendous loss in performance.

Figure 1 illustrates this point with the *WordCount* benchmark on a single compute node of the Comet cluster at the San Diego Supercomputing Center (cluster details are presented in Section IV). We note that while MR-MPI provides the necessary functionality for this computation, it experiences significant slowdown in performance for datasets larger than 4 GB, even though the node itself contains 128 GB of memory. Consequently, increasing the dataset size from 4 GB to 64 GB results in nearly three orders of magnitude degradation in performance.

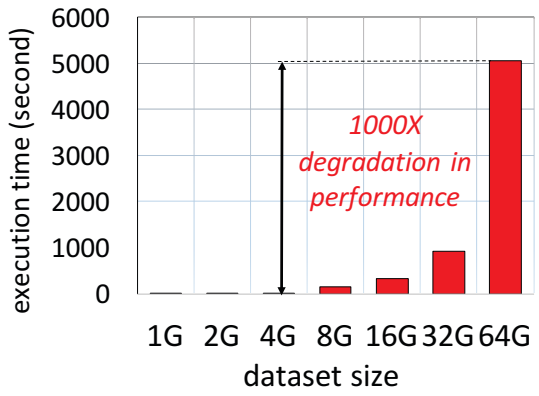

Fig. 1: Single-node execution time of *WordCount* with MR-MPI on Comet.

The goal of the work presented here is to overcome such inefficiencies and design a memory-efficient MapReduce library for supercomputing systems. To this end, we present a new MapReduce implementation over MPI, called *Mimir*. Mimir inherits the core principles of MR-MPI while redesigning the execution model to incorporate a number of sophisticated optimization techniques that significantly reduce the amount of memory used. Our experiments demonstrate that for problem sizes where MR-MPI can execute in memory, Mimir achieves equal or better performance than does MR-MPI. At the same time, Mimir allows users to run significantly larger problems in memory, compared with MR-MPI, thus achieving significantly better performance for such problems.

The rest of this paper is organized as follows. We provide a brief background of MapReduce and MR-MPI in Section II. In Section III, we introduce the design of Mimir and present experimental results demonstrating its performance in Section IV. Other research related to our paper is presented in Section V. We finally draw our conclusions in Section VI.

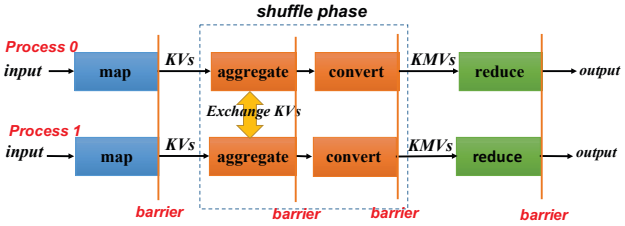

Fig. 2: The *map*, *shuffle*, and *reduce* phases in MR-MPI.

II. Background

In this section, we provide a high-level overview of the MapReduce programming model and the MR-MPI implementation of MapReduce.

A. MapReduce Programming Model

MapReduce is a programming model intended for data-intensive applications [10] that has proved to be suitable for a wide variety of applications. A MapReduce job usually involves three phases: *map*, *shuffle*, and *reduce*. The *map* phase processes the input data using a user-defined map callback function and generates intermediate $\langle key, value \rangle$ (KV) pairs. The *shuffle* phase performs an all-to-all communication that distributes the intermediate KV pairs across all processes. In this phase KV pairs with the same key are also merged and stored in $\langle key, \langle value1, value2... \rangle \rangle$ (KMV) lists. The *reduce* phase processes the KMV lists with a user-defined reduce callback function and generates the final output. A global barrier between each phase ensures correctness. The user needs to implement the map and reduce callback functions, while the MapReduce runtime handles the parallel job execution, communication, and data movement.

Several successful implementations of the MapReduce model exist, such as Hadoop [1] and Spark [30]. These frameworks seek to provide a holistic solution that includes the MapReduce engine, job scheduler, and distributed file system. However, large supercomputing facilities usually have their own job scheduler and parallel file system, thus making deployment of these existing MapReduce frameworks in such facilities impractical.

B. MapReduce-MPI (MR-MPI)

MR-MPI is a MapReduce implementation on top of MPI that supports the logical *map-shuffle-reduce* workflow in four phases: `map`, `aggregate`, `convert`, and `reduce`. The `map` and `reduce` phases are implemented by using user callback functions. The `aggregate` and `convert` phases are fully implemented within MR-MPI but need to be explicitly invoked by the user. Figure 2 shows the workflow of MR-MPI. The `aggregate` phase handles the all-to-all movement of data between processes. Within the `aggregate` phase, MR-MPI calculates the data and buffer sizes and exchanges the intermediate KV pairs using `MPI_Alltoallv`. After the exchange, the `convert` phase merges all received KV pairs based on their keys.

Similar to traditional MapReduce frameworks, MR-MPI uses a global barrier to synchronize at the end of each phase.