The Best of Both Worlds: Big Data Programming with Both Productivity and Performance

Fan Yang Yuzhen Huang Yunjian Zhao Jinfeng Li Guanxian Jiang James Cheng
Department of Computer Science and Engineering
The Chinese University of Hong Kong
{fyang, yzhuang, yjzhao, jfli, gxjiang, jcheng}@cse.cuhk.edu.hk

ABSTRACT

Coarse-grained operators such as *map* and *reduce* have been widely used for large-scale data processing. While they are easy to master, over-simplified APIs sometimes hinder programmers from fine-grained control on how computation is performed and hence designing more efficient algorithms. On the other hand, resorting to domain-specific languages (DSLs) is also not a practical solution, since programmers may need to learn how to use many systems that can be very different from each other, and the use of low-level tools may even result in bug-prone programming.

In [7], we proposed *Husky* which provides a highly expressive API to solve the above dilemma. It allows developers to program in a variety of patterns, such as MapReduce, GAS, vertex-centric programs, and even asynchronous machine learning. While the Husky C++ engine provides great performance, in this demo proposal we introduce PyHusky and ScHusky, which allow users (e.g., data scientists) without system knowledge and low-level programming skills to leverage the performance of Husky and build high-level applications with ease using Python and Scala.

Keywords

Distributed system; Data-parallel processing; Programming model

1. INTRODUCTION

Distributed programming in a declarative and functional framework (such as Spark [10] and Flink [1]) has become more and more popular in recent years, as it allows users to create big data pipelines using simple coarse-grained operators such as *map* and *reduce*. However, these operators over-simplify a lot of fine-grained computation in many classes of algorithms (e.g., machine learning, graph analytics) and hinder developers from gaining more fine-grained control on how computation is performed. Thus, when performance is critical or the resource cost is a concern (e.g., paying for resources on the cloud), people usually favor specialized architectures (e.g., Parameter Servers for machine learning) over directly using coarse-grained functional operators. On the other hand, resorting to specialized systems incur a steeper learning curve and more configuration/maintenance efforts.

SIGMOD'17, May 14-19, 2017, Chicago, IL, USA

DOI: http://dx.doi.org/10.1145/3035918.3058735

The Husky framework (http://www.husky-project.com/) aims to solve the above problems. It provides much more flexibility than existing frameworks such as Spark and Flink, allowing users to employ different programming paradigms (called *patterns* in [7]), or even create customized patterns for specific problems or performance-critical paths, while keeping a simple set of core primitives and a variety of algorithm libraries. However, it will be desirable if users can also program in high-level languages such as Python and Scala, and apply various mature library packages (e.g., visualization, crawling) supported by these languages, in order to achieve lower development cost and higher productivity.

To this end, we started two open source projects, PyHusky and ScHusky, which developed the Python and Scala frontend of Husky, respectively. PyHusky and ScHusky allow users (e.g., data scientists) with little or no system programming background to exploit the advantages (e.g., succinctness and high productivity) of high-level languages, while making little compromise on performance with the powerful Husky backend running underneath.

PyHusky and ScHusky have their own data collection representations, allowing Python-specific and Scala-specific operations to be applied on them. Users write their programs declaratively just like in Spark and Flink, forming a DAG of operators. They are lazily evaluated and executed using a scheduler. However, the most important highlight is that PyHusky and ScHusky are able to leverage functionalities in native Husky—the scheduler will move execution to Husky when it finds that the computation involves no Python/Scala-specific operations, or run some parts of the computation in Python/Scala side and then shift to native Husky whenever possible for best performance.

The Husky Demo. While in [7] we presented the concepts and implementation details of Husky, in this demo we focus on showing SIGMOD attendees how PyHusky and ScHusky unleash the power of Husky in a much easier and user-friendly way. SIGMOD attendees will be able to use high-level languages (Python and Scala) to leverage a variety of programming paradigms (e.g., Parameter Servers, MapReduce, Asynchronous ML, Pregel, and so on) just like in native Husky. In particular, attendees will see how PyHusky and ScHusky bring the best of both worlds (i.e., high productivity of PyHusky/ScHusky and high performance of Husky) together by seeing how quickly it is to build end-to-end high-level applications from scratch using PyHusky and ScHusky, such as a crawler with visualizer and a streaming application with real-time analysis, while at the same time enjoying high performance and scalability of native Husky.

We will also dive deeper and elaborate the gain of shifting performance-critical paths of a big data application to native Husky. In addition, we will also show interested developers more about the in-

ternals and how developers can expose a customized Husky library to PyHusky/ScHusky by coding a small piece of binding code.

2. SYSTEM OVERVIEW

We first introduce some key ideas in Husky and its execution engine. Then we show some performance figures to give readers an understanding of its performance. After that we introduce programming in PyHusky and ScHusky, the overall architecture, and how it works with Husky.

2.1 Core Concepts

Husky is an in-memory system running on a cluster of machines based on a shared nothing architecture. Each machine can run multiple workers, and each worker performs computation on its own partitions of objects. There is also a master coordinating the workers.

Lists of Structured Objects. Objects are fine-grained concepts. Different types of objects can be defined in Husky, for example, vertex objects in graph applications, or parameters in the context of machine learning. Objects are compoundable, for example, subclassing a `Vertex` class from a `TeraSort` class may result in vertices that are able to sort themselves (e.g., by their PageRank values). Objects are mutable. Objects are organized into object lists, which are a coarse-grained concept and facilitate the application of coarse-grained transformations such as *map* and *reduce*.

Object Interaction. In Husky, computation happens by object interaction. Husky supports both push-based and pull-based communication, meaning that objects can push messages to each other, or pull messages from each other. Objects have two different visibility levels. Local objects are only visible to objects in the local worker. Global objects are globally visible. Besides push and pull operations, an object can also migrate to different workers, which is not only used to implement effective fault tolerance and load balancing, but also enables flexible programming.

Consistency. Husky supports switching between synchronous and asynchronous executions. The synchronous execution adheres to strict BSP-style consistency, while the asynchronous mode with relaxed consistency improves the throughput for graph and machine learning workloads.

Patterns. The way that objects interact is called an *object interaction pattern* (or *pattern* for short). Husky is able to capture many existing programming paradigms as different patterns, as shown Figure 1. For example, the Pregel model [6] is just connecting global objects (i.e., vertices in a graph) using pure push-based messaging. The Parameter Server (PS) model [4] is captured by `Client` objects pulling parameters from and pushing updates to `Server` objects. A chain of MapReduce jobs can be modeled by implementing the shuffle using pushes and generating new objects based on push messages. Developers may just pick the model they are familiar with to lower the learning curve and gain high productivity. They may also create customized patterns in order to efficiently solve a specific problem.

2.2 Performance

While we refer readers to [7] for detailed system implementation and optimization of native Husky, here we briefly report some of its performance results by showing how we can compose a highly efficient workflow with Husky. Suppose that we want to recommend some Wikipedia pages to Wikipedia editors. In addition, we make the PageRank scores of the pages available so that we can recommend more influential pages. To do this, we construct a workflow

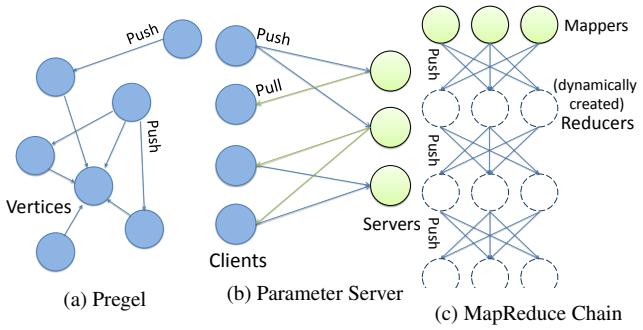


Figure 1: Different object interaction patterns

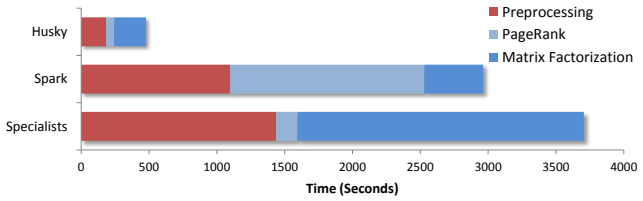


Figure 2: Performance on a Wikipedia workflow

as follows: (1) *Preprocessing*: parse the Wikipedia raw data, construct the page-link graph, and extract the page-editor sparse matrix; (2) *PageRank*: compute the PageRank values of the pages in the page-link graph; and (3) *Matrix Factorization*: model the page-editor relation using alternating least squares (ALS). We create a customized pattern based on the pull operation to solve the preprocessing, use Husky's Pregel API for PageRank, and create a customized pattern to implement blocked ALS algorithm.

The above Wikipedia workflow includes both *non-iterative coarse-grained* workloads and *iterative fine-grained* workloads. We compared Husky with Spark and a combination of specialized systems. The specialized systems include the use of Hadoop to do the preprocessing, GraphLab [3] for PageRank, and Petuum [4] for matrix factorization (we used its fastest method, ALS).

Figure 2 reports the results. With the flexibility to choose suitable patterns for different subtasks and the capability of handling both coarse-grained and fine-grained workloads, Husky is able to process mixed types of workloads in one system with remarkable efficiency.

2.3 PyHusky and ScHusky

We now present some key concepts and components of PyHusky and ScHusky.

2.3.1 Data Abstraction

In order to store and manipulate Python and Scala objects, we have `PythonObjectList` and `ScalaObjectList`, respectively, mirroring the `ObjectList` in native Husky. Unlike the `ObjectList` in native Husky, these are data abstractions, meaning that they are not necessarily materialized. Users can apply a number of operators (to be introduced shortly) on them, forming a DAG of operators, and the system will evaluate the results lazily. In other words, the system will first optimize the DAG, schedule the tasks in Python/Scala side as well as native Husky side, and then compute the final results.

2.3.2 Operators

Operators can be classified into two types, namely, non-native operators and native operators. Non-native operators include *map*, *reduce*, and so on. They allow users to manipulate Python-specific data with Python-specific libraries and tools (same for Scala), leveraging the productivity and succinctness of employing existing tools, or to perform works that are more convenient in high-level languages (e.g., use *map* with `urllib2` in Python for crawling, as crawling is not CPU-bounded).

Native operators represent functionalities in native Husky, including algorithmic computation such as machine learning and graph computation, as well as some frequently used ETL functionality for parsing data from common input sources.

2.3.3 Scheduling and Execution

When the materialization process is triggered, the DAG constructed by the user is first processed by the PyHusky/ ScHusky frontend. The frontend will first simplify the DAG (e.g., concatenation push-down). After that, it performs translations upon the operators and breaks shuffling operations into two multiple phases. One reason of doing that is for applying optimization. For example, breaking *reduce* into two parts facilitates concatenate elimination. Another reason is that we need to break down operators into multiple phases in order to match with the corresponding primitives in native Husky. These operators are then serialized into a queue. The serialization makes sure that all workers will execute in the same order. This serialized queue is later requested by all PyHusky/ScHusky workers.

The computation will take place in both PyHusky/ScHusky and native Husky. The two sides will use inter-process communication (IPC) to share data and cooperate with each other. We also implemented various data loaders to transform Python/Scala-specific data types to Husky data types in order to facilitate performance-critical computation in native Husky.

There is one major difference between PyHusky and ScHusky. As depicted in Figure 3, in ScHusky, each native Husky worker thread manages one ScHusky JVM thread, since JVM has good support for threading. However, for PyHusky, each native Husky worker thread manages one PyHusky *process*, since the most popular Python interpreter implementation (i.e., CPython) has the GIL (Global Interpretation Lock), making threading in PyHusky non-trivial.

2.3.4 Optimization

There are various optimizations applied in PyHusky/ScHusky in order to achieve high performance. Most importantly, PyHusky/ScHusky always tries to shift computation to native Husky whenever possible. For a DAG with native operators only, which may even possibly involve interaction with external systems such as HDFS and MongoDB, the data path will not go through Python/Scala runtime at all.

For loading data from external sources to PyHusky/ScHusky, we make a back-pressure loader in Husky in order to balance the load among different PyHusky/ScHusky instances, which allows maximal re-use of existing Husky components while at the same time maintaining the load balance. The back-pressure mechanism ensures that faster threads will tend to read more data from the source and handle larger data collection.

For consecutive one-to-one or one-to-many mapping operators, including *map*, *flat_map*, *filter*, and so on, in the actual execution of PyHusky/ScHusky, they are all compressed into only one operation, in order to improve locality and avoid scanning the data collection multiple times.

Another useful optimization is to bypass the Husky thread whenever communication can be directly done between two PyHusky/ScHusky threads. This avoids costly IPC (or thread communication) between the PyHusky/ScHusky process and Husky process.

3. DEMONSTRATION PLANS

We will demonstrate (1) the expressiveness of Husky in implementing different programming paradigms in one unified framework, (2) the usefulness of combining the productivity of Python/Scala and the efficiency of native Husky using PyHusky/ScHusky, (3) the flexibility of Husky in integrating with the Hadoop ecosystem and more, such as HBase, Hive and MongoDB, and (4) the high performance of PyHusky/ScHusky, compared with both general-purpose systems (e.g., Spark [10], Flink [1]) and domain-specific systems (e.g., Petuum [4], GraphLab [3]), on various workloads, as well as its scalability and fault tolerance.

3.1 Demo Setup

The back-end engine of Husky will be deployed and run on a Linux computing cluster with 20 to 30 nodes, where each machine has 24 cores and 48GB RAM, connected by Gigabit Ethernet as well as InfiniBand. HDFS, Hive, HBase, MongoDB, Redis, Cassandra, Parquet, etc., will also be connected with the Husky system, and serve as data sources for Husky. Apart from large real datasets used in [7, 5], we will also use streaming/crawled data, with which we will demonstrate how quickly we can build an end-to-end big data application using PyHusky/ScHusky.

In addition to distributed computing, we will also show SIGMOD attendees how to run Husky on a multi-core laptop (or even in a Windows box) for data small enough to fit in main memory of a single machine. This is especially useful for prototyping and debugging.

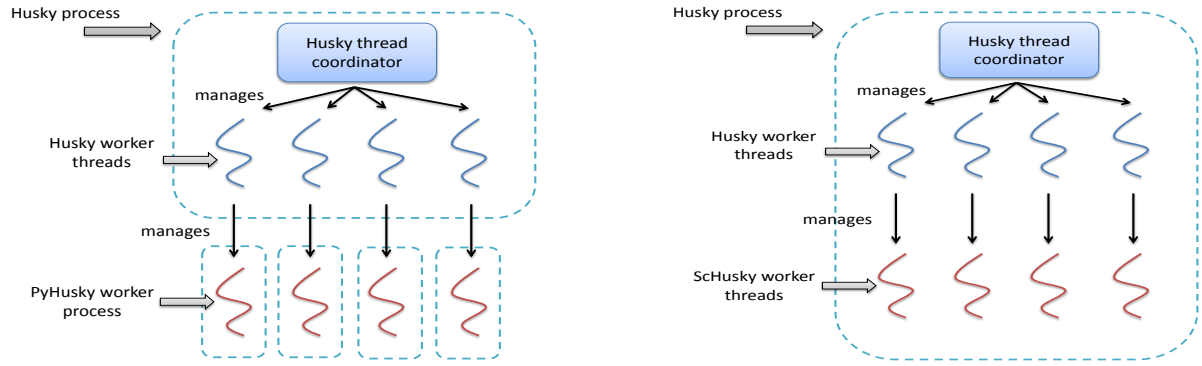
3.2 Demo Details

The demo will focus on giving SIGMOD attendees an experience of data analytics with PyHusky/ScHusky, especially for developers who are interested in building high-performance big data applications on top of Husky with the help of high-level languages. Both PyHusky and ScHusky are new, and almost all parts of the demo are new and not shown in [7].

Interactive Data Analysis. Husky's high-performance engine allows us to support interactive data analytics. This feature is very useful in exploratory data analysis, as well as debugging machine learning applications. For example, an end user may submit some ad-hoc top-k (or k-nearest-neighbor) queries based on some user-defined criteria. In this way, the user can use Husky to learn a statistical model (e.g., using stochastic gradient descent, alternating least squares, or logistic regression) from the data, query the model interactively, tune the parameters based on the query results, and in this way obtain a more accurate model. SIGMOD attendees will be able to interact with Husky via PyHusky/ScHusky, and try out interactive data analytics on Husky.

Building End-to-End Big Data Application. PyHusky and ScHusky aim to make big data applications easy to build while having little compromise on performance. In particular, SIGMOD attendees will be able to investigate and try out the following applications:

- A distributed crawler that automatically accumulates and analyzes data. The crawler has high performance due to its stateful implementation in Husky, and is easy to deploy and tune with the help of PyHusky/ScHusky.



(a) PyHusky (b) ScHusky

Figure 3: Architectural difference between PyHusky and ScHusky

- A real-time streaming analysis system, with the core data analysis functionality done in native Husky and the visualizer implemented in Python/Scala.

SIGMOD attendees will be able to try out different ideas on PyHusky/ScHusky via an easy interactive scripting interface (e.g., the Jupyter Notebook).

Integration with the Hadoop Ecosystem. One goal in the development of Husky is the interoperability with existing systems and tools in the Hadoop ecosystem. For example, Husky can load data from Hive by issuing some SQL statements on it. SIGMOD attendees can choose to read from different data sources and in different data formats, e.g., HDFS, HBase, MongoDB, Redis, Cassandra, Parquet, etc., and perform analytics over the data.

Performance Improvement. SIGMOD attendees will have the chance to compare different solutions and understand the benefits of using PyHusky/ScHusky. We will compare with baselines (e.g., Spark [10], Flink [1], Hadoop) where distributed computation are all done in Python or JVM, and show the benefits of PyHusky and ScHusky where performance-critical routines can be handled by native Husky, including:

- Using PyHusky/ScHusky to leverage asynchronous machine learning inside native Husky, such as NOMAD [9] and butterfly mixing [2].
- Using PyHusky/ScHusky to shift computation-intensive work to Husky, such as tensor/matrix factorization [8] and graph analytics.
- Understanding how a developer can write performance-critical paths in Husky and then expose to PyHusky and ScHusky with a small piece of binding code.

Fault Tolerance and Elastic Scalability. To show that Husky is fault tolerant, during the demonstration, we will occasionally kill some of the Husky processes to simulate machine failures. We will also randomly start new Husky processes in different machines to simulate that new computing nodes are added into the cluster. SIGMOD attendees will see that Husky is able to sustain failures, and automatically scale when new nodes join the cluster.

4. CONCLUSIONS

This demo plans to show how users can use PyHusky/ScHusky to enjoy the best of both worlds, i.e., the productivity and succinctness of high-level languages, and the flexibility and high performance of native Husky. We will showcase how PyHusky/ScHusky unleashes the power of Husky for building high-performance big data applications, and how easily this can be done via an easy interactive scripting interface. In addition, through hands-on experiences and comparisons, we will also illustrate the benefits of shifting performance-critical paths to native Husky.

Acknowledgments. This work was partially supported by ITF 6904079, Grants (CUHK 14206715 & 14222816) from the Hong Kong RGC, and Grant 3132821 funded by the Research Committee of CUHK.

5. REFERENCES

[1] Apache Flink. https://flink.apache.org/.

[2] J. Canny and H. Zhao. Butterfly mixing: Accelerating incremental-update algorithms on clusters. In *Proceedings of the 13th SIAM International Conference on Data Mining, May 2-4, 2013. Austin, Texas, USA.*, pages 785–793, 2013.

[3] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[4] S. Lee, J. K. Kim, X. Zheng, Q. Ho, G. A. Gibson, and E. P. Xing. On model parallelization and scheduling strategies for distributed machine learning. In *NIPS*, pages 2834–2842, 2014.

[5] J. Li, J. Cheng, Y. Zhao, F. Yang, Y. Huang, H. Chen, and R. Zhao. A comparison of general-purpose distributed systems for data processing. In *IEEE International Conference on Big Data*, pages 378–383, 2016.

[6] G. Malewicz, M. H. Austern, A. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[7] F. Yang, J. Li, and J. Cheng. Husky: Towards a more efficient and expressive distributed computing framework. *PVLDB*, 9(5):420–431, 2016.

[8] F. Yang, F. Shang, Y. Huang, J. Cheng, J. Li, Y. Zhao, and R. Zhao. Lftf: A framework for efficient tensor analytics at scale. *PVLDB*, 10(7), 2017.

[9] H. Yun, H. Yu, C. Hsieh, S. V. N. Vishwanathan, and I. S. Dhillon. NOMAD: nonlocking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. *PVLDB*, 7(11):975–986, 2014.

[10] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.