A Comparative Survey of the HPC and Big Data Paradigms: Analysis and Experiments

HamidReza Asaadi †, Dounia Khaldi † and Barbara Chapman∗†
†Institute for Advanced Computational Science
Stony Brook University, Stony Brook, NY
Email: {hamidreza.asaadi, dounia.khaldi, barbara.chapman}@stonybrook.edu
∗Dept. of Computer Science
University of Houston, Houston, TX
Email: bchapman@uh.edu

Abstract—Many scientific data analytic applications need huge amounts of input, which can often consist of more than several TBs of data. This emphasizes the high I/O and processing/computational cost requirements of these algorithms. Tasks in these programs can induce more I/O operations than computations or the opposite. Hardware also includes nodes with large storage devices and/or nodes with sophisticated computational capabilities. To embrace the heterogeneity of the hardware systems in non-cloud and cloud environments, the issues of resource and job allocation in these environments need to be revisited. High-Performance Computing (HPC) models, under the leadership of MPI (plus OpenMP) parallel APIs, have mostly met users’ requirements in terms of high computational performance, while Big Data frameworks such as Spark have performed likewise in terms of high-level programming, resiliency and I/O handling. Therefore, in order to meet the specialized needs of scientists, there is a need for convergence between HPC and Big Data ecosystems.

This paper presents a data-supported, comparative survey of the main current HPC and Big Data programming interfaces, namely MPI, OpenMP, PGAS (OpenSHMEM), Spark, and Hadoop, and their software stacks. A comprehensive experimental study of these interfaces on a set of benchmarks, namely reduction and I/O microbenchmarks, the StackExchange AnswersCount benchmark, and PageRank Benchmark has been performed on a single platform in order to achieve a fair comparison. These experiments lead to a thorough discussion about whether the envisioned convergence is indeed needed or not, efficient or not, and in particular whether it is the best solution to tackle future computational challenges.

I. INTRODUCTION

Future exascale systems, with hundreds of times the compute capacity of today’s leading edge Petascale computers, are expected to emerge during the next decade. Processor cores in these systems want data delivery to be fast and efficient, which is difficult to achieve in large computations, like those in scientific and data analytics problems. With the growth in parallelism and system complexity, especially within individual compute nodes, portable programming approaches that offer intuitive programmability while ensuring scalability and fault-tolerance are necessary to support users from different scientific domains. Exascale Application Programming Interfaces (APIs) will need to facilitate the exploitation of exceptional amounts of concurrency in applications, enable the processing of significant amounts of data, and support several distinct architectures, including those based upon heterogeneous cores. The APIs and their implementations will need to carefully manage different kinds of memories within each node. Moreover, the need to conserve energy has led to increased focus on reducing data motion at all levels of the memory hierarchy, requiring a rethinking of algorithms as well as of the entire HPC software stack. Exascale execution software systems will need to ensure that jobs continue to run despite the occurrence of system failures and other kinds of hard and soft errors. Exascale APIs and their implementations must also support these requirements.

Legacy programming interfaces for expressing parallelism (in particular MPI [1], OpenMP [2] and OpenACC [3]) are being further developed to achieve some of these requirements. On the other hand, Big Data frameworks (in particular Spark [4] and Hadoop [5]) already fulfill fault tolerance and programmability requirements. However, it is still unclear which paradigm is a natural fit for expressing computations and handling data in a broad range of scientific application codes, nor the extent of the code modification required to do so. There is a tension between the need to reduce data motion and the potential to dynamically schedule and execute tasks; the role of the user in balancing these is also still debated.

Yet, is there a fundamental difference between HPC and Big Data or does the difference only reside in the application and software usage? While HPC mostly focuses on large computational loads, Big Data targets applications that need to handle very large and complex data sets; these data sets are typically of the order of multiple terabytes or exabytes in size. Big Data applications are thus very demanding in terms of storage, to accommodate such a massive amount of data, while HPC is usually thought more in terms of sheer computational needs. To help clear the issue, a discussion about the possible convergence between Big Data and HPC frameworks has been recently initiated [6] in order to enable application developers to express computations and data accesses conveniently and in a manner that allows the implementation, including the execution environment, to optimize the code for performance, energy, and reliability.

To provide evidence to help address this issue in an objective manner, our paper presents a survey and a comparison study of
the most significant current Big Data and HPC software stacks. This will include theoretical and experimental discussions using OpenMP, Hadoop, Spark, OpenSHMEM, and MPI. The paper will also provide proper definitions of the different related concepts. Unlike previous related work that uses a range of different platforms for testing each paradigm, our paper uses a single cluster machine and thus provides a fair comparison between the two stacks. The goal is to help remove confusion and misunderstanding of different concepts such as HPC vs. Big Data, computing center vs. data center, etc. Finally, and more generally, our work draws a possible roadmap for the convergence between the two paradigms and helps language designers come up with improved or new paradigms to leverage exascale challenges.

After this introduction, we survey existing HPC and Big Data commonly-used APIs in Section II. We describe the main exascale parallelism issues in Section III. Big Data and HPC software stacks are introduced in Section IV. Experimental results using microbenchmarks and the PageRank benchmarks are shown in Section V. We follow-up with a discussion about various aspects of each paradigm based on these experimental results, in Section VI. We review related work in Section VII and finally discuss future work and conclude in Section VIII.

II. HPC AND BIG DATA COMMONLY USED APIs

A. OpenMP

OpenMP [2] is an HPC application program interface providing a multi-threaded programming model for shared memory parallelism. It uses directives to extend sequential languages with bindings for C, C++, and Fortran. While OpenMP (from version 2.5 onward) describes OpenMP codes as consisting of a collection of tasks (OpenMP 3.0), not all of its constructs lend themselves to exascale levels of execution. Task dependencies using the depend clause were introduced in OpenMP 4.0 for describing data flow execution. Moreover, the target construct creates tasks to be executed on accelerators in an offload mode. OpenMP currently has few features to map its tasks and data to components of the system. Yet it still includes synchronization features that do not scale and its features for expressing locality and affinity are limited. Today, it cannot target multiple system nodes.

B. MPI

MPI [1] is an HPC library specification for message passing used to program shared and distributed memory systems with bindings for C, C++, and Fortran. Both point-to-point and collective communications are supported. The starter process may be a separate process that is not part of the MPI application, or the rank 0 process may act as a starter process to launch the remaining MPI processes of the MPI application. MPI processes are created when MPI_Init is called, which also initializes MPI_COMM_WORLD communicator for all processes, driving various communications. A communicator determines the scope of communications. Processes are terminated using MPI_Finalize.

The MPI 1.0 standard was released in 1994 and provided functionality for performing two-sided and collective communications. The support for one-sided communications has been added to the MPI 2.0 standard released in 1997. However, there were several missing features in the MPI 2.0 RMA specification. In September 2012, the MPI 3.0 was introduced; it included a significant update to the semantics and features of the RMA interface in order to provide better support for one-sided and global-address-space models. RMA capability has been added to MPI via the notion of windows. Any memory segment that is part of a window can be remotely accessed by other processes via put/get RMA operations.

Initially, MPI supported sequential I/O approach, in which only the master process reads from files and send chunks of the files to other processes for processing. In the case of write operation, all processes send data to the master process, and then the master process writes the collected data to files. This approach is limiting in terms of performance and scalability for many applications. Parallel I/Os, on the other hand, enables the participation of multiple processes in reading data from or writing data to a common file in parallel. MPI parallel I/O was introduced in MPI 2.0. It provides a mechanism for specifying collective operations and noncontiguous data layout in memory and files. In this paper, we use these routines in our MPI benchmarks to implement I/Os.

C. PGAS Programming Model (OpenSHMEM)

In the Partitioned Global Address Space (PGAS) [7] programming model, there is a partitioned global memory that may be accessed in a manner akin to shared memory, but with the programmer being able to distinguish between local and nonlocal accesses of this memory. The model also typically allows for data distribution across the memory partitions. Thus, data movement may be explicitly controlled by the programmer. Additionally, the model may also provide provisions for defining locality domains within the system and spawning tasks at specified locales. Some of the APIs that are designed for the HPC community and are based on the PGAS model include Global Arrays [8], CAF [9], UPC [10], Chapel [11], X10 [12] and OpenSHMEM [13]. Note that these PGAS APIs have a growing user community. OpenSHMEM [13], an HPC library-based API for the Partitioned Global Address Space (PGAS) programming model with bindings for C, C++, and Fortran, is the result of a community effort to standardize the SHMEM API. It is designed with the chief aim of performance, exploiting support for the Remote Direct Memory Access (RDMA) mode available in modern network interconnects. OpenSHMEM programs follow an SPMD-like execution model, where all processing elements (PEs) are launched at the beginning of the program; each PE executes the same code and the number of processes remains unchanged during execution. OpenSHMEM PEs are initialized when the shmem_init function is called. OpenSHMEM is particularly advantageous for applications with many small put/get operations and/or irregular communication patterns across compute nodes, since it offloads...
communication operations to the hardware whenever possible. As a result, OpenSHMEM is particularly well suited for applications that use algorithms in the fields of graph traversal, sorting, and codes with dynamic behavior (e.g., recursive data structures).

D. Hadoop MapReduce

Hadoop MapReduce is a subproject of Apache Hadoop [5] that provides an implementation of the MapReduce programming model for large-scale data processing. A MapReduce program consists of fine-grained Mapper and Reducer tasks. Both tasks have key-value pairs as input and output. The Mapper has a map method that transforms the input pairs into any number of intermediate pairs. The Reducer has a reduce method that transforms the intermediate pairs and aggregates them into any number of output pairs. The main features of the MapReduce model are automatic load balancing, and recovery from failed tasks. Indeed, failed tasks are re-executed automatically. Apache Hadoop MapReduce is an open-source implementation of Google MapReduce, which was introduced in [14].

Each query in Hadoop reads data from disk and runs as a separate MapReduce job. However, Spark (see next subsection) enables in-memory iterative processing. Thus, the user can query repeatedly on a dataset without having to perform intermediate disk operations.

E. Spark

Spark [15] is an increasingly successful framework for Big Data computations. The main feature of Spark is the RDD (Resilient Distributed Data) [16] abstraction. RDDs are exposed in the Spark API where each dataset is represented as a read-only object and transformations are invoked using methods on these objects. RDDs are partitioned across a set of nodes. Spark implements RDDs using a form of graph-based representation in order to track data across a wide range of transformations. RDDs have actions, which return values, and transformations, which return pointers to new RDDs; an RDD won’t be materialized unless an action is called on that RDD. This is called lazy evaluation and it allows Spark to tune the whole execution based on data locality and stage dependencies.

RDDs are fault tolerant by nature because they provide an interface based on coarse-grained transformations (e.g., map, filter and join) that apply the same operation to multiple data items. Tracking their use via the logs of transformations applied to RDDs builds a dataset (its lineage) rather than the actual data. If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute just that partition. Thus, costly replication is avoided.

B. Memory Model

Several different approaches have been used to logically partition a global address space and map it to system resources, including the distribution of arrays (e.g., HPF, UPC) and their replication (e.g., OpenSHMEM). Typical distribution strategies included blocked, round-robin and user-defined distributions, e.g., via libraries such as in Chapel [17]. On future platforms, with potentially greater heterogeneity and specialization of computing cores, data mappings will need to take into account the likely range of placements of the tasks that use it.

Spark RDDs (Resilient Distributed Datasets) [16] are read-only objects that are partitioned across a set of nodes. RDDs bear a certain resemblance to the PGAS model because each RDD is divided into several partitions and the Spark engine knows which nodes each partition is on; an array in PGAS is distributed into portions and knows which nodes each portion of the array is on as well. The map transformation on RDDs is similar to parallel loops in HPC models.

C. Storage and I/O

Many scientific data analytic applications are becoming I/O-bound in modern systems, such as seismic algorithms, which are examples of applications that need huge amounts of input data. For instance, for the Kirchhoff migration algorithm [18], a widely-used depth migration computation for obtaining 3D images of the earth subsurface, the number of traces can sometimes be over 500 million, which consists of several
TBs of data. This will require giving a special attention to storage and I/Os. In fact, storage systems can have a profound impact on future algorithm design and system development in HPC and Big Data. Parallel I/Os have been already adopted in MPI in order to write/read to different disks in parallel. However, parallel I/O does not solve the problem of storage contention if the application is embarrassingly parallel and is reading/writing huge data at the same time. Spark provides an API for RDDs where save can be used to output data sets into a storage system. Also, the persist method is used to indicate which RDDs will be reused in the future; Spark keeps them in memory or spill them to disk if there is not enough RAM. Another work on those lines where non-volatile memory was exploited as a secondary memory partition using NVMalloc library is presented in [19]. Thus, users can explicitly allocate and manipulate memory regions therein. A memory hierarchy resource-aware read algorithm that relieves read contention issues on SSDs is presented in [20]. In this latter, thresholds are set to control the read contention on SSDs depending on the number of parallel readers and total read size.

Moreover, the Legion [21] programming interface explicitly addresses memory hierarchy issues. Its key features are tasks and logical regions. A logical region consists of a set of objects to describe the organization of data and to make relationships useful for locality and independence explicit. They may be partitioned into disjoint or aliased (overlapping) subregions that are mapped to the memory hierarchy. Tasks specify the regions they use and mode of access. Regent [22], which relies on compiler translation to convert its features into Legion, also provides tasks and logical regions to the application developer, but task dependencies are determined by the implementation. The Regent compiler determines the necessary communications and synchronizations and generates code for execution by the Legion runtime. The constructs are not very intuitive and, so far, published results have used no more than 8 nodes.

D. Handling Heterogeneous Cores

OpenMP 4.0 specified the target and target data constructs for offloading computation to target devices, or accelerators [23], and to define and manage the data environment on the target device, respectively. Given the very high cost of transferring data between host and device on existing platforms, and the scarcity of device memory, both OpenACC and OpenMP have developed relatively complex interfaces for managing allocations, transfers, updates and synchronization of data. In addition, there is a difference between the current generation of accelerators (e.g., Nvidia GPUs, “Knight’s Corner” (KNC) [24]) and those in which the device memory is unified with the host’ (e.g. AMD GPUs, “Knight’s Landing” (KNL) [25]). So the trend toward heterogeneity of the cores, and very powerful attached accelerators, greatly exacerbates the programming challenge.

Spark, on the other hand, is a CPU-only computing platform. However, there has been multiple works to support GPUs using a Spark-like framework. For instance, HeteroSpark [26] is a GPU-accelerated, heterogeneous architecture integrated within Spark that combines the power of GPU and the scalability of Spark. Also, cuSpark [27] offers primitives just like Apache Spark. It follows the lazy execution pattern of Spark to reduce data movement and uses Cuda streams to hide data movement latency. In these two works, there is no new syntax specific to GPUs added to Spark; the implementations take care of everything, especially data partitioning, so that data can fit in the CPU memory, since it is much larger than the GPU’s memory.

IV. Big Data and HPC Software Stacks

A flexible and integrated software stack is essential to building efficient and persistent parallel computing cluster systems. Such a stack is the set of software, including operating system, provisioning, remote console/power management, cluster monitoring, parallel file system and scheduling, and development and performance monitoring tools that enable users to work with the target clusters. The underlying software stacks for HPC and Big Data are fundamentally different, mainly due to the differences present in their target class of applications [28]. Each stack component is, in fact, different in the two stacks; we provide examples below, and summarize the two stacks in Figure 1.

**Fig. 1. HPC and Big Data Software Stacks**

The following provides examples for each layer/component for the two stacks.

- **APIs:** Fortran, C, and C++ are the commonly used languages in HPC. Java, Scala, R, Python etc. are used mostly in Big Data solutions.
- **Debugging and profiling tools:** There is more transparency in HPC models when it comes to the debugging of a distributed application. Multiple tools such as Scalasca [29], Tau [30], etc. can be used for debugging/profiling HPC applications. However, there is no sufficient tooling in the Hadoop ecosystem unless for special cases (e.g., running Spark in interactive shell mode and using Spark standalone cluster).
- **Filesystem:** Most of the Big Data frameworks that are built around the Hadoop ecosystem support a variety of input formats, including different databases and queueing...
systems. However, an essential part of Hadoop is its fully-distributed filesystem, called HDFS. HDFS splits files into blocks and distributes them over different network nodes in order to reach higher availability and improved data read performance while serving high-level web services. On the other hand, the HPC solutions mostly rely on NFS filesystems in order to provide access to the same data from different locations on clusters.

- **Resource manager:** YARN, Mesos etc., are used in Big Data, while Slurm/Torque is used in HPC.
- **Operating system:** Many Big Data frameworks are build using JVM-based languages, which allow them to be ported to any OS with JVM support. HPC frameworks usually rely on lower-level languages such as C and C++ that require them to recompile applications for different operating systems.
- **Interconnect hardware:** One of the main goals of Hadoop was to use conventional hardware in order to make it easier and less expensive to scale. Therefore all standard distributions of Hadoop-based Big Data frameworks use conventional Ethernet sockets in order to communicate data sets and perform orchestration across clusters. However, HPC software makes efficient use of RDMA technology such as Infiniband interconnect.

V. EXPERIMENTAL RESULTS

This section discusses experimental results of some HPC and BigData frameworks, namely Spark, Hadoop, MPI, and OpenMP, on three sets of benchmarks: (1) reduction and I/O microbenchmarks, which contain code designed to test the performance of reduction and parallel file read operations, (2) the StackExchange AnswersCount benchmark, which counts the average number of answers for questions asked on stackexchange.com, and (3) PageRank Benchmark.

A. Experimental Setup

All our tests are conducted on Comet [31], the SDSC’s newest HPC resource cluster that contains 1984 nodes. Every node on the Comet cluster has the characteristics highlighted in Table I.

<table>
<thead>
<tr>
<th>Processor type</th>
<th>Intel Xeon E5-2680v3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sockets #</td>
<td>2</td>
</tr>
<tr>
<td>Cores/socket</td>
<td>12</td>
</tr>
<tr>
<td>Clock speed</td>
<td>2.5 GHz</td>
</tr>
<tr>
<td>Flop speed</td>
<td>960 GFlop/s</td>
</tr>
<tr>
<td>Memory capacity</td>
<td>128 GB DDR4 DRAM</td>
</tr>
<tr>
<td>Interconnect</td>
<td>Hybrid Fat-Tree, FDR InfiniBand</td>
</tr>
<tr>
<td>Local scratch memory</td>
<td>320 GB SSD</td>
</tr>
</tbody>
</table>

TABLE I

On Comet, we used the following frameworks: the default Spark running with IP over InfiniBand (IPoIB) (v.1.5.2), Spark-RDMA (v.0.9.1), MPI (OpenMPI implementation v.1.8.4), Hadoop MapReduce (v.2.6.0), and OpenMP (GCC v.4.9.2). Note that Scala (v.2.10.4) was used to implement BigDataBench PageRank, while Java was used for all other Hadoop and Spark codes used in this section (Oracle JDK 7). Also, except for the StackExchange AnswersCount benchmark, described in Section V-C, other test cases have been performed only on Spark (and its RDMA variant when applicable) and MPI frameworks, to give a more focused analysis.

B. Microbenchmarks

The microbenchmarks used in this paper are designed to compare MPI and Spark frameworks in terms of computation and I/O performance.

1) Computation Performance - Reduce Microbenchmark: We used the MPI reduce microbenchmark from the OSU MPI Microbenchmarks [32]. This microbenchmark performs reduction on an array replicated across all processes and returns an array of the same size: each element of the result array is the sum of all the corresponding elements across all the processes. For Spark, we developed an equivalent version in terms of reduction operations. Note the fact that the number of reduction operations for MPI is equal to the number of processes \times size of the array; in Spark, this number should be equal to the size of the array being parallelized. Therefore, the size of the array being reduced in Spark should be equal to the number of processes \times size of the array in MPI. This is because Spark returns the sum of all the distributed elements of an input array in one scalar. Figure 2 provides a Spark code snippet containing the reduction operations.

```java
// 'size' is equal to number_of_processes * MPI_array_size
Float[] arrayofZeros = new Float[size]
// initialization code
List<Float> listOffZeros = Arrays.asList(arrayOfZeros);
JavaRDD<Float> listOfZerosRDD = sc.parallelize(listOffZeros);
Float result = listOfZerosRDD.reduce(new Function2<Float, Float, Float>() {
    public Float call(Float a, Float b) {
        return a + b;
    }
});
```

Fig. 2. Spark Code Snippet for Reduce Microbenchmark (Java)

Figure 3 shows the reduce microbenchmark results on 8 nodes and 8 processes/node for Spark and MPI. MPI outperforms Spark because of the asynchronous and non-blocking nature of MPI runtime routines. Also, the reduction and communication algorithms used in MPI implementations are well tuned depending on the array size and other parameters...
used in the reduction. On the other hand, the use of the driver program in Spark to ensure completion and success of data distribution causes an extra overhead before performing the actual reduction. Note that the use of Spark RDMA does not significantly improve the results since this microbenchmark does not require shufflings large amount of data around the network and therefore the effect of using the RDMA shuffle engine is marginal. In fact, the MPI version takes full advantage of RDMA since MPI uses Infiniband interconnect for all types of communication between nodes. However, the Spark RDMA implementation uses Infiniband only for data shuffling purposes while the orchestration messages use conventional Java sockets.

2) I/O Performance - Parallel File Read Microbenchmark: This microbenchmark reads an input file (8GB and 80GB files in this experiment) in parallel (using provided mechanisms in the frameworks) and reports the required execution time. Since Spark does not materialize RDDs unless an action is called over them, we added a counting operation to both MPI and Spark implementations to make the comparison fair.

In the case of Spark, we used two configurations: (1) the input file is hosted on HDFS (for which the data is stored on the scratch filesystem (SSD disks) and the replication level is set to the number of nodes available); (2) the input file is copied to the local scratch filesystems of the allocated nodes. This is useful to measure the impact of having HDFS as an additional layer for data access. For MPI implementation, we replicated the input file to local scratch filesystem of every node. Results for this microbenchmark are provided in Table II.

![Graph](image)

**Fig. 3. Reduce Microbenchmark Results for 64 Processes (8 processes/node)**

Spark uses HDFS as preferred filesystem; this gives Spark the ability to continue running jobs in case of a filesystem fault. In other words, unlike the case of local files, failure at HDFS level is handled automatically at that level and will not propagate to the application level, i.e. Spark code. As depicted in the results, we observed a 25% overhead in using HDFS compared to the local filesystem, but this overhead is acceptable especially when the jobs are long-running and the failures in the filesystem could result into loss of data or job reset.

The flexibility and autonomous adaptation provided by services like HDFS do not completely waive the need for manual tuning and management by users. One of our observations during our experiments on this microbenchmark was that the Spark results do not scale well when we increase the number of executor nodes. Since HDFS promises even distribution of data blocks (distributed over all cluster nodes with a replication factor of 3, by default), Spark should not face any data locality problems. However, we found that some data blocks are not local to any of the executor nodes and therefore transferred over the network, leading to a running time overhead. We found out that, in some cases, the Spark cluster manager does not evenly distribute the executors among the nodes. In order to solve this problem, we increased the replication factor of HDFS and made it equal to the number of executor nodes in order to ensure that all executors are local to any requested data block.

C. StackExchange AnswersCount Benchmark

StackExchange (see http://stackexchange.com) is a network of public questions and answers website. We used a subset StackExchange question and answer dataset in text format, counted the average number of answers for all the available questions and measured the execution time. We show in Figure 4 the comparison results between the OpenMP, MPI, Spark, and Hadoop versions we wrote for this benchmark. Note that we used 8 processes per node. In the case of OpenMP, since it can only run on a single node, we only provide results for 8- and 16-core configurations.

We used an 80 GB dataset file for our tests in order to make this benchmark an I/O intensive test. As shown in Figure 4, Spark and Hadoop scale better than OpenMP and MPI when processing this data size. Also, Hadoop relies heavily on disk operations and persists intermediate results on disk. Spark, in contrast, minimizes disk operations (by caching data in memory if possible), moves data around, i.e., performs shuffling, and introduces lazy calculations for RDDs for data locality. This justifies the noticeable difference between the Hadoop and Spark execution times.

As for MPI, note that we used MPI parallel I/Os to implement this benchmark. Since the file size is 80 GB, MPI could not support this amount of data unless the number of processes is greater than 40. This is due to the fact that MPI I/O routines (MPI_File_read_at_all) use an integer as a parameter for expressing the size of the chunks assigned to each process. This is a limitation in the case of large files (bigger than 2GB,

<table>
<thead>
<tr>
<th>Model</th>
<th>Filesystem</th>
<th>Spark on HDFS (scratch fs)</th>
<th>Spark on HDFS (scratch fs)</th>
<th>MPI (scratch fs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8GB execution time</td>
<td>8.2s</td>
<td>6.5s</td>
<td>1.2s</td>
<td></td>
</tr>
<tr>
<td>80GB execution time</td>
<td>46.751s</td>
<td>29.9s</td>
<td>14.16s</td>
<td></td>
</tr>
</tbody>
</table>
which is the size of the largest file that can be represented by
\texttt{MAX\_INT} in C). In this case, specifically when using an input
file of 80 GB of the StackExchange AnswersCount benchmark,
we had to use more than 40 processes to make it working.
This will divide the file into chunks of 2GB or less to be used
by each process. This makes MPI non-scalable and shows a
fundamental issue with the parallel I/Os of MPI that cannot
be overcome by using MPI-3 features.

\begin{figure}[h]
\centering
\includegraphics[width=\linewidth]{Fig4.pdf}
\caption{StackExchange AnswersCount Benchmark Results (8 processes/node)}
\end{figure}

\subsection*{D. PageRank Benchmark}

In this experiment, we compare the performance of the
MPI and Spark implementations of the PageRank algorithm
over a dataset containing 1,000,000 vertices. We used Big-
DataBench [33] as the reference implementation for both
frameworks. As shown in Figure 6, MPI code performs almost
the same while Spark is highly scalable. Also, using the Spark
RDMA implementation does not improve the performance
of our tests. This is due to the fact that the BigDataBench
implementation of PageRank is tuned to avoid data shuffling.
In this tuned case, the materialized RDDs are persisted into
memory after each step of execution (see Figure 5). This
simple change does not only improve the performance of
the Spark implementation by a factor of 3, but it makes the
implementation of data shuffling over RDMA insignificant
because each stage keeps its data local to the worker node.
However, we found that Spark RDMA performs better when
the HiBench [34] implementation of PageRank is used. In
order to test this, we also executed the same test using the
HiBench implementation using both the default and RDMA
versions of Spark (see Figure 7). These results suggest that,
when the rate of data shuffling is high and with the increase
in the number of nodes, the Spark RDMA implementation
outperforms the default implementation.

\begin{figure}[h]
\centering
\includegraphics[width=\linewidth]{Fig6.pdf}
\caption{BigDataBench PageRank Benchmark Results (16 processes/node)}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\linewidth]{Fig7.pdf}
\caption{HiBench PageRank Benchmark Results (16 processes/node)}
\end{figure}

VI. DISCUSSION

This section discusses findings and draw conclusions with
regard to maintainability, execution flow, performance, scala-

```scala
var ranks = links.mapValues(v => 1.0).persist(StorageLevel.MEMORY\_AND\_DISK)
for (i <- 1 to iters) {
  val contribs =
    links.join(ranks).values.flatMap {
      case (urls, rank) =>
        val size = urls.size
        urls.map(url => (url, rank / size))
    }.persist(StorageLevel.MEMORY\_AND\_DISK)
  // This caching is not done in HiBench Implementation
  ranks = contribs.reduceByKey(\_ + \_).mapValues(0.15 + 0.85 * \_)
}
```

```
D. PageRank Benchmark

In this experiment, we compare the performance of the
MPI and Spark implementations of the PageRank algorithm
over a dataset containing 1,000,000 vertices. We used Big-
DataBench [33] as the reference implementation for both
frameworks. As shown in Figure 6, MPI code performs almost
the same while Spark is highly scalable. Also, using the Spark
RDMA implementation does not improve the performance
of our tests. This is due to the fact that the BigDataBench
implementation of PageRank is tuned to avoid data shuffling.
In this tuned case, the materialized RDDs are persisted into
memory after each step of execution (see Figure 5). This
simple change does not only improve the performance of
the Spark implementation by a factor of 3, but it makes the
implementation of data shuffling over RDMA insignificant
because each stage keeps its data local to the worker node.
However, we found that Spark RDMA performs better when
the HiBench [34] implementation of PageRank is used. In
order to test this, we also executed the same test using the
HiBench implementation using both the default and RDMA
versions of Spark (see Figure 7). These results suggest that,
when the rate of data shuffling is high and with the increase
in the number of nodes, the Spark RDMA implementation
outperforms the default implementation.

VI. DISCUSSION

This section discusses findings and draw conclusions with
regard to maintainability, execution flow, performance, scala-

```scala
var ranks = links.mapValues(v => 1.0).persist(StorageLevel.MEMORY\_AND\_DISK)
for (i <- 1 to iters) {
  val contribs =
    links.join(ranks).values.flatMap {
      case (urls, rank) =>
        val size = urls.size
        urls.map(url => (url, rank / size))
    }.persist(StorageLevel.MEMORY\_AND\_DISK)
  // This caching is not done in HiBench Implementation
  ranks = contribs.reduceByKey(\_ + \_).mapValues(0.15 + 0.85 * \_)
}
```

Fig. 5. Spark Code Snippet from the BigDataBench implementation of
PageRank in Scala

Fig. 6. BigDataBench PageRank Benchmark Results (16 processes/node)

Fig. 7. HiBench PageRank Benchmark Results (16 processes/node)

bility, and fault tolerance based on the experimental results we
obtained on different HPC and Big Data frameworks.

A. Maintainability

With the increase of complexity in distributed tasks and
therefore size in codebases, quality attributes such as maintain-
ability and readability become more important. In this work,
we analyzed the code implemented for our benchmarks from two perspectives: first, the total number of lines of code and, second, the amount of boilerplate code required to run the distributed code. We show the result of this analysis in Table III. While we did not use the same programming language for all implementations (C/C++ for MPI and OpenMP, and Java for Hadoop and Spark), we found that the number of code lines is highly affected by the parallel API being used.

The approach used in OpenMP by inserting pragmas to code the benchmarks resulted in the smallest code sizes and a small amount of boilerplate code. However, the semantics expressed by these directives are decoupled from the actual parallel code. Note that, today, none of the commonly used IDEs support OpenMP-specific directives and thus it is not possible to use features such as intelligent code generation.

MPI uses library function calls to describe parallelism. Using this approach, the developers have to take care of the detailed flow of the application to ensure correct data communication and synchronization in the application (e.g., by using MPI_Barrier, while avoiding deadlocks).

In Hadoop, the execution path is not made explicit in the code. The developer has no control on how the map and reduce classes will be executed. The mapper and reducer instances have to be submitted to the Hadoop engine, which will decide at run time about their interaction and execution flow. The Spark API proceeds using a different approach. First, the transformations are scoped and encapsulated in methods. Then, the logical execution path matches the actual code flow.

Regarding setup cost of the different frameworks, Spark is built on top of the Hadoop ecosystem. In such enterprise architecture, a number of runtime components work in cooperation and each of the components can be deployed and managed separately. This layered architecture allows Spark to be easier to setup out-of-the-box for its users. At the same time, advanced users can simply modify and tune all the participating components from JVM to filesystem and cluster managers to obtain the desired resource allocation.

B. Execution Control Flow

MPI routines provide low-level control of the execution flow to developers. In fact, MPI developers write code for several processes imperatively. However, Spark uses a more descriptive approach for its API. In other words, while MPI code is directly executed on target machines, Spark code is parsed and managed by the Spark driver program and code segments are then submitted to the cluster machines for execution. While Spark’s approach provides a more manageable and high-level API, the MPI developers have extreme control over the code being executed, which helps design code for border cases with much higher performance. In Spark, on the other hand, there is no chance of intercommunication of executors at run time, except for simple constructs such as Accumulators and Broadcast variables.

C. Performance and Scalability

HPC frameworks perform better when the benchmark is computationally intensive. On the other hand, Spark handles better intensive I/Os and the distribution of big files over the network.

While using high-performance interconnects (Infiniband) in order to shuffle data around the executors/processes is an important factor to improve performance, we found that this may not be the largest part in play in many cases such as in the tuned version of PageRank. In fact, comparing the results for Spark-RDMA (which uses RDMA only for data shuffling) with default Spark and also MPI (that uses RDMA) showed that, in the case where the volume of data movement around the network is relatively low, MPI still performs better; using Infiniband does not affect significantly the overall performance of the Spark code. One future direction might be to implement all Spark communications using RDMA and not only the data shuffling operations.

Caching data in memory has a big benefit as well. As seen in the PageRank results, explicitly asking Spark to cache intermediate results via a simple API call resulted in large speedups. While Spark API eliminates a lot of distribution-specific coding effort (that is needed in MPI and other HPC languages), the mere existence of API calls such as persist() shows the possibility for future work on finding ways to automate such distribution decisions as well.

Finally, we observed better scalability from Spark compared to MPI. We think this is due to the fact that I/O implementation in MPI still needs better handling and more optimizations to meet the requirements of data-intensive scientific applications.

D. Fault Tolerance

While most distributed frameworks (such as MPI) use different checkpointing/restarting algorithms to handle faults that occurred during execution, Spark stores a lineage graph for its RDDs and reruns all tasks required in order to materialize lost RDDs. While this approach may result in computation replication, it can address all types of failure by assigning the task to another available node. For Spark however, the master node is a single point of failure and cannot be recovered.

Spark also benefits from using external data sources (like HDFS) in order to reach higher availability. In the Parallel File Read Microbenchmark, we discussed how using HDFS makes data node failures transparent to Spark at job-level. Note that Spark supports different I/O methods, which boosts the portability of the framework.
VII. RELATED WORK

Spark uses conventional Java sockets in order to shuffle data partitions across clusters. Lu et al. [35] implemented a new shuffle engine for Spark that uses RDMA technology to send and receive data partitions over networks. The main important characteristics of this implementation are: (1) the new engine affects only the data shuffling between nodes and not coordination messaging and other commands that are exchanged in the cluster; (2) this engine has been developed as a plugin for Spark and less than 100 lines of the Spark codebase has been modified in order to be integrated; (3) the new engine uses a Staged Event-Driven Architecture (SEDA). Their evaluation shows an improvement in performance using RDMA (in contrast to using IPoIB) from 20% to 83% based on the cluster and the number of cores being used. Note that this is the implementation we used in our experiments for Spark RDMA. The paper does not discuss the reason why there is a large gap between performance improvements observed when using different clusters.

A scalable implementation for Map-Reduce in MPI using MPI_Scatter and MPI_Reduce operations has been proposed in [36]. The paper also offers an optimization to this implementation by using non-blocking collective operations added in MPI-3. As for performance evaluation, it provides a comparison between MPI-based Map-Reduce functions with and without optimization and shows a performance improvement of about 25%. However, this paper does not provide any comparison to reference implementations of Map-Reduce such as Hadoop. They also discussed the currently missing features in MPI so that it can be used to implement a complete Map-Reduce system. On top of this list is the way MPI handles faults and the fact that MPI does not recover from faults at run time. The other drawback is the lack of support for arbitrary data types and sizes in MPI reduce operations.

Plimpton et al. [37] used MPI_Send and MPI_Receive basic functions to implement a fully synchronized Map-Reduce engine in MPI. They implemented an out-of-core method in order to handle the cases where data partitions do not fit into the page size allocated to the cores. They have implemented a number of graph algorithms, including PageRank, using Map-Reduce in the framework and compared the results with Hadoop and the Trillinos linear algebra library. While their implementation shows more than 100x improvement over standard Hadoop, it is considerably slower than special-purpose libraries such as Trillinos. Moreover, this paper discusses the details of implementing simple operators required by a MapReduce framework in MPI, but it does not provide any suggestion to deliver key features such as checkpointing and fault tolerance.

The subject of comparing the HPC and Big Data paradigms has been addressed in a recent paper [38], where a comparison analysis between the ecosystem of high-performance computing and the Apache-Hadoop paradigm is provided. It used the clustering benchmark k-means to evaluate the two paradigms. However, for the experiments, it used a range of different platforms for each paradigm, which makes it difficult to judge or compare both. The paper uses Hadoop, Mahout, MPI, Python, HARP, and Spark to illustrate the performance differences. Our work in this paper is different because we analyze/criticize both paradigms using, admittedly, a more HPC-heavy angle; unlike [38], we focus more on HPC models (MPI, OpenSHMEM, and OpenMP) and study how one can make them converge to Hadoop or Spark, or the other way around. Also, we use a single platform (Comet) for all our experimental results.

VIII. CONCLUSION

This paper presented a survey of the main existing HPC and Big Data programming interfaces, namely MPI, OpenMP, PGAS (OpenSHMEM), Spark, and Hadoop, and their software stacks. We performed a comprehensive experimental study of these interfaces on a set of benchmarks: (1) reduction and I/O microbenchmarks, which contain code designed to test the performance of reduction and parallel file read operations, (2) the StackExchange AnswersCount benchmark, which counts the average number of answers for questions asked on stackexchange.com, and (3) PageRank Benchmark. We completed this experimental analysis by a comprehensive discussion section that included a set of conclusions and findings with regard to maintainability, execution flow, performance, scalability, and fault tolerance.

The overall goal of this paper is to initiate discussions and thinking on why and how the HPC and Big Data paradigms can be made to converge. The main takeaway lesson is that many insights can be gained from both paradigms. These ideas should be investigated and deployed to help construct a programming model that will handle both computational and data intensive applications while meeting users’ expectations with regard to programmability, performance portability, and fault tolerance.

Future work will address applying fault tolerance and I/O handling from Spark to HPC models. We are also looking into proposing a new HPC directive-based programming language where data distribution and handling are inspired from Spark.

ACKNOWLEDGMENTS

This research work is supported by the US Office of the Assistant Secretary of Defense for Research and Engineering (OASD(R&E)) under agreement number FA8750-15-2-0119. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the US NSF, or the Office of the Assistant Secretary of Defense for Research and Engineering (OASD(R&E)) or the U.S. Government.

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575.