Received: 00 Month 0000 | Revised: 00 Month 0000 | Accepted: 00 Month 0000

DOI: xxx/xxxx

RESEARCH

Twister2: Design of a Big Data Toolkit

Supun Kamburugamuve* | **Kannan Govindarajan** | **Pulasthi Wickramasinghe** | **Vibhatha Abeykoon** | **Geoffrey Fox**

[1]School of Informatics, Computing, & Engineering, Indiana University, Bloomington, Indiana, USA

Correspondence
*Supun Kamburugamuve. Email: skamburu@indiana.edu

Summary

Data-driven applications are essential to handle the ever-increasing volume, velocity, and veracity of data generated by sources such as the Web and Internet of Things devices. Simultaneously, an event-driven computational paradigm is emerging as the core of modern systems designed for database queries, data analytics, and on-demand applications. Modern big data processing runtimes and asynchronous many task (AMT) systems from high performance computing (HPC) community have adopted dataflow event-driven model. The services are increasingly moving to an event-driven model in the form of Function as a Service (FaaS) to compose services. An event-driven runtime designed for data processing consists of well-understood components such as communication, scheduling, and fault tolerance. Different design choices adopted by these components determine the type of applications a system can support efficiently. We find that modern systems are limited to specific sets of applications because they have been designed with fixed choices that cannot be changed easily. In this paper, we present a loosely coupled component-based design of a big data toolkit where each component can have different implementations to support various applications. Such a polymorphic design would allow services and data analytics to be integrated seamlessly and expand from edge to cloud to HPC environments.

KEYWORDS:
Big data, Event-driven computing, Dataflow, High Performance Computing

1 | INTRODUCTION

Big data has been characterized by the ever-increasing velocity, volume, and veracity of the data generated from various sources, ranging from web users to Internet of Things devices to large scientific equipment. The data have to be processed as individual streams and analyzed collectively, either in streaming or batch settings for knowledge discovery with both database queries and sophisticated machine learning. These applications need to run as services in cloud environments as well as traditional high performance clusters. With the proliferation of cloud-based systems and Internet of Things, fog computing (1) is adding another dimension to these applications where part of the processing has to occur near the devices.

Parallel and distributed computing are essential to process big data owing to the data being naturally distributed and processing often requiring high performance in compute, communicate and I/O arenas. Over the years, the High Performance Computing community has developed frameworks such as message passing interface to execute computationally intensive parallel applications efficiently. HPC applications target high performance hardware, including low latency networks due to the scale of the applications and the required tight synchronous parallel operations. Big data applications have been developed for commodity hardware with Ethernet connections seen in the cloud. Because of this, they are more suitable for executing asynchronous parallel applications with high computation to communication ratios. Recently, we have observed that more capable hardware comparable to HPC clusters is being added to modern clouds due to increasing demand for cloud applications in deep learning

[0]**Abbreviations:** Big data, Serverless Computing, Event-driven

and machine learning. These trends suggest that HPC and cloud are merging, and we need frameworks that combine the capabilities of both big data and HPC frameworks.

There are many properties of data applications that influence the design of those frameworks developed to process them. Numerous application classes exist, including database queries, management, and data analytics, from complex machine learning to pleasingly parallel event processing. A common issue is that the data can be too big to fit into the memory of even a large cluster. In another aspect, it is impractical to always expect a balanced data set from the processing standpoint across the nodes. This follows from the fact that initial data in the raw form is usually not load balanced and often require too much time and disk space to balance the data. Also, the batch data processing is often insufficient, as much data is streamed and needs to be processed online with reasonable time constraints before being stored to disk. Finally, the data may be varied and have processing time that varies between data points and across iterations of algorithms.

Even though MPI is designed as a generic messaging framework, a developer has to focus on file access, with disks in case of insufficient memory and relying mostly on send/receive operations to develop higher level communication operations in order to express communication in a big data application. Adding to this mix is the increasing complexity of hardware, with the explosion of many-core and multi-core processors having different memory hierarchies. It is becoming burdensome to develop efficient applications on these new architectures using the low-level capabilities provided by MPI. Meanwhile, the success of Harp (2) has highlighted the importance of the Map-Collective computing paradigm.

The dataflow (3) computation model has been presented as a way to hide some of the system-level details from the user in developing parallel applications. With dataflow, an application is represented as a graph with nodes doing computations and edges indicating communications between the nodes. A computation at a node is activated when it receives events through its inputs. A well-designed dataflow framework hides the low-level details such as communications, concurrency, and disk I/O, allowing the developer to focus on the application itself. Every major big data processing system has been developed according to the dataflow model, and the HPC community has also developed asynchronous many tasks systems (AMT) according to the same model. AMT systems mostly focus on computationally intensive applications, and there is ongoing research to make them more efficient and productive. We find that big data systems developed according to a dataflow model are inefficient in computationally intensive applications with tightly synchronized parallel operations (4), while AMT systems are not optimized for data processing.

At the core of the dataflow model is an event-driven architecture where tasks act upon incoming events (messages) and produce output events. In general, a task can be viewed as a function activated by an event. The cloud-based services architecture is moving to an increasingly event-driven model for composing services in the form of Function as a Service (FaaS). FaaS is especially appealing to IoT applications where the data is event-based in its natural form. Coupled with microservices and server-less computing, FaaS is driving next-generation services in the cloud and can be extended to the edge.

Because of the underlying event-driven nature of both data analytics and message-driven services architecture, we can find many common aspects among the frameworks designed to process data and services. Such architectures can be decomposed into components such as resource provisioning, communication, task scheduling, task execution, data management, fault tolerance mechanisms, and user APIs. High-level design choices are available at each of these layers that will determine the type of applications a framework composed of these layers can support efficiently. We observe that modern systems are designed with fixed sets of design choices at each layer, rendering them only suitable for a narrow set of applications. Because of the common underlying model, it is possible to build each component separately with clear abstractions supporting different design choices. We propose to design and build a polymorphic system by using these components to produce a system according to the requirements of the applications, which we term the toolkit approach. We believe such an approach will allow the system to be configured to support different types of applications efficiently. The authors are actively pursuing a project called Twister2, encompassing the concept of the toolkit. Server-less FaaS is a good approach to building cloud native applications (5, 6) and in this way, Twister2 will be a cloud native framework.

This paper provides the following contributions: 1) A study of different application areas and how a common computation model fits them; 2) Design of a component-based approach for data analysis with various choices available at each component and how they affect the applications. The rest of the paper is organized as follows. Section 2 discusses the related work in the area. Next section 3 categorizes data applications into broad areas and introduces the processing requirements. Section 4 discusses the components of our approach. The next section 5 details implications of the design and section 6 concludes the paper.

2 | RELATED WORK

Hadoop (7) was the first major open-source platform developed to process large amounts of data in parallel. The map-reduce (8) functional model introduced by Hadoop is well understood and adapted for writing distributed pleasingly parallel and one-pass applications. Coupled with Java, it provides a great tool for average programmers to process data in parallel. Soon enough, though, the shortcomings of HadoopâĂŹs simple API and its disk-based communications (9) became apparent, and systems such as Apache Spark (10) and Apache Flink (11) were developed to overcome them. These systems are designed according to the dataflow model and their execution models and APIs closely follow dataflow semantics. Some other examples of batch processing systems include Microsoft Naiad (12), Apache Apex and Google Dataflow (13). It is interesting to note that even with all its well-known inefficiencies, Hadoop is still being used by many people for data processing. Apart from the batch processing systems mentioned above, there are also streaming systems that can process data in real time which also adhere to the dataflow model. Further open source streaming system examples include Apache Storm (14), Twitter Heron (15), Google Millwheel (16), Apache Samza (17) and Flink (11). Note that some of the systems process both streaming and batch data in a unified way such as Apache Apex, Google Dataflow, Naiad, and Apache Flink. Apache Beam (13) is a project developed to provide a unified API for both batch and streaming pipelines. It acts as a compiler and can translate a program written in its API to a supported batch or streaming runtime. Prior to modern distributed streaming systems, research was done on shared memory streaming systems, including StreamIt (18), Borealis (19), Spade (20) and S4 (21).

There are synergies between HPC and big data systems, and authors (22, 23) among others (24) have expressed the need to enhance these systems by taking ideas from each other. In previous work (25, 26) we have identified the general implications of threads and processes, cache, memory management in NUMA (27), as well as multi-core settings for machine learning algorithms with MPI. DataMPI (28) uses MPI to build Hadoop-like systems while (29) uses MPI communications in Spark for better performance. Our toolkit approach as proposed in Twister2 makes interoperability easier at the usage level, as one can change lower level components to fit different environments without changing the programmatic or user interface.

There is an ongoing effort in the HPC community to develop AMT systems for realizing the full potential of multicore and many-core machines, as well as handling irregular parallel applications in a more robust fashion. It is widely accepted that writing efficient programs with the existing capabilities of MPI is difficult due to the bare minimum capabilities it provides. AMT systems model computations as dataflow graphs and use shared memory and threading to achieve the best performance out of many-core machines. Such systems include OCR (30), DADuE (31), Charm++ (32), COMPS (33) and HPX (34), all of which focus on dynamic scheduling of the computation graph. A portability API is developed in DARMA (35) to AMT systems to develop applications agnostic to the details of specific systems. They extract the best available performance of multicore and many-core systems while reducing the burden of the user having to write such programs using MPI. Prior to this, there was much focus in the HPC community on developing programs that could bring automatic parallelism to users such as Parallel Fortran (36). Research has been done with MPI to understand the effect of computer noise on collective communication operations (37, 38, 39). For large computations, computer noise coming from an operating system can play a major role in reducing performance. Asynchronous collective operations can be used to reduce the noise in such situations, but it is not guaranteed to completely eliminate the burden.

In practice, multiple algorithms and data processing applications are combined together in workflows to create complete applications. Systems such as Apache NiFi (40), Kepler (41), and Pegasus (42) were developed for this purpose. The lambda architecture (43) is a dataflow solution for designing such applications in a more tightly coupled way. Amazon Step functions (44) are bringing the workflow to the FaaS and microservices.

In task execution management and scheduling to acquire a fault tolerant system, Akka framework has provided a pluggable implementation to manage task execution in other systems. The actor-based model in Akka offers a versatile implementation in obtaining a fault-tolerant and scalable solution. With the actor model, various topologies can be designed to meet the requirements in a system.

3 | BIG DATA APPLICATIONS

Here we highlight four types of applications with different processing requirements: 1) Streaming, 2) Data pipelines, 3) Machine learning, and 4) Services. With the explosion of IoT devices and the cloud as a computation platform, fog computing is adding a new dimension to these applications, where part of the processing has to be done near the devices.

Streaming applications work on partial data while batch applications process data stored in disks as a complete set. By definition, streaming data is unlimited in size and hard (to say nothing of unnecessary) to process as a complete set due to time requirements. Only temporal data set observed in data windows can be processed at a given time. In order to handle a continuous stream of data, it is necessary to create summaries of the temporal data windows and use them in subsequent processing of the stream. There can be many ways to define data windows, including time-based windows and data count-based windows. In the most extreme case, a single data tuple can be considered as the processing granularity.

Data pipelines are primarily used to extract, transform and load (ETL) operations even though they can include steps such as running a complex algorithm. They mostly deal with unstructured data stored in raw form or semi-structured data stored in NoSQL (45) databases. Data pipelines work on arguably the largest data sets possible out of the three types of applications. In most cases, it is not possible to load complete data sets into memory at once and we are required to process data partition by partition. Because the data is unstructured or semi-structured, the processing has to assume unbalanced data for parallel processing. The processing requirements are coarse-grained and pleasingly parallel. Generally, we can consider a data pipeline as an extreme case of a streaming application, where there is no order of data and the streaming windows contain partitions of data.

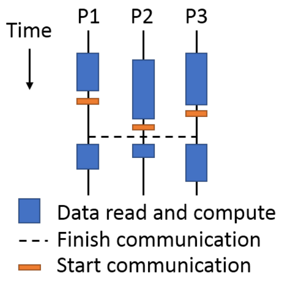

FIGURE 1 Load imbalance and velocity of data

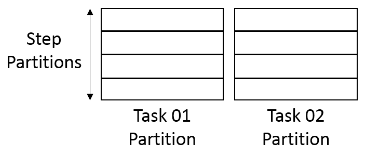

FIGURE 2 Hierarchical data partitioning of a big data application

Machine learning applications execute complex algebraic operations and can be made to run in parallel using synchronized parallel operations. In most cases the data can be load balanced across the workers as curated data is being used. The algorithms can be regular or irregular and may need dynamic load balancing of computations and data.

Services are moving towards an event-driven model for scalability, efficiency, and cost effectiveness in the cloud. The old monolithic services are being replaced by leaner microservices. These microservices are envisioned to be composed of small functions arranged in a workflow (44) or dataflow to achieve the required functionality.

3.1 | Data Processing Requirements

Data processing requirements are different compared to traditional parallel computing applications due to the characteristics of data. For example, some data are unstructured and hard to load balance for data processing. Data can be in heterogeneous sources including NoSQL databases and distributed file systems. Also, it can arrive at varying velocities in streaming use cases. Compared to general data processing, machine learning applications can expect curated data in a more homogeneous environment.

Data Partitioning: A big data application requires the data to be partitioned in a hierarchical manner due to memory limitations. Fig. 2 shows an example of such partitioning of a large file containing records of data points. The data is first partitioned according to the number of parallel tasks and then each partition is again split into smaller partitions. At every stage of the execution, such smaller examples are loaded into the memory of each worker. This hierarchical partitioning is implicit in streaming applications, as only a small portion of the data is available at a given time.

Hiding Latency: It is widely recognized that computer noise can play a huge role in large-scale parallel jobs that require collective operations. Many researchers have experimented with MPI to reduce performance degradation caused by noise in HPC environments. Such noise is much less compared to what typical cloud environments observe with multiple VMs sharing the same hardware, I/O subsystems, and networks. Added to this is the Java JVM noise which most notably comes from garbage collection. The computations in the dataflow model are somewhat insulated from the effects of such noise due to the asynchronous nature of the parallel execution. For streaming settings, the data arrives at the parallel nodes with different speeds and processing time requirements. Because of these characteristics, asynchronous operations are the most suitable for such environments. Load balancing (46) is a much harder problem in streaming settings where data skew is more common because of the nature of applications.

Overlapping I/O and Computations: Because of the large data transfers required by data applications, it is important to overlap I/O time with computing as much as possible to hide the I/O latencies.

3.2 | MPI for Big Data

MPI is the de facto standard in HPC for developing parallel applications. An example HPC application is shown in Fig. 3 where a workflow system such as Kepler (41) is used to invoke individual MPI applications. A parallel worker of an MPI program does computations and communications within the same process scope, allowing the program to keep state throughout the execution. An MPI programmer has to consider low-level details such as I/O, memory hierarchy and efficient execution of threads to write a parallel application that scales to large numbers of nodes. With the increasing availability of multi-core and many-core systems, the burden on the programmer to get the best available performance has increased dramatically (26, 25). Because of the inherent load imbalance and velocity of the data applications, an MPI programmer has to go into great detail to program efficient data applications in such environments. Another important point is that MPI is a message level protocol with low-level message abstractions. Data applications such as pipelines, streaming and FaaS require higher level abstractions than low level message abstractions. When data is in a more curated form as in machine learning, the authors have shown that MPI outperforms other technologies by a wide margin (4).

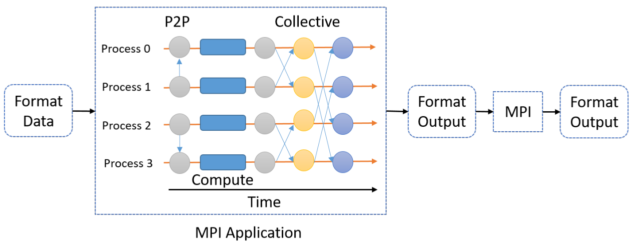

FIGURE 3 MPI applications arranged in a workflow

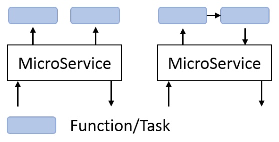

FIGURE 4 Microservices using FaaS, Left: Functions using a workflow, Right: Functions in a dataflow

3.3 | Dataflow for Big Data

Data-driven computing is becoming dominant for big data applications. A dataflow program can hide details such as communication, task execution and data management from the user while giving higher level abstractions including task APIs or data transformation APIs. One can make different design choices at these core components to tune a dataflow framework for supporting different types of applications.

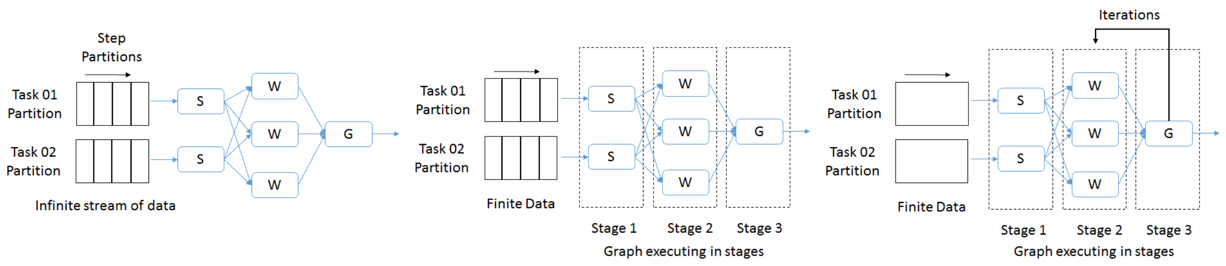

FIGURE 5 Dataflow application execution, Left: Streaming execution, Middle: Data pipelines executing in stages, Right: Iterative execution

3.3.1 | Streaming Applications

Streaming applications deal with load imbalanced data coming at varying rates to parallel workers at any given moment. Unless very carefully designed using asynchronous operations, an MPI application processing this data will increase the latency of the individual events. Fig. 1 shows this point with an example where three parallel workers process messages arriving at different speeds, sizes, and processing times. If an MPI collective operation is invoked, it is clear that the collective has to wait until the slowest task finishes, which can vary widely. Also, to handle streams of data with higher frequencies, the tasks of the streaming computation must be executed in different CPUs arranged in pipelines. The dataflow model is a natural fit for such asynchronous processing of chained tasks.

3.3.2 | Data Pipelines

Data pipelines can be viewed as a special case of streaming application. They work on hierarchically partitioned data as shown in Fig 2 . This is similar to streaming where a stream is partitioned among multiple parallel workers and a parallel worker only processes a small portion of the assigned partition at a given time. Data pipelines deal with the same load imbalance as streaming applications, but the scheduling of tasks is not equal between them. Usually, every task in a data pipeline is executed in each CPU sequentially, so only a subset of tasks is active at a given time in contrast to all the tasks being active in streaming applications. Streaming communication operations only need to work on data that can be stored in memory, while data pipelines do communications that require a disk because of the large size of data. It is necessary to support iterative computations in data pipelines in case they execute complex data analytics applications.

3.3.3 | Machine Learning

Complex machine learning applications work mostly with curated data that are load balanced. This means tight synchronizations required by the MPI-style parallel operations are possible because the data is available around the time the communication is invoked. It is not practical to run complex machine learning algorithms ($> O(n^2)$) on very large data sets as they have polymorphic time requirements. In those cases, it is required to find heuristic approaches with lower time complexities. There are machine learning algorithms which can be run in a pleasingly parallel manner as well. Because of the expressivity required by the machine learning applications, the dataflow APIs should be close enough to MPI-type programming, but it should hide details such as threads and I/O from users. Task-based APIs as used by AMT systems are suitable for such applications. We note that large numbers of machine learning algorithms fall into the map-collective model of computation as described in (47, 48).

3.3.4 | Services

The services are composed of event-driven functions which can be provisioned and scaled without the user having to know the underlying details of the infrastructure. The functions can be directly exposed to the user for event-driven applications or by proxy through microservices for request/response applications. Fig. 4 shows microservices using functions arranged in a workflow and in a dataflow.

4 | TOOLKIT COMPONENTS

Considering the requirements of different applications, we have designed a layered approach for big data with independent components at each level to compose an application. The layers include: 1. Resource allocations, 2. Data Access, 3. Communication, 4. Task System, and 5. Distributed Data. Among these communications, task system and data management are the core components of the system with the others providing auxiliary services. On top of these layers, one can develop higher-level APIs such as SQL interfaces which are not a focus of this paper. Fig. 6 shows the runtime architecture of Twister2 with various components. Even though Fig. 6 shows all the components in a single diagram, one can mix and match various components according to their needs. Fault tolerance and security are two aspects that affect all these components. Table. 1 gives a summary of various components, APIs and implementation choices.

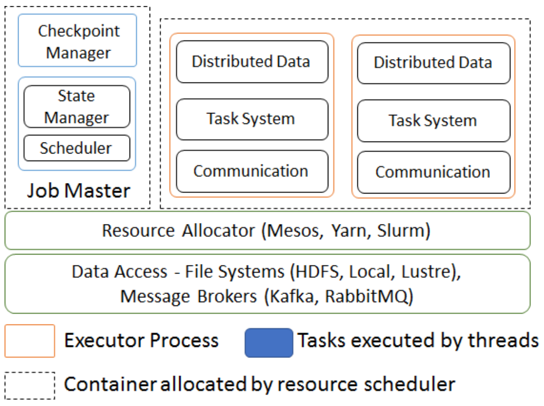
Checkpoint Manager
State Manager
Scheduler
Job Master
Distributed Data
Task System
Communication
Distributed Data
Task System
Communication
Resource Allocator (Mesos, Yarn, Slurm)
Data Access - File Systems (HDFS, Local, Lustre), Message Brokers (Kafka, RabbitMQ)
Executor Process
Tasks executed by threads
Container allocated by resource scheduler

FIGURE 6 Runtime architecture of Twister2

4.1 | Architecture Specification

System specification captures the essentials of a parallel application that will determine the configuration of the components. We identify execution semantics and coordination points as the two essential features that define the semantics of a parallel application.

Coordination Points: To understand and reason about a parallel application, we introduce a concept called a coordination point. At a coordination point, a program knows that a parallel computation has finished. With MPI, a coordination point is implicitly defined when it invokes and completes a communication primitive. For example, when AllReduce operation finishes a parallel task, it knows that the code before the AllReduce has been completed. For data driven applications, the coordination happens at the data level. Depending on the abstractions provided, the coordination can be seen at communication level, task level or the distributed data set level. For example, a task is invoked when its inputs are satisfied. So the coordination of tasks happens at the beginning of such executions. No coordination between parallel tasks are allowed inside the tasks. At the data level, the coordination occurs when the data sets are created and its subsequent operations are invoked. HPC also has coordination points at the end of jobs. These are managed in workflow graphs with systems like Kepler, Taverna, and Pegasus. The data driven coordination points are finer-grained than workflow and similar to those in HPC systems where computing phases move to communication phases.

TABLE 1 Components of the Twister2 Toolkit

Component	Area	Implementation	Comments; User API
Architecture Specification	Coordination Points	State and Configuration Management; Program, Data and Message Level	Change execution mode; save and reset state
	Execution Semantics	Mapping of Resources to Bolts/Maps in Containers, Processes, Threads	Different systems make different choices - why?
Job Submission	(Dynamic/Static) Resource Allocation	Plugins for Slurm, Yarn, Mesos, Marathon, Aurora	Client API (e.g. Python) for Job Management
Communication	Dataflow Communication	MPI Based, TCP, RDMA	Define new Dataflow communication API and library
	BSP Communication	Conventional MPI, Harp	MPI P2P and Collective API
Task System	Task migration	Monitoring of tasks and migrating tasks for better resource utilization	Task-based programming with Dynamic or Static Graph API; FaaS API; Support accelerators (CUDA,KNL)
	Elasticity	OpenWhisk	
	Streaming and FaaS Events	Heron, OpenWhisk, Kafka/RabbitMQ	
	Task Execution	Process, Threads, Queues	
	Task Scheduling	Dynamic Scheduling, Static Scheduling, Pluggable Scheduling Algorithms	
	Task Graph	Static Graph, Dynamic Graph Generation	
Data Access	Static (Batch) Data	File Systems, NoSQL, SQL	Data API
	Streaming Data	Message Brokers, Spouts	
Distributed Data Management	Distributed Data Set	Relaxed Distributed Shared Memory (immutable data), Mutable Distributed Data	Data Transformation API; Spark RDD, Heron Streamlet
Fault tolerance	Check pointing	Lightweight barriers, Coordination Points, Upstream backup; Spark/Flink, MPI and Heron models	Streaming and batch cases distinct; Crosses all components
Security	Messaging, FaaS, Storage	Research	Crosses all components

Execution semantics: Execution semantics of an application define how the allocated resources are mapped to execution units. Cluster resources are allocated in logical containers and these containers can host processes that execute the parallel code of the application. Execution semantics define the mapping of computation tasks into the containers using processes or a hybrid approach with threads and processes.

4.2 | Job Submission & Resource Allocation

Cluster resource allocation is often handled by specialized software that manages a cluster such as Slurm, Mesos or Yarn. Such frameworks have been part of the HPC community for a long time and the existing systems are capable of allocating a large number of jobs in large clusters. Yarn and Mesos are big data versions of the same functionality provided by Slurm or Torque with an emphasis on fault tolerance and cloud deployments. In particular, both are capable of handling node failures and offer applications the opportunity to work even when the nodes fail by dynamically allocating resources. Twister2 will use a pluggable architecture for allocating resources utilizing different schedulers available. An allocated resource including CPUs, RAM and disks are considered as a container. A container can run a single computation or multiple computations using processes/threads depending on the system specification. For computationally expensive jobs, it is important to isolate the CPUs to preserve cache coherence while I/O-bound jobs can benefit from the idle CPUs available. In case of node failures, Twister2 can get a new node and start failed processes to achieve fault tolerance. For cloud deployments with FaaS, resource management frameworks such as Docker can be exploited to scale the applications.

4.3 | Communication

Communication is a fundamental requirement of parallel computing because the performance of the applications largely revolves around efficient implementations. High-level communication patterns as identified by the parallel computing community are available through frameworks such as MPI (49). Some of the heavily used primitives are Broadcast, Gather, Reduce, AllGather and AllReduce (50). The naive implementation of these primitives using point-to-point communication in a straightforward way produces worst-case performance in practical large-scale parallel applications. These patterns can be implemented using data distribution algorithms that minimize the bandwidth utilization and latency of the operation. In general, they are termed collective algorithms. Twister2 will support message-level and data-level BSP-style communications as in MPI, and solely data-level communications as in data flow programs. The dataflow-style communications will be used for data pipeline and streaming applications. One can choose to use BSP style or dataflow style for machine learning algorithms. Table. 2 summarizes some of the operations available in BSP and dataflow communications.

TABLE 2 MPI and dataflow communication operations

	Collectives							
BSP (MPI)	Reduce, AllReduce	Gather, AllGather	Broadcast, Scatter	Barrier	–	–	–	–
Dataflow	Reduce, Keyed Reduce	Gather, Keyed Gather	Broadcast	–	Union	Join	Partition	Sort

4.3.1 | BSP Communications

In MPI, collective operations and other point-to-point communication operations are driven by computations. This means that the programmer knows exactly when to execute the communication primitives as well as the parameters. Once the program is ready to communicate, it can initiate the appropriate operations which will invoke the network functions. The asynchronous communications are slightly different than synchronous operations in the sense that after their invocation, the program can continue to compute while the operation is pending. It is important to note that even with asynchronous operations the user needs to employ other operations such as wait/probe to complete the pending operation. The underlying implementation for MPI collective can use different algorithms based on factors including message size. Significant research has been done on MPI collectives (50, 51) and the current implementations are optimized to an extremely high extent. A comprehensive summary of MPI collective operations and possible algorithms is found in (52). BSP communications can be used as in MPI, or by the task system. Harp (2) is a machine learning-focused collective library that supports the standard MPI collectives as well as some other operations like rotate, push and pull.

4.3.2 | Dataflow Communications

A dataflow communication pattern defines how the links are arranged in the task graph. For instance, a single task can broadcast a message to multiple tasks in the graph when they are arranged in a broadcast communication pattern. One of the best examples of a collective operation in dataflow is Reduce. Reduce is the opposite of broadcast operation and multiple nodes link to a single node. The most common dataflow operations include reduce, gather, join (53), union and broadcast. MPI and big data have adopted the same type of collective communications but sometimes they have diverged in supported operations.

It is important to note the differences between MPI and dataflow communication primitives. In a dataflow graph, the messages (communications) drive the computation rather than computation driving the communication as in MPI. Also, dataflow communications work at data level rather than the message level as in MPI. For example, a dataflow communication can reduce a whole data set as a single operation that runs in many steps using hierarchical partitioning. In case of insufficient memory, the communications can use disks to save intermediate data of the operation. Also, the semantics of the data flow primitives are different compared to the MPI collectives, with keyed operations, joins, unions, and partitioning.

The system specification dictates that a task can only send and receive data via its input and output ports (coordination points) and they cannot communicate with each other while performing computations. If they communicate inside the tasks, that will introduce another coordinating point inside the task and the concept of the task will be broken. The authors of this paper propose collective operations as a graph enrichment, which introduces sub-tasks to the original dataflow graph. Fig.7 and Fig.8 show the naive implementation and our proposed approach for dataflow collective operations. In this approach, the collective operations computation is moved to a sub-task under which the collective operation depends. These sub-tasks can be connected to each other according to different data structures like trees and pipes in order to optimize the collective communication. This model preserves the dataflow nature of the application and the collective does not act as a synchronization barrier. The collective

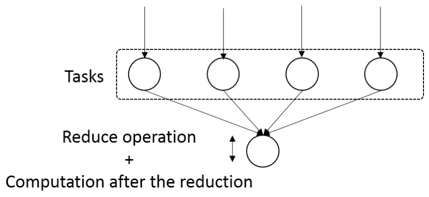

FIGURE 7 Default implementation of a dataflow reduce

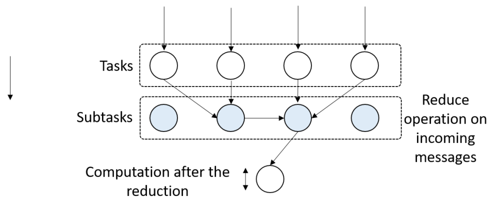
Tasks
Subtasks
Reduce operation on incoming messages
Computation after the reduction

FIGURE 8 Optimized dataflow reduce operation with sub-tasks arranged in a tree

operation can run even as data becomes available to each individual task, and the effects of unbalanced load and timing issues in MPI are no longer applicable. For collective operations such as broadcast and scatter, the original tasks will be arranged according to data structures required by such operations. We identify several requirements for a dataflow collective algorithm.

1. The communication and the underlying algorithm should be driven by data.
2. The algorithm should be able to use disks when the amount of data is larger than the available memory.
3. The collective communication should work at the data level taking into account partitions of the data

The dataflow collectives can be implemented on top of MPI send/receive operations, directly using TCP socket API, and using RDMA (Remote Direct Memory Access). These options will give the libary the ability to work in cloud environments as well as HPC environments. Twister2 dataflow communication library can be used by other big data frameworks to be efficient in HPC environments.

4.3.3 | High Performance Interconnects

RDMA (Remote Direct Memory Access) is one of the key areas where MPI excels. MPI implementations support a variety of high-performance communication fabrics and perform well compared to Ethernet counterparts. Recently there have been many efforts to bring RDMA communications to big data systems, including HDFS (24), Hadoop (54) and Spark (55). The big data applications are primarily written in Java and RDMA applications are written in C/C++, requiring the integration to go through JNI. Even by passing through additional layers such as JNI, the application still performs reasonably well with RDMA. One of the key forces that drags down the adoption of RDMA fabrics is their low-level APIs. Nowadays with unified API libraries such as Libfabric (56) and Photon (57), this is no longer the case.

4.4 | Task System

In order to develop an application at the communication layer, one needs a deep understanding about threaded execution, efficient use of communications and data management. The task layer provides a higher-level abstraction on top of the communication layer to hide the details of execution and communication from the user, while still delegating data management to the user. At this layer, computations are modeled as task graphs which can be created statically as a complete graph or dynamically as the application progresses.

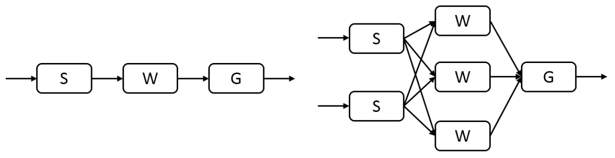

FIGURE 9 Left: User graph, Right: execution graph of a data flow

4.4.1 | Task graph

A node in the task graph represents a task while an edge represents a communication link between nodes. Each node in the graph holds information about the inputs and its outputs (edges). Also, a node contains an executable user code. The user code in a task is executed when events arrive at the inputs of the task. The user will output events to the output edges of the task graph and they will be sent through the network by the communication layer. A task can be long-running or short-running depending on the type of application. For example, a stream graph will have long running tasks while a dataflow graph without loops will have short running tasks. When loops are present, long running tasks can be appropriate to reduce task creation overheads.

4.4.2 | Execution Graph

Execution graph is a transformation of the user-defined task graph, created by the framework for deploying on the cluster. This execution graph will be scheduled onto the available resource by the task scheduler. For example, some user functions may run on a larger numbers of nodes depending on the parallelism specified. Also, when creating the execution graph, the framework can perform optimizations on the user graph to increase efficiency by reducing data movement and overlapping I/O and computations. Fig. 9 shows the execution graph and the user graph where they run multiple W operations and S operations in parallel. When creating the execution graph, optimizations can be applied to reduce data movement and preserve data locality.

4.5 | Task Scheduling

Task scheduling is the process of scheduling multiple task instances into the cluster resources. The task scheduling in Twister2 generates the task schedule plan based on the per job policies, which places the task instances into the processes spawned by the resource scheduler. It aims to allocate a number of dependent and independent tasks in a near optimal manner. The optimal allocation of tasks decreases the overall computation time of a job and improves the utilization of cluster resources. Moreover, task scheduling requires different scheduling methods for the allocation of tasks and resources based on the architectural characteristics. The selection of the best method is a major challenge in the big data processing environment. The task scheduling algorithms are broadly classified into two types, namely static task scheduling algorithms and dynamic task scheduling algorithms. Twister2 supports both types. It considers both the soft (CPU, disk) and hard (RAM) constraints and serializes the task schedule plan in the format of Google Protocol Buffers (58). Additionally, the Google Protobuf contains information about the number of containers to be created and the task instances to be allocated for each one. Additionally, it houses the required resource information such as CPU, disk memory, and RAM for the containers and the task instances to be allocated in those containers.

4.5.1 | Task Scheduling for Batch and Streaming Jobs

The task scheduling for batch jobs can be performed prior to the processing based on the knowledge of input data and the task information for processing in a distributed environment. Moreover the resources can be statically allocated prior to the execution of jobs. Nevertheless, the task scheduling for streaming jobs is considerably more difficult than batch jobs due to the continuous and dynamic nature of input data streams that requires unlimited processing time. The task scheduling should consider the availability of resource and resource demand as important parameters while scheduling the streaming tasks. Also, it should give more importance to the network parameters such as bandwidth and latency. Streaming task components (59) that communicate each other should be scheduled in close network proximity to avoid the network delay in the streaming jobs processing. Dynamic task scheduling is more suitable than static task scheduling for handling the dynamic streams of data or streaming jobs.

4.5.2 | Static Task Scheduling Algorithm

In static task scheduling, the jobs are allocated to the nodes before the execution of a job and the processing nodes are known at the compile time. Once the tasks are assigned to an appropriate resource, the execution continues to run until finishing the execution of the task. The main objective of the static task scheduling strategy is to reduce the scheduling overhead which occurs during the runtime of the task execution. Some static task scheduling strategy examples are Capacity Scheduler, Data Locality-Aware Scheduling, Round Robin Scheduling, Delay Scheduling, FIFO Scheduling, First Fit Scheduling, Fair Scheduling and Matchmaking Scheduling.

Twister2 is implemented with the following static task scheduling algorithms: (1) Round Robin (RR) Task Scheduling, (2) First Fit (FF) Task Scheduling, and (3) Data Locality-Aware (DLA) Task Scheduling. The round-robin task scheduling algorithm generates the task scheduling plan in which the task instances are allocated to the containers in a round robin manner without considering any priority to the task instances. It has the support to launch homogeneous containers of equal size of disk, memory, CPU and heterogeneous nature of task instances. Round-robin-based task (heterogeneous) instance allocation in the (homogeneous) containers is represented in Fig. 10 . The FF task scheduling algorithm generates the task scheduling plan in which the task instances are allocated to a finite number of containers with the objective of minimizing the number of containers and reducing the waste of underlying resources. In contrast to the round-robin task scheduling, it provides the support for launching heterogeneous containers and the heterogeneous nature of task instances. Fig. 11 shows the FF-based task (heterogeneous) instances allocation in the (heterogeneous) containers. The data locality-aware task scheduling algorithm is implemented with an awareness of data locality (i.e. the

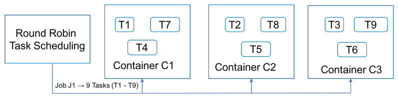

FIGURE 10 RR Task Scheduling (Homogeneous Containers and Heterogeneous Task Instances)

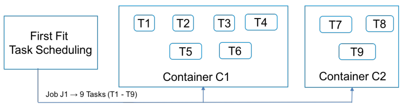

FIGURE 11 FF Task Scheduling (Heterogeneous Containers and Heterogeneous Task Instances)

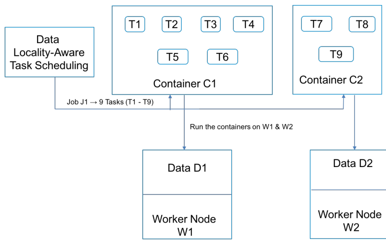

FIGURE 12 DLA Task Scheduling (Execution on the Data Nodes)

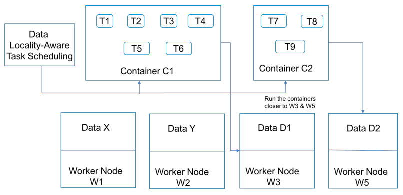

FIGURE 13 DLA Task Scheduling (Execution Closer to the Data Nodes)

distance between the data node that holds the data and the task execution node). Scheduling of tasks to the execution node which has the input data or closest to the input data maximizes the overall response time of a job. However, in some scenarios the execution of a node requires the data which has been distributed in nature, hence the data locality-aware task scheduling algorithm should consider that case while scheduling the tasks. Fig. 12 and Fig. 13 show the data locality-aware task scheduling scenarios handled in the Twister2 framework.

4.5.3 | Dynamic Task Scheduling Algorithm

In dynamic scheduling strategy, jobs are allocated to the nodes during the execution time of tasks. It is assumed that the user has complete knowledge about their application requirements, such as the maximum size of the container (CPU, disk, and RAM) or the required number of containers while submitting the jobs to the Twister2 framework. Thus the task scheduling algorithm should be able to generate an appropriate task scheduling plan using that information. However, the static task scheduling algorithm does not consider the availability of resources and the resource demand, which can lead to over-utilization or under-utilization of the resources and thus pave the way for inefficiencies. Contrary to the static task scheduling, the dynamic task scheduling evaluates the scheduling decisions during the execution of the job. It provides the support or triggers the task migration based on the status of the cluster resources and the workload of the application. Resource-Aware Scheduling, Deadline-Aware Scheduling and Energy-Aware Scheduling are examples of dynamic scheduling strategy. As such, Twister2 will be empowered with a dynamic task scheduling algorithm which considers the deadline of the job, inter-node traffic, inter-process traffic, resource availability and resource demand with the objective of minimizing the makespan (i.e. total execution time of all the tasks) of a job and effectively utilizing the underlying resources.

4.6 | Task Execution

Depending on the system specification, a process model or a hybrid model with threads can be used for execution. It is important to handle both I/O and task execution within a single execution module so that the framework can achieve the best possible performance by overlapping I/O and computations. The execution is responsible for managing the scheduled tasks and activating them with data coming from the message layer. To facilitate dynamic task scheduling, scaling of tasks for FaaS environments and high frequency messaging, it is vital to maintain high-performance concurrent message queues. Much research has been done on improving single queue multiple-threaded consumer bottlenecks for task execution, as shown in (60).

Unlike in MPI applications where threads are created equal to the number of CPU cores, big data systems typically employ more threads than the cores available to facilitate I/O operations. With I/O offloading and advanced hardware, the decision to choose the correct model for a particular environment becomes a research question. When performing large data transfers or heavy computations, the threads will not be able to attend to computing or I/O depending on the operation being performed. This can lead to unnecessary message buildups in upstream tasks or in the task itself. The ability to model such behaviors and pick the correct execution model (61) is important for achieving optimum performance. It has been observed that using a single task executor for both these applications would bring inferior performance (62).

4.6.1 | Multi-core machines

For an application running on multi-core (multiple CPUs) machines with multiple sockets, the effects of context switching can be significant due to cache misses and memory access latency, especially when crossing NUMA (non-uniform memory access) domains. The data migration cost is very important when threads cross the contexts (63). With NUMA, the data locality is considered and the tasks are allocated in a way that shared memory access can be gained for the task executors by binding them to specific NUMA domains which contain the expected memory blocks (63).

With many core machines now having large numbers of hardware threads, a single process can expect to deal with larger memory and more parallelism within a process. Having efficient ways to share resources (locks) is important, especially when the number of threads increases significantly. Languages such as Java require garbage collection (GC) to reclaim memory, and having processes with very large memory allocated can cause long pauses in GC. Because of this a balance for number of processes per node must be maintained.

4.7 | Data Access

Data access abstracts out various data sources including files and streaming sources to simplify the job of an application developer. In most distributed frameworks, the data is presented as a higher level abstraction to the user, such as the RDD in Apache Spark and DataSet for Apache Flink. Since the goal of Twister2 is to provide a toolkit which allows developers to choose the desired components, Twister2 includes a lower level API for data access in addition to a higher level abstraction. For example, the abstraction of a File System allows Twister2 to support NFS, HDFS, and Luster, which enables the developer to store and read data from any file by specifying only the URL. In addition to the data sources that are supported by the framework, the pluggable architecture allows users to add support for any data source by implementing the relevant interfaces.

Another important role of the data access layer is to handle data partitioning and data locality in an efficient manner. An unbalanced set of data partitions will create stragglers, which will increase the execution time of the application. The data access layer is responsible for providing the developer with appropriate information regarding data locality. Data locality directly affects the execution time since unnecessary data movements will degrade the efficiency of the application. In addition to the built-in functions of Twister2, the developer is given the option to plug in custom logic to handle data partitioning and locality.

4.8 | Distributed Data

The core of most dataflow frameworks is a well-defined high level data abstraction. RDDs in Apache Spark and DataSets in Apache Flink are well-known examples for higher level data abstractions. Twister2 provides an abstraction layer so that developers can develop applications using data transformation APIs that are provided. The distributed data abstraction used in Twister2 is termed a DataSets. DataSets are the main unit of parallelism when programs are developed using the data flow model in the framework. The number of splits or partitions that a DataSet is broken into determines the number of parallel tasks that will be launched to perform a given data flow operation. Twister2 DataSets support two primary types; immutable and mutable. The immutable version is most suitable for traditional data flow applications. Mutable DataSet's allow the data sets to be modified, but a given task may only alter the entries from the partition that is assigned to that task. The DataSet API provides the developer with a wide variety of transformations and actions that allow the developer to build the required application logic effortlessly.

DataSets are loaded lazily, which means that the actual data is not read until execution of a data flow operation is performed. This allows many optimizations such as pipelining transformations and performing local data reads to be implemented. Fault tolerance is built into the distributed data abstraction; if a task or a node fails the required calculation will be redone automatically by the system and the program can complete without any problems. Distributed DataSets leverage functionalists provided by the data access APIs, therefore the data partitioning and data locality is managed by the data access layer, removing the burden from the DataSets implementation. Leveraging the lower level APIs adheres to the toolkit approach taken by Twister2 and allows each system component to be modified and updated with little effect to the other components.

The framework generates an execution graph based on the transformations and actions that are performed on the distributed data set. This execution graph takes into account the number of partitions in the data set and the localities of the data partitions. Fig. 14 shows an example of such an execution graph. It demonstrates the execution graph of an application which applies the $logFile.map(...).filter(...).reduceByKey(...).forEach(...)$ sequence of transformations to a data set that has 4 partitions.

4.9 | Fault Tolerance

A crucial feature in distributed frameworks is fault tolerance since it allows applications to recover from various types of failures that may occur during the application runtime. Fault tolerance has become more and more important with the usage of larger commodity computer clusters to run applications. However, the overheads caused by fault tolerance mechanisms may reduce the application's performance, so it is important to keep them as lightweight as possible. Most distributed frameworks such as Spark and Flink have inherent support for fault tolerance. There are several

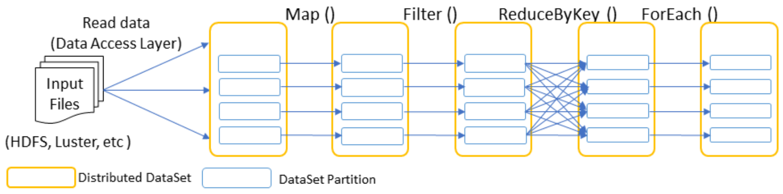

FIGURE 14 Example execution graph for $logFile.map(...).filter(...).reduceByKey(...).forEach(...)$

projects such as (64) and (65) which provide fault tolerance for MPI applications. It is important to allow the application developer to determine the level of fault tolerance required. This enables applications which run on reliable hardware with very large mean times of failure to run without the burden of fault tolerance. Checkpointing is a well-known mechanism used to handle failures in distributed frameworks. Dataflow application can have automatic checkpointing mechanisms at the coordination points to recover from failures. The checkpointing mechanism works differently for streaming data and batch data. Opting out of checkpointing does not mean that the application will fail as soon as one failure occurs. Instead the system can automatically restart the application or recover from the failure using cached intermediate data if available.

4.9.1 | Fault Tolerance For Streaming Data

Twister2 provides fault tolerance to streaming applications through lightweight distributed checkpoints. The model used is based on the stream barrier-based distributed snapshots described in (11). This checkpointing method injects barriers into the data stream and uses them to create snapshots so that every item in the stream before the barrier is processed completely. This helps guarantee exactly once semantics for stream processing applications. It also allows developers to choose the checkpointing frequency just by specifying the intervals at which the barriers are injected into the stream. Developers can completely forgo checkpointing, removing the overhead of fault tolerance if they choose. There are three main types of message processing guarantees that are required by various stream processing applications: exactly once, at least once and at most once. The fault tolerance mechanism provides support for all three given that some required conditions are met. For instance, to provide exactly once guarantee, the streaming source is required to have the capability to replay the source from a certain point. It is also important to note that stricter guarantees result in higher overheads for the fault tolerance mechanism.

4.9.2 | Fault Tolerance For Batch Data

Applications based on batch data can vary from pleasingly parallel applications to complex machine learning applications. Providing fault tolerance for pleasingly parallel applications is relatively simple because of the narrow dependencies involved. The system can relaunch a task when it fails without affecting any other running task. On the other hand, complex algorithms typically consist of wide dependencies, recovering from a failure is much more complex for such scenarios. Twister2 provides fault tolerance for batch data at two levels, namely checkpoint-based and cache-based mechanism. Checkpoint-based fault tolerance is the main mechanism while the cache-based model can be used to reduce overhead of checkpointing based on the application.

Checkpoint-based fault tolerance develops snapshots of the runtime application. These snapshots are created at coordination points in the applications, a natural candidate for a checkpoint since the runtime has the least amount of moving parts, such as messages at this point. This allows the checkpoints to be lightweight and simple. The developer has the flexibility to specify the checkpoints based on the application requirements. If a failure occurs, the framework recovers by loading the data and state from the checkpoint and relaunching the necessary tasks. The amount of tasks that need to be relaunched depends on the task dependencies. If the applications have narrow dependencies it may suffice to relaunch tasks for a subset of the data partitions.

Cached-based fault tolerance provides a more lightweight mechanism to reduce the need for checkpointing. It is important to note that this is not a full-fledged alternative to the checkpoint-based model and cannot handle node level failures. Once a task level failure occurs, the system first checks if the necessary intermediate data partitions are available in the cache. If so, the framework will relaunch the tasks without rolling back all the way to the most recent checkpoint. Developers are given the ability to specify which intermediate results need to be cached according to the application requirements.

4.10 | State Management

State management is an important aspect in distributed systems as it touches on most of the core components of the system. State of the system encompasses various parameters and details of the system at runtime. State management needs to be addressed at two levels: job level state and task level state. Job level state consists of information that is required to run the distributed application as a whole. Job level state is particularly important for fault tolerance and load distribution. Keeping a job level state allows tasks to be migrated within the cluster since the required state information is accessible to any worker node in the system. If one worker is overloaded, some of its tasks can be easily migrated to a worker that is underutilized so that the load can be distributed. The same state information allows the framework to recover from a node failure by relaunching the tasks on a new worker node. Job level state management is achieved via a distributed shared memory. Checkpointing mechanisms needs to take state into consideration when creating checkpoints of the system. Job level state is managed by a separate processes that makes sure the global state is consistent and correct . Task level state is runtime information that can be kept local to a task. Task level state is saved when checkpoints are performed and is used during the recovery process. This is especially important for long-running stateful tasks such as streaming tasks. In most scenarios the loss of information that falls under task level state does not affect the application as a whole and can be recovered.

4.11 | API

Over the years, there have been numerous languages and different types of APIs developed for programming data-driven applications. Twister2 supports three different levels of APIs with the objective of handling different programming and performance requirements for the applications. These three levels are classified as: 1) Communication API, 2) Task/FaaS API, and 3) Distributed Data API. The user can adopt the communication API to program parallel applications with auxiliary components such as data access API at the lowest level. It will give the maximum possible performance of the system because the user controls both the task execution and data placement, but at the same time it will be the most difficult way to program. Next the user can create or generate a task graph to create an application. The task graph can be made either statically or dynamically depending on the application requirements. By using the Task/FaaS API, the user can control data placement among the tasks while the framework will handle the communication and execution. At the highest level, the user can adopt the Distributed Data API, which will allow the framework to control every aspect of the application. At this level, programming will be easier for certain types of applications and the performance will be considerably less compared to the same application written in other layers. Fig. 15 provides a summary of the points discussed above and lists types of applications that are most suitable to be implemented at each level.

For efficient message transfers, it is necessary to use low level abstractions to communicate in order to reduce the burden of serialization. Using complex objects at the communication level adds a serialization overhead which can be significant in some applications. When we go up the API levels, we must utilize complex objects to represent data and use these abstractions for communication.

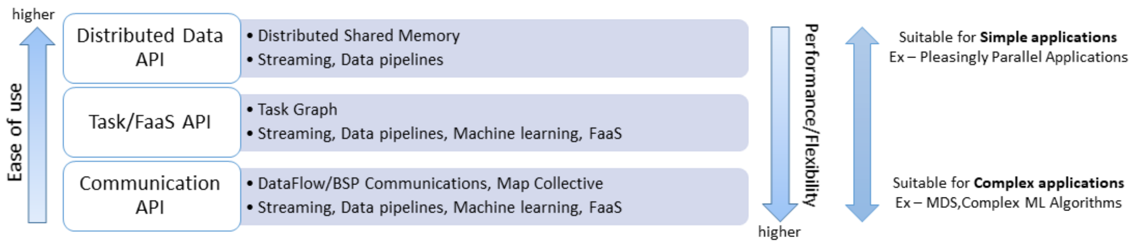
higher
Ease of use
Distributed Data API
• Distributed Shared Memory
• Streaming, Data pipelines
Task/FaaS API
• Task Graph
• Streaming, Data pipelines, Machine learning, FaaS
Communication API
• DataFlow/BSP Communications, Map Collective
• Streaming, Data pipelines, Machine learning, FaaS
Performance/Flexibility
higher
Suitable for Simple applications
Ex – Pleasingly Parallel Applications
Suitable for Complex applications
Ex – MDS,Complex ML Algorithms

FIGURE 15 Twister2 Big Data Toolkit API levels

5 | DISCUSSION

With our previous work (4) we have observed that various decisions made at different components of a big data runtime determine the type of applications that can be executed efficiently. The layered architecture proposed in this work will eliminate the monolithic designs and empower components to be developed independently and efficiently. The Twister2 design has the following implications: 1) It will allow developers to **choose only the components** that they need in order to develop the application. For example, a user may only want MPI-style communication with a static

TABLE 3 Requirements of applications

Type of applications	Capabilities		
	Data	Task System	Communications
Streaming	Distributed Data Set	Static Graph	Dataflow Communications
Data Pipelines	Distributed Data Set	Static Graph or Dynamic Graph	Dataflow Communications
Machine Learning	Distributed Shared Memory	Dynamic Graph	Dataflow Communications / BSP Communications
FaaS	Stateless	Dynamic Graph	Dataflow, P2P Communication

scheduling and distributed shared memory for their application; 2) Each component will have **multiple implementations**, allowing the user to support different types of applications, e.g., the toolkit can be used to compose a system that can perform streaming computations as well as data pipelines.

We identify communications, task system, and distributed shared memory as the three main components required by an application. The user APIs will be available to these components to program an application. Table 3 shows the different capabilities expected from different types of big data applications described herein. It is important to note that one can build streaming, data pipeline or machine learning algorithms with only the communication layer. Later, they can add the task system on top of communication to further enhance the ease of programming, and finally they can add the data layer to give the framework the highest possible control while reducing the burden on the programmer.

In general, it is safe to assume that machine learning algorithms require complex communications and executions. It is worth noting that there are a large group of machine learning algorithms that work with minimal parallel communications, and such algorithms are similar to data pipelines and can be scaled easily. Machine learning algorithms that work with large data sets also use heuristic methods to lower the parallel computation complexity in order to make them run in a more pleasingly parallel manner.

Security and fault tolerance are two areas that crosses all the components of the toolkit. In order to be fault tolerant, each component has to be able to work under node failures. We recognize security as an important aspect of this approach, but reserve a lengthy discussion to a subsequent work.

6 | CONCLUSIONS & FUTURE WORK

We foresee that the share of large-scale applications driven by data will increase rapidly in the future. The HPC community has tended to focus mostly on heavy computational-bound applications, and with these new developments, there is an opportunity to explore data-driven applications with HPC features such as high-speed interconnects and many-core machines. Data-driven computing frameworks are still in the early stages, and as we discussed there are four driving application areas (streaming, data pipelines, machine learning, and service) with different processing requirements. In this paper, we discussed the convergence of these application areas with a common event-driven model. We also examined the choices available in the design of frameworks supporting big data with different components. Every choice made by a component has ramifications for the performance of the applications the system can support. We believe the toolkit approach gives user the required flexibility to strike a balance between performance and usability and allows the inclusion of proven existing technologies in a unified environment. This will enable a programming environment that is interoperable across application types and system infrastructure including both HPC and clouds, whereas in the latter case it supports a cloud-native framework (5). The authors are actively working on the implementation of various components of the toolkit and APIs in order to deliver the promised flexibility across various applications and systems.

ACKNOWLEDGMENT

This work was partially supported by NSF CIF21 DIBBS 1443054, NSF RaPyDLI 1415459 and the Indiana University Precision Health initiative.

References

[1] Bonomi Flavio, Milito Rodolfo, Zhu Jiang, Addepalli Sateesh. Fog Computing and Its Role in the Internet of Things. In: .

[2] Zhang B., Ruan Y., Qiu J.. Harp: Collective Communication on Hadoop. In: :228-233; 2015.

[3] Halbwachs Nicholas, Caspi Paul, Raymond Pascal, Pilaud Daniel. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*. 1991;79(9):1305–1320.

[4] Kamburugamuve Supun, Wickramasinghe Pulasthi, Ekanayake Saliya, Fox Geoffrey C. Anatomy of machine learning algorithm implementations in MPI, Spark, and Flink. ;.

[5] Cloud Native Computing Foundation https://www.cncf.io/Accessed: 2017-Aug-06; .

[6] Gannon Dennis, Barga Roger, Sundaresan Neel. Cloud Native Applications. *IEEE Cloud Computing Magazine special issue on cloud native computing*. to be published;.

[7] White Tom. *Hadoop: The Definitive Guide*. Sebastopol, CA, USA: O'Reilly Media, Inc.; 1st ed.2009.

[8] Dean Jeffrey, Ghemawat Sanjay. MapReduce: A Flexible Data Processing Tool. *Commun. ACM*. 2010;53(1):72–77.

[9] Ekanayake Jaliya, Li Hui, Zhang Bingjing, et al. Twister: A Runtime for Iterative MapReduce. In: HPDC '10:810–818ACM; 2010; New York, NY, USA.

[10] Zaharia Matei, Chowdhury Mosharaf, Franklin Michael J., Shenker Scott, Stoica Ion. Spark: Cluster Computing with Working Sets. In: HotCloud'10:10–10USENIX Association; 2010; Berkeley, CA, USA.

[11] Carbone Paris, Katsifodimos Asterios, Ewen Stephan, Markl Volker, Haridi Seif, Tzoumas Kostas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*. 2015;36(4).

[12] Murray Derek G., McSherry Frank, Isaacs Rebecca, Isard Michael, Barham Paul, Abadi Martín. Naiad: A Timely Dataflow System. In: .

[13] Akidau Tyler, Bradshaw Robert, Chambers Craig, et al. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing. *Proc. VLDB Endow.*. 2015;8(12):1792–1803.

[14] Toshniwal Ankit, Taneja Siddarth, Shukla Amit, et al. Storm@Twitter. In: SIGMOD '14:147–156ACM; 2014; New York, NY, USA.

[15] Kulkarni Sanjeev, Bhagat Nikunj, Fu Maosong, et al. Twitter Heron: Stream Processing at Scale. In: SIGMOD '15:239–250ACM; 2015; New York, NY, USA.

[16] Akidau Tyler, Balikov Alex, Bekiroğlu Kaya, et al. MillWheel: Fault-tolerant Stream Processing at Internet Scale. *Proc. VLDB Endow.*. 2013;6(11):1033–1044.

[17] Ranjan R.. Streaming Big Data Processing in Datacenter Clouds. *IEEE Cloud Computing*. ;.

[18] Thies William, Karczmarek Michal, Amarasinghe Saman. StreamIt: A Language for Streaming Applications:179–196. Berlin, Heidelberg: Springer Berlin Heidelberg 2002.

[19] Balazinska Magdalena, Balakrishnan Hari, Madden Samuel R., Stonebraker Michael. Fault-tolerance in the Borealis Distributed Stream Processing System. *ACM Trans. Database Syst.*. 2008;33(1):3:1–3:44.

[20] Gedik Bugra, Andrade Henrique, Wu Kun-Lung, Yu Philip S., Doo Myungcheol. SPADE: The System s Declarative Stream Processing Engine. In: SIGMOD '08:1123–1134ACM; 2008; New York, NY, USA.

[21] Neumeyer L., Robbins B., Nair A., Kesari A.. S4: Distributed Stream Computing Platform. In: :170-177; 2010.

[22] Fox Geoffrey, Qiu Judy, Jha Shantenu, Ekanayake Saliya, Kamburugamuve Supun. Big Data, Simulations and HPC Convergence:3–17. Cham: Springer International Publishing 2016.

[23] Fox G.C., Qiu J., Kamburugamuve S., Jha S., Luckow A.. HPC-ABDS High Performance Computing Enhanced Apache Big Data Stack. In: :1057-1066; 2015.

[24] Islam N. S., Rahman M. W., Jose J., et al. High Performance RDMA-based Design of HDFS over InfiniBand. In: SC '12:35:1–35:35IEEE Computer Society Press; 2012; Los Alamitos, CA, USA.

[25] Ekanayake Saliya, Kamburugamuve Supun, Fox Geoffrey C.. SPIDAL Java: High Performance Data Analytics with Java and MPI on Large Multicore HPC Clusters. In: HPC '16:3:1–3:8Society for Computer Simulation International; 2016; San Diego, CA, USA.

[26] Ekanayake S., Kamburugamuve S., Wickramasinghe P., Fox G. C.. Java thread and process performance for parallel machine learning on multicore HPC clusters. In: :347-354IEEE; 2016; Washington, DC, USA.

[27] Blagodurov Sergey, Zhuravlev Sergey, Fedorova Alexandra, Kamali Ali. A Case for NUMA-aware Contention Management on Multicore Systems. In: PACT '10:557–558ACM; 2010; New York, NY, USA.

[28] Liang Fan, Feng Chen, Lu Xiaoyi, Xu Zhiwei. Performance Benefits of DataMPI: A Case Study with BigDataBench. In: Zhan Jianfeng, Han Rui, Weng Chuliang, eds. *Big Data Benchmarks, Performance Optimization, and Emerging Hardware: 4th and 5th Workshops, BPOE 2014, Salt Lake City, USA, March 1, 2014 and Hangzhou, China, September 5, 2014, Revised Selected Papers*, :111–123Springer International Publishing; 2014; Cham.

[29] Anderson Michael, Smith Shaden, Sundaram Narayanan, et al. Bridging the Gap Between HPC and Big Data Frameworks. ;.

[30] Mattson T. G., Cledat R., CavÃl' V., et al. The Open Community Runtime: A runtime system for extreme scale computing. In: :1-7; 2016.

[31] Bosilca George, Bouteiller Aurelien, Danalis Anthony, Herault Thomas, Lemarinier Pierre, Dongarra Jack. DAGuE: A generic distributed DAG engine for High Performance Computing. *Parallel Computing*. 2012;38(1):37 - 51. Extensions for Next-Generation Parallel Programming Models.

[32] Kale Laxmikant V., Krishnan Sanjeev. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In: OOPSLA '93:91–108ACM; 1993; New York, NY, USA.

[33] Conejero Javier, Corella Sandra, Badia Rosa M, Labarta Jesus. Task-based programming in COMPSs to converge from HPC to big data. ;.

[34] Sterling Thomas, Anderson Matthew, Bohan P. Kevin, Brodowicz Maciej, Kulkarni Abhishek, Zhang Bo. Towards Exascale Co-design in a Runtime System:85–99. Cham: Springer International Publishing 2015.

[35] Hollman David, Lifflander Jonathon, Wilke Jeremiah, et al. *DARMA v. Beta 0.5*. 2017.

[36] Bozkus Z., Choudhary A., Fox G., Haupt T., Ranka S.. Fortran 90D/HPF Compiler for Distributed Memory MIMD Computers: Design, Implementation, and Performance Results. In: Supercomputing '93:351–360ACM; 1993; New York, NY, USA.

[37] Hoefler Torsten, Schneider Timo, Lumsdaine Andrew. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In: SC '10:1–11IEEE Computer Society; 2010; Washington, DC, USA.

[38] Hoefler T., Schneider T., Lumsdaine A.. The impact of network noise at large-scale communication performance. In: :1-8; 2009.

[39] Agarwal Saurabh, Garg Rahul, Vishnoi Nisheeth K.. The Impact of Noise on the Scaling of Collectives: A Theoretical Approach:280–289. Berlin, Heidelberg: Springer Berlin Heidelberg 2005.

[40] Apache NiFi https://nifi.apache.org/Accessed: July 19 2017; .

[41] LudÃd'scher Bertram, Altintas Ilkay, Berkley Chad, et al. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*. 2006;18(10):1039–1065.

[42] Deelman Ewa, Singh Gurmeet, Su Mei-Hui, et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*. 2005;13(3):219–237.

[43] Marz Nathan, Warren James. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Greenwich, CT, USA: Manning Publications Co.; 1st ed.2015.

[44] AWS Step Functions https://aws.amazon.com/step-functions/Accessed: July 19 2017; .

[45] Han Jing, E Haihong, Le Guan, Du Jian. Survey on NoSQL database. In: :363-366; 2011.

[46] Nasir M. A. U., Morales G. De Francisci, Garcia-Soriano D., Kourtellis N., Serafini M.. The power of both choices: Practical load balancing for distributed stream processing engines. In: :137-148; 2015.

[47] Chu Cheng-Tao, Kim Sang Kyun, Lin Yi-An, et al. Map-reduce for Machine Learning on Multicore. In: NIPS'06:281–288MIT Press; 2006; Cambridge, MA, USA.

[48] Ghoting A., Krishnamurthy R., Pednault E., et al. SystemML: Declarative machine learning on MapReduce. In: :231-242; 2011.

[49] Gabriel Edgar, Fagg Graham E., Bosilca George, et al. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation:97–104. Berlin, Heidelberg: Springer Berlin Heidelberg 2004.

[50] Thakur Rajeev, Gropp William D.. Improving the Performance of Collective Operations in MPICH:257–267. Berlin, Heidelberg: Springer Berlin Heidelberg 2003.

[51] Pješivac-Grbović Jelena, Angskun Thara, Bosilca George, Fagg Graham E., Gabriel Edgar, Dongarra Jack J.. Performance analysis of MPI collective operations. *Cluster Computing*. 2007;10(2):127–143.

[52] Wickramasinghe Udayanga, Lumsdaine Andrew. A Survey of Methods for Collective Communication Optimization and Tuning. *arXiv preprint arXiv:1611.06334*. 2016;.

[53] Barthels Claude, Müller Ingo, Schneider Timo, Alonso Gustavo, Hoefler Torsten. Distributed Join Algorithms on Thousands of Cores. *Proc. VLDB Endow.*. 2017;10(5):517–528.

[54] Lu X., Islam N. S., Wasi-Ur-Rahman M., et al. High-Performance Design of Hadoop RPC with RDMA over InfiniBand. In: :641-650IEEE; 2013; New York, NY, USA.

[55] Lu X., Shankar D., Gugnani S., Panda D. K. D. K.. High-performance design of apache spark with RDMA and its benefits on various workloads. In: :253-262; 2016.

[56] Grun P., Hefty S., Sur S., et al. A Brief Introduction to the OpenFabrics Interfaces - A New Network API for Maximizing High Performance Application Efficiency. In: :34-39; 2015.

[57] Kissel E., Swany M.. Photon: Remote Memory Access Middleware for High-Performance Runtime Systems. In: :1736-1743; 2016.

[58] Google Protocol Buffers https://developers.google.com/protocol-buffers/Accessed: August 20 2017; .

[59] Boyang Peng, Mohammad Hosseini, Zhihao Hong. R-Storm: Resource-Aware Scheduling in Storm. In: Annual Middleware ConferenceACM; 2015.

[60] Alistarh Dan, Kopinsky Justin, Li Jerry, Shavit Nir. The SprayList: A Scalable Relaxed Priority Queue. *SIGPLAN Not.*. 2015;50(8):11–20.

[61] Rosti Emilia, Serazzi Giuseppe, Smirni Evgenia, Squillante Mark S.. Models of Parallel Applications with Large Computation and I/O Requirements. *IEEE Trans. Softw. Eng.*. 2002;28(3):286–307.

[62] Chen C. T., Hung L. J., Hsieh S. Y., Buyya R., Zomaya A. Y.. Heterogeneous Job Allocation Scheduler for Hadoop MapReduce Using Dynamic Grouping Integrated Neighboring Search. *IEEE Transactions on Cloud Computing.* 2017;PP(99):1-1.

[63] Stenström Per, Joe Truman, Gupta Anoop. Comparative Performance Evaluation of Cache-coherent NUMA and COMA Architectures. *SIGARCH Comput. Archit. News.* 1992;20(2):80–91.

[64] Fagg Graham, Dongarra Jack. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. *Recent advances in parallel virtual machine and message passing interface.* 2000;:346–353.

[65] Hursey Joshua, Graham Richard, Bronevetsky Greg, Buntinas Darius, Pritchard Howard, Solt David. Run-through stabilization: An MPI proposal for process fault tolerance. *Recent advances in the message passing interface.* 2011;:329–332.

□