STEP: A Distributed Multi-threading Framework Towards Efficient Data Analytics

Yijie Mei, Yanyan Shen, Yanmin Zhu, Linpeng Huang
Department of Computer Science and Engineering
Shanghai Jiao Tong University
{myjisgreat, shenyy, yzhu, lphuang}@sjtu.edu.cn

ABSTRACT
Various general-purpose distributed systems have been proposed to cope with high-diversity applications in the pipeline of Big Data analytics. Most of them provide simple yet effective primitives to simplify distributed programming. While the rigid primitives offer great ease of use to savvy programmers, they probably compromise efficiency in performance and flexibility in data representation and programming specifications, which are critical properties in real systems. In this paper, we discuss the limitations of coarse-grained primitives and aim to provide an alternative for users to have flexible control over distributed programs and operate globally shared data more efficiently. We develop STEP, a novel distributed framework based on in-memory key-value store. The key idea of STEP is to adapt multi-threading in a single machine to a distributed environment. STEP enables users to take fine-grained control over distributed threads and apply task-specific optimizations in a flexible manner. The underlying key-value store serves as distributed shared memory to keep globally shared data. To ensure ease-of-use, STEP offers plentiful effective interfaces in terms of distributed shared data manipulation, cluster management, distributed thread management and synchronization. We conduct extensive experimental studies to evaluate the performance of STEP using real data sets. The results show that STEP outperforms the state-of-the-art general-purpose distributed systems as well as a specialized ML platform in many real applications.

1. INTRODUCTION
Big Data analytics is broadly defined to be a pipeline involving several distinct phases, from data acquisition and cleaning to data integration, and finally data modeling and interpretation [26]. This pipeline poses various challenges in developing distributed systems for Big Data analytics. In particular, the system should fulfill multiple design goals: (1) be flexible to cope with high-diversity applications in the pipeline; (2) be efficient and scalable to handle ever-increasing data; (3) provide easy-to-use APIs to shorten the learning curve for programmers; and (4) be insensitive to underlying operation systems to facilitate easy deployment.

Numerous efforts have been devoted to addressing this “multi-objective optimization” problem, which consequently lead to a proliferation of general-purpose distributed systems for Big Data analytics over the last few decades. Existing general-purpose distributed frameworks typically fall into two categories: (1) disk-based solutions including MapReduce [11] (and its open-source implementation Hadoop [1]) and Dryad [19]; (2) memory-based solutions such as Spark [42]. Spark introduces an in-memory computation model to eliminate expensive I/O cost for iterative computation tasks and is reported to be over 10x faster than Hadoop [42].

The generalization ability of existing distributed systems comes from their simple yet effective functional programming primitives, e.g., map and reduce functions in Hadoop, transformation and action operations over immutable data abstraction – RDDs in Spark. While these high-level operations simplify parallel programming by separating programming paradigm from implementation details, they are still limited in expressiveness and functionality [33,40]. More importantly, the adoption of fixed primitives potentially drive the development of systems towards ease-of-use extreme with compromise of other objectives.

First of all, high-level operations inhibit the opportunities of task-specific optimizations. Particularly, compared with specialized distributed systems, many general-purpose systems perform poorly in real applications. For instance, specialized graph processing systems such as PowerGraph [15] and Giraph [2] can be 10 times faster than Spark and its graph analytics extension GraphX [16,40]; and the distributed machine learning platform Petuum [39] outperforms MLlib [29] (ML library based on Spark) significantly for K-means clustering (in our experiment), thanks to its soft synchronization mechanism and the core component parameter server for global parameter management.

Second, enforcing rigid interfaces limits the flexibility in data representation and programming specification. Real-life data sets may come in various types: structured, semi-structured and unstructured with latent semantics encoded. Transforming these data into predefined forms such as RDDs or key-value pairs can be time-consuming and even cause information loss. Furthermore, specific programming models implemented beneath the interfaces does not allow fine-grained user access control. For example, data processing in Spark are performed via a sequence of RDD transformations while a MapReduce job consists of a map phase followed by a reduce phase. Designing algorithms that fit into these programming models is often a non-trivial task. This can be verified by the emergence of several Spark extensions, e.g., GraphX and MLlib that enable natural expressions for graph analytics and machine learning tasks, respectively.

Finally, we find most provided operations are insufficient to manage globally shared data. Note that Spark does not provide explicit global data management and data sharing is performed inefficiently via broadcasting. However, managing shared data is inevitably important in most machine
learning applications, where the model (e.g., coefficients for regression analysis) to be trained is globally accessed and updated by a cluster of compute nodes iteratively. Things become even worse in the presence of Big Model training, e.g., deep learning models with billions of parameters and topic models with up to hundreds of topics.

To this end, we propose a novel general-purpose distributed framework, named STEP, towards flexible and efficient data analytics. The key idea of STEP is to adapt multi-threading in a single machine to a distributed environment, which is inspired by two important observations: 1) very often, users feel more comfortable writing their programs in procedural languages without fixed primitives; 2) multi-threaded programming such as Pthreads [31] has achieved great success in expressing various programming models and accelerating task execution in a single machine. Hence, rather than define primitive operations, STEP allows users to write distributed multi-threaded programs in a similar way as they accelerate task execution with multi-threading in a centralized manner. We believe that STEP is a suitable alternative for users who would like to have more control over distributed systems to deliver efficient programs.

With STEP, users can specify the number of threads to run over each compute node in a cluster. Furthermore, they are allowed to take fine-grained control over threads’ behaviors via a user-defined thread variants function, where object-oriented programming and various task-specific optimizations can be easily performed. Unlike multi-threading in a single machine, distributed threads running in different compute nodes do not have shared memory space. STEP addresses this problem by leveraging off-the-shelf in-memory key-value stores so that data maintained in key-value stores are globally accessible to all threads in STEP cluster.

The implementation of STEP is challenging due to the following two reasons. First and foremost, the flexibility and expressiveness of distributed multi-threading should not increase the difficulty of programming. High-level interfaces for remote thread management and communication are strongly demanded to make the system easy to use. Second, shared data manipulation requires user programs to interact with the underlying key-value store frequently. However, the dependency on specific key-value store implementation may easily make user programs less reusable.

To address the challenges, STEP offers plentiful interfaces in terms of cluster management, distributed thread management and synchronization, which hide complex implementation details of distributed multi-threading. Most interfaces are designed to be “aligned” with Pthreads APIs so that programmers who are familiar with Pthreads programming can write distributed multi-threaded STEP programs effectively. Furthermore, STEP provides abstraction layers to decouple user programs with specific key-value store implementation and supplies users with easy-to-use distributed shared data manipulation interfaces in C++. By using these interfaces, manipulating shared data in STEP can be expressed as simply as operating local variables. For example, programmers can use normal operator “=” for shared variable assignment. All these efforts differentiate STEP system from MPI-based data analytics solutions [18,22,37], which have some deficiencies in programming, e.g., users are responsible for handling low-level message passing by hand and exploiting globally shared data via intra-node data transfer.

To summarize, the main contributions of our work include:

- We propose a novel general-purpose distributed framework named STEP, which facilitates efficient multi-threaded programming in a distributed manner. With STEP, users can perform fine-grained control over distributed programs and enforce various task-specific optimizations.
- We develop various easy-to-use interfaces in STEP for distributed shared data manipulation, cluster and distributed thread management, which enable users to deliver distributed multi-threaded STEP programs more effectively.
- STEP leverages key-value stores to maintain shared data among distributed threads. We provide abstraction layers to separate shared data access with specific key-value store implementation. An additional cache layer is used to alleviate the heavy burden on key-value store throughput.
- We conduct extensive experiments to evaluate the performance of STEP using various applications and real data sets. The experimental results show that STEP outperforms Spark on logistic regression, K-means and NMF by 8.6x-29x, and runs up to 5.4x faster than specialized ML platform Petuum on K-means and NMF, and up to 3.4x faster than general purpose distributed platform Husky on PageRank.

The remainder of the paper is organized as follows. Section 2 contains some background for STEP system. Section 3 provides an overview of STEP framework. Section 4 introduces key interfaces of STEP. The implementation details are described in Section 5. We evaluate our system in Section 6 and discuss related work in Section 7. Finally, we conclude our paper in Section 8.

2. BACKGROUND

Pthreads Programming. POSIX threads (Pthreads) [31] is an implementation of the standardized C language multi-threading programming interface introduced by IEEE POSIX 1003.1c standard. It can be found on almost any modern POSIX-compliant OS and is ideally suited for parallel programming where multiple threads work concurrently. Pthreads APIs allow developers to manage the life cycle of threads (e.g., creation, joining, terminating) and express various programming models to maximize thread-level parallelism, e.g., boss-worker, pipeline, peer. Every thread has access to a globally shared memory as well as its own private memory space. Pthreads library also includes APIs to synchronize the accesses to shared data and coordinate the activities among threads e.g., semaphore, barrier. We refer readers to more details in [31].

Pthreads library gains great popularity due to its flexible programming models and the lightweight shared memory management. This inspires us to believe that Pthreads programming has promising potential in flexible and efficient distributed computing for Big Data analytics. However, implementing Pthreads APIs in a distributed environment (especially the shared-nothing architecture) is not merely an extension of the centralized version, and hence we propose STEP as an end-to-end solution to this challenge.

| APIs        | Definitions                                    |
|-------------|------------------------------------------------|
| Get          | Getting the value of a key                     |
| Set          | Setting the value of a key                     |
| MGet         | Getting values of multiple keys                |
| Insert       | Atomic key-value pair insertion                |
| Inc/Dec      | Atomic increment/decrement on Integers         |
| Delete       | Delete an existing key and its value           |

Table 1: Support from key-value stores
Cluster Manager
by multi-threaded programs in a single node. Together which is similar to centralized shared memory used accessible to all threads running in STEP cluster. At a high slaves. STEP master and slaves have their own local mem-
in the cluster is selected as the master and the others are system is deployed on a cluster of well-connected compute how distributed multi-threading is performed in STEP.

The
3. THE STEP FRAMEWORK
This section introduces the STEP system and describes how distributed multi-threading is performed in STEP.
We show the architecture of STEP in Figure 1. STEP system is deployed on a cluster of well-connected compute nodes together with a distributed in-memory key-value store. It employs the master-slave architecture, where one node in the cluster is selected as the master and the others are slaves. STEP master and slaves have their own local memory space and manage their own threads. They use the in-memory key-value store as distributed shared memory (DSM). That is, data maintained in the key-value store is accessible to all threads running in STEP cluster. At a high level, distributed shared memory (DSM) ties all threads together which is similar to centralized shared memory used by multi-threaded programs in a single node.

STEP master plays four roles: main thread, cluster manager, DSM manager, and sync controller. Specifically, the main thread is the entry point of a submitted STEP job. Upon creation, the main thread decides the set of slaves and the number of working threads in each slave. It can also declare globally shared data (stored in distributed key-value store) that is accessible to all threads, and specify the behaviors of threads, e.g., when to reach a global synchronization point. The cluster manager is responsible for setting up the cluster and establishing communication channels among STEP master and slaves during initialization. The DSM manager initializes the distributed in-memory key-value store and broadcasts store information (e.g., IP addresses and ports of key-value store servers) to the slaves. The sync controller coordinates all the slaves via message passing. It forwards synchronization messages to the slaves and collects responding messages. It also decides when to resume the execution of blocked working threads in slaves.

Every STEP slave manages a couple of working threads that execute user-defined program over a subset of data concurrently. Working threads in the same slave leverage multiprocessor to achieve thread-level parallelism while those in different slaves perform distributed computing and communicate with each other via shared data in DSM or network messages. Therefore, similar to STEP master, every STEP slave consists of a DSM manager to access shared data in distributed key-value store and a synchronizer that is responsible for processing synchronization messages from master to block/unblock working threads accordingly.

Execution overview. The execution of a STEP job consists of three phases, initialization, followed by distributed multi-threaded execution and finally an output phase.
- Phase 1: During initialization, STEP slaves are first started, waiting for the connection from the master. STEP master sets up the cluster and the distributed key-value store. It then establishes connections with selected slaves. Master’s main thread then declares globally shared data in DSM and creates working threads on each slave via STEP interfaces for distributed thread management. After that, it notifies slaves of the entry function (i.e., thread.proc function) to be executed by slaves’ local working threads.
- Phase 2: Working threads in STEP slaves start their execution by invoking the entry function in parallel with each other. Note that different working threads may share the same entry function that is user-defined. To express different computation logic for working threads, we assign an identifier to each working thread and allow entry function to use thread identifier as one of its input arguments. Typically, an entry function is designed to process a subset of input data, operate shared data in DSM and communicate with other threads (main or working threads) based on STEP interfaces. As we adapt the idea of Pthreads programming to STEP system, various computation and communication patterns among threads can be flexibly expressed by users (similar to multi-threaded programming in a single node). Upon completion of the entry function, all threads can send synchronization messages to master’s sync controller to perform a global synchronization.
- Phase 3: The output of a STEP job is application-dependent. For machine learning tasks, the output is typically the computed model parameters that reside in DSM for global access. For graph analytics tasks such as PageRank computation, each working thread performs PageRank computation over a subset of vertices iteratively and outputs the resulting PageRank values to its local storage. In either case, the main thread in STEP master can optionally collect output in DSM or slaves’ storage to get the complete final results.

STEP allows users to get full control over distributed threads in master and slaves, including the computation logics and communication patterns. However, programming difficulty is increased as a side effect. In particular, this difficulty arises from two aspects: i) complex interaction with underlying key-value store for shared data manipulation; ii) fine-grained cluster and thread management in a distributed manner. In what follows, we introduce easy-to-use STEP in-
4. PROGRAMMING INTERFACES

STEP provides effective programming interfaces to simplify DSM data operations, cluster and distributed thread management, synchronization and vector accumulation. We illustrate STEP programming interfaces in C++ language.

4.1 DSM Data Declaration and Manipulation

We consider three kinds of shared data, shared variables, shared objects and shared arrays, to be store in DSM. Shared data can be declared in a primitive type (i.e., int, float, double, etc) or a reference type pointing to a shared object or shared array.

Shared variable. STEP allows developers to use the macro DefGlobal(NAME,TYPE) to define a shared variable in DSM, where NAME is the variable name and TYPE is the variable type. After declaration, developers can manipulate shared variables in the same way as normal local variables. For example, they can assign a new value to a shared variable by operator “=”, and the data stored in DSM will be updated accordingly.

Shared object. STEP supports key object-oriented features for shared classes in DSM, including templates, dynamic dispatch for virtual functions, encapsulation, inheritance and polymorphism. We introduce a base class DObject in STEP and developers should extend DObject or a subclass of it to construct their own shared classes. Similar to shared variables, developers can use macro Def (NAME,TYPE) in the class body to declare the member variable of a shared class. In STEP, all instances (i.e., objects) of a shared class are shared objects that will be stored in DSM. We provide two APIs NewObj and DelObj to create and delete a shared object, respectively. NewObj function returns a reference def<CLASS_NAME> to the newly created shared object in DSM which behaves like a normal pointer in C++. The members of a shared object can be accessed by the “->” operator through the object reference.

Shared array. STEP provides NewArray and DelArray APIs to allocate and deallociate a shared array in DSM, respectively. References to shared arrays are defined by a template class Array<TYPE>. Developers can use indexing operator “[ ]” to access elements in a shared array, as normal arrays in C++. Moreover, STEP allows developers to perform batch operations over arrays. For example, CopyTo function copies a shared array in DSM to a local array, and CopyFrom function copies data in the opposite direction. Below is an example showing how to operate a shared array arr in STEP.

```
List 1: Shared array example in STEP

Array<float> arr = NewArray<float>(10); //shared array
arr[4] = 3.14;
float local_buf[3] = {1, 2, 3}; //local array
arr->CopyTo(local_buf, 0, 3);
DelArray(arr);
```

4.2 Cluster and Distributed Thread Management

List 2 shows the main interfaces for cluster and distributed thread management in STEP.

```
List 2: Main APIs for cluster and distributed thread management

Cluster management. The HelperInitCluster API is responsible for initializing STEP environment and establishing connections among the compute nodes during initialization. This function acts differently on the master and slaves. Specifically, it parses the arguments from the command line, and then decides whether the current process will run under master mode or slave mode. In master mode, HelperInitCluster initializes the cluster by i) reading the settings from configuration file, ii) connecting STEP master to the selected slaves and key-value store servers, and iii) forwarding configuration information to all slaves. In slave mode, HelperInitCluster makes the slave node wait for the connection from the master and respond to the master’s requests.

The CloseCluster API is used to shut down the cluster, which can only be invoked by STEP master.

Thread management. STEP allows users to specify the number of working threads to be created in each slave. This is achieved by using the DThread class, a pre-defined shared class in STEP. To declare a working thread on a slave node, users can create a DThread object using NewObj API.

The constructor of DThread takes three arguments: i) func is a user-defined entry function (i.e., thread_proc function) for working threads; ii) node_id is the ID of the slave node where the working thread is created; iii) param is the parameter forwarded to the user-defined entry function. Users can get the state of a thread (i.e., alive or completed) via the member function GetState. Note that we declare DThread as a shared class so that all its instances are stored in DSM and publicly available to STEP cluster, which is critical for communication among STEP master and slaves.

4.3 Distributed Thread Synchronization

List 3 lists important STEP interfaces for synchronizing distributed threads. Both DBarrier and DSemaphore are encapsulated as shared classes (i.e., inherited from DObject) whose instances are accessible to all threads in STEP cluster.

```
List 3: Synchronization APIs

The DBarrier class provides barrier synchronization pattern to keep distributed threads (i.e., main thread and working threads) in the same pace, which is useful in performing synchronous iterative computation. Typically, a DBarrier
object is created by the main thread in STEP master. The
constructor in DBARRIER is then invoked to create a barrier
and specify the total number of threads to be synchronized
on the barrier. The reference to a DBARRIER object can be
stored in a shared global variable so that all threads in the
cluster can share this barrier. After setting up all the work-
ing threads in slaves, the main thread calls Enter function
and waits at the barrier until all the working threads reach
the barrier. When the last thread arrives at the barrier, all
the threads will resume their normal execution.

The DSemaphore class allows a specified number of threads
to access a resource. During the creation of a RSemaphore
object, we set a non-negative resource count as its value.
There are two ways to manipulate a semaphore. Acquire
function is used to request a resource and auto-decrement
the resource count; Release function is used to release the
resource and auto-increment the resource count. Threads
that request a resource with non-positive semaphore value
will be blocked until other threads release that resource and
the semaphore value becomes positive.

The above synchronization interfaces provide basic build-
ing blocks for user applications and are designed to be aligned
to Pthreads APIs. With this similarity, developers who are
familiar with traditional multi-threaded programming are
able to perform distributed multi-threading with STEP
effectively.

4.4 Accumulator

We found that many real applications require to perform
vector-wise updates. For example, in PageRank compu-
tation, each working thread maintains a subset of vertices
with their outgoing edges. During each iteration, a working
thread computes the PageRank credits from its own vertices
to the destination vertices along the edges. The credit vec-
tors from different working threads are summed together to
produce new PageRank values for all vertices.

A straightforward way to perform vector accumulation with
STEP is to ask working threads to transfer local vec-
tors to DSM, and then choose one thread to fetch all vectors,
perform final accumulation locally, and forward newly com-
puted vector elements to DSM or the corresponding threads.
Let N be the number of working threads. The above method
incurs high network cost, i.e., the size of data to be trans-
ferred is at least \((2N + 1) \times \text{vector size}\).

STEP provides DAddAccumulator class for users to per-
form vector accumulation more efficiently as well as hide
data transfer details involved in vector accumulation. Users
can create a shared DAddAccumulator object and initialize it
with the number N of working threads and an output shared
array in DSM. The working threads can invoke Accumulate
function defined in DAddAccumulator to send out their local
vectors and STEP will compute the final accumulated result
and store it in the output shared array automatically. The
Accumulate function also acts like a synchronization point
which will not return until all the N threads send out their
local vectors. Our implementation of Accumulate function
reduces the data transfer cost to \((N + 1) \times \text{vector size}\) (see
details in Section 5.2).

4.5 Example: Putting Them All Together

Now we illustrate how to develop applications with STEP
interfaces using logistic regression as an example. Logistic
regression\(^8\) is a widely used discriminative model for
two-class classification\(^\text{3}\). Given a d-dimensional expla-
natory vector \(x\), the logistic regression model is to predict the
probability of a binary response variable \(y\) taking 0 or 1,
based on logistic function. That is,
\[
p(y = 1|x) = \sigma(\theta^T x) = \frac{1}{1 + \exp(-\theta^T x)}(1)
\]
where \(\alpha\) is the step size, and \(\delta = \sum_{p \in B} (y_p - \sigma(\theta^{(i)}^T x_p))x_p\)
is the gradient over mini-batch \(B\) of training data.

The STEP program for the entry function (i.e., thread_proc
function) of working threads and the shared data declara-
tion in logistic regression is provided as follows. We omit
initialization and finalization details for simplicity.

| struct DataPoint{ |
| --- |
| float y; |
| float *x; |
| } |
| float* theta = InitialParam(); |
| std::vector<DataPoint> points = LoadTrainPoint( tid ); |
| float* local_grad = new float[param_len]; |
| for (int i = 0; i < ITERATIONS; i++) { |
| std::fill_n(local_grad, param_len, 0); |
| float dot = 0; |
| for (int j = 0; j < param_len; j++) |
| dot += theta[j] * p.x[j]; |
| for (int j = 0; j < param_len; j++) |
| local_grad[j] *= (p.y - 1 / (1 + exp(-dot))) * p.y; |
| } |
| accu->Accumulate(local_grad, param_len); |
| for (int j = 0; j < param_len; j++) |
| theta[j] += step_size * grad[j]; |
| } |
| } |

\(^3\)We consider binary logistic regression for simplicity. The
solution can be easily extended to multinomial case.

In our design, every working thread maintains the param-
eter vector \(\theta\) in a local array theta (line 10) whose length is
stored in a shared variable param_len. We use a shared array
\(\text{grad}\) in DSM to keep the global gradient vector, and asso-
ciate \(\text{grad}\) with an accumulator accu that sums over thread-
level gradients \(\text{local} \_\text{grad}\) and stores the accumulated result
to \(\text{grad}\). We partition the training set into disjoint mini-
batches and assign them to the working threads uniformly
via a user-defined partition function LoadTrainPoint (line
11). In each iteration, every working thread computes lo-
cal gradient \(\text{local} \_\text{grad}\) based on its mini-batch training data
(line 15-21), and the accumulator accu is used to sum up all
local gradients \(\text{local} \_\text{grad}\) (line 22) to compute the global
gradient \(\text{grad}\). Finally, each working thread updates its local
parameter vector theta by adding step_size*\(\text{grad}\) with theta
(line 23-24). Such computation is repeated by ITERATION
times.
Discussion. Various implementations of thread proc function (i.e., slave proc in the above example) can be adopted to further optimize the performance for logistic regression. For instance, we find that the global gradient grad is fetched by all threads in the same node (line 24). One can improve the example code by letting only one thread in each slave fetch the global gradient and share it with other threads via a local array (since threads within the same node share the local memory space). Moreover, one can use a single thread in each slave to combine all local gradients from that node and then accumulate the combined results via the accumulator. Both optimizations can help reduce data transfer cost between DSM and local memory. Note that the above fine-grained optimizations can hardly be achieved using programming primitives from existing general-purpose distributed systems; and more importantly, the STEP interfaces are useful to achieve these optimizations in a natural and efficient way.

5. IMPLEMENTATION DETAILS

The following subsections discuss the technical details of STEP implementation.\footnote{The full code of STEP is available in \url{https://github.com/STEP-dev/STEP}}

5.1 Distributed Shared Memory Management

STEP provides distributed shared memory (DSM) to store globally shared data among threads in STEP cluster. We implement DSM following a three-layer architecture, as shown in Figure 2.

- The bottom layer contains the off-the-shelf distributed in-memory key-value store which keeps all shared data physically. We use memcached\cite{memcached}, a simple yet powerful object caching system, in our current implementation. We associate each piece of shared data with a unique key and store the (key, shared data) pair into memcached. All the requests sent to the bottom layer are key-value store specific.
- The middle layer, named DSM internal layer, separates user-level DSM data access from specific key-value store implementation. Particularly, it handles unified DSM API calls (e.g., Get() and Set()) from STEP programs, and transforms them into the operations provided by the underlying key-value store. This transformation is important in face of the rapid evolution of key-value stores. That is, we can easily switch to another more efficient key-value store without any modification in user programs.
- The top layer is a DSM cache that leverages spatial and temporal locality to facilitate fast shared data access in DSM. We implement a directory-based distributed memory cache in this layer. It absorbs DSM data access when there is a cache hit. We also allow some DSM API calls to skip DSM cache layer atomic operations on the shared data, e.g., atomic-increment or atomic-decrement on shared counters.

In what follows, we first present our design of shared memory address space in STEP and then provide implementation details in DSM management.

Shared Memory Address Space. STEP allows 64-bit shared memory addresses and organizes the complete DSM space in the granularity of word, i.e., 32 bits. That is, every 64-bit shared memory address identifies a 32-bit chunk in DSM. STEP interprets each shared memory address as a high-order \( x \)-bit object_id plus a low-order \((64 - x)\)-bit field_id. A 32-bit word and 64-bit address size can support up to \( 2^{64} \times 4 \) bytes DSM space. For a data type with over 32 bits, any of its instances will occupy multiple words in DSM. By default, we set \( x \) to 32, which allows STEP to support up to \( 2^{32} \) shared objects and \( 2^{12} \) fields for each of them.

To access a field of an object in DSM, STEP runtime system will fetch its object_id and field_id to compose a 64-bit shared memory address. This address will be used as a key to access the object’s field value in key-value store. In addition to objects, STEP also allows arrays and variables to be stored in DSM. The object_id for an array is a unique array identifier and the field_id is the index of an array element. For shared variables, STEP runtime allocates a unique field_id for each of them. STEP system has a virtual object with object_id equal to 0, which holds all shared variables in the program.

Key-value Store Layer. STEP implements key-value store layer based on libmemcached\cite{libmemcached}, an open-source C client library and tool for memcached server. Note that memcached is originally designed as a fully-associative distributed cache and will discard the oldest data periodically or when running out-of-memory. To address the problem, we disable automatic cache data eviction in memcached via appropriate system parameter settings. However, when the cache is full, subsequent insertions will still throw out-of-memory exceptions. This limitation can be solved by leveraging persistence enabled key-value stores such as memcachedDB\cite{memcachedDB}, which we leave as future work.

STEP stores all shared data as key-value pairs in memcached where every key is a 64-bit shared memory address and the value contains a word-sized chunk. We refer to this implementation as fine-grained DSM. However, fine-grained DSM achieves merely 33\% effective key-value store usage due to the fact that only the value part in a pair is used to store real shared data and the key part is typically unused. Furthermore, in fine-grained DSM, reads and writes to shared data in large sizes may involve a number of network requests and thus is inefficient in terms of data transmission between local memory and DSM.

In face of the above problem, STEP introduces coarse-grained DSM that associates a shared memory address (i.e., the key) with several consecutive words, called a package, and stores (key, package) as a key-value pair in memcached. By default, a package contains 32 words and STEP guarantees that key-value pairs are stored at package-size-aligned shared memory addresses. Coarse-grained DSM re-
duces the number of data transmission requests for large-
size shared data access and hence improves the overall DSM
throughput. Unfortunately, this solution will increase DSM
access latency. That is, an update to a single word in the
package requires to access the whole package from DSM.

STEP allows developers to decide which DSM mode to use
via STEP configuration file. According to our experimental
results, coarse-grained DSM achieves better performance
than fine-grained DSM in many real-world applications.

**DSM Internal Layer.** DSM internal layer provides a
set of DSM APIs functions for getting and setting values
of shared data of user programs on DSM. Specifically, users
access STEP shared data in the same way as operating data
in local memory (recall the details in Section 4.1) and STEP
library transforms all DSM data accesses into DSM API
calls. We want to emphasize that DSM APIs are invisible
to users and only accessible to STEP library internally.

STEP implements all DSM APIs based on the interfaces
from the underlying key-value store (see Table 1). Typically,
DSM APIs include two kinds of operations: setting or
getting shared data given its address. All the set functions
have three parameters. The first and second parameters
represent the object_id and field_id of the shared data, re-
spectively. Concatenating these two parameters produces a
64-bit shared memory address to locate the data. The third
parameter is the updated data value. The address-value
pair will be passed to the underlying key-value store API
functions, e.g., memcached_set() in memcached.

Similarly, all the get functions involve two parameters rep-
resenting the object_id and field_id of the shared data, respec-
tively. In get functions, we first concatenate the input pa-
rameters to compose a shared memory address. We then
call the corresponding key-value store API method (e.g.,
memcached_get() in memcached) with the shared memory
address as the key. The Finally, the get functions return the
data fetched by the key-value store.

**DSM Cache Layer.** The development of DSM cache
layer is motivated by two important observations. First, fre-
quent shared data access incurs a large amount of network
communication cost. Second, the performance of STEP is
compromised by the limited throughput of the underlying
key-value store. Hence, we implement a write-through dis-
tributed DSM cache with the purpose of reducing network-
communication cost. Second, the performance of STEP is
sensitive to the sub-vectors in local memory and updates the corre-
responding elements in the output array. This method reduces
the amount of transferred data to \((N+1)\times\text{vector.size}\).

In some applications, the vectors to be accumulated are
sparse, i.e., with few non-zero elements. Hence, STEP pro-
vides three modes of accumulator: dense, sparse, auto. The
sparse mode behaves as described above. In sparse mode,
a vector is represented by (index, non-zero element) pairs.
Transferring pairs for non-zero elements incurs lower net-
work cost if the vector is sparse. In auto mode, STEP checks
if it is beneficial to convert vectors to the pairs and auto-
matically chooses the mode with lower network cost.

**5.3  Synchronization**

We implemented two thread synchronization mechanisms
in STEP. The first mechanism is barrier. Specifically, STEP
master maintains a counter for each distributed barrier in
STEP cluster. When a thread (main or working) enters the
barrier, it sends a message to the sync controller on master
node to increase the counter by 1. Every thread entering the
barrier waits for the release of the barrier. When the counter
reaches the threshold defined on barrier creation, sync con-
troller broadcasts a “release” command to all the threads
blocked by this barrier. The synchronizer of each slave node
then resumes the execution of the blocked threads.

The second synchronization mechanism is semaphore. The
implementation of semaphore is similar to that of barrier.
The sync controller in STEP master manages a counter
upon the creation of a distributed semaphore. When a thread
Acquires control over a semaphore, sync controller
will check the count of the semaphore. If the count is non-
positive, it puts the thread into a waiting queue for the
semaphore. Otherwise, the counter is decreased by 1 and
the thread is left unblocked. When a thread holding the
semaphore calls Release method, sync controller will in-
crease the counter by 1. When the counter becomes positive, sync controller resumes the execution of the first thread in the waiting queue (in FIFO manner) if any, and auto-decrements the counter.

5.4 Fault Tolerance

STEP leverages heartbeat messages to detect node failures. That is, every slave node is requested to send heartbeats to STEP master periodically. If the master does not receive the heartbeat message from a slave over a fixed time interval, the slave is considered to be dead. Upon failures, STEP master will send “recovery” message to all the remaining slaves and the recovery process starts immediately.

STEP adopts checkpoint-based recovery mechanism. For synchronous iterative applications, we make checkpoints every a few iterations. This is achieved by inserting appropriate checkpoint logic right before the barrier is released. When doing a checkpoint, STEP uses fault-tolerant distributed file system to maintain a consistent copy of the data in DSM and any important information to be materialized. During recovery, STEP master creates new working threads in healthy nodes to replace the failed ones. All the threads then rollback to the latest checkpoint, re-load input data if necessary and redo the computation with the latest consistent copy of DSM. Besides, STEP provides an abstract Checkpoint class with two functions DoCheckpoint() and DoRestart() for users to store extra information for recovery. Users may extend Checkpoint class to specify the variables to be persisted during checkpointing. They can also transform program-specific state to a Checkpoint instance and vice versa, while STEP is responsible for invoking DoCheckpoint() during checkpointing (to persist Checkpoint instance) and calling DoRestart() during recovery (to restore program state from the materialized Checkpoint instance). Doing checkpointing for non-iterative (or asynchronous) computation tasks is more subtle because no barrier is available. To address the problem, STEP master is able to send a “checkpoint” command to pause the execution of all slaves. This command is used to enforce a virtual barrier. Upon receiving the command, every slave stops all its working threads (after finishing the computation in hand) to create a checkpoint. Similarly, users can leverage Checkpoint class to indicate a consistent state of the program that will be persisted by STEP automatically.

6. EXPERIMENTAL EVALUATION

6.1 Experiment Setup

The experimental study was conducted on our in-house cluster. The cluster consists of 16 Dell M630 compute nodes, each of which is equipped with one Intel E5-2609-V3 CPU, 64GB RAM and 600GB hard disk, running CentOS 6.6 operating system. All the nodes are hosted on one rack and connected by a 10Gbps switch.

We compare STEP with two Spark extensions – MLib [29] and GraphX [16], Husky [40] and a specialized machine learning platform Petuum [39]. We installed version-2.1.1 of Spark and the latest versions of Petuum and Husky. We also carefully tuned the settings to get the best system performance. For STEP, we use auto mode for the accumulator and coarse-grained DSM mode by default.

6.2 Applications and Datasets

Applications. We evaluate the performance of different distributed systems using four applications: logistic regression, K-means, non-negative matrix factorization (NMF) and PageRank. These applications are considered to capture different workloads: machine learning tasks, graph analytics tasks, requiring little or substantial amount of shared data to be managed. We implemented all the applications in STEP system. For Spark, Petuum and Husky, we directly used the implementations from the libraries or examples shipped with these open-source systems. We guarantee that each application is implemented using the same algorithm in different systems. For instance, both STEP and Spark adopt the mini-batch SGD algorithm for logistic regression.

Table 2 shows the available implementations for the applications and the corresponding code lengths provided by MLlib and GraphX in Spark, Petuum Bosen and Husky. For STEP applications, each of them is written in one source file using C++. Spark applications are implemented in the corresponding RDDs. We measure the code lengths of KMeans.scala, LogisticRegression.scala and PageRank scala from Spark packages excluding the comments and unrelated code. For Petuum, we consider the codes in .cpp, .hpp, .h files for each application under the app directory and exclude the relevant codes from Petuum’s native library. We count the lines of .cpp files for Husky PageRank application under the directory of benchmarks with relevant system library source codes excluded.

We can see that STEP applications require comparable code lengths compared with Spark and Petuum over all the applications, which illustrates the effectiveness of programming with STEP. Husky requires the shortest code length for PageRank. This is because Husky utilizes Boost library to parse input data, leading to shorter code length in Husky.

Datasets. Table 3 provides a detailed description for all the datasets used throughout the experiments.

- Logistic regression. We consider two datasets for logistic regression: GENE and LRS. GENE [32] is a gene-expression dataset (accession number: E-TABM-185). It contains 22K data rows over about 6K features representing different cell lines, tissue types, etc. LRS [1] is a synthetic dataset generated by the COUNT library in R language. LRS contains 30K features and we use LRS to evaluate the effects of feature dimensionality on system performance.

- K-means. We use two datasets for K-means computation: FOREST and KMS. FOREST [4] contains forest cover types for observations (30x30 meter cell) from US Forest Service Region 2 Resource Information System. Each observa-

Table 2: Code lengths in available implementations

| Application      | STEP | Petuum | Husky | Spark |
|------------------|------|--------|-------|-------|
| Logistic Regression | 323  | -      | -     | 213   |
| K-means          | 285  | 1446   | -     | 372   |
| NMF               | 311  | 1144   | -     | -     |
| PageRank         | 279  | -      | 101   | 215   |

http://archive.ics.uci.edu/ml/datasets/Covertype

https://spark.apache.org/docs/2.1.1/mllib-guide.html
https://spark.apache.org/docs/2.1.1/graphx-programming-guide.html
https://github.com/petuum/bosen/tree/stable/app
https://github.com/husky-team/husky
http://finzi.psych.upenn.edu/library/COUNT/html/logit_sva.html
http://archive.ics.uci.edu/ml/datasets/Covertype
Table 3: Dataset descriptions

| Datasets     | #features | #data rows |
|--------------|-----------|------------|
| Logistic     | GENE      | 6896       | 22283      |
|              | LRS       | 30720      | 30000      |
| K-means      | FOREST    | 34         | 581012     |
|              | KMS       | 4096       | 200000     |
| NMF          | NETFLIX   | 17700      | 384000     |
|              | NMFS      | 8192       | 500000     |
| PageRank     | LF        | 3997962    | 34681189   |
|              | FRIEND    | 656898666  | 1800067135 |

Table 4: Parameter ranges

| Parameter          | Range          |
|--------------------|----------------|
| K-means            | #clusters(KMS) 10, 20, 40, 60, 120, 256 |
| NMF                | Factorization rank 8, 16, 32, 64, 128 |
| ALi                | #iterations 6, 8, 10, 12, 14 |

Figure 3: Results on different DSM modes (NMF)

(a) NETFLIX
(b) NMFS

Figure 4: Logistic regression results on GENE dataset

(a) # of iterations
(b) # of nodes

Figure 5: Logistic regression results on LRS dataset

(a) # of Iterations
(b) # of nodes

6.3 Fine-grained DSM vs Coarse-grained DSM

We first evaluate the performance of STEP using two different modes of DSM: fine-grained mode and coarse-grained mode (see details in Section 6.1). Figure 3 shows the results of two modes for NMF. Coarse-grained mode outperforms fine-grained mode over two datasets. On average, STEP with coarse-grained DSM runs 2.7 and 1.6 times faster than that with fine-grained DSM on NETFLIX and NMFS datasets, respectively. In NMF, we need to retrieve the factorized matrices from DSM in each iteration. Bulk loading of shared data reduces the number of data access requests and makes better use of the bandwidth of the underlying key-value store. Furthermore, coarse-grained DSM is more robust to the number of iterations, i.e., the increasing rate of the running time is lower than that with fine-grained DSM. We observe similar results on other applications and omit the figures due to redundancy. Henceforth, all the results of STEP system are based on coarse-grained DSM.

6.4 Logistic Regression

The algorithm for logistic regression has been discussed in Section 4.5. Figure 4(a) shows the running time of different systems with various number of iterations on GENE dataset. STEP outperforms Spark over all the iteration numbers. On average, it runs 21.6 times faster than Spark. Both STEP and Spark requires longer running time with the increase in the number of iterations. In Spark, partially updated regression parameters are first forwarded to the master who will aggregate all the updates and broadcast the result to all the workers. In contrast, STEP uses accumulator to perform aggregation in parallel and leverages DSM to keep the shared parameters, both of which reduce the computation and communication burden on the master. Figure 4(b) provides the results of varying the number of compute nodes over GENE dataset. STEP runs much faster than Spark over all the node numbers. We observe both systems do not benefit significantly from using more nodes. The reason may be that the performance gain of adding more nodes is suppressed by the increase of communication cost.

Figure 5(a) shows the logistic regression results on LRS dataset. Both STEP and Spark require longer running time as the number of iterations increases. Similar to GENE dataset, STEP runs much faster than Spark over all the iterations, i.e., 29 times faster than Spark on average. Figure 5(b) provides the running time on LRS dataset by varying the number of compute nodes. When the number of nodes increases from 8 to 16, the running time of STEP is decreased linearly. Spark achieves the best performance with 12 nodes and the performance is decreased by using
16 nodes. This is because LRS is a large high-dimensional dataset. When using more nodes, Spark master has to aggregate more updated parameters and broadcast the results to more workers, which incurs very high communication cost and compromises the gain in computation time.

### 6.5 K-means

We first study the effects of varying the number of clusters on K-means clustering using KMS dataset. Figure 6(a) shows the running time for various K. Spark requires the longest running time over all the values of K, and the disadvantage becomes more significant when K increases. For K=256, Spark is over 10 times slower than STEP. We attribute the poor performance to the deficiency of the programming language of Spark and the master node who is responsible for computing and broadcasting K centers in each iteration. STEP performs better than Petuum for all values of K. For K=256, STEP runs 5.4 times faster than Petuum. The reason is that Petuum’s K-means application transfers K center vectors by sparse vectors, which may result in more communications than using dense vectors in some datasets. While the accumulator in STEP can automatically detect the sparsity of the vector, choosing more efficient way to transfer the shared data. The running time of all three systems increases when K becomes larger, due to the higher computation and communication cost.

Figure 6(b) provides the running time on KMS dataset when the number of iterations varies from 6 to 14. All the systems require longer running time as the number of iterations increases. On average, STEP runs 5.4 and 8.6 times faster than Petuum and Spark, respectively. The advantage of STEP is consistent over all the iteration numbers. Figure 6(c) reports the results with various node numbers. STEP is more efficient than Petuum and Spark when the number of nodes increases from 8 to 16. With 8 compute nodes, STEP is 2.7 and 12 times faster than Petuum and Spark, respectively. When we double the number of nodes from 8 to 16, Petuum achieves the highest speedup of 1.96, followed by a speedup of 1.83 by Spark. STEP achieves 1.67 speedup, which is lower than Petuum and Spark. This is because the computation in STEP is already efficient with a small number of compute nodes, and the performance does not benefit too much from using more nodes.

Figure 7 provides K-means results on FOREST dataset. STEP outperforms Spark and Petuum over all the values of K. Both STEP and Petuum takes longer running time when K becomes larger. This is because larger values of K require more distance computations and comparisons. We observe similar results with the increase of iteration numbers on FOREST dataset, where STEP runs 1.5 times faster than Petuum and 28 times faster than Spark on average.

### 6.6 NMF

Given a factorization rank $k$, NMF tries to factorize an $n \times m$ input matrix $R$ into two matrices $P$ ($n \times k$) and $Q$ ($k \times m$) so that the loss function $L = \|R - PQ\|^2$ is minimized. Both STEP and Petuum adopt the SGD algorithm that learns two matrices $P$ and $Q$ iteratively.

Figure 8 shows the running time of STEP and Petuum on NMFS dataset. We first evaluate the effects of varying the number of data points, i.e., data size. We generated 500K data points for NMFS and allow the system to load a subset of data to get a smaller data size. Figure 8(a) provides the running time. When the number of data points increases, both STEP and Petuum require longer running time to learn $P$ and $Q$. This is because of the longer data loading time and more complex training process. STEP outperforms Petuum over all the data point numbers, i.e., 1.1 times faster than Petuum on average. We observe the computation time in STEP is less than that in Petuum. This may result from the simple shared memory abstraction and efficient shared data manipulation in STEP. Figure 8(b) shows the effects of varying the number of iterations on the performance. For both STEP and Petuum, the running time increases linearly with the number of iterations. On average, STEP is 1.14 times faster than Petuum over all the iteration numbers. Figure 8(c) provides the speedup by varying the number of nodes. STEP outperforms Petuum over all the node numbers. Both systems achieve similar speedup (i.e., 1.8) when the node number increases from 8 to 16.
Figure 8: NMF results on NMFS dataset

Figure 9: NMF results on NETFLIX dataset

Figure 10: PageRank results

Figure 9 shows the results of NMF on NETFLIX dataset. STEP outperforms Petuum when the factorization rank exceeds 16. The difference becomes larger with the increase of factorization rank. Since NMF with a large factorization rank puts more burden on both CPU and network, this result verifies the efficiency of STEP over the workloads with high computation and communication cost. Both STEP and Petuum require longer running time when the number of iterations increases. STEP runs 1.16 times faster than Petuum over all the iteration numbers. We observe that Petuum requires more computation time than STEP and hence achieves better speedup by adding more nodes.

6.7 PageRank

We now present the results of PageRank computation. Spark runs more than 2 hours over the large FRIEND dataset. We found Spark incurs huge memory consumption during iterative computation and RDDs are flushed to disk from time to time. Hence, we only report the results of STEP and Husky on FRIEND dataset.

Figure 10(a) shows the running time by varying the number of iterations on LJ dataset. On average, STEP runs 20.8 times faster than Spark, and outperforms Husky by a factor of 1.75. Husky performs vertex-to-vertex message forwarding for PageRank computation and the communication cost in each iteration is proportional to the number of edges. In contrast, STEP leverages accumulator to compute PageRank values in parallel where the communication cost is proportional to the number of vertices. Furthermore, the coarse-grained DSM exhibits high performance in accessing PageRank values for all vertices.

Figure 10(b) provides the running time of different systems by varying the number of nodes on LJ dataset. On average, STEP runs 2.1 and 19.6 times faster than Husky and Spark, respectively. STEP achieves lower speedup than Husky (1.4 vs 2) when the node number increases from 8 to 16, as the computation in STEP is already efficient with 8 compute nodes. The running time of Spark does not vary too much when the number of compute nodes is doubled. While adding more nodes improves thread-level parallelism, the network communication cost is increased as a side effect.

Figure 10(c) presents the running time on FRIEND dataset. STEP runs about 3 times faster than Husky over all the iteration numbers. The running time of both systems increases linearly with the number of iterations. Figure 10(d) shows the results of varying the number of compute nodes. The running time of STEP and Husky decreases linearly with the increase of compute nodes, which illustrates the scalability of both systems. STEP outperforms Husky over all the node numbers. We attribute the advantage of STEP to the efficient accesses and updates of PageRank values via coarse-grained DSM and accumulator. As mentioned before, the communication cost of STEP in each iteration is proportional to the number of vertices, which is much lower than the total size of messages forwarded in Husky.

6.8 Fault Tolerance
We now evaluate the performance of fault tolerance in STEP. We tested two recovery methods used in STEP as follows. In single-node recovery, when a node fails, one healthy node will take over the task for the failed node. Namely, all the failed working threads will be recreated in a healthy node. In multi-node recovery, we recreate failed threads in multiple compute nodes that perform recovery task collaboratively. To simulate a node failure, we disconnect a slave in the 6-th iteration when running K-means over a 16-node cluster. Figure 11 shows the recovery performance of STEP. Each of the first 5 iterations takes about 46ms. In iteration 6, a node with 4 working threads fails. In single-node recovery, STEP requires 196ms to load data using one healthy node and redo the computation for the 6-th iteration (we may require all the nodes to redo all the iterations since the latest checkpoint in other applications). Among the 196ms, we find that data loading occupies 169ms and K-means computation takes 27ms. multi-node recovery initializes four threads in different healthy nodes. It outperforms single-node recovery by using only 63ms to recover from the failure, where 40ms and 23ms are used for data loading and recomputation, respectively. This is because multi-node recovery performs data loading and recomputation in a distributed manner. We also observe that both methods used shorter time to do the recomputation, compared with the normal execution time (about 46ms) before and after iteration 6. The reason may be the reduced competition in network bandwidth where only the recovering threads need to interact with DSM and enter barrier, while the others have already finished the iteration and been waiting at the barrier asynchronously.

7. RELATED WORK

Most existing works [5,10,11,19,21,36,42] focused on developing general-purpose distributed systems for efficient Big Data analytics. They provide functional primitives to offer great ease-of-use to developers with the compromise of efficiency and flexibility. In contrast, STEP provides flexible interfaces that allow users to take fine-grained control over distributed threads in a simple yet effective way. Recently Husky [40] was introduced as a flexible computing framework. Similar to STEP, it allows object-oriented programming for distributed workloads. However, the objects in STEP and Husky are used in different ways. Husky adopts object-centric programming model where the data are viewed as objects and objects can communicate with each other via in-class methods. In STEP, we regard thread as computation unit that is able to manipulate objects (or other kinds of data) and communicates with other threads via distributed shared memory (based on key-value stores). Moreover, we propose distributed shared data manipulation interfaces to simplify shared data operations and provide an abstraction stack separating user programs from specific key-value store implementation. STEP is also evaluated to be more efficient than Husky over data in large size.

To achieve better performance, many efforts have been devoted to developing specialized systems for particular classes of applications such as graph analytics [2,15,27,28,34,35] and machine learning tasks [6,38,39]. Pregel [28] follows the Bulk Synchronous Parallel (BSP) model and proposes a vertex-centric computation model which is more efficient than MapReduce-based frameworks in distributed graph processing. GraphLab [27] provides asynchronous graph computation to get further performance improvement. Petuum [39] emerges as a distributed platform for ML applications. It uses parameter server to store intermediate results in the form of matrices. It also introduces Stale Synchronous Parallel (SSP) to trade off between fully synchronous and fully asynchronous modes for model training. Our work focuses on developing general-purpose distributed systems to cope with complex data analytics pipeline. STEP offers high flexibility to express different classes of applications effectively. We also experimentally show the high efficiency of STEP compared with the specialized systems.

Prior DSM designs [24] provided strong consistency like sequence consistency, which incurs high communication cost for applications with frequent writes. Recent researches [9,30] adopted Partitioned Global Address Space (PGAS) model to exploit data locality, where each partition of the global address space is local to a node. Different from existing DSM solutions, STEP leverages distributed key-value stores [12,14,26] to maintain globally shared data. Different key-value store implementation provides slightly different interfaces and functionalities. STEP has decoupled the specific key-value store implementation from shared memory management by introducing a DSM internal layer. We use memcached in our current implementation and STEP can perform a light-weight switch to other key-value stores.

8. CONCLUSION

In this paper, we proposed a distributed framework named STEP, towards flexible and efficient data analytics. It enables users to perform fine-grained control over distributed multi-threading and apply various application-specific optimizations effectively. Pthreads-like interfaces for cluster and thread management are offered to simplify programming with STEP. STEP leverages the off-the-shelf key-value stores to keep globally shared data. It contains distributed shared data manipulation interfaces so that operations on shared data can be expressed in a similar way as operating local variables. We also develop abstraction layers to separate user programs with specific key-value store implementation. We evaluate the performance of STEP in real applications, and showed that the system is both flexible and efficient.
9. REFERENCES

[1] http://hadoop.apache.org/
[2] http://giraph.apache.org/
[3] http://libmemcached.org/libMemcached.html
[4] http://memcachedb.org/
[5] https://flink.apache.org/
[6] http://mahout.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. M. Bishop. Pattern Recognition and Machine Learning. Springer, 2006.
[9] B. L. Chamberlain, D. Callahan, and H. P. Zima. Large scale parallel I/O for scientific simulations. In EuroSys, pages 397–408, 2004.
[10] M. A. Heroux, R. A. Bartlett, V. E. Coffman, S. D!]
in the hands of multimedia users. In *Proceedings of the 23rd ACM international conference on Multimedia*, pages 25–34. ACM, 2015.

[39] E. P. Xing, Q. Ho, W. Dai, J.-K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. In *KDD*, pages 1335–1344, 2015.

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

[41] J. Yuan, F. Gao, Q. Ho, W. Dai, J. Wei, X. Zheng, E. P. Xing, T.-Y. Liu, and W.-Y. Ma. Lightlda: Big topic models on modest computer clusters. In *WWW*, pages 1351–1361, 2015.

[42] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, pages 10–10, 2010.