Designing Twister2: Efficient Programming Environment Toolkits for Big Data

Supun Kamburugamuve
School of Informatics and Computing
Indiana University
Bloomington, IN, USA
skamburu@indiana.edu

Geoffrey Fox
School of Informatics and Computing
Indiana University
Bloomington, IN, USA
gcf@indiana.edu

ABSTRACT

Data-driven applications are required to adapt to the ever-increasing volume, velocity and veracity of data generated by a variety of sources including the Web and Internet of Things devices. At the same time, an event-driven computational paradigm is emerging as the core of modern systems designed for both database queries, data analytics and on-demand applications. MapReduce has been generalized to Map Collective and shown to be very effective in machine learning. However one often uses a dataflow computing model, which has been adopted by most major big data processing runtimes. The HPC community has also developed several asynchronous many tasks (AMT) systems according to the dataflow model. From a different point of view, the services community is moving to an increasingly event-driven model where (micro)services are composed of small functions driven by events in the form of Function as a Service(Faas) and serverless computing. Such designs allow the applications to scale quickly as well as be cost effective in cloud environments.

An event-driven runtime designed for data processing consists of well-understood components such as communication, scheduling, and fault tolerance. One can make different design decisions for these components that will determine the type of applications a system can support efficiently. We find that modern systems are designed in a monolithic approach with a fixed set of choices that cannot be changed easily afterwards. Because of these design choices their functionality is limited to specific sets of applications. In this paper we study existing systems (candidate event-driven runtimes), the design choices they have made for each component, and how this affects the type of applications they can support. Further we propose a loosely coupled component-based approach for designing a big data toolkit where each component can have different implementations to support various applications. We believe such a polymorphic design would allow services and data analytics to be integrated seamlessly and expand from edge to cloud to high performance computing environments.

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 both in streaming and 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 [10] is adding another dimension to these applications where part of the processing has to occur near the devices.

Parallel and distributed computing is 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 (HPC) community has developed frameworks such as message passing interface (MPI) to execute computationally intensive parallel applications efficiently. The 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 targeting 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. Lately 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 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. There are many application classes including database queries, management, and data analytics from complex machine learning to pleasingly parallel event processing. Common issues include that the data can be too big to fit into the memory of even a large cluster. Another important aspect is that 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 raw form are usually not load balanced and often require too much time and disk space to balance the data. Also the batch data processing is not enough 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 using mostly 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 [58] has highlighted the importance of the Map-Collective computing paradigm.

The dataflow [28] 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 (AMT) systems 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 [35], 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 serverless 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 that the 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.

This paper provides the following contributions: 1) A study of different application areas and how a common computation model fits them; 2) Design choices of different systems and how they affect each application area; 3) Presenting a vision of a big data toolkit (Twister2) that can execute applications from each area efficiently. Furthermore the paper provides comparisons of big data and MPI styles of programs to gain better insight into the system requirements.

2. RELATED WORK

Hadoop [54] was the first major open source platform developed to process large amounts of data in parallel. The map-reduce [17] 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 [19] became apparent, and systems such as Apache Spark [57] and Apache Flink [13] were developed to overcome them. These systems are developed 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 [44], Apache Apex and Google Dataflow [5]. 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. Some examples of open source streaming systems include Apache Storm [53], Twitter Heron [38], Google Millwheel [4], Apache Samza [49] and Flink [13]. Note that some of the systems process both streaming and batch data in a unified way. Apache Beam [5] 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 [52], Borealis [6], Spade [25] and S4 [46].

There are synergies between HPC and big data systems, and authors [22, 23] among others [33] have expressed the need to enhance these systems by taking ideas from each other. In previous work [20, 21] we have identified the general implications of threads and processes, cache, memory management in NUMA [9], as well as multi-core settings for machine learning algorithms with MPI.

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 best performance out of many-core machines. Some examples of such systems are OCR [43], DADuE [11], Charm++ [34], COMPS [15] and HPX [50], all of which focus on dynamic scheduling of the computation graph. A portability API is developed in DARMA [32] 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 writing 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 [12]. Research has been done with MPI to understand the effect of computer noise on collective communication operations [31, 30, 3]. 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 [1], Kepler [41], and Pegasus [18] were developed for this purpose. The lambda architecture [42] is a dataflow solution to designing such applications in a more tightly coupled way. Amazon Step functions [2] is bringing the workflow to the FaaS and microservices.

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 (and unnecessary) to process as a complete set due to time requirements. Only temporal data sets 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 for 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 [29] 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 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.

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 the 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 [2] or dataflow to achieve the required functionality.

3.1 Dataflow Applications

Parallel computing and distributed computing are two of the general computing paradigms available for doing computations on large numbers of machines. MPI is the de facto standard in HPC for developing parallel applications. It provides a basic but powerful tool to develop parallel applications. 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 multicore and many core systems, the burden on the programmer to get the best available performance has increased dramatically [21, 20]. Because of the load imbalance and velocity of the big data applications, an MPI program written with tight synchronized operations across parallel workers may not perform well. An example HPC application is shown in Fig. 1 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.

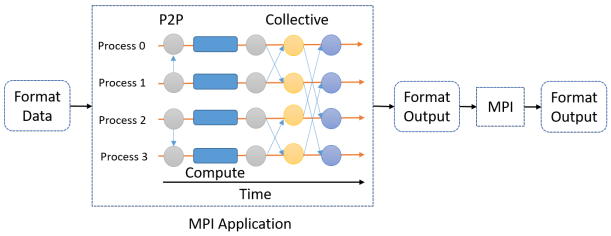

Figure 1: MPI applications arranged in a workflow

Dataflow computing has been around in various forms for a long time. A dataflow program is a computation graph with nodes doing computations and edges passing messages between the nodes. Computation at a node is invoked when its input data dependencies are satisfied. It is a largely accepted truth that dataflow programs are easier to write than MPI-style programs for applications that fit the dataflow model well. In a dataflow program, the user has to program the computations in the nodes and define how the nodes are connected to each other. The dataflow framework handles the details, such as executing the tasks using threads, scheduling, data placement and communications. A carefully designed framework can be tuned to run in different hardware with NUMA boundaries, caches and memory hierarchies.

3.1.1 Dataflow Application APIs

Over the years, there have been numerous languages and different types of APIs developed for creating dataflow applications. Task-based programming and data transformation-based programming are two popular approaches for dataflow parallel applications.

Data transformation APIs are used primarily by big data systems. Data transformation APIs employ a functional programming approach to create the dataflow graph implicitly. In this approach, distributed data is represented

in some abstract form and functions are applied to it that return other distributed data. A function takes a user defined operator as an argument and defines the communication between the operators. An example function is a partition function, often called a map in data flow runtimes. A map function works on partitions of a data set and presents the partitioned data to the user defined operators. The output of a user defined operator is connected to another user defined operator by way of another function.

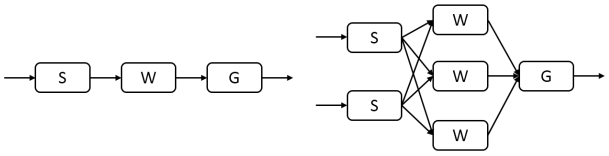

Figure 2: Left: User graph, Right: execution graph of a data flow

Task-based APIs are primarily used by the HPC community with AMT systems. A task-based API usually creates the dataflow tasks at the runtime as the program progresses. A normal program is used to create the tasks, which can use complex control operations such as 'if/else' and 'for' loops to control the dataflow dynamically at runtime.

3.1.2 Execution Graph

The graph executed by the dataflow runtime is termed execution/physical graph. This is created by the framework when the user graph is deployed on the cluster. For example, some user functions may run in larger numbers of nodes depending on the parallelism specified. Also when creating the execution graph, the framework can apply optimization to make some dataflow operations more efficient by reducing data movement and overlapping I/O and computations. Fig. 2 shows the execution graph and the user graph where it runs multiple W operations and S operations in parallel. Each user defined task runs on its own program scope without access to any state regarding other tasks. The only way to communicate between tasks is by messaging, as tasks can run in different nodes.

3.1.3 Data Partitioning

A big data application requires the data to be partitioned in a hierarchical manner due to memory limitations. Fig. 4 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.

3.1.4 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 subsystem and networks. Added to this is the Java JVM noise which most notably comes from

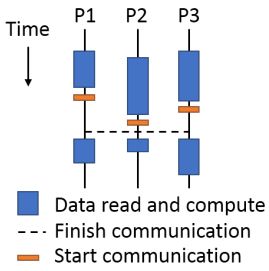

Figure 3: Load imbalance and velocity of data

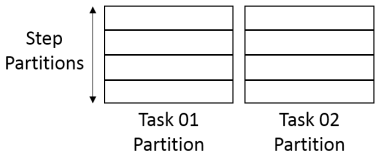

Figure 4: Hierarchical data partitioning of a big data application

garbage collection. The computations in 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, dataflow operations are the most suitable for such environments. Load balancing [45] is a much harder problem in streaming settings where data skew is more common because of the nature of applications.

3.2 Dataflow for Big Data Applications

3.2.1 Streaming Applications

Streaming applications deal with load imbalanced data coming at different rates to parallel workers at any given moment. Having an MPI application processing this data will increase the latency of the individual events. Fig. 3 shows this point with an example where three parallel workers process messages arriving at different speeds and sizes(different 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.

3.2.2 Data Pipelines

Data pipelines can be viewed as a special case of streaming applications. They work on hierarchically partitioned data as shown in Fig 4. 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 different in streaming and data pipeline applications. Usually every task in a data pipeline is executed in each CPU sequentially so only a subset of tasks

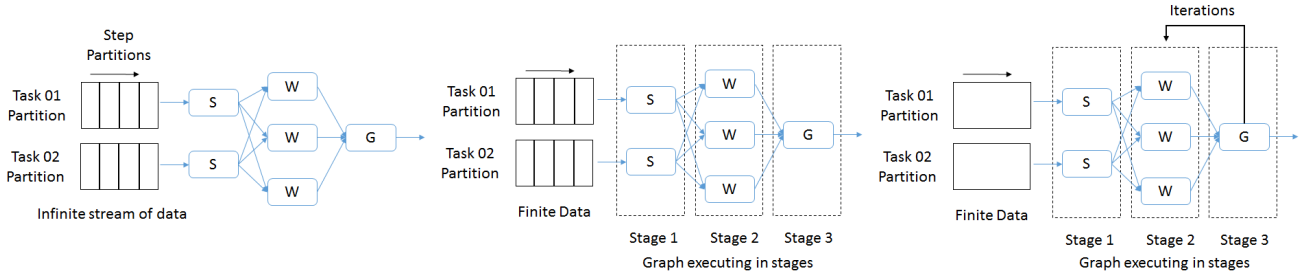

Figure 5: Dataflow application execution, Left: Streaming execution, Middle: Data pipelines executing in stages, Right: Iterative execution

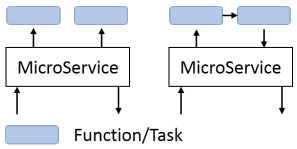

Figure 6: Microservices using FaaS, Left: Functions using a workflow, Right: Functions in a dataflow

are 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.

3.2.3 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 should hide the 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 [14, 26].

3.2.4 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. 6 shows microservices using functions arranged in a workflow and in a dataflow.

4. RUNTIME ARCHITECTURE

The general architecture of a runtime designed for big data is shown in Fig. 7. An application is created using a graph API and an optimizer can be used to make the graph execution efficient. A resource scheduler then allocates the required computing resources to run the processes required, including a master process to manage the job and a set of executors to execute the tasks. There can be additional processes to manage state and gather statistics but these are not essential. The executors use threads to invoke the tasks and manage the communications. The task scheduler can be distributed to run in each executor or be central. The task execution model adopted is a hybrid model where both processes and threads are used. A single executor can host multiple tasks of the execution graph and execute them using threads.

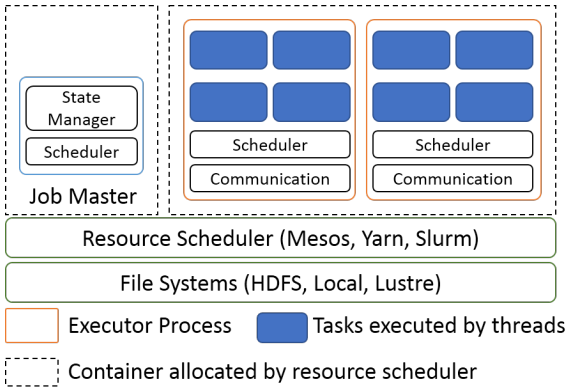

Figure 7: Dataflow runtime architecture

4.1 Communication

Communication is a fundamental requirement of distributed computing because the performance of the applications largely revolves around efficient implementations. The communication patterns that can involve more than two parallel tasks are termed collective communications. These patterns as identified by the parallel computing community are available through frameworks such as MPI [24]. Some of the heavily used communication patterns are Broadcast, Gather, Reduce, AllGather and AllReduce [51].

The naive implementation of these communication patterns using point-to-point communication in a straightforward way produces worst-case performance in practical large-

scale parallel applications. These communication patterns can be implemented using algorithms that minimize the bandwidth utilization and latency of the operation. In general they are termed collective algorithms.

4.1.1 MPI Collective Operations

In MPI, collective operations and other point-to-point communication operations are driven by control operations. This means the programmer knows exactly when to execute the send or receive functions. Once the program is ready to receive or send data, 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 use other operations such as wait/probe to complete the pending operation. MPI has clear standard APIs defined for collective communication patterns and all MPI implementations follow these specifications. The underlying implementation for such a communication pattern can use different algorithms based on factors including message size among others. Significant research has been done on MPI collectives [51, 48] and the current implementations are optimized to an extremely high extent. A comprehensive summary of MPI collective operations and possible algorithms is found in [55].

4.1.2 Dataflow Collective Operations

A communication pattern defines how the links are arranged in the dataflow graph. For instance a single node can broadcast a message to multiple nodes 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 [7] and broadcast.

MPI and big data have adopted the same type of collective communications but sometimes they have diverged in supported operations. Table 1 shows some of the collective operations and their availability in MPI and dataflow systems. Even though some MPI collective operations are not present in big data systems, they can be effective. Harp [58] is a machine learning focused collective library and supports the standard MPI collectives as well as some other operations like rotate, push and pull.

It is important to observe that dataflow applications use keyed collective operations. Unlike in MPI where the operations happen in-place, dataflow operations happen between individual tasks. Without keyed operations, it is not possible to direct the outcome of a collective operation to a task.

4.1.3 Optimized Dataflow Collective Operations

Each task in a dataflow graph can only send and receive data via its input and output ports and parallel tasks cannot communicate with each other while performing computations. The authors of this paper propose collective operations as a dataflow graph enrichment, which introduces sub-tasks to the original dataflow graph. Fig **??** and Fig. 8 show the naive implementation and our proposed approach for dataflow collective operations. In this approach, the collective operationâĂŹs computation is moved to a sub-task

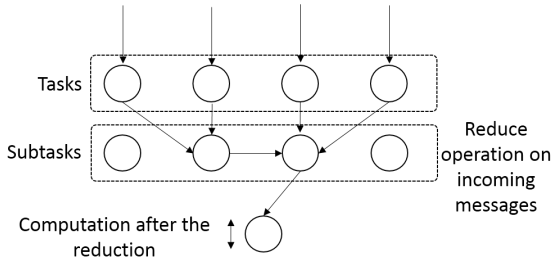

Figure 8: Collective operation with sub-tasks arranged in a tree

under which the collective operation depends. These subtasks 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 operation can run 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 on partitions of data and need to finish only after all the partitions are processed.

4.1.4 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 performs well compared to Ethernet counterparts. Some RDMA fabrics are developed especially targeting MPI [8]-type applications. Recently there have been many efforts to bring RDMA communications to big data systems, including HDFS [33], Hadoop [39] and Spark [40]. 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 [27] and Photon [36], this is no longer the case.

4.2 Task Scheduling, Threads & Processes

Task scheduling is a key area in which MPI, static and dynamic dataflow systems differ. From an MPI perspective, the task scheduling is straightforward, as MPI only spawns processes to run. It is the responsibility of the user to spawn threads and assign the computations appropriately. This process becomes harder for the MPI programmer when designing applications to run on many-core and multicore

Table 1: MPI and big data collective operations

MPI	Big Data	Algorithms available	
		Small Messages	Large Messages
Reduce	Reduce, Keyed Reduce	Flat/Binary/N-ary/Binomial Tree	Pipelined/Double/Split Binary Tree, Chain
AllReduce	N/A	Recursive Doubling, Reduce followed by Broadcast	Ring, Rabenseifner Algorithm, Recursive Doubling, Vector Halving with Distance Doubling
Broadcast	Broadcast	Flat/Binary Tree	Pipelined/Double/Split Binary Tree, Chain
Gather	Aggregate, Keyed Aggregate	Flat/Binary/N-ary/Binomial Tree	Pipelined/Double/Split Binary Tree, Chain
AllGather	N/A	Recursive Doubling, Reduce followed by Broadcast	Ring, Rabenseifner Algorithm, Recursive Doubling, Vector Halving with Distance Doubling
Barrier	N/A	Flat/Binary/Binomial Tree	
Scatter	N/A	Flat/Binary Tree	Pipelined/Double/Split Binary Tree, Chain
N/A	Join	Distributed radix hash, sort merge	

systems. Here it is worth noting that the vast majority of programmers are not comfortable with threads, let alone possess the skill to get good performance from them by efficient use of locks.

Static and dynamic scheduling are the two main paradigms used in scheduling tasks.

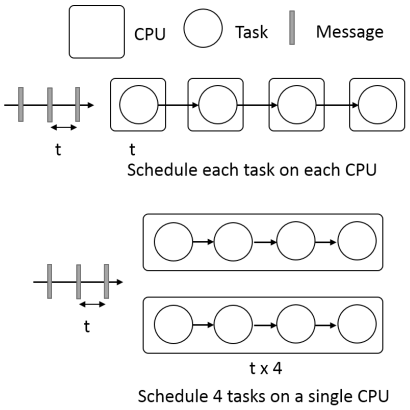

Figure 9: Top: Stream task scheduled in 4 CPU in a chain. Bottom: All streaming tasks scheduled in a single CPU.

Static graph scheduling necessitates the graph be available from the beginning. Because streaming systems require the entire graph to run continuously, this is the only way for such systems to operate. However that does make it harder to express complex applications, especially when containing loops. So this approach is suitable for data parallel applications such as streaming and data pipelines. Because the entire graph is available upon submission, graph optimization techniques can be applied to obtain the execution graph.

Dynamic graph scheduling allocates and schedules tasks at the run-time as the computation progresses. Because the graph is generated on the fly by a control program, more complex operations can be specified easily. AMT systems are dynamic graph execution systems. Being driven by a normal program, this method is not suitable for streaming data applications.

4.2.1 Streaming & Batch Task Scheduling

Streaming systems need to allocate all the tasks of the graph to run continuously, as well as optimize for latency. To illustrate requirements of stream task scheduling, let us take a hypothetical example where we have 4 computations to execute on a stream of messages with each computation taking t CPU time. Assume we have 4 CPUs available and the data rate is 1 msg per t CPU time. If we run all 4 tasks on a single CPU as shown in Fig. 9, it takes $t \times 4$ time to process one message and the computation cannot keep up with the stream using 1 CPU. So we need to load balance between the 4 CPUs and the order of the processing is lost unless explicitly programmed with locks to keep the state across 4 CPUs. But it is worth noting that the data remains in a single thread while the processing happens, thus preserving data locality. If we perform the schedule as in Fig. 9 the data locality is lost but the task locality is preserved. [37] describes stream computation scheduling on multicore systems in great detail.

For batch dataflow applications, the tasks are executed as the computation progresses. This means sequential tasks can run on a single CPU as time passes, unlike in a streaming system. Usually a single thread is scheduled to run on a single core and when the tasks become ready to run, this thread can execute them.

4.2.2 Execution of Tasks

As described earlier, a thread-based shared memory model is used for executing the tasks of the dataflow. This allows both pipelined execution of tasks and sharing of data through memory. Such pipelined execution of tasks is critical for streaming and data pipeline application to achieve efficient computations. Because of the large number of cores available in modern systems, it is required to strike a balance between the number of executor processes run in a single node. If fewer executors run in a node, that means a single

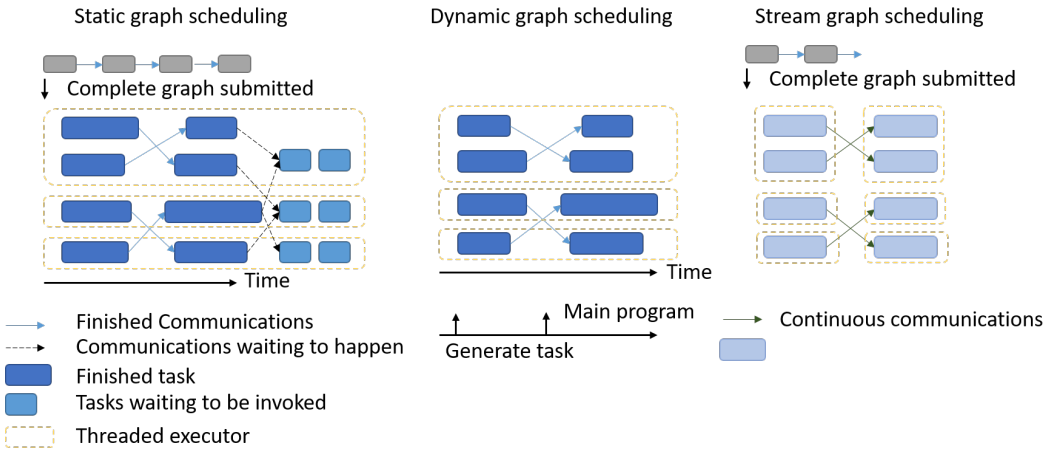

Figure 10: Task scheduling and execution of dataflow frameworks

executor has to cope with larger memory, which can lead to TLB misses and long JVM GC pauses. If more executors are used, the data has to be moved among the processes.

4.3 Data Management & Fault Tolerance

Big data applications work with input data and intermediate data generated through calculations. Apart from this, some state has to be maintained about the computations. In most applications the input data is not updated and model data is updated frequently. A MPI application delegates the complete data management to user, allowing her to load data, partition the data and place the data as needed. The same data set can be updated though-out the computations allowing efficient use of memory. Because partitioning and placement of the data is controlled by the user, this approach allows the best possible optimizations.

Even-though MPI approach is flexible, user needs to work with low level details in-order to write such applications. Distributed shared memory architectures have been proposed to ease some of the burdens of data management from user. Dataflow runtimes can use distributed shared memory [47] (DSM) for data and state management. In general a DSM presents distributed memory as a continuous global address space to the programmer. In such systems, the tasks of the graph are only allowed to work with DSM and no local process level state is kept. This permits the tasks of a dataflow graph to be migrated freely among the nodes. The migration allows the system to recover from node failures and balance load at runtime. HPC community has developed systems such as partitioned global address space (PGAS) [16] distributed memory. The big data community developed DSM technologies like RDD [56] for the same purpose.

RDD and other big data DSMs are a relaxed implementation of general DSM architecture where only coarse-gained operations are allowed. These DSMs are immutable, meaning once created they cannot be changed. Because tasks of a dataflow application work on partitions of such distributed memory, when a task fails it can be migrated and recalculated without any side effects. If there are global data dependencies for a partition of such a distributed data set, complete operations may be required to execute again, so checkpointing mechanisms are employed to save state in order to reduce deep recalculations. Because these data sets are immutable, they need to be created every time there are updates to them. This can lead to unnecessary resource consumption for complex iterative application where the models change frequently during calculations. These DSMs do not allow random access and fine grained control of data, which can lead to inefficiencies in complex applications [35]. It is worth noting that, they work extremely well for pleasingly parallel data pipeline type applications.

Fault tolerance is tightly coupled with how the state is managed. If fine-grained control of state is allowed, the checkpointing has to take these into account. The frequency of such checkpointing is mostly given as an option to the application developer. Such an architecture would require recalculation of large portions of the application in case of a failure depending on the rate of checkpointing.

5. SYSTEMS DESIGN

We have chosen a representative set of runtimes that can support at least one of the application areas comfortably, then studied their strengths and weaknesses related to the design choices they have made. Table 2 shows these systems along with design choices and their applicability to the three application areas. MPI is the default choice for HPC applications and HPX-5 is an AMT system. Both can handle applications with tightly synchronized parallel operations. Spark and Flink are mainly data pipeline systems, with Flink being able to handle streaming computations naively because of its static scheduling. Spark is a dynamic scheduling system and cannot handle low latency streaming computations. Storm/Heron are streaming systems that cannot handle batch applications but are well suited for streaming analytics.

5.1 A Toolkit for Big Data

Our study has identified high level features of a big data runtime that determine the type of application which can be executed efficiently. This toolkit aims to compose systems using well-defined components, and has the following implications: 1) It will allow developers to **choose only the components** that they need in order to develop the appli-

Table 2: Design choices of current runtimes

	Frameworks					
	MPI	HPX-5	Spark	Flink	Naiad	Storm/Heron
Task Scheduling	Static	Dynamic	Dynamic	Static	Static	Static
Execution	User control	Task-based threads	Task-based threads	Task-based threads	Task-based threads	Task-based threads
API	In-place communication	Task-based	Data transformation	Data transformation	Data transformation	Explicit graph creation
Optimized commns	Yes	Yes	No	No	No	No
RDMA	Yes	Yes	No	No	No	No
DSM	No	Fine grained	Coarse Grained	Coarse Grained	Coarse Grained	No
Streaming	No	No	Yes - high latency	Yes	Yes	Yes
Data pipelines	No	No	Yes	Yes	Yes	N/A
Machine learning	Yes	Yes	Performance can be poor	Performance can be poor	N/A	N/A

cation. For example, a user may only want MPI-style communication with a static 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 observed that communications, task scheduling and distributed shared memory (if required) are the three main factors affecting the applications. The API needs to adapt to each of these choices as well. Table 3 shows the different capabilities expected from different types of big data applications described herein.

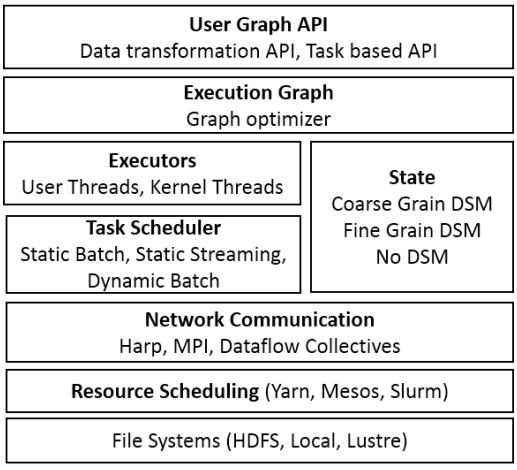

Figure 11: The toolkit approach

We propose a high performance communication library including both MPI and dataflow style communications, a task scheduler capable of scheduling according to different requirements, a thread-based task executor, and a low-level graph representation API as the basis of the toolkit. This can be further enhanced with a distributed shared memory implementation supporting coarse-grained and fine-grained operations. Different APIs can be built on top of such a system including low-level and high-level APIs to support different use cases. High level APIs can be in the form of languages, SQL queries or domain specific APIs. The support of classic HPC communications such as MPI will further enhance the system's adaptability to existing legacy applications.

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. The 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, 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 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. The authors are actively working on the implementation of various components of the toolkit and APIs in order to deliver on the promised flexibility across various applications.

Acknowledgments

This work was partially supported by the Indiana University Precision Health Initiative and by NSF CIF21 DIBBS 1443054 and NSF RaPyDLI 1415459. We thank Intel for their support of the Juliet system, and extend our gratitude to the FutureSystems team for their support with the infrastructure.

7. REFERENCES

[1] Apache NiFi.
[2] AWS Step Functions.

Table 3: Requirements of applications

Type of applications	Capabilities			
	Scheduling	API	Communications	Data and State
Streaming	Static Scheduling	Static graph API	Optimized Dataflow Collectives	Coarse grain DSM, Local
Data Pipelines	Static/Dynamic Scheduling	Static or dynamic graph generation	Optimized dataflow collectives	Coarse grain DSM, Local
Machine learning	Dynamic Scheduling	Dynamic graph generation	Optimized dataflow/MPI collectives	Fine grain DSM, Local
FaaS	Dynamic Scheduling	Dataflow or Workflow	P2P Communication	Local

[3] S. Agarwal, R. Garg, and N. K. Vishnoi. *The Impact of Noise on the Scaling of Collectives: A Theoretical Approach*, pages 280–289. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[4] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044, Aug. 2013.

[5] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing. *Proc. VLDB Endow.*, 8(12):1792–1803, Aug. 2015.

[6] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker. Fault-tolerance in the Borealis Distributed Stream Processing System. *ACM Trans. Database Syst.*, 33(1):3:1–3:44, Mar. 2008.

[7] C. Barthels, I. Müller, T. Schneider, G. Alonso, and T. Hoefler. Distributed Join Algorithms on Thousands of Cores. *Proc. VLDB Endow.*, 10(5):517–528, Jan. 2017.

[8] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak. Intel x00AE; Omni-path Architecture: Enabling Scalable, High Performance Fabrics. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 1–9, Aug 2015.

[9] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali. A Case for NUMA-aware Contention Management on Multicore Systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 557–558, New York, NY, USA, 2010. ACM.

[10] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. fog computing and its role in the internet of things.

[11] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. DAGuE: A generic distributed DAG engine for High Performance Computing. *Parallel Computing*, 38(1):37 – 51, 2012. Extensions for Next-Generation Parallel Programming Models.

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

[13] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[14] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *Proceedings of the 19th International Conference on Neural Information Processing Systems*, NIPS'06, pages 281–288, Cambridge, MA, USA, 2006. MIT Press.

[15] J. Conejero, S. Corella, R. M. Badia, and J. Labarta. task-based programming in compss to converge from hpc to big data.

[16] M. De Wael, S. Marr, B. De Fraine, T. Van Cutsem, and W. De Meuter. partitioned global address space languages.

[17] J. Dean and S. Ghemawat. MapReduce: A Flexible Data Processing Tool. *Commun. ACM*, 53(1):72–77, Jan. 2010.

[18] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[19] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: A Runtime for Iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, New York, NY, USA, June, 2010. ACM.

[20] S. Ekanayake, S. Kamburugamuve, and G. C. Fox. Spidal java: High performance data analytics with java and mpi on large multicore hpc clusters. In *Proceedings of the 24th High Performance Computing Symposium*, HPC '16, pages 3:1–3:8, San Diego, CA, USA, 2016. Society for Computer Simulation International.

[21] S. Ekanayake, S. Kamburugamuve, P. Wickramasinghe, and G. C. Fox. Java thread and process performance for parallel machine learning on multicore hpc clusters. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 347–354, Washington, DC, USA, Dec 2016. IEEE.

[22] G. Fox, J. Qiu, S. Jha, S. Ekanayake, and S. Kamburugamuve. *Big Data, Simulations and HPC Convergence*, pages 3–17. Springer International Publishing, Cham, 2016.

[23] G. Fox, J. Qiu, S. Kamburugamuve, S. Jha, and A. Luckow. Hpc-abds high performance computing enhanced apache big data stack. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 1057–1066, New York, NY, USA, May 2015. IEEE.

[24] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. *Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation*, pages 97–104. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[25] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: The System s Declarative Stream Processing Engine. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1123–1134, New York, NY, USA, 2008. ACM.

[26] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *2011 IEEE 27th International Conference on Data Engineering*, pages 231–242, April 2011.

[27] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, and J. M. Squyres. A Brief Introduction to the OpenFabrics Interfaces - A New Network API for Maximizing High Performance Application Efficiency. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 34–39, Aug 2015.

[28] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[29] J. Han, H. E, G. Le, and J. Du. Survey on nosql database. In *2011 6th International Conference on Pervasive Computing and Applications*, pages 363–366, Oct 2011.

[30] T. Hoefler, T. Schneider, and A. Lumsdaine. The impact of network noise at large-scale communication performance. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–8, May 2009.

[31] T. Hoefler, T. Schneider, and A. Lumsdaine. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[32] D. Hollman, J. Lifflander, J. Wilke, N. Slattengren, A. Markosyan, H. Kolla, and F. Rizzi. Darma v. beta 0.5, Mar 2017.

[33] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. High performance rdma-based design of hdfs over infiniband. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 35:1–35:35, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[34] L. V. Kale and S. Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM.

[35] S. Kamburugamuve, P. Wickramasinghe, S. Ekanayake, and G. C. Fox. anatomy of machine learning algorithm implementations in mpi, spark, and flink.

[36] E. Kissel and M. Swany. Photon: Remote Memory Access Middleware for High-Performance Runtime Systems. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1736–1743, May 2016.

[37] M. Kudlur and S. Mahlke. Orchestrating the Execution of Stream Programs on Multicore Platforms. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 114–124, New York, NY, USA, 2008. ACM.

[38] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.

[39] X. Lu, N. S. Islam, M. Wasi-Ur-Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda. High-performance design of hadoop rpc with rdma over infiniband. In *2013 42nd International Conference on Parallel Processing*, pages 641–650, New York, NY, USA, Oct 2013. IEEE.

[40] X. Lu, D. Shankar, S. Gugnani, and D. K. D. K. Panda. High-performance design of apache spark with RDMA and its benefits on various workloads. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 253–262, Dec 2016.

[41] B. LudÃd'scher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[42] N. Marz and J. Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2015.

[43] T. G. Mattson, R. Cledat, V. CavÃl', V. Sarkar, Z. BudimliÄĞ, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, M. Lee, B. Meister, B. Nickerson, N. Pepperling, B. Seshasayee, S. Tasirlar, J. Teller, and N. Vrvilo. The open community runtime: A runtime system for extreme scale computing. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, Sept 2016.

[44] D. G. Murray, F. McSherry, R. Isaacs, M. Isard,

P. Barham, and M. Abadi. naiad: A timely dataflow system.

[45] M. A. U. Nasir, G. D. F. Morales, D. Garcia-Soriano, N. Kourtellis, and M. Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *2015 IEEE 31st International Conference on Data Engineering*, pages 137–148, April 2015.

[46] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. In *2010 IEEE International Conference on Data Mining Workshops*, pages 170–177, Dec 2010.

[47] B. Nitzberg and V. Lo. distributed shared memory: a survey of issues and algorithms. *Computer*.

[48] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. Performance analysis of MPI collective operations. *Cluster Computing*, 10(2):127–143, 2007.

[49] R. Ranjan. streaming big data processing in datacenter clouds. *IEEE Cloud Computing*.

[50] T. Sterling, M. Anderson, P. K. Bohan, M. Brodowicz, A. Kulkarni, and B. Zhang. *Towards Exascale Co-design in a Runtime System*, pages 85–99. Springer International Publishing, Cham, 2015.

[51] R. Thakur and W. D. Gropp. *Improving the Performance of Collective Operations in MPICH*, pages 257–267. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[52] W. Thies, M. Karczmarek, and S. Amarasinghe. *StreamIt: A Language for Streaming Applications*, pages 179–196. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[53] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM.

[54] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., Sebastopol, CA, USA, 1st edition, 2009.

[55] U. Wickramasinghe and A. Lumsdaine. A survey of methods for collective communication optimization and tuning. *arXiv preprint arXiv:1611.06334*, 2016.

[56] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[57] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[58] B. Zhang, Y. Ruan, and J. Qiu. Harp: Collective Communication on Hadoop. In *2015 IEEE International Conference on Cloud Engineering*, pages 228–233, March 2015.