Husky: Towards a More Efficient and Expressive Distributed Computing Framework

Fan Yang Jinfeng Li James Cheng
Department of Computer Science and Engineering
The Chinese University of Hong Kong
{fyang,jfli,jcheng}@cse.cuhk.edu.hk

ABSTRACT

Finding efficient, expressive and yet intuitive programming models for data-parallel computing system is an important and open problem. Systems like Hadoop and Spark have been widely adopted for massive data processing, as coarse-grained primitives like map and reduce are succinct and easy to master. However, sometimes over-simplified API hinders programmers from more fine-grained control and designing more efficient algorithms. Developers may have to resort to sophisticated domain-specific languages (DSLs), or even low-level layers like MPI, but this raises development cost—learning many mutually exclusive systems prolongs the development schedule, and the use of low-level tools may result in bug-prone programming.

This motivated us to start the *Husky* open-source project, which is an attempt to strike a better balance between high performance and low development cost. Husky is developed mainly for in-memory large scale data mining, and also serves as a general research platform for designing efficient distributed algorithms. We show that many existing frameworks can be easily implemented and bridged together inside Husky, and Husky is able to achieve similar or even better performance compared with domain-specific systems.

1. INTRODUCTION

Distributed computing systems, such as Spark [31], Dryad [15], FlumeJava [8], and Flink [3], offer functional or declarative programming interfaces, with the basic idea that, programmers express the application logic in simple coarse-grained primitives like *map* and *reduce* [9], while the underlying system handles distributed execution and fault tolerance. This simplifies programming for distributed algorithms, as users need not worry about the underlying optimization or network topology.

However, such coarse-grained programming paradigm often does not result in efficient programs [11, 23]. In situations where performance is critical, programmers may want to have fine-grained control of the data access pattern, or even explicitly express the global data interaction pattern in order to reduce network traffic. For example, to support more efficient graph analytics, GraphX [13] extends Spark, and Gelly [3] extends Flink, to expose to programmers fine-grained control over the access patterns on vertices and their communication patterns.

Over-simplified functional or declarative programming interfaces with only coarse-grained primitives also limit the flexibility in designing efficient distributed algorithms. For instance, there is recent interest in machine learning algorithms that compute by frequently and asynchronously accessing and mutating global states (e.g., some entries in a large global key-value table) in a fine-grained manner [14, 19, 24, 30]. It is not clear how to program such algorithms using only synchronous and coarse-grained operators (e.g., *map*, *reduce*, and *join*), and even with immutable data abstraction (e.g., Spark's RDDs).

While easy application development by systems such as Spark is attractive, performance is equally critical, especially in cloud services where one pays for computing resources. Consider computing *PageRank* or *single source shortest path*, which are commonly used for social network analysis, Spark/GraphX can be 10 times slower than a domain-specific system like Giraph [4] and GraphLab [12] according to our experiments in Section 5.2; such poor performance implies that users would need to increase their budget for the cloud service by 10 times.

There are also other domain-specific frameworks developed to offer satisfactory performance, e.g., the Pregel [20] or GAS [12] computing frameworks for graph computing, and the Parameter Server (PS) framework [14, 18, 25] for machine learning. In addition, many low-level distributed programs are also available for solving specific problems. However, these frameworks and programs rely on completely different DSLs and internal data abstraction (e.g., vertices and edges in Pregel/GAS, but key-value stores and training points in PS). Thus, to develop real-world application with them, programmers not only need to spend time learning many different frameworks, but also have to suffer from the cost of *context switch*—data have to be dumped to distributed file system, loaded into the next system and then parsed, between consecutive stages that use different domain-specific systems, which involves unnecessary disk IO, network traffic and serialization/deserialization. The context switch overhead can make the overall process even slower than general-purpose systems like Spark, which can move seamlessly between stages (even though each individual stage may be slower).

The above discussion reveals the weaknesses in both existing general-purpose systems and domain-specific systems, which motivates us to develop the **Husky** open-source system. Husky gives developers freedom to express more data interaction patterns, but at the same time minimizes programming complexity and guarantees automatic fault tolerance and load balancing. In Husky, computation happens by fine-grained object interaction, where objects

Proceedings of the VLDB Endowment, Vol. 9, No. 5
Copyright 2016 VLDB Endowment 2150-8097/16/01.

are meaningful data abstractions such as `Customers`, `Products` and `Pages`. Husky objects are mutable, and can be easily composed with other objects so as to minimize programming efforts. Moreover, we can realize existing frameworks such as MapReduce, Pregel, or PS in Husky, so that these different domain-specific frameworks can now be integrated into a unified framework, while the Husky API still allows users to program in the same way as using familiar existing APIs as in MapReduce, Pregel, or PS.

Husky offers a simple and yet expressive set of object interaction primitives, and can serve as a platform for developing distributed algorithms with comparable efficiency as low-level codes (e.g., MPI), while allowing much more succinct coding. For example, we show that a sophisticated asynchronous matrix factorization program [30], which has more than 2000 lines of low-level code in MPI, can be expressed using Husky's primitives in just 100 lines of code in total. More importantly, in Husky such highly efficient programs can be composed into an integrated workflow, which may include specialized codes implemented in MapReduce (e.g. non-iterative bulk workloads), Pregel (e.g., iterative, fine-grained graph analytics), PS (e.g., asynchronous machine learning), etc. Underlying such an expressive API, Husky offers high performance with its efficient backend engine, and is able to achieve similar or even better performance compared with existing specialized systems or programs.

In the following sections, we first present the programming model and key concepts (Section 2), discuss the system implementation details (Section 3), and then demonstrate how to program in Husky using a list of applications (Section 4). We then discuss experimental results (Section 5) and related work (Section 6), followed by a conclusion of our work (Section 7).

2. PROGRAMMING MODEL

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 manages its own partitions of objects.

2.1 Core Concepts

Lists of structured objects. Husky supports different types of objects. The Husky library provides a variety of commonly used objects. Users may use objects from the library, or define their own objects. For example, a graph application uses `Vertex` objects, a machine learning developer may use `Vector` and `Matrix` objects, and a text mining application such as TF-IDF may define objects like `Term` and `Document`.

All the objects are subclasses of the ***base object class***, which automatically handles data racing, socket programming, and data layout under the hood. Users just extend the base class with specific application semantics (e.g., adding the neighbors of a `Vertex` object in a graph). Different objects can also be composed to create new object types. For example, users may further subclass `Vertex` with a `TeraSort` object in the library, such that the new compounded objects can be sorted (e.g., by the PageRank values of the `Vertex` objects).

Objects are organized into ***object lists*** by workers according to their types. The native support of reusable, arbitrarily structured objects reduces development efforts and increase the overall system expressiveness.

Workers. A worker can be regarded as a special type of objects that can read from external sources, e.g., *Hadoop distributed file system* (*HDFS*), and create objects. Husky allows users to specify a `partition` function to assign objects to different workers. An application can achieve better performance if the `partition` function groups objects that frequently interact with each other in the same partition.

Execution. The `execute` function of an object type specifies how an object performs local computation and global interaction, by pushing/pulling messages and/or migration (described later). To simultaneously invoke `execute` for all objects, users can invoke the `list_execute` function of a worker by providing an object list as the argument. We may also further extend `list_execute` to support more coarse-grained operations. Thus, Husky naturally supports both coarse-grained computation as in MapReduce and Spark, and fine-grained execution as in Pregel and PS.

Global and local objects. We classify objects into two ***visibility levels***: ***global objects*** and ***local objects***. A global object is visible globally by any object, while a local object is only visible by objects in the same worker. Global object facilitates communication across workers. Local object is an advanced feature of Husky, which helps optimize local computation when it can proceed independently or asynchronously without any global synchronization. Users can explicitly specify an object list to be local so that the runtime can apply optimization accordingly.

Interaction protocol. There are four ways in which objects interact with each other in Husky, listed as follows:

- Global/local objects can *push* messages to global objects.
- Global/local objects can *pull* messages from global objects.
- Global/local objects can *push* messages to or *pull* messages from local objects in the same worker.
- Global/local objects can *broadcast* messages to one or more workers (regarded as global objects), and the messages are then accessible by all objects in the respective worker.

Dynamic object creation and migration. Besides instructing a worker to create objects before computation starts, objects can also be created dynamically during computation. In addition, objects can be created by existing object via messaging—an object can *push* a message to a *not-yet-existing* object, which is then constructed upon the receipt of the message using the *message constructor*. This feature provides applications a new, flexible way to dynamically create new objects based on the current states of existing objects, and it is handy for various purposes which we will illustrate in Section 4.

Newly created objects can be inserted into an existing object list or a new object list, while existing objects can be removed from an object list. An application can also dynamically change a global object list to a local list, and vice versa. Besides, objects have the ability to *migrate* from one worker to another worker. Object migration is particularly useful in operations such as system load balancing which we will discuss in Section 3.5.

Synchronous and asynchronous computation. Husky supports switching between synchronous and asynchronous computation. In synchronous mode, conceptually all objects are processed in one round. At the end of a round the workers flush out the outgoing communications generated by the objects in the current round. This computation pattern adheres to BSP-style consistency, maximizes network traffic reduction, and is easy to reason about. In the actual implementation, it is possible to pipeline these processes to increase throughput.

```
class BaseWorker:
    def load(url, format)
    def create_list(name)
    def globalize_list(obj_list)
    def localize_list(obj_list)
    def add_object(obj_list, obj)
    def list_execute{obj_list, mode}
```

Listing 1: Husky Worker API

```
class BaseObject:
    def partition() # return partition id
    def execute()
    # The following are used inside execute()
    def get_msgs()  # get incoming messages
    def push(msg, id)
    def pull(id)
    def migrate(worker_id)
    def broadcast(msg, worker_id)
```

Listing 2: Husky Object API

Developers are free to choose asynchronous mode for an application, in which computation does not proceed through synchronized rounds, but in an asynchronous and non-deterministic way. For algorithms that do not require strict consistency guarantee (e.g., Stochastic Gradient Descent [30]), asynchronous computation leads to faster convergence.

Listing 1 and 2 summarize the Husky API discussed in this section.

2.2 The Husky Approach

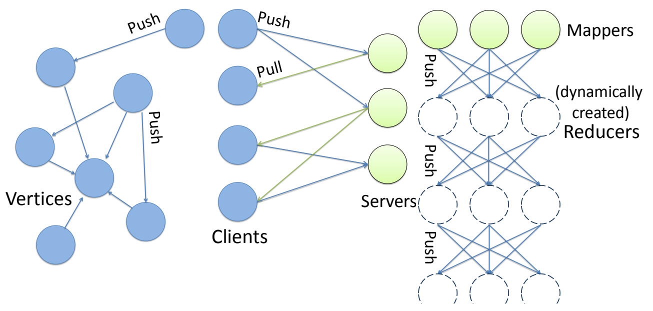
There have been a lot of distributed frameworks and programs developed for various purposes, but they cannot cooperate with each other in an efficient way. We now describe the Husky approach that composes different efficient programs into a unified framework with little overhead.

Cooperation of Different Objects

A real-world application usually involves the cooperation of many different components. For example, the PageRank values computed using a graph framework may later be used by a machine learning component as training features. This can be easily achieved in Husky, since all objects can cooperate through the basic *pull*, *push*, and *migrate* primitives, even though they belong to different types (e.g., the PageRank values of `Vertex` objects can be directly pushed to or pulled by objects in a machine learning component).

Composing Object Interaction Patterns with Husky

The way that objects interact is called an *object interaction pattern*, or simply a *pattern*. We can naturally create patterns of many existing frameworks in Husky, as shown in Figure 1. For example, connecting global objects (i.e., vertices in a graph) using pure *push*-based messaging results in the Pregel framework. Parameter Server is created by letting local objects (i.e., clients in PS) *pull* (read) and *push* (update) parameter data from a set of partitioned global objects (i.e., servers in PS). We can also create a pattern for a chain of MapReduce jobs as shown in Figure 1. Note that Reducers in each round of MapReduce are dynamically created, and there is no Mappers following Reducers since one-to-one transformation can be directly applied from the outputs of Reducers in the previous round to Reducers in the next round.



(a) Pregel (b) Parameter Server (c) MapReduce Chain

Figure 1: Different object interaction patterns

There are far more possibilities in addition to the above three patterns. For example, with the asynchronous mode enabled, by letting objects migrating and pulling local training points, we created a sophisticated pattern dedicated for high-performance machine learning, which will be discussed in Section 4.3.2.

Another great advantage of such a design is that users who are not familiar with Husky can simply import predefined frameworks, e.g., MapReduce and Pregel (implemented as patterns), from the Husky library, then code in MapReduce and Pregel styles, and obtain a pipelined workflow with in-memory MapReduce and Pregel programs. Husky also allows developers the freedom to create specialized patterns to solve specific problems so as to maximize performance, as is shown in our TF-IDF example in Section 4.1.

3. SYSTEM IMPLEMENTATION

This section discusses the details of how the concepts discussed in Section 2 are implemented, as well as other important details such as load balancing and fault tolerance.

3.1 Master-Worker Architecture

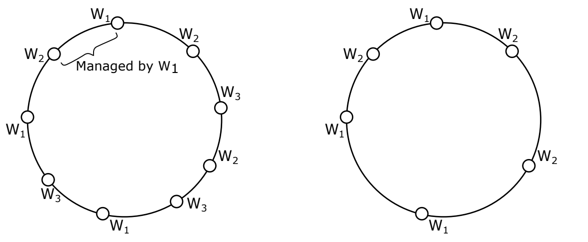
A Husky cluster consists of one master and multiple workers. The master is responsible to coordinate the workers, and workers perform actual computations.

To perform a round of computation (by calling `list_execute`), a worker first receives all incoming communications[1] (if any) from other workers, dispatches messages to objects and invokes their `execute` function, and finally flushes the outgoing communications that the objects have generated.

In *synchronous* mode, at the i-th execution round, a worker will only process incoming communications (if any) generated from the $(i-1)$-th round. Any incoming communication of the i-th round will be (asynchronously) received and cached for the $(i+1)$-th round, even during the process when the worker itself is still working on the i-th round. Usually a worker flushes outgoing communications only after all workers on its same host finishes the current execution round. The purpose is to exploit the effective *shuffle combiner* technique (to be discussed in Section 3.3.2) to maximize message reduction. However, when no combiner is provided, Husky will batch and flush messages right after they are generated, in order to interleave CPU usage and networking IO.

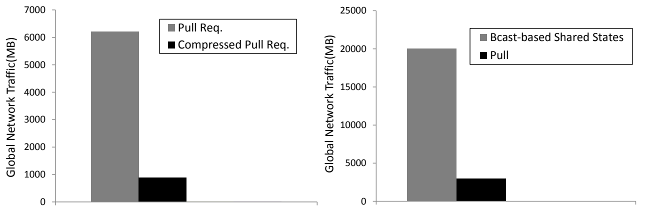
In *asynchronous* mode, object computation is triggered right after the worker receives any incoming communication. Outgoing communications are also flushed asynchronously right after they are generated.

[1]We use the term "communication" to refer to both messages and migrating objects.



(a) Normal hash ring (b) Hash ring after W_3 removed

Figure 2: Consistent hashing with virtual nodes



(a) *pull* request compression (b) *pull* vs. *broadcast*

Figure 3: Performance of *pull*

3.2 Consistent Hashing-based Object Management

In Husky, global objects are partitioned and distributed to different workers based on *consistent hashing* [17]. The consistent hashing-based layout supports dynamic addition and removal of machines without the need of rehashing everything, and it is also important for Husky in designing load balancing and fault tolerance strategies. A few default `partition` functions (e.g., hash functions for numbers and for strings) are available, while users can supply their own `partition` functions specifically designed for their data. Object ids are hashed to $\{0 \ldots H_{max} - 1\}$, where H_{max} is usually set as a large number (e.g., 2^{32}).

The hash range can be visualized as a hash ring with perimeter H_{max}, and workers are inserted into the ring. Let W_1 be a worker on the ring and W_2 be the worker next to W_1 in counter-clockwise direction. All objects falling into the range between W_2 (exclusive) and W_1 (inclusive) are distributed to and managed by W_1, as shown in Figure 2a. The master keeps a copy of the hash ring arrangement, and each worker caches it locally. The master will notify workers when there is any change to the hash ring.

We also apply the *virtual node* technique [10], by which a worker is hashed to multiple locations on the ring and owns multiple ranges, as shown in Figure 2a. This helps load balancing, especially when the distribution of objects is non-uniform.

Local objects are managed locally without the restriction of consistent hashing. Communication among local objects does not go through the network and is more efficient. Lookup of a local object does not need to use a global `partition` function, but simply use a local index to locate the object. In jobs for which heavy computation needs to be performed on local data, it can be efficient to create local objects to work on the local data. If subsequent pipelined jobs require more global interactions among the objects, they can be converted to global ones and the system will automatically partition and migrate these global objects. For example, a developer may work on local objects in the process of extracting graph structures from a Wikipedia corpus, but later convert the graph global to computing PageRank, where global interaction happens (see an example in Section 4.4).

3.3 Implementation of Primitives

Any Husky application boils down to the composition of primitives such as *push*, *pull*, and *migrate*. We discuss the implementation of these primitives in this subsection.

3.3.1 Compressed Pull

There are a wide range of applications that rely on globally shared states (e.g., machine learning applications that operate on globally shared matrix), and they need *pull* to retrieve these states.

In systems such as Hadoop and Spark, the shared state problem is usually solved by *broadcasting* all the global states to all workers. Spark provides the broadcast primitive, and Hadoop provides distributed cache. But such a design has two limitations: 1) many applications may just access a fraction of the global states in each round, so broadcasting all shared objects every time is too expensive; and 2) the shared objects have to be read-only and immutable—a worker will not be aware of the modification done by another worker. In fact, these limitations necessitate the distributed mutable key-value table design like Parameter Server, for many machine learning applications.

Let us first consider the synchronous mode. A standard implementation is to represent a *pull* request as a pair, (W_{id}, o_{id}), where W_{id} is the worker id of the requester and o_{id} is the id of the global object being requested. At the end of an execution round, worker W_{id} flushes all its *pull* requests, and waits for the responses to come in the next execution round. One problem with this design is that compared with a *push* communication, a *pull* consists of a request and a response, thus doubling the network traffic.

To reduce network traffic, we use *Bloom filter* to compress *pull* requests. Worker W_{id} creates a Bloom filter B for all requests sent to another worker W'_{id}, and inserts (the hash values of) the ids of all these requestees in B. Then, worker W_{id} only sends (W_{id}, B) to worker W'_{id}. This effectively reduces the network traffic to $O(|B|)$ if we regard the number of workers as a constant; thus, the *pull* operation can enjoy almost the same efficiency of a *push*.

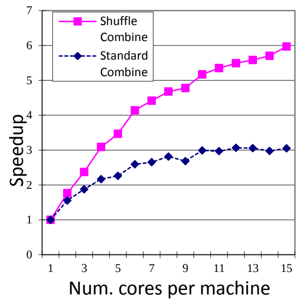
We can further reduce the network traffic by sending compressed Bloom filters (e.g., by run-length encoding), which also addresses the issue when the number of requestees is small. Figure 3a shows the effectiveness of the *pull* request compression, while Figure 3b shows the reduction of network traffic by *pull*, compared with the broadcast strategy. The results were measured in the process of extracting a Wikipedia link graph (see details in Section 4.4).

Another effective optimization to reduce network traffic is to share responses to the same machine. Multiple objects in multiple workers sitting in the same machine may request the same information from a global object. We can combine all identical responses to the same machine into a single response. Then, the workers in the same machine share a buffer for incoming responses, from which the requesting objects obtain their responses.

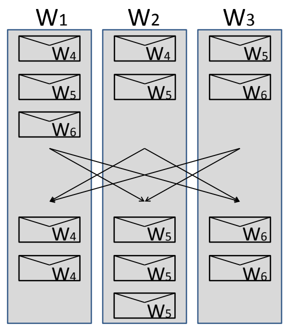
The asynchronous mode is different in that, after a *pull* request, the corresponding response may come at a non-deterministic time. We leave the flexibility to developers and they can choose to instruct the objects to wait for the latest responses, or allow them to use the old ones, resulting in a relaxed consistency model similar to the *Stale Synchronous Parallel* [14].

3.3.2 Shuffle Combiner

Many systems use *combiner* to reduce outbound messages of a worker, as fewer messages mean lower network latency and shorter message processing time (e.g., serialization and de-serialization). For example, in MapReduce [9], users can specify a combiner to combine the output of a mapper. In Pregel [20], users can use a



(a) Effect of combiner designs



(b) Shuffle combiner

Figure 4: Combiner design

combiner to combine the outgoing messages of a worker.

However, such a design of applying combiner to each single worker/mapper poses a potential scalability problem when the system is deployed in a modern cluster with many multi-core machines. In general, a multi-core machine hosts multiple workers. In this setting, when each worker only combines its own messages, more workers means less efficiency of the combiner. With the number of workers increasing, there exists a critical point beyond which the cost of processing more messages overtakes the benefit of more workers. For example, when running PageRank on a Twitter graph with this *standard combiner* design, as shown in Figure 4a, the speedup relative to a single thread implementation becomes less obvious when the number of workers in each machine increases, and the system almost stops to scale when there are more than 8 workers per machine.

We propose a ***shuffle combiner*** design to address the above-mentioned scalability problem as follows. After the first pass of message combining, workers in the same machine shuffle outgoing messages among each other according to the destinations of the messages, as shown in Figure 4b. Then, each worker further combines the messages that are shuffled to it in a second pass, and flushes them to the destination worker(s).

Although the shuffle combiner requires synchronization as a tradeoff, the overall running time can be significantly reduced in a Gigabit (or slower) Ethernet network. As shown in Figure 4a, shuffle combiner not only has greater speedup, but also has much better scalability than standard combiner. This technique benefits all models (e.g., Pregel, PS, and MapReduce) built on Husky that apply combiner on *push* messages.

3.3.3 *Cache-Aware Optimization*

To handle fine-grained operations inside an object list, the system has to support fast object lookup. A standard way is to implement a hashmap or a tree structure to organize the objects, which facilitates fast lookup, insertion and deletion. However, such a data structure may not be efficient for coarse-grained operations due to poor locality. For example, batch object creation and incoming communication handling essentially become random walks over main memory, resulting in cache thrashing and degraded performance.

In order to support both fine-grained and coarse-grained operations, we design dedicated data structures for Husky as follows. Consider a single object list for simplicity. We store the list of objects in contiguous memory spaces for better spatial locality. The objects are sorted according to their ids. In addition, we use two auxiliary data structures, a *bitmap* and a *hashmap*. The bitmap is used to facilitate lazy deletion—object deletion is done by marking the corresponding position in the bitmap. The hashmap is used for dynamic object creation, where the new objects are appended to the end of the object list and indexed by the hashmap. Thus, the object list consists of two parts, where the first part is ordered by object ids and the second part indexed by the hashmap. When the number of deleted objects and dynamically created objects reaches a threshold, the system automatically rebuilds the whole list, and in this way the cost of sorting during the rebuild is amortized.

With this design, batch object creation is done by building and sorting the object list. During execution, dynamically created objects are appended to the unordered part and then indexed by the hashmap. An object lookup consists of a binary search over the ordered part and then a hashmap lookup. As messages come in order (due to sort-based combiner), our object lookup design is significantly faster than a standard hashmap lookup design, since consecutive searches access similar locations and therefore exploit cache temporal locality. We observed that this brings around 20% improvements for applications that heavily depend on messaging (which requires extensive object lookups).

3.4 Fault Tolerance

The basic fault tolerance mechanism in Husky is *checkpoint-based recovery*. The master asks workers to save their current states to HDFS at every fixed time interval (e.g., every 10 minutes). Workers periodically send heartbeats to the master, and the master marks a worker "failed" if it does not receive the heartbeats of the worker within a time limit. In this case, the master instructs all workers to rollback to the most recent checkpoint, and restart from there.

While checkpoint-based recovery is necessary as otherwise recovery would have to start from the beginning, it is a time consuming process. To support efficient recovery, Husky provides an additional fault tolerance mechanism based on consistent hashing and upstream backup. *Upstream backup* is a technique proven to be robust and widely used in stream processing systems which emphasize low-latency data processing [1, 2, 6, 26]. In upstream backup, each processing node logs the messages before pushing them to the downstream nodes. Failed nodes are re-created on healthy machines, and their upstream nodes push logged messages to them to recover the lost states.

A simple way to apply this technique in Husky is to let each worker log all the outgoing communication before flushing them. To hide I/O latency, the logging can be done asynchronously while the worker is performing other tasks. For fault recovery, the recovering workers obtain objects in the failed workers from the last checkpoint, and receive logged communications (that were sent to the failed workers) from healthy workers, while filtering out logged communications to healthy workers. Then, the recovering workers only need to communicate among themselves to repeat all the lost execution rounds until they catch up with the healthy workers, by then the recovery completes.

The above recovery process, however, is still not efficient enough as we show by the following analysis. Suppose the total time for one execution round in a failed worker is $T_{comp} + T_{comm}$, where T_{comp} is the computation time and T_{comm} is the communication time. The above method may significantly reduce T_{comm} but not T_{comp} , unless a more powerful recovering worker is used to replace a failed worker. Thus, this method is inefficient when T_{comp} is relatively large.

While it is a must to create new processing units to replace failed ones in streaming systems, we do not have to create new workers in Husky by employing consistent hashing for fault recovery. When fault occurs, the Husky master simply takes away the failed worker(s) from the hash ring, and so objects in its ranges (including its virtual nodes) are automatically merged into the ranges of neighboring healthy workers, as shown in Figure 2b where we remove W_3. In this way, the recovery workload is parallelized and shared by all workers, hence reducing both T_{comp} and T_{comm}. We show in Section 5.5 that fault recovery in Husky is much more efficient than that by Spark's lineage recovery.

3.5 Load Balancing

In distributed computing, the appearance of stragglers is usually the sign of load imbalance. Stragglers may be due to skewed data distribution, heterogeneous machine configurations, or other local tasks running in parallel contending for resources. The first case can be solved in the application layer by investigating (e.g., sampling) the data characteristics and making a better partitioning scheme, and we discuss how we handle the last two cases as follows.

A common way to achieve load balancing is to configure each machine individually by taking the capacity differences of the machines into account, which is similar to the Hadoop practice. However, this works well only if developers have prior knowledge of the machine configurations.

In situations when the above strategy is ineffective or inapplicable (e.g., machine configurations are not known), Husky also provides a *dynamic load balancing* scheme to address any imbalanced workloads during computation. The master monitors the progress of workers. Each worker samples its running time statistics and sends to the master, and the master determines whether a worker is a straggler. In the current Husky implementation, a worker reports $(T_{comp} + T_{outcomm})$ as its running time, where T_{comp} is the computation time taken by a worker and $T_{outcomm}$ is the time of processing outgoing communications. We omit T_{incomm} (time for processing incoming communications) because it can be affected by stragglers and cannot reflect the true capacity of the worker. Upon the receipt of the statistics, the master identifies stragglers as those workers whose running time is larger than $(1 + p) \cdot t_{avg}$, where t_{avg} is the average reported time and p ($p > 0$) indicates the tolerance level of load imbalance (larger p means greater tolerance, 0.1 by default). Workers may report more statistics (e.g., network condition, CPU rate) to the master, so that the master may make better decisions (we leave this to future work).

When stragglers are identified, the master then models the capacities of workers according to the running times they reported, and adjusts the arrangement on the hash ring. Specifically, the master first selects $\lceil \frac{t_{slow} - t_{avg}}{t_{slow}} |P_{slow}| \rceil$ partitions from the partitions of a straggler, where t_{slow} is the time reported by the straggler and $|P_{slow}|$ is the number of data partitions assigned to the straggler. This process repeats for all stragglers. Then the master merges the selected partitions into the partitions (or ranges) of the neighboring non-stragglers, in order to preserve data locality. The master may also apply heuristics and choose faster workers to absorb more partitions from the stragglers.

The master then broadcasts the new hash ring arrangement to all workers, which triggers a bulk migration of objects among workers. During the process, most objects stay intact, and only objects affected by the new arrangement move (from stragglers) to their new owners. This process is usually fast (as verified in Section 5.6) because with the aid of consistent hashing, each object migration does not need to be globally announced.

4. APPLICATIONS

In this section, we illustrate how we can build applications on Husky with a number of examples, including non-iterative bulk workloads, iterative computation, asynchronous algorithms, and pipelined tasks. Performance figures of these applications will be reported in Section 5.

```
1st_map(line):
    emit ((t, d), 1)
1st_reduce((t, d), [1, 1, ...])
    emit ((t, d), count)

2nd_map((t, d), count):
    emit (d, (t, count))
2nd_reduce(d, [(t_1, c_1), (t_1, c_2), ...]):
    emit ((t, d), tf)

3rd_map((t, d), tf):
    emit (t, (d, tf))
3rd_reduce(t, [(d_1, tf_1), (d_1, tf_2), ...]):
    emit ((t, d), tfidf)
```

Figure 5: TF-IDF in MapReduce

4.1 Bulk Transformation Workloads

MapReduce [9] and Hadoop [5] were primarily developed for handling non-iterative workloads that involve bulk data movements and transformations, such as computing word counts or TF-IDF from a large corpus, data preprocessing (e.g., extracting structures from raw data), etc. We discuss how to compute TF-IDF using Husky as an example.

TF-IDF reflects the importance of a *term* t to a *document* d in a corpus D, defined as $tf\text{-}idf(t, d, D) = tf(t, d) \times idf(t, D)$, where $tf(t, d) = count(t, d)/|d|$ and $idf(t, D) = \log(|D|/|\{d \in D : t \in d\}|)$: $count(t, d)$ is the count of t in d, $|d|$ is the total number of terms in d, and $|D|$ is the number of documents in D. A common MapReduce algorithm to compute TF-IDF involves three rounds of Map and Reduce as illustrated in Figure 5.

Users may use the predefined MapReduce framework (discussed in Section 2.2) from the Husky library to compute TF-IDF, and then seamlessly pass them to any other framework (e.g., use them as ML features) in Husky. MapReduce on Husky achieves good performance compared with popular in-memory MapReduce systems such as Spark, but Husky is usually faster due to the use of shuffle combiner. However, we are more interested in how a developer may be able to design a more efficient algorithm using Husky's native primitives. We present the details of the algorithm as follows (Figure 6 shows the object interaction pattern):

1. Workers read the corpus from HDFS. As a worker reads a document with id $= d$, for each term t it reads in the document, it *pushes* a message "1" to a local `TermDoc` object with id $= (t, d)$ (which may not exist yet) on the fly. Upon finishing the document, the worker creates a local `Doc` object with id $= d$ and records the total number of terms in it, denoted by $|d|$.

2. Each `TermDoc` object, which is created (with id $= (t, d)$) upon receiving the first message from its worker, counts the number of received messages to obtain $count(t, d)$. Then, it *pulls* $|d|$ from the local `Doc` object with id $= d$ to calculate $tf(t, d)$.

3. Each `TermDoc` object, with id $= (t, d)$, also *pushes* a message "1" to a global `Term` object with id $= t$ (which may not exist yet). Upon the receipt of the messages, the global `Term` object with id $= t$ knows the number of documents containing t, and so derives $idf(t, D)$.

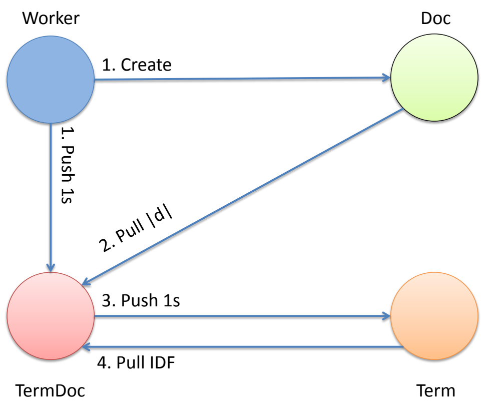


Figure 6: TF-IDF in Husky's native primitives

4. Each `TermDoc` object, with id $= (t, d)$, *pulls* $idf(t, D)$ from the global `Term` object with id $= t$, and multiplies $idf(t, D)$ by $tf(t, d)$ to obtain $tf\text{-}idf(t, d, D)$.

The above algorithm is intuitive as users think of meaningful objects related to the TF-IDF equation, just like object-oriented programming, instead of key-value lists and functional primitives like map and reduce. It can also better utilize the computing cluster, bringing about 8.6 to 16 times improvement as reported in Section 5.1. Specifically, the use of local objects eliminates network traffic (Steps 1 and 2); while the fine-grained *pull* operation ensures that each object only picks what it needs from a large vocabulary (Step 4), without shuffling the whole vocabulary. This helps avoid a large amount of unnecessary network traffic.

4.2 Graph Analytics

Graph analytics has received a lot of attention in recent years. The Husky library implements the Pregel framework [20], as discussed in Section 2.2, though other existing graph-parallel models such as GraphLab (or PowerGraph [12]) can also be implemented.

Users can simply design their algorithms based Pregel's "think-like-a-vertex" philosophy, write vertex programs and run them on Husky. We implemented PageRank and single source shortest path (SSSP) to test Husky's capability in performing graph analysis. PageRank involves both heavy computation and heavy network traffic, as all vertices are active in every execution round. SSSP requires fine-grained updates to only partial individual vertices during the computation, and hence communication loads can be more biased. Section 5.2 shows that Husky can be 5.0, 5.6, 16.4 and 36.6 times faster than GraphLab, Naiad [21], Giraph and GraphX for PageRank, respectively, and 4.6, 5.5, 7.2 and over 50 times faster for SSSP.

4.3 Collaborative Filtering

Collaborative Filtering (*CF*) is a technique popularly used in many recommender systems. A common technique for CF is to mathematically interpret the original recommendation problem as a *Matrix Factorization* (*MF*) problem, which models users and items as fixed-size vectors of latent factors.

We discuss two very different ways of training this model, the synchronous and iterative *Alternative Least Squares*, and the asynchronous *Stochastic Gradient Descent*. The latter is more efficient, but hard to implement in frameworks that do not support asynchronous messaging.

4.3.1 Alternating Least Squares (ALS)

Using the Parameter Server (PS) framework from the Husky library, we can easily implement ALS. However, in this case, each update is pushed to the server object and then pulled by client objects in need, resulting in doubled network traffic. Besides, PS does not allow interaction between clients, limiting the flexibility of designing more efficient algorithm.

We consider the sparse rating matrix as a bipartite graph between users and items. Then, we define `User` objects and `Item` objects, where each `User` (`Item`) object keeps the ids of its neighboring `Item` (`User`) objects in the bipartite graph. At each training iteration, neighboring `User` objects and `Item` objects exchange their latent factors. The transmission of these latent factors is done using Husky's *broadcast* primitive, instead of the *push* primitive so as to eliminate identical variables in the *pushes* of an object to its many neighbors, thereby reducing network traffic. Note that Husky does not *broadcast* variables to workers who do not need them.

Our experiments showed that ALS on Husky significantly outperforms ALS on existing systems such as Petuum [14, 18] (the state-of-the-art PS), GraphLab, and Spark. This is mainly because Husky's fine-grained broadcast saves network traffic, and Husky is also free of the *client-server-client* indirect network communication overhead in PS.

4.3.2 Stochastic Gradient Descent (SGD)

Theoretically, SGD for MF has linear complexity while ALS has cubic complexity. But in practice, efficient SGD MF implementation is not common in existing distributed systems, as it requires a computing framework that supports *fast asynchronous fine-grained updates*. Otherwise, since each iteration of SGD tends to be very short and it usually requires thousands of execution rounds until convergence, frequent global synchronization leads to poor network and CPU utilization.

We implemented an asynchronous SGD algorithm called NOMAD [30], which runs much faster compared with the synchronous counterparts, as reported in [30]. Here we briefly discuss how this algorithm is implemented. We define three types of objects: the global `User` object (storing the latent factors of a user), the global `DataBlock` object (storing a partition of training data points), and the local `Item` object (storing the latent factors of an item). Each training data point is a triplet (*user*, *item*, *rating*), and the `User` objects and `DataBlock` objects are co-partitioned across workers by user ids. During the computation, each `Item` object i continuously *migrates* from one worker to another worker, and upon arrival it performs the following operations:

1. *Pull* the corresponding training data points (*user*, *item* $= i$, *rating*) from the `DataBlock` object in the worker where i has just migrated to.

2. *Pull* the corresponding `User` objects and update the gradients.

We run Husky in asynchronous mode as required by the algorithm, and achieve performance very close to the native MPI implementation in [30]. While the native MPI program has over 2000 lines of low-level code, the implementation in Husky takes only around 100 lines, using just the *pull* and *migrates* primitives.

4.4 A Demonstration of Pipelined Workflow

Our final application is a demonstration how we can efficiently compose a workflow, which consists of tasks with different characteristics. Suppose 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. Thus, the problem is how to obtain the PageRank scores of the pages and the latent factors of both the pages and editors.

Conceptually, we can implement the workflow by the following pipelined steps:

1. Load the Wikipedia XML data.
2. Extract the page graph.
3. Extract the page-editor sparse matrix.
4. Compute PageRank on the page graph.
5. Factorize the page-editor sparse matrix.

The Wikipedia datasets consist of the page data and the revision data. The page data contain pages in Wikipedia (including Categories, Talks, etc.). The revision data contain the revisions of the pages, which we use to extract the history of the modifications done by the editors.

The page data are loaded page by page. After loading each page, we create a local `Page` object containing all the information about the page. However, a problem is that the links in the Wikipedia pages are shown as titles instead of page ids. We may use the title of a page as its id; but for message passing in PageRank computation, sending textual titles usually incur much higher communication cost than sending integer ids. Using title hashes may potentially lead to hash collisions and inaccurate results as the number of pages is large. With Husky's model, we can easily create the corresponding page id for a page title that appears as a link in another page, which we show as follows.

Each `Page` object *pushes* a message to create a global `Title` object with its title as the object id, where `Title` object also keeps the id of the `Page` object. This is essentially creating a dictionary with title as key and `Page` id as value. In the next step, we can use `Title` to get its corresponding `Page` id from the dictionary as follows. For each `Page` object, p, let T be the set of links in the page of p (i.e., the titles of other pages that p links to), do: for each title t in T, p *pulls* the id of the corresponding `Page` object from the `Title` object with key t. This process constructs the set of out-neighbors for each `Page` object and hence the page graph.

After that, we convert the `Page` objects to global objects to facilitate the processing in subsequent steps. Then, we read the revision data and send the editor ids to the corresponding `Page` object upon reading a revision record. After reading the revision data, each `Page` object now also contains its editors. Each `Page` object then also initializes its latent factors, and *pushes* messages to create and initialize `Editor` objects (as well as their latent factors). This process constructs the page-editor sparse matrix.

At this point we have done the first three steps in the job pipeline. The remaining two steps are just to perform PageRank on the `Page` objects (Step 4) and Matrix Factorization (e.g., by ALS) on the `Page` objects and `Editor` objects (Step 5). Readers may refer to Sections 4.2 and 4.3 for PageRank and MF computation in Husky.

5. EXPERIMENTAL EVALUATION

We conducted extensive experiments to verify our arguments that the design of Husky leads to low system overhead, while at the same time allowing users to express a variety of algorithms and frameworks. In particular, we tested Husky with different workloads, including *non-iterative coarse-grained workloads, iterative fine-grained graph analytics, synchronous and asynchronous machine learning, and pipelined workflow of mixed jobs*. We also evaluated Husky's *scalability* and its performance on *fault tolerance* and *load balancing*.

We ran all experiments on a Linux computing cluster, where each machine is equipped with 48GB RAM, two 2.0GHz Intel(R) Xeon(R) CPU (12 physical cores in total), a 450GB SATA disk (6Gb/s, 10k rpm, 64MB cache), and a Broadcom Gigabit Ethernet NIC. We allocate 300 cores of our cluster for all experiments. The Hadoop version we use is 2.6.0. Spark (version 1.3.1) was deployed Standalone Mode and carefully tuned for best performance. For GraphLab [12], Giraph [4], Petuum [14, 18], and Naiad [21], we installed the latest versions according to their manuals. For all systems we tested in each of our experiments, we used the best settings that gave the best performance. All graph and machine learning implementations were taken from the libraries or the authors of the respective systems, if provided.

Husky was implemented in C++. The source codes of Husky and all its applications presented in this paper are available on `http://www.husky-project.com/`. The latest version also supports a Python API.

5.1 Performance on Bulk Workload

We first tested TF-IDF, which involves bulk data movements and is non-iterative computing. Such workload is what MapReduce is designed for, and hence we compared with Hadoop and Spark. The dataset we used is the English Wikipedia corpus[2], denoted by "enwiki $\times$ 1", which contains over 4.8 million documents and 1.7 billion terms. We also tested heavier workloads by duplicating "enwiki $\times$ 1" by 2 and 3 times, denoted by "enwiki $\times$ 2" and "enwiki $\times$ 3".

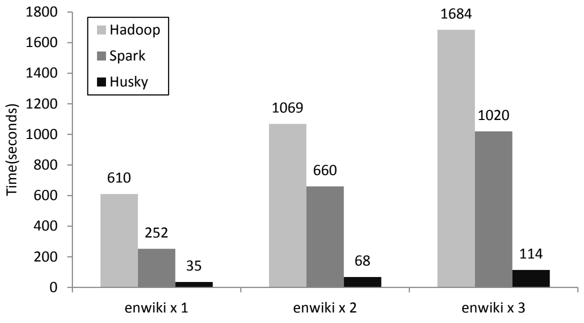


Figure 7: Performance on TF-IDF

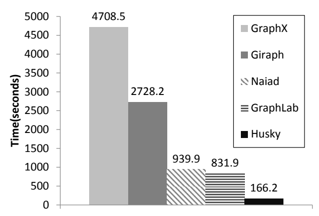
As reported in Figure 7, Spark is 39% to 58% faster than Hadoop, but Husky is even around 10 times faster than Spark. Spark outperforms Hadoop mainly because Spark is in-memory MapReduce system and hence it does not do context switch (i.e., dump/load data to/from HDFS) between consecutive stages, which saves a lot of time. Husky's superior performance can be explained as follows.

Due to bulk data movements, TF-IDF is essentially network-bounded. In this case, the use of local objects and fine-grained *pull* in Husky reduce a large amount of network traffic. To verify it, we collected the number of bytes transmitted in different systems, and observed that the network traffic of Husky is over an order of magnitude lower than Spark and Hadoop. The advantage is more significant when the datasets get larger.

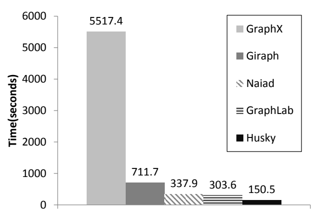
5.2 Performance on Graph Analytics

Next, we tested Husky on iterative graph analytics workloads. We used two large publicly available graphs, WebUK and Twitter,

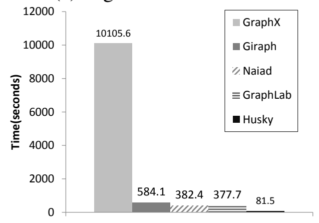
[2]https://en.wikipedia.org/wiki/Wikipedia:Database_download/



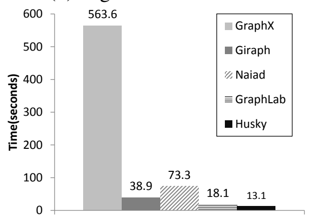
(a) PageRank on WebUK



(b) PageRank on Twitter



(c) SSSP on WebUK



(d) SSSP on Twitter

Figure 8: Performance on PageRank (30 iterations) and SSSP

and Table 1 lists their number of vertices and edges, and maximum and average degree. We implemented two popular graph algorithms, PageRank and SSSP, as discussed in Section 4.2. PageRank updates all vertices and generates $|E|$ messages at each round, while SSSP requires fine-grained updates to only partial vertices at each round.

	$\lvert V\rvert$	$\lvert E\rvert$	max-deg	avg-deg
WebUK	133,633,040	5,507,679,822	22,429	41.21
Twitter	52,579,682	1,963,263,821	779,958	37.33

Table 1: Graph datasets

We compared Husky with Giraph, GraphLab PowerGraph, Spark GraphX, and Naiad. Giraph and GraphLab are two popular efficient graph computing systems. GraphX is built on the MapReduce-based system Spark. Naiad abstracts a graph as a stream of edges and was reported to achieve good performance for graph computing [21]. The graphs were loaded with the default settings of the respective systems, without doing any advanced partitioning as this is not our focus.

Figure 8 reports the results. GraphLab, Giraph, and Naiad all outperform GraphX significantly, among which GraphLab shows the best performance, but is still slower than Husky. We observed that the improvement of Husky over GraphLab largely comes from the use of *shuffle combiner* (see Section 3.3.2), with which the network traffic of each iteration in Husky is around 3 times lower than that in GraphLab. To further confirm the effect of shuffle combiner, we turned it off and observed that the Husky result was degraded and close to that of GraphLab.

All the systems, except GraphX, support fine-grained data access, which is essential for graph computing. On the contrary, in GraphX, graph operations are transformed into maintenance and joins of several immutable distributed tables, resulting in high memory footprint. For example, in running SSSP on WebUK, GraphX ran out all the available memory and started spilling to disks. Note that the poor performance is not because GraphX is JVM-based, since Giraph is also based on JVM, but it implements the right Pregel model and hence used less than 30% of the memory. The coarse-grained primitives of Spark significantly worsen the performance of graph computing especially for SSSP, where there are a lot of short-lived iterations and only a fraction of data is touched in most iterations.

5.3 Performance on Machine Learning

Next, we tested Husky's performance on two collaborative filtering algorithms, ALS and asynchronous SGD. ALS is a synchronous algorithm and can be easily parallelized, which is popularly adopted in various systems. The asynchronous SGD, on the other hand, are not effectively supported by many systems as it requires the support of asynchronous and fine-grained updates. We used the song rating data from Yahoo! Music[3] and Netflix movie rating data [7]. Some statistics of the datasets are shown in Table 2.

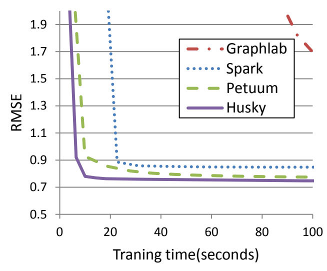
	Users	Items	Ratings
Netflix	480,189	17,770	100,480,507
YahooMusic	1,823,179	136,736	699,640,226

Table 2: Rating datasets

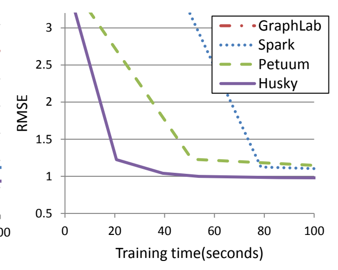
We compare Husky with Petuum, GraphLab, and Spark. Petuum is the state-of-the-art Parameter Server implementation. GraphLab is able to perform machine learning by modeling it as a graph problem. Both Petuum and GraphLab were written in C++. We measured how efficient a system can compute a high quality result based on the given input, by reporting training *root-mean-square error* (*RMSE*) versus training time. *The faster the RMSE decreases over time, the better is the performance of the system.*

Alternating Least Squares. Petuum provides the CCD++ [28] implementation, accelerated by the STRADS scheduler [18], where CCD++ can be regarded as an improved ALS algorithm. GraphLab models parameters as shared states and ensures consistency through locking (it may lock across network). Spark uses the blocked ALS algorithm in its MLlib. ALS implementation in Husky is discussed in Section 4.3.1. The regularization parameter is set to 0.1 and the rank is set to 20 for all systems.

We obtained the results as shown in Figures 9. The overhead of locking over network in GraphLab is too high (also observed in [30]), and we cannot properly plot the GraphLab lines as others. As a dedicated machine learning framework, Petuum performs much better than Spark, while Husky further outperforms Petuum as Husky overcomes the shortage of Parameter Server by avoiding indirect communication.



(a) ALS on Netflix



(b) ALS on YahooMusic

Figure 9: Performance on ALS

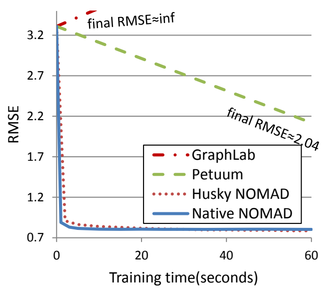
We also remark that comparing the easiness of development using different systems, in Husky computation happens by object interaction and hence it is intuitive to reason the algorithm logic as

[3]http://webscope.sandbox.yahoo.com/

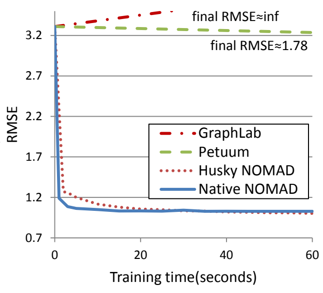
users and products (objects) exchanging their information (latent factors), while mapping such algorithm logic into coarse-grained functional operators is not easy work.

Stochastic Gradient Descent. We implemented NOMAD [30] in Husky, as discussed in Section 4.3.2, and we also compared with its specialized native MPI program by [30] in order to study the system overhead of Husky. We also compared with Petuum and GraphLab as references, both of which are able to exploit asynchronization. It is not clear how to implement asynchronous algorithm in Spark and hence we did not include Spark in this experiment. We set the SGD regularization parameter to 0.05 and the learning rate to 0.01 for all systems.

Figure 10 shows that the native MPI program achieves great efficiency, but the performance of Husky is very close to it. In contrast, we found that the support for asynchronization in both Petuum and GraphLab is still quite limited and the performance is far worse. For GraphLab, the model diverged in all trials. The performance of Petuum is better but still far from satisfactory when compared with Husky and the native MPI program (both converged in several seconds). Note that the performance of Petuum (version 0.93) is similar to that reported in Figure 3 of the Petuum paper [14].



(a) SGD on Netflix



(b) SGD on YahooMusic

Figure 10: Performance on SGD

Compared with the native MPI program, the Husky framework frees researchers from the complexity of developing distributed programs in low-level MPI from scratch. Moreover, porting this pattern into the Husky library results in a re-usable component, which can be composed together with other components into pipelined workflows in Husky with little overhead.

5.4 Performance on Wikipedia Pipeline

Next, we demonstrate that Husky is efficient for pipelined jobs that require context switches. There are a number of dataflow systems, for example, Dryad [15], DryadLINQ [29] and Naiad [21] by Microsoft Research, and also Spark [31] and epiC [16]. We only compared with Spark and Naiad, since Naiad has significantly better performance than Dryad/DryadLINQ [21] and epiC is not open source.

We used the Wikipedia pipelined workflow described in Section 4.4. We also used specialized systems to run each job in the pipeline, i.e., Hadoop for preprocessing (i.e., Steps 1-3 of the workflow), GraphLab for PageRank (Step 4), and Petuum for MF (Step 5).

The result, reported in Figure 11, verifies the efficiency of Husky, as both its overall running time and the running time of each individual job are significantly shorter than those of the other systems. The overhead of context switch between jobs in Husky is negligible. Spark has better overall performance than specialized systems, even though GraphX in Spark is much slower than GraphLab for running PageRank. This is partly because Spark has negligible context switch overhead, and partly because the page-editor matrix is quite sparse and Petuum cannot handle a large sparse matrix efficiently compared with others. Compared with Husky, Spark is still 6.2 times slower.

For Naiad, we used the Naiad LINQ interface to do preprocessing in a similar way as Hadoop and Spark. However, We did not plot the Naiad figures since the preprocessing time for Naiad is even larger than the overall running time of the other systems. We tried but failed to fix the data loading problem in Naiad[4]. However, we did observe that Naiad achieved satisfactory performance for PageRank computation on the page graph (i.e., Step 4 of the workflow), which takes 160 seconds (compared with 157 seconds using GraphLab, 59 seconds using Husky, and 1431 seconds using GraphX). The result of graph computing is also consistent with our results reported in Section 5.2.

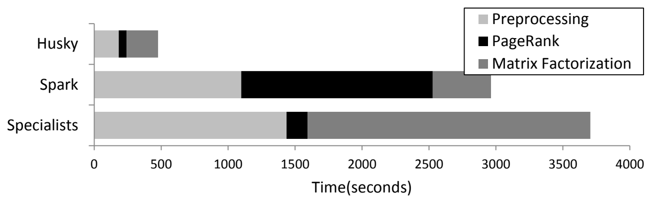


Figure 11: Performance on pipelined jobs

5.5 Performance on Fault Tolerance

We evaluated fault tolerance by randomly disconnecting a *physical* machine during the computation of a job. We compared Husky with Spark, which provides lineage-based recovery—a method that shows better performance than traditional checkpoint recovery [31]. Both Husky and Spark ran iterative PageRank on the Twitter graph, and we simulated a machine failure at the 15-th iteration.

Figure 12 shows that, while Husky took a short time to recover, the recovery time of Spark is even longer than re-running the job. This result verifies the effectiveness of Husky's fault recovery technique. We also examined the cause to Spark's costly fault recovery, which we explain as follows.

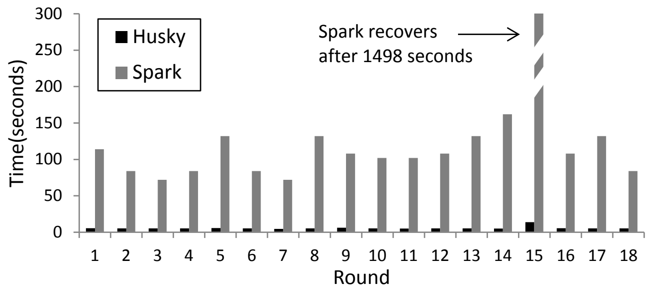


Figure 12: Performance on fault tolerance

There are two possible cases of fault recovery in Spark. The first one is as follows: healthy workers read the shuffle output on the disks of failed machines to recover lost partitions. This works for situations where the machine is running fine but the workers on it

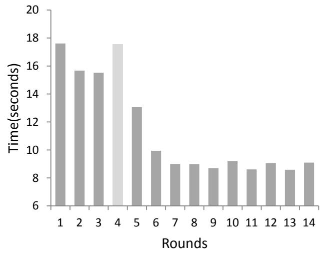
[4]We also communicated with one of the authors for assistance, but learnt that Naiad was no longer a maintained project.

disappear for some reason. The second one is the lineage recovery that can sustain general failures like machine outage and network partition. We found that when we used the first recovery by killing a worker process, the recovery of Spark was fast as reported in [31]. However, when we used lineage recovery by disconnecting a physical machine, the recovery time of Spark was approximately the same as re-running the whole job. This is because jobs like PageRank, as well as most graph algorithms and machine learning algorithms, have *shuffle dependencies* between RDDs, and for these jobs the lineage recovery effectively reduces to a complete re-execution [31]. The lineage of a data partition can also grow too long and cause stack overflow. This is a serious problem for jobs requiring many iterations, e.g., SGD and k-core computation.

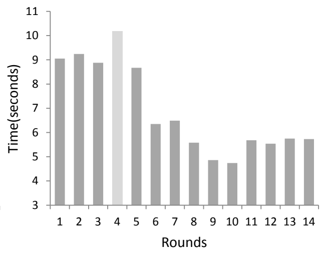
5.6 Performance on Load Balancing

According to Section 3.5, the design of dynamic load balancing strategy should allow Husky to dynamically balance the workloads across different machines according to their workload situations. We tested the effectiveness of this method as well as the overhead, through a machine learning algorithm (ALS on YahooMusic dataset) and a graph algorithm (PageRank on Twitter graph). To simulate heterogeneous configuration and resource contention, we deliberately limit the CPU rate of one machine to half of its original capacity. We set the interval required for the master to collect enough statistics from workers to detect a load imbalance to be 4 rounds (note that an application may adjust the interval length to increase the sensitivity).

As shown in Figures 13a and 13b, the master confirmed the load imbalance at Round 4 and initiated load re-balancing. The master adjusted the hash ring, which triggered a bulk migration among workers. All migrating objects settled at Round 5. Consistent hashing makes load balancing in Husky simple and fast, as the load balancing overhead in Rounds 4 and 5 was only a small fraction of the running time at each round. In contrast, the benefit brought was obvious as the running time at each round after Round 5 was almost halved, since before Round 5 the stragglers were taking the same workloads as other workers that had twice of their computing power.



(a) ALS



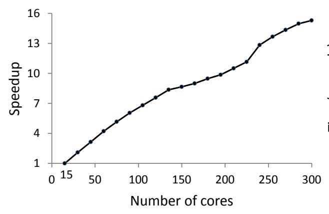
(b) PageRank

Figure 13: Performance on load balancing

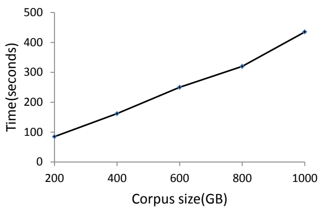
5.7 Performance on Scalability

Finally, we tested the scalability of Husky by an iterative task and a non-iterative task, PageRank and WordCount. We ran PageRank on the WebUK graph. PageRank is both CPU-intensive and network-intensive, and we tested the scalability of Husky by increasing the number of cores used. Then we tested the scalability of Husky by increasing the input data size by running WordCount. As WordCount is I/O-intensive, we fixed the machine settings and varied the size of a synthetic corpus (generated by Hadoop RandomTextWriter) from 200GB to 1,000GB.

The results are reported in Figures 14a and 14b, which verify that Husky achieves nearly linear scalability in both cases.



(a) PageRank



(b) WordCount

Figure 14: Performance on scalability

6. RELATED WORK

Recently there is great interest in building general-purpose distributed systems that support high-level languages, especially functional primitives (e.g., Spark [31], FlumeJava [8], and Flink [3]). The key idea is that functional primitives like map and reduce can be trivially parallelized and they hide the underlying distributed execution from developers. However, it has been shown [11, 23] that although these coarse-grained primitives are ideally suited for bulk processing, they may not be efficient for iterative and fine-grained computation. They also hinder programmers from having finer control and composing more efficient programs. The above discussion indicates that a more efficient distributed framework should support coarse-grained transformations, but also natively allow fine-grained data access when necessary(e.g., pushing messages to a specific object, or migrating some objects to another worker).

In order to attain higher performance in large-scale graph mining or machine learning, researchers have been exploring new computing models. The Pregel model [20] frees from the inefficiency of map and reduce operators by implementing graph-specific message-passing operations, and achieves substantially better performance in distributed graph processing. Blogel [27] introduces the block-centric model to eliminate Pregel's performance bottlenecks caused by real-world graph characteristics, namely skewed degree distribution, large diameter, and high density. With a similar idea as Pregel but in a more general approach, Piccolo [23] enables developers to program by accessing and mutating entries of global key-value tables in a fine-grained manner. Such approach is further exploited in the Parameter Server (PS) model [14, 19, 25], as machine learning applications generally need to have fine-grained or even asynchronous access to the global statistical model. The epiC system [16] uses a similar architecture as PS but stores global states on HDFS. Compared with PS, Piccolo and epiC do not support asynchronous operations, while all of them lack features like Husky's fine-grained object migration and dynamic object creation. The PS systems are specialized for fine-grained machine learning tasks, and do not support coarse-grained operations (e.g., map and reduce) as in Husky. Besides, PS and epiC force all data communication through global key-value tables, which incurs indirect communication and unnecessary network traffic. In addition to the above differences, Husky also supports pipelined tasks and mixed data interaction patterns from different domains (including Pregel, PS, MapReduce, as shown in Section 2.2).

SDG [11] allows developers to parallelize single-machine imperative Java programs by adding annotations. Some features of Husky are also implemented in SDG, for example, independent local states and fine-grained data access. However, SDG focuses more on parallelizing imperative machine learning algorithms. On the contrary, Husky provides a rich set of object primitives and bridges many existing models and components inside one system (as discussed in Sections 2.2 and 4.4).

Stream processing systems such as S4 [22], Storm [26] and MillWheel [1], achieve real-time computation over streaming data. Naiad [21] generalizes the streaming model and abstracts general distributed computation as stream processing problems. Naiad is able to achieve satisfactory performance for distributed computation on both streaming and static graphs, by casting graph problems as processing streams of edges. In contrast, in Husky developers work with meaningful objects and think about their interactions. This can be more intuitive for modeling general offline applications, which is the focus of Husky in this paper. Online stream processing is an important topic, but is beyond the scope of this work and we leave it to future work.

7. CONCLUSIONS

We presented Husky, which is an attempt towards a more efficient and expressive computing model with an easy-to-use interface. We showed that Husky is able to implement applications of different characteristics, for example, *coarse-grained and fine-grained, iterative and non-iterative, synchronous and asynchronous workloads*, and achieves performance close to or better than specialized systems and programs. We also verified that Husky is *scalable, efficient in fault recovery, and effective in load balancing*.

Acknowledgements. The authors would like to thank the reviewers for the encouraging and valuable feedbacks. The authors are partially supported by the Hong Kong GRF 2150851.

8. REFERENCES

[1] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant stream processing at internet scale. *PVLDB*, 6(11):1033–1044, 2013.

[2] Apache Aurora. https://aurora.apache.org/.

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

[4] Apache Giraph. http://giraph.apache.org/.

[5] Apache Hadoop. https://hadoop.apache.org/.

[6] Apache Samza. https://samza.apache.org/.

[7] J. Bennett and S. Lanning. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35, 2007.

[8] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *PLDI*, pages 363–375, 2010.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[11] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Making state explicit for imperative big data processing. In *USENIX ATC*, pages 49–60, 2014.

[12] 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.

[13] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

[14] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *NIPS*, pages 1223–1231, 2013.

[15] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[16] D. Jiang, G. Chen, B. C. Ooi, K. Tan, and S. Wu. epiC: an extensible and scalable system for processing big data. *PVLDB*, 7(7):541–552, 2014.

[17] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, pages 654–663, 1997.

[18] 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.

[19] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, pages 583–598, 2014.

[20] 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.

[21] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP*, pages 439–455, 2013.

[22] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: distributed stream computing platform. In *ICDMW*, pages 170–177, 2010.

[23] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, pages 293–306, 2010.

[24] B. Recht, C. Re, S. J. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701, 2011.

[25] A. J. Smola and S. M. Narayanamurthy. An architecture for parallel topic models. *PVLDB*, 3(1):703–710, 2010.

[26] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. V. Ryaboy. Storm@twitter. In *SIGMOD*, pages 147–156, 2014.

[27] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, 2014.

[28] H. Yu, C. Hsieh, S. Si, and I. S. Dhillon. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In *ICDM*, pages 765–774, 2012.

[29] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, pages 1–14, 2008.

[30] 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.

[31] 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.