# TOWARDS HPC AND BIG DATA CONVERGENCE IN A COMPONENT BASED APPROACH

Supun Kamburugamuve

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the School of Informatics and Computing & Engineering
Indiana University
May 2018

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Geoffrey Fox, Ph.D.

Martin Swany, Ph.D.

Judy Qiu, Ph.D.

Minje Kim, Ph.D.

Ryan Newton, Ph.D.

Date of Defense May 14, 2018

To my wife Chathuri, son Seth, and our parents.

# Acknowledgements

I have been blessed to be among many who have been guiding me and supporting me throughout the years dedicated to this research.

I would like to thank my advisor Prof. Geoffrey Fox who gave invaluable and brilliant research advice and directions. His remarkable insights and deep understanding of parallel and distributed computing made possible this research. I'm privileged to be able to conduct this research with his guidance.

I would like to thank Prof. Judy Qiu, whom I have worked with on different areas including research, teaching and different software projects for her guidance and support. Prof. Martin Swany gave great insights for communication improvements. I'm grateful to Prof. David Crandall for his inputs on robotics algorithms we worked together. I would like to thank Prof. Minje Kim and Prof. Ryan Newton for their valuable inputs.

I owe a lot to Dr. Sanjiva Weerawarana who encouraged me to pursue a graduate degree. Without his encouragement, I would not have enrolled in a PhD program. Special thanks go to Saliya Ekanayake whom I had the privilege to work with while he was an PhD candidate under the same advisor. We shared ideas and worked on projects together and it was a pleasure working with him. I would like to thank Henjing He who was a visiting student from Dept. of Electrical Engineering, Tsinghua University in developing robotics algorithms.

Further, I would like to thank team members of Twister2 project who are contributing to making the Twister2 project a reality. Especially I would like to thank Pulasthi Wickramasinghe whom I worked on many research projects prior to Twister2. Kannan Govindarajan, Ahmet Uyar, Gurhan Gunduz and Vibhatha Abeykoon are major contributors to the Twister2 project. I would like to thank systems administrators, especially Allan Streib, of our lab who provided an excellent service to make this research a success.

Words cannot explain the love, support, and encouragement of my parents. They have taught me to be ambitious and do things otherwise I would not have done.

None of this research would have been possible if not for my wife Chathuri who has been with me throughout highs and lows of graduate life. Her love and support guided me through the graduate life. My son Seth is too little to understand what his daddy is doing but his smiles kept his dad strong.

Supun Kamburugamuve

## TOWARDS HPC AND BIG DATA CONVERGENCE IN A COMPONENT BASED APPROACH

Data-driven applications are essential to handle the ever-increasing volume, velocity, and veracity of data generated by Web and Internet of things devices. 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. An event-driven distributed computing platform consists of well-understood components such as communication, task management, data management and fault tolerance. Different design choices adopted by these components determine the type of applications a system can support efficiently.

Stream and batch data analytics applications are dominant with systems such as Hadoop, Spark, and Heron are designed to execute them. This thesis investigates modern Apache big data systems with different applications and compares their designs and performance with classic HPC techniques. It shows that by using HPC features such as high performance interconnects one can improve the performance of big data systems. Further, it recognizes that these systems are mostly developed as single projects with fixed design choices at different components making them limited to specific applications.

To overcome design limitations of modern big data systems, a loosely coupled component-based big data toolkit is proposed and its design and implementation are discussed. This approach allows components to have different implementations to support different applications. Further, the thesis recognizes the requirements of both dataflow used in Apache Systems and the Bulk Synchronous Processing style in HPC for different applications. Message passing interface(MPI) implementations are dominant in HPC, but there are no such standard libraries available for data processing. The thesis presents a novel stand-alone, highly optimized dataflow style parallel communication library that can be used by big data systems both in clouds and HPC environments.

Geoffrey Fox, Ph.D.

Martin Swany, Ph.D.

Judy Qiu, Ph.D.

Minje Kim, Ph.D.

Ryan Newton, Ph.D.

# Contents

# List of Figures

CHAPTER 1

# Introduction

## 1.1. Overview

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, edge computing [32] 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 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 API, 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 [153] has highlighted the importance of the Map-Collective computing paradigm.

The dataflow [75] 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 [95], 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.

## 1.2. Research Challenges

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.

The thesis shows the results of Machine learning algorithms and data processing workloads on Spark, Flink, and MPI. Also, it describes the contributions to integrate high performance interconnects to Heron and how the performance of Storm was improved using algorithmic techniques. The thesis also describes an end-to-end batch application and a streaming application developed using MPI and a streaming engine. With this work, following conclusions were made regarding current Big data frameworks.

(1) Major big data systems are developed according to the dataflow model

(2) There are many decisions to be taken at different aspects of a framework that determine the type of applications it can support

(3) Every system rely on the same components, but they have different implementations with different semantics

(4) One cannot use these components independently or improve them easily because of the tight integration

This thesis recognizes the need to clearly define the architecture of next generation systems that will take advantage of both HPC and Cloud environments. Because of the common underlying event-driven model, it is possible to build each component separately with clear abstractions supporting different design choices. We present a design and implementation of a polymorphic system 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 project is called Twister2, encompassing the concept of the toolkit. Twister2 will be a cloud native framework [4, 64] including Server-less FaaS.

Further, the thesis addresses the core question of what is the difference between HPC and Big data processing. It is fair to say that HPC mostly revolves around Message Passing Interface standard which is a Bulk Synchronous Processing model. Consequently, we recognize the requirements of both dataflow used in popular Apache Systems and the BSP communication style common in High-Performance Computing(HPC) for different applications. Message passing interface implementations are dominant in HPC but there are no such standard libraries available for big data. Twister:Net is the communication component of Twister2, which has been developed as a stand-alone library for big data applications supporting dataflow communications. Bulk synchronous parallel(BSP) communication as implemented by MPI or Harp [154] can be used in Twister2 as well. The dataflow communications in Twister:Net are implemented on top of TCP and MPI communications, allowing it to be deployed in both HPC and cloud environments. In this thesis we present, dataflow communications of Twister:Net and compare it to the existing big data frameworks in order to show it can achieve equivalent performance or better with existing frameworks. Also, we look at MPI communications and big data requirements and compare different applications in those settings.

## 1.3. Outline

Rest of the thesis is organized as chapters. Chapter 6 discusses related work in the area of HPC and big data computing for streaming and batch big data processing. Chapter 3 presents different application areas, frameworks, and three applications developed using HPC and big data tools. It compares the performance of these applications and showcases the differences between HPC and big data frameworks. This chapter forms the basis of the thesis by discussing the strengths and weaknesses of existing systems.

Chapter 4 describes an end-to-end machine learning algorithm developed using HPC technologies and big data. Chapter 5 describes an end-to-end streaming application for analyzing robotics data. These two chapters discuss the requirements of batch and streaming applications. Chapter 7 introduces the requirements and expectations of next generation parallel and distributed systems.

Chapter 8 introduces the Twister2 framework with different requirements and components. Chapter 9 introduces a novel communication library of Twister2 for big data processing. Chapter 10 briefly introduces the Twister2 task system and resource scheduler and Chapter 11 summarizes and concludes the thesis with future work.

CHAPTER 2

# Related Work

## 2.1. Batch Systems

Hadoop [144] was the first major open-source platform developed to process large amounts of data in parallel. The map-reduce [47] 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 [52] became apparent, and systems such as Apache Spark [151] and Apache Flink [38] 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 [118], Apache Apex [13] and Google Dataflow [19]. It is interesting to note that even with all its well-known inefficiencies, Hadoop is still being used by many people for data processing. 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 [19] 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 [138], Borealis [26], Spade [65] and S4 [121].

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.

## 2.2. Streaming Systems

There are several other DSPFs available, each solving specific problems with their own strengths. Such open source DSPFs includeApache Storm [140], Apache Heron [100], Apache Samza [125], Apache S4 [122], Apache Flink [1] and Apache Spark Streaming [152], with commercial solutions including Google Millwheel [18], Azure Stream Analytics and Amazon Kinesis. Early research in DSPFs include Aurora [15], Boreiles [11] and Spade [66]. Apache Spark Streaming uses micro-batch jobs on top of its batch engine to achieve streaming computations while Samza is a storage-first processing framework; both target high throughput applications rather than low latency applications. Apache Flink is comparable to Storm in its underlying execution and supports low latency message processing.

Neptune [36] and Heron [100] improve some of the inefficiencies of Apache Storm. Heron addresses issues in the task scheduling, enhance flow control by handling back pressure and improves connection management for large applications and task isolation for better performance in Storm. Neptune streamlines the throughput of applications by using multiple threads for processing and efficient memory management. Apache Flink employs efficient fault tolerance and message processing guarantee algorithms [38] that are lightweight and introduce minimal overheads to the normal processing flow when compared to the upstream backup [85] mechanisms utilized by DSPFs such as Apache Storm. Further, Nasir et al. [120] utilize a probabilistic algorithm in Storm's load balancing message distribution operator to avoid imbalance workloads in tasks. Adaptive scheduling [24] improves the scheduling of Storm by taking into account the communication patterns among the tasks and Cardellini et al. [39] has improved scheduling to take QoS into account.

All these systems follow the data flow model with comparable features to each other. Stream bench [104] is a benchmark developed to evaluate the performance of these systems in detail. These systems are primarily optimized for commodity hardware and clouds. There has been much research done around Apache Storm and Apache Heron to improve its capabilities. [61] described architectural and design improvements to Heron that improved its performance much further.

### 2.3. HPC Frameworks

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 [115], DADuE [33], Charm++ [88], COMPS [44] and HPX [135], all of which focus on dynamic scheduling of the computation graph. A portability API is developed in DARMA [81] 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 [35]. Research has been done with MPI to understand the effect of computer noise on collective communication operations [17, 79, 80]. 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.

## 2.4. HPC Intergration to big data

In large part, HPC and Big Data systems have evolved independently over the years. Despite this, there are common requirements that raise similar issues in both types of systems. Some of these issues are solved in HPC and some in Big Data frameworks. As such, there have been efforts to converge both HPC and Big Data technologies to create better systems that can work in different environments efficiently. SPIDAL [60] and HiBD are two such efforts to enhance the Big Data frameworks with ideas and tools from HPC. This work is part of an ongoing effort by the authors to improve stream engine performance using HPC techniques. Previously we showed [89] how to leverage shared memory and collective communication algorithms to increase the performance of Apache Storm. Also, the authors have looked at various available network protocols for Big Data applications [139].

There are synergies between HPC and big data systems, and authors [58, 60] among others [86] have expressed the need to enhance these systems by taking ideas from each other. In previous work [53, 54] we have identified the general implications of threads and processes, cache, memory management in NUMA [31], as well as multi-core settings for machine learning algorithms with MPI. DataMPI [102] uses MPI to build Hadoop-like systems while [22] 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.

In recent years there have been multiple efforts to integrate high performance interconnects with machine learning and big data frameworks. InfiniBand has been integrated into Spark [110] where the focus is on the Spark batch processing aspects rather than the streaming aspects. Spark is not considered a native stream processing engine and only implements streaming as an extension to its batch engine, making its latency inadequate for low latency applications. Recently InfiniBand has been integrated into Hadoop [108], along with HDFS [86] as well. Hadoop, HDFS, and Spark all use Java runtime for their implementations, hence the RDMA was integrated using JNI wrappers to C/C++ codes that invoke the underlying RDMA implementations. Recent work has added RDMA support for the Tensorflow [16] machine learning framework.

High performance interconnects have been widely used by the HPC community, and most MPI(Message Passing Interface) implementations have support for a large number of interconnects that are available today. Some early work that describes in detail about RDMA for MPI can be found in [103]. There has even been some work to build Hadoop-like systems using the existing MPI capabilities [109]. Photon [96] is a higher level RDMA library that can be used as a replacement to libfabric. RoCE [28] and iWARP [126] are protocols designed to use RDMA over Ethernet to increase the maximum packets per second processed at a node while decreasing latency. Heron doesn't take into account the network when scheduling tasks into available nodes and it would be interesting to consider network latencies as described in [87, 97] for task scheduling.

## 2.5. Messaging for Big Data

Message Passing Interface (MPI) [12] is primarily responsible for addressing messaging in parallel computing. It mainly supports two types of communications, point-to-point, and collective communication. The traditional reduce/aggregate communication pattern in Spark [8] sends all the partitions reduced data values into the driver program. However, the driver program occupies a linear amount of time in those partitions that can lead to a performance bottleneck if there are multiple data partitions and the data partition size is large. To reduce the bottleneck, Apache Spark 1.1 [9] has introduced new communication patterns named TreeReduce and TreeAggregate which are based on the multi-level aggregation tree technique. In this technique, the data partitions are combined into a small set of executors in a partial manner before they are sent to the driver program, which reduces the load of the driver program and improves the performance.

Apache Flink [37] follows the data-streaming paradigm, thereby providing a unified architecture for both batch and streaming processing in the programming model and execution engine. In Flink, the streams distribute the data based on the various communication patterns, namely point-to-point, broadcast, re-partition, fan-out and merge. In Storm and Heron [10], a spout is a source of input data streams for the topology and a bolt is a component which processes those topologies. The stream grouping is an important concept which defines the method to partition the data stream into bolt tasks. It consists of eight built-in custom stream grouping concepts, namely shuffle grouping, field grouping, partial key grouping, all grouping, global grouping, none grouping (similar to shuffle grouping), direct grouping, and local or shuffle grouping. Harp [154] is a framework which is mainly designed to run big data analytic algorithms on High Performance Computing architectures. It is comprised of two main layers, computation and communication. The communication library is implemented similar to MPI collective communication operations which are highly optimized for big data analytics and machine learning algorithms. COMPS [44] is a task-based environment for Spark-like applications in HPC.

MRNet [130] is a software-based reduction network specifically designed for scalable tools to achieve scalable performance and multicast support. It uses a communicator for representing groups of network points. Similar to MPI, MRNet provides the support for point-to-point and multicast or broadcast communications. DataMPI [106] communication library is an extended MPI specification to achieve Hadoop-like communications. Currently, it only supports single program and multiple data (Common), but they intended to support MapReduce, Streaming, and Iterations in the future. There are many ongoing efforts to incorporate HPC features into existing big data frameworks, including RDMA support for frameworks such as Spark [107], Hadoop [105], and Apache Heron [94]. The authors previously worked on improving Apache Storm's performance using collective algorithms [89]. MPIgnite is an effort to bring BSP-style communication into Spark [117].

## 2.6. Workflow Systems

In practice, multiple algorithms and data processing applications are combined together in workflows to create complete applications. Systems such as Apache NiFi [2], Kepler [112], and Pegasus [48] were developed for this purpose. The lambda architecture [114] is a dataflow solution for designing such applications in a more tightly coupled way. Amazon Step functions [3] are bringing the workflow to the FaaS and microservices.

There are many tools to develop large-scale workflows from both the HPC and Big Data communities. Some noteworthy examples of HPC workflows include Tavarna [148], Kepler [111] and Airavata [113]. Technologies such as Hadoop [144], Spark [151] and Flink [1] are also available for developing dataflows. Developing workflows on top of such frameworks provides other capabilities including fault tolerance and automatic parallelization for analyzing large data sets. To develop big data workflows, tools such as Apache Beam [19] and Crunch, which work as a layer on top of big data frameworks described above, can be used. A generic dataflow language such as Apache Beam can avoid locking into a single big data platform. The workflow described in this paper is implemented using Hadoop and Crunch and later will be implemented in Apache Beam (open source Google Cloud Dataflow).

CHAPTER 3

# Big data landscape

## 3.1. 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 [76] 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.

**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 [3] or dataflow to achieve the required functionality.

**3.1.1. 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. 3.2 shows an example of such partitioning

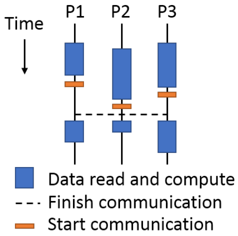


FIGURE 3.1. Load imbalance and velocity of data

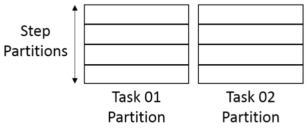


FIGURE 3.2. Hierarchical data partitioning of a big data application

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 [119] 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.

### 3.1.2. MPI for Big Data.

MPI is the de facto standard in HPC for developing parallel applications. An example HPC application is shown in Fig. 3.3 where a workflow system such as Kepler [112] 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 [53, 54]. 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 [95].

### 3.1.3. Dataflow for Big Data.

Data-driven computing is becoming dominant for big data applications. Dataflow has been recognized and accepted as the preferred mechanism for processing large data sets. A dataflow program can hide details such as communication, task execution, and data management from the user while giving higher level abstractions

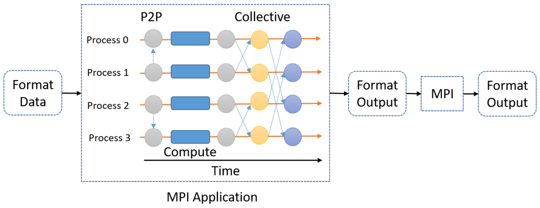


Figure 3.3. MPI applications arranged in a workflow

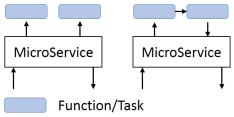


Figure 3.4. Microservices using FaaS, Left: Functions using a workflow, Right: Functions in a dataflow

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.

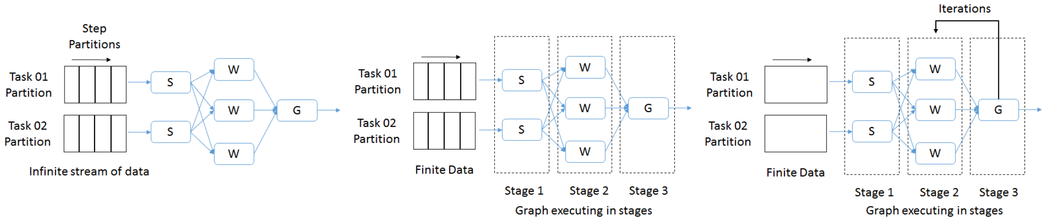


Figure 3.5. Dataflow application execution, Left: Streaming execution, Middle: Data pipelines executing in stages, Right: Iterative execution

3.1.3.1. *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. 3.1 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.

3.1.3.2. *Data Pipelines.* Data pipelines can be viewed as a special case of streaming application. They work on hierarchically partitioned data as shown in Fig 3.2. 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.

3.1.3.3. *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 [42, 67].

3.1.3.4. *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. 3.4 shows microservices using functions arranged in a workflow and in a dataflow.

## 3.2. Data flow and MPI

**3.2.1. Execution models.** Data flow frameworks model computation as a graph where nodes represent the operators that are applied to data and edges represent the communication channels between the operators. The input and output of an operator are sent through the edges. Operators are user defined code that executes on the data and produces other data. The graph is created using data flow functions provided by the framework. These data flow functions act upon the data to create the graph. An example function is a partition function, often called a map in data flow engines. A map function works on partitions of a data set and presents the partitioned data to the user defined operators at the nodes. The output of a user defined operator is connected to another user defined operator using a data flow function. Map and reduce are such two widely used data flow functions.

The data flow model consists of a logical graph and an execution graph. The logical graph is defined by the user with data, data flow functions and user operators. The execution graph is created by the framework to deploy the logical graph on the cluster. For example,

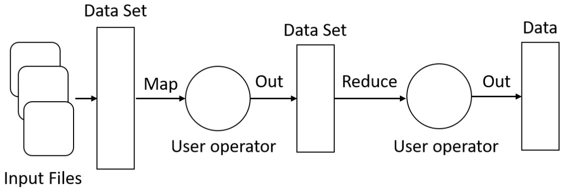


FIGURE 3.6. Logical graph of a data flow

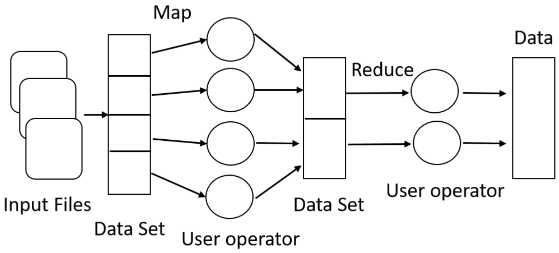


FIGURE 3.7. Execution graph of a data flow

some user functions may run in larger numbers of nodes depending on the user defined parallelism and size of input. Also when creating the execution graph the framework can apply optimizations to make some data flow operations more efficient. An example logical graph is shown in Fig. 8.4 where it displays a map and reduce data flow functions. Fig. 3.8 shows an execution graph for this logical graph where it runs multiple map operations and reduce operations in parallel. It is worth noting that each user defined function runs on its own program scope without access to any state about previous tasks. The only way to communicate between tasks is using messaging, as tasks can run in different places.

For task execution, frameworks use a thread model with fewer task managers executing the parallel tasks for a node. This leads to a smaller number of TCP connections between the nodes, but the individual tasks are subjected to interference by other task executions.

Message passing interface (MPI) is a generic model for doing parallel computations. It is used most widely in the single program multiple data (SPMD) style of parallel programs.

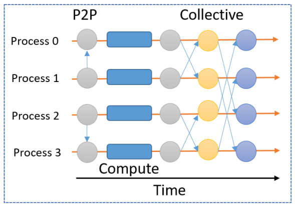


Figure 3.8. Execution flow of MPI program with compute and communication regions

The parallel processes can employ MPI functions for both message passing and synchronization between the processes. Fig. 3.8 shows the execution model of MPI. Unlike in data flow, the same process does the communion and computations throughout the execution. The processes are normal Unix versions spawned by MPI.

**3.2.2. Programming APIs.** Popular data flow engines such as Spark, Flink and Google Data Flow all enable a functional programming API. The data is abstracted as high level language objects. A large distributed data set is represented as a programming construct. For example, in Flink, the data structure is called a DataSet, while in Spark it is an RDD. These data sets are considered immutable, meaning they cannot be changed once created.

The user defined operators are implemented according to the functional programming paradigm with no side effects, meaning a user defined code cannot modify out-of-scope data sets or passed-in data sets. The only way an operator can produce a new data set is by outputting it, thus creating a different version every time. This clean programming paradigm makes it easier for an average programmer to write data-flow programs faster and error-free. But it comes with the cost of creating new structures every time a data set has to be modified. In MPI style programs, we almost always modify the existing memory structures without producing new data sets every time a small operation is applied to the data. The data flow functions are always applied to these data objects. Usually, a data flow function implements a user defined function to act upon the data it receives, although there are functions that don't accept user defined programs.

The data loading and data saving are abstracted out in terms of data sources and sinks. Data source and sinks are responsible for reading and writing the data while hiding the details of different file systems such as local file systems and distributed file systems like HDFS. The default data sources are sufficient for large sets of applications but for more advanced use cases users often end up writing their own readers and writers.

## 3.3. Big data frameworks

**3.3.1. Apache Flink.** Apache Flink is a batch and stream processing engine that models every computation as a data flow graph which is then submitted to the Flink cluster. The nodes in this graph are the computations and the edges are the communication links. Flink closely resembles both the data flow execution model and API. The user graph is transformed into an execution graph by Flink before it is executed on the distributed nodes. While undergoing this transformation, Flink optimizes the user graph, taking into account the data locality. Flink uses thread based worker model for executing the data flow graphs. It can chain consecutive tasks in the work flow in a single node to make the run more efficient by reducing data serializations and communications.

Even though Flink has a nice theoretically sound data flow abstraction for programming, we found that it is difficult to program in a strictly data-flow fashion for complex programs. This primarily due to the fact that control flow operations such as if conditions and iterations are harder code in data flow paradigm.

**3.3.2. Apache Spark.** Apache Spark is a distributed in-memory data processing engine. The data model in Spark is based on RDDs [150], and the execution model is based on RDDs and lineage graphs. The lineage graph captures dependencies between RDDs and their transformations. The logical execution model is expressed through a chain of transformations on RDDs by the user. This lineage graph is also essential in supporting fault tolerance in Apache Spark.

RDD's can be read in from a file system such as HDFS, and transformations are applied to the RDDs. Spark transformations are lazy operations and actual work is only done when an action operation such as count, reduce are invoked. By default, intermediate RDDs created through transformations are not cached and will be recomputed when needed. The user has the ability to cache or persist intermediate RDDs by specifying this explicitly. This is very important for iterative computation where same data sets are being used over and over again.

Spark primarily uses a thread based worker model for executing the tasks. Unlike in Flink where the user submits the execution graph to the engine, Spark programs are controlled by a driver program. This driver program usually runs in a separate master node and the parallel regions in this driver program are shipped to the cluster to be executed. With this model complex control flow operations that needs to run serially such as iterations and if conditions run in master while data flow operators are executed in worker nodes. While this model makes it easier to write complex programs, it is harder to do complex optimizations on the data flow graph as it needs to be executed on the fly.

**3.3.3. Apache Storm.** Every DSPF consists of two logical layers identified as the application layer and the execution layer. The application layer provides the API for the user to define a stream processing application as a graph. The execution layer converts this user defined graph into an execution graph and executes it on a distributed set of nodes in a cluster.

3.3.3.1. *Storm Application Layer.* A Storm application called a topology determines the data flow graph, with streams defining the edges and processing elements defining the nodes. A stream is an unbounded sequence of events flowing through the edges of the graph, and each such event consists of a chunk of data. A node in the graph is a stream operator implemented by the user. The entry nodes in the graph acting as event sources to the rest of the topology are termed Spouts while the rest of the data processing nodes are called Bolts. The spouts generate event streams to the topology by connecting to external event sources such as message brokers. From here onwards we refer to both spouts and bolts as processing elements (PEs) or operators. Bolts consume input streams and produce output streams. The user code inside a bolt executes when an event is delivered to it on incoming links. The topology defines the logical flow of data among the PEs in the graph by using streams to connect PEs. A user can also define the parameters necessary to convert this user defined graph into an execution graph. The physical layout of the data flow is mainly defined by the parallelism of each processing element and the communication strategies defined among them. This graph is defined by the user who deploys it to the Storm cluster to be executed. Once deployed, the topology runs continuously, processing incoming events until it is terminated by the user. An example topology is shown in Figure 3.9 where it has a spout connected to a bolt by a stream and a second bolt connected to the first bolt by another stream.

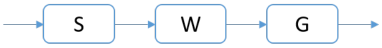


FIGURE 3.9. A sample stream processing user defined graph

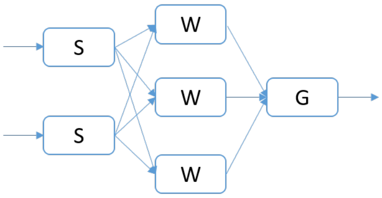


FIGURE 3.10. A sample stream processing execution graph

3.3.3.2. *Storm Execution layer.* Storm master (known as Nimbus) converts logical graph of processing elements to an execution graph by taking the number of parallel tasks for each logical PE and the stream grouping 6.1.1 into account. For example, Figure 3.10 displays an execution graph of the user graph shown in Figure 3.9 where two instances of $S$, three instances of $W$ and one instance of $G$ are running. The stream grouping between $S$ and $W$ is a load balancing grouping where each instance of $S$ distributes its output to the 3 $W$ instances in a round-robin fashion. A runtime instance of a node in the execution graph is called a task.

After converting logical graph to execution graph, master node takes care of scheduling of the execution graph and also manages the stream processing applications running in the cluster. Each slave node runs a daemon called a supervisor, which is responsible for executing a set of worker processes which in turn execute the tasks of the execution graph. Tasks in an execution graph will get assigned to multiple workers running in the cluster. Figure 3.11 shows one configuration of the example topology assigned to two nodes both running two workers. Each worker can host multiple tasks of the same graph, and the worker assigned a thread of execution to every task. If multiple tasks run in the same worker, multiple threads execute the user codes in the same worker process.

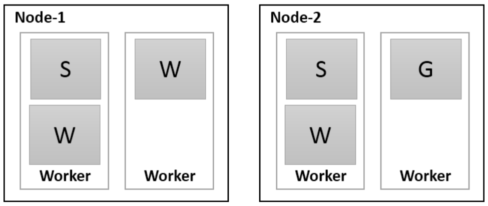


FIGURE 3.11. Storm task distribution in multiple nodes

## 3.4. Algorithms on MPI, Spark and Flink

In this section we will look at three algorithms with MPI, Spark and Flink to compare their performance. These algorithms are Multidimensional Scaling, K-Means and Terasort.

**3.4.1. Multidimensional Scaling.** Multidimensional scaling is a popular, well established machine learning technique for projecting high dimensional data into a lower dimension so that they can be analyzed. It has been extensively used to visualize high dimensional data by projecting into 3D or 2D models. MDS is a computationally expensive algorithm. The best algorithms are in the range of $O(N^2)$, where N is the number of data points. When applied to a larger data set, the computation time increases exponentially. The algorithm can be made to run efficiently in parallel to reduce the computation time requirements.

Parallel version of MDS requires multiple parallel regions and nested iterations for its computations. We will not go through the details of the algorithm as it is already described in several previous works. The initial work [131] illustrates how the original MDS algorithms complexity is being lowered from $O(N^3)$ to $O(N^2)$. The proceeding papers [53] [54] describe techniques for improving the implementation efficiency using Java and MPI. For the purpose of this study, we only investigate the algorithm in terms of parallel operations it requires, their complexity and how they can be run in parallel as we try to analyze this algorithm on three execution platforms.

Given a high dimensional distance matrix, the goal of MDS algorithm is to produce a point file with the target dimension. Optionally the algorithm can take a weight matrix. The flow of the algorithm is shown in flow diagram 3.12. MDS algorithm begins by reading two files which contain the pair of matrices. If the data set has $N$ high dimensional data points, the size of these matrices will be $N \times N$ with each cell containing 2 bytes of data in the form of a Short Integer. One matrix file contains the distances and the other contains the weights. Each worker in the computation only reads part of the $N \times N$ matrix. Aside from these two matrices, the algorithm requires a $N \times M$ matrix where $M$ is the target projection dimension. In most cases $M$ will be 3 or 2.

The pseudo code of the algorithm is shown in 1 emphasizing on the important parallel operations. Since this particular implementation of MDS uses deterministic annealing(DA) as an optimization technique, the algorithm has an outer loop involving temperature. The temperature is lowered after each loop until it reaches a configured low value. For each degree reading, the solution to the MDS equation is found as an optimization. For each iteration, the stress is lowered until the difference between consecutive stresses are minimal enough. This involves the last two loops, where the middle loop is over stress and the inner loop is for the optimization computation.

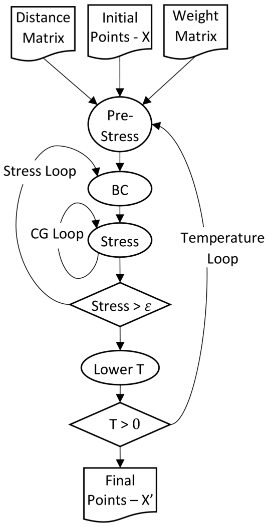


FIGURE 3.12. Multidimensional Scaling algorithm with Deterministic Annealing control flow

The core of the computation involves several matrix multiplications. The remaining computations are mainly serial in nature. These matrix multiplications are present in the two inner loops and largely involve $N \times N$ into $N \times M$ matrices. The algorithm is made parallel around these matrix multiplications.

Apart from the 3 matrices already described, three other matrix of size $N \times M$ are required, called $BC$,$MMr$ and $MMa$ and one array $V$ of size $N$. For this algorithm we can achieve a balanced load across the parallel workers. The two $NN$ matrices read are partitioned rowwise. The $N \times M$ matrix, which contains the projected points, is maintained as a whole in all the workers along with along with $BC$,$MMr$ and $MMa$.

3.4.1.1. *MPI Implementation.* The MPI implementation [1] is a BSP style program written in Java using the OpenMPI Java bindings. Computations in DA-MDS grow $O(N^2)$ and communications $O(N)$. The algorithm involves gather/reduce of large matrices. Because

[1] https://github.com/DSC-SPIDAL/damds

```
$d, w_i, P, n, m$    ▷ Where d - partition of D, w - partition of W, p - Initial points, N - number of points, M - target dimension, K - number of parallel tasks
$P = [N, M]$ point matrix
$b = \frac{N}{K}$
$d = [b, N]$  partition of D matrix
$w = [b, N]$  partition of W matrix
$BC = [N, M]$  NxM matrix
$V = [b]$  Array
$T$  DA temperature
    while $T > 0$ do

    end
    $preStress = calculateStress(d, w, P, T)$ while $stressDiff > delta$ do

    end
    $BC = calculateBC(d, w, P, T)$
$P = conjugateGradient(d, w, P, T, BC)$
$stress = calculateStress(d, w, P, T)$
$stressDiff = preStress - stress$
$preStress = stress$

$T = \alpha T$

function conjugateGradient($d, w, P, T, BC$)
    $MMr = [N, M]$  NxM matrix
    $MMr = calculateMM(w, V, P)$
    $BC = BC - MMr$
    $MMr = BC$
    $rTr = innerProduct(MMr, MMr)$
    $testEnd = rTr * cgThreshold$ while $itr < cgCount$ do

    end
    $MMAp = calculateMM(w, V, BC)$
    $\alpha = rTr / innerProduct(BC, MMAp)$
    $P = P + \alpha * BC$ if $rTr < testEnd$ then

    end
    break

    $MMr = MMr - \alpha * MMAp$
    $temp = innerProduct(MMr, MMr)$
    $\beta = temp / rTr$
    $rTr = temp$
    $BC = MMr + \beta * BC$

    return $P$
end function
function calculateStress($d, w, P, T$)                                  ▷ parallel operation
    calculate partial value of Stress (Double value)
    All Reduce partial stress (sum)
    return $stress$
end function
function calculateBC($d, w, P, T$)                                      ▷ parallel operation
    calculate partial value of BC for $[b, M]$ matrix
    gather the other parts from peers (AllGather)
    return $BC$
end function
function calculateMM($w, V, A$)                                         ▷ parallel operation
    calculate $wA$ using $V$ for diagnol values of $w$
    AllGather $[b, M]$ parts from peers
    return the collected $[N, M]$ matrix
end function
function innerProduct($A, B$)
    return sum of vector dot products
end function
```

**Algorithm 1:** MDS Parallel operations

of this the communication between the workers is very intensive. The MPI implementation uses shared memory explicitly to reduce the cost of collective operations by doing local operation to a node first before doing the global collective operation.

3.4.1.2. *Apache Flink Implementation.* Flink implementation [2] starts with custom input formats for reading the binary matrix files and text based point files required by MDS. Compared to MPI implementation it was much easier to deal with the high level APIs of the input format for handling data partitioning and reading. The parallel operations in the MDS algorithm are mostly implemented using Map, Reduce, and GroupReduce operations in Flink API. Also, the broadcast functionality is used heavily throughout the program.

As of this work, Flink doesn't support nested iterations and only supports single level iterations. Because of this limitation, only the last loop is implemented as a data-flow. To handle the two outer loops, the implementation used separate jobs. As such each job executes the innermost loop and saves the computation to disk. Then the next outer loop starts a new job with the saved state. This is a very inefficient way of implementing the iterations because Flink has to schedule the job, load data and save intermediate data for each of the two outer loops. We noticed that data loading doesn't add much overhead, but scheduling the tasks does.

Flink does not support outputting variables created inside an iteration. This leads to the creation of unified data sets with both the loop variable and output variable in a single data set. Because of this, we had to pass these two data sets to distributed operations that only handle one of them at a time. Flink has no mechanism to load the same data set over

[2] https://github.com/DSC-SPIDAL/flink-apps

all the workers and maintain such structures across all the nodes throughout different data flow operations. So the point matrix is created only in a single node. Every time there is a parallel operation requiring the point matrix, it has to be broadcast to the workers. This applies to other matrices such as BC and MMr as well.

The data flow operators are scheduled by Flink without much control from the user about where to place the data and operators in a cluster. For a complex data flow application such as MDS, it is important to have some control over where the data and operators are placed in the cluster while doing the computation to apply many application-specific optimizations. For simple data flow application, this gives a perfect abstraction and an easy API for users to write efficient programs. But for complex applications, it may not translate well as in the easy case. Also, it is worth noting that, Flink has to model the serial operation in the algorithm according to the data flow style.

Even though Flink programming model can become complex and non-intuitive for complex algorithms, we found that the programs were surprisingly free of programming errors compared to MPI programs. This is primarily due to the functional style of programming API where the user cannot change state.

3.4.1.3. *Spark Implementation.* The programming model [3] of Spark does not allow tasks to communicate with one another, therefore all-to-all collectives are not supported in Spark. As a result, at each iteration the data needs to be collected at the master process and then a new set of tasks has to be created with the new data for the next iteration. This adds two additional overheads to the algorithm. The first is task creation and distribution at

[3] https://github.com/DSC-SPIDAL/damds.spark

each iteration. The other is caused by the additional I/O that needs to be done at each step to perform reduce and broadcast instead of an AllReduce operation. Because of these limitations, the main loops of the MDS algorithm are executed in the driver program and large tasks such as calculateBC, calculateMM and calculateStress are performed as distributed map-reduce tasks. This results in a large number of map reduce phases. The resulting values are then broadcast from the driver program to the cluster before the next iteration is executed. Several RDD's that contain distance data and weight data are cached to improve performance.

Because Spark does not allow in-place changes on RDDs, the algorithm generates intermediate data sets that are not required in the MPI implementation. These intermediate data sets increase the memory usage of the algorithm, which is problematic because memory is a limiting factor when running MDS on very large matrices.

**3.4.2. K-Means algorithm.** K-Means is an efficient and practical classical machine learning algorithm for data clustering. The algorithm maintains K cluster points called centroids. There are many variations of K-means available, but for these experiments, we use the simplest form. As described in the pseudo code 2, parallel algorithm works as follows: the input to the system is *N* points and *K* initial centroids generated randomly. The *N* points are partitioned among the parallel tasks and each parallel processes read the *K* initial centroids. After this step, every task calculates its nearest centroid for each point. The local average of these points for each centroid is used in a global average to get the new centroids position. This is essentially an AllReduce operation with sum.

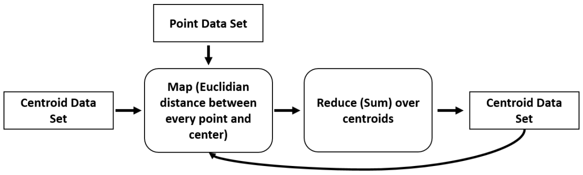

Point Data Set
Centroid Data Set
Map (Euclidian distance between every point and center)
Reduce (Sum) over centroids
Centroid Data Set

FIGURE 3.13. K-Means data flow graph for Flink and Spark

3.4.2.1. *MPI Implementation.* Each parallel MPI implementation reads its partition of the point data set and calculates the nearest centroid for each point. The average of these local values are summed over all the ranks using the MPI AllReduce operation to find the new centroids.

3.4.2.2. *Spark & Flink Implementations.* Spark [4] and Flink [5] K-Means data flow graph is shown in Fig 3.13. At each iteration, a new set of centroids are calculated and fed back to the beginning of the iteration. The algorithm partitions the points into multiple map tasks and uses the full set of centroids in every one. Each map task assigns its points to their nearest centroid. The average of points is reduced (sum) for each centroid to get the new set of centroids, which are broadcast to the next iteration. Spark MLib provides a implementation of K-Means, which is used for evaluations.

In MPI after the AllReduce operation, the centroids are updated within the program in-place. On the other hand, for Spark and Flink these centroids need to be broadcast back to all the tasks that do the nearest neighbor calculation in the next iteration. We can argue that all reduce operation in MPI is equivalent to reduce + broadcast, which is the mechanism used in Flink and Spark. But it is worth noting that AllReduce can be optimized

[4] https://github.com/DSC-SPIDAL/SparkKmeans
[5] https://github.com/DSC-SPIDAL/flink-apps

for greater efficiency than running reduce and broadcast separately by using algorithms such as recursive doubling.

**3.4.3. Terasort.** Sorting terabytes of data is a utility algorithm used by larger machine learning applications. Spark, Flink and MPI implementations of the algorithm use the strategy shown in Fig 3.14 to sort 1 terabyte of data in parallel. At the initial stage, the data is partitioned into equal size chunks among the processes. The processes load these chunks into memory and uses a sample set of data to find an ordered partitioning of the global data set. It does this by sorting the samples gathered from a configurable number of processes. Given that the data is well balanced and the number of sample partitions are reasonably high, this step normally generates a balanced partitioning of the data. In the next step, this global partitioning is used by all the processes to send the data to the correct parallel task. We call this the shuffling phase. The parallel version of the algorithm is described in code 3. After a process receives the data it requires from other processes, it sorts and writes them to a file. This creates a globally sorted data set across the parallel tasks.

```
$W$ number of parallel processes
$P$ point partition
$C$ initial centers
$C_1 = C$ for $i = 1$ to iterations do
    end
    $C_{i+1} = 0$ next set of centers for $p = in\ P$ do
    end
    Calculate the nearest center $i$ for $p$ in $C_i$
add $p$ to the $i$ center in $C_{i+1}$
All reduce sum on $C_{i+1}$ and take average (no of assigned points)
```

**Algorithm 2:** Parallel K-Means algorithm

```
$W$ number of parallel processes
$P$ point partition
$R$ Final points of this process
$s \subseteq P$ take samples from P
$S = \bigcup s_i$ Gather the samples from parallel processes
$S = sort(S)$ Sort the samples gathered
$T \subseteq S$ T is a partition that divides the samples across the parallel processes
    **for** $p = in\ P$ **do**
    **end**
    Calculate the partition $i$ which $p$ belongs to using $T$
Send $p$ to $i$
    **for** $j = 1\ to\ W$ **do**
    **end**
    Receive points from $i$th process and gather them to $R$
sort $R$
save $R$
```

**Algorithm 3:** Terabyte sort parallel algorithm

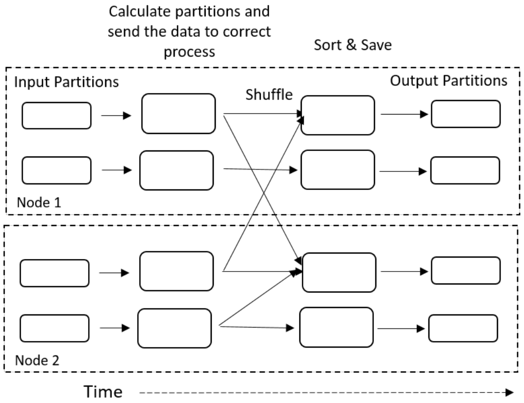


Figure 3.14. Parallel sorting flow. Large boxes show the processes/tasks and small boxes are input files and output files.

The input to the system is according to the format defined for Indy sort by sortbenchmark.org [6]. Each point is 100 bytes long with a 10 bytes key and 90 bytes of random data. The data is sorted using only the key part of the data but final output contains the full 100 bytes for each point.

[6] http://sortbenchmark.org/

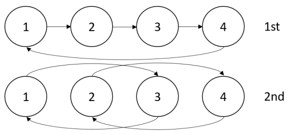


FIGURE 3.15. MPI's data shuffling algorithm. Processes send and receive in a ring topology.

3.4.3.1. *MPI Implementation.* In the MPI implementation [7] the initial sampling is done in memory by choosing a subset of ranks. The samples are gathered to a single process to sort and create the partitioning. This partitioning is then broadcast to all the parallel tasks. The algorithm used send/receives along with iprobe to send the data to correct ranks. A chained send/receive topology in a ring is used, as shown in Fig 3.15. Unlike in Spark and Flink case lot of functionality had to be written in-order to get the algorithm working in a memory limited environment with file based sorting. Also to send the data efficiently the algorithm gathered large enough data set before sending it to the correct process.

3.4.3.2. *Spark and Flink Implementations.* Terasort can be implemented in Spark and Flink [8] using the built-in capabilities of the platform. The data is loaded from HDFS cluster running in the same nodes. The data partitioning is done using the HDFS file system by reading the sample chunks in a single node; sorting them and writing the partitions back to HDFS. The sorting algorithm uses this partitioning to send the data to correct maps. The data is sorted using the sorting functions of Flink and Spark.

For Terasort, there was no apparent difference in the communication or the computation used by MPI, Flink or Spark. The MPI algorithm used send/receive operations which

[7] https://github.com/DSC-SPIDAL/mpi-terasort

[8] https://github.com/DSC-SPIDAL/terasort

is essentially the mechanism used by Spark and Flink. Flink and Spark buffer the data internally while we buffered the data manually for sending using MPI.

### 3.5. Performance of MPI, Spark and Flink

The experiments were run on Juliet, which is an Intel Haswell HPC cluster. Up to 64 nodes were used for the experiments. These nodes have 24 cores (2 sockets x 12 cores each) per node with 128GB of main memory and 56Gbps Infiniband interconnect and 1Gbps dedicated Ethernet connections. MPI experiments were conducted using TCP instead of Infiniband to make it comparable with Spark and Flink. For Flink and Spark experiments, the data was loaded from a HDFS cluster running on the same nodes. For MPI experiments data was copied to local storage in the nodes. We only used up to 20 cores in each node to reduce interference from other processes such as HDFS data nodes, Yarn, Flink TaskManager that run on these. MPI can utilize all the cores as it doesn't include additional processes.

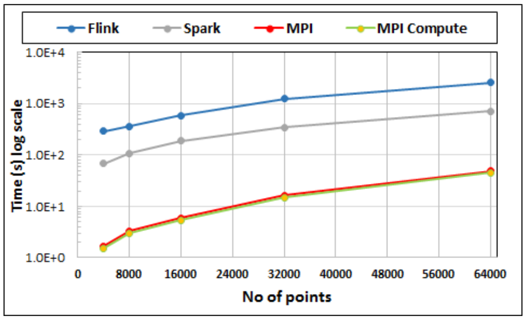


FIGURE 3.16. MDS execution time on 16 nodes with 20 processes in each node with varying number of points

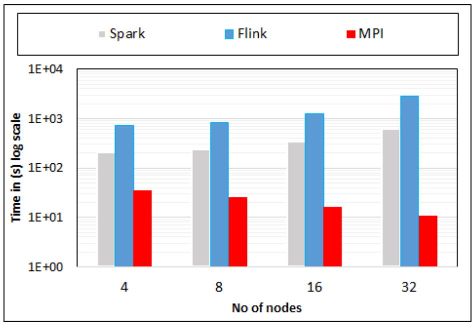


FIGURE 3.17. MDS execution time with 32000 points on varying number of nodes. Each node runs 20 parallel tasks.

For the experiments we ran MDS with a limited number of iterations with 5 temperature loops, 2 stress loops and 8 conjugate gradient iterations to limit the time required for experiments. Fig. 3.16 shows time (displayed in log scale) for running MDS on 16 nodes with 20 parallel tasks in each node. The points are varied from 4000 to 64000. Flink performed very poorly as expected because it doesn't support nested iterations. Spark did considerably well compared to Flink but still proved much slower than MPI. This figure also shows the compute time of the MPI algorithm. It is evident that in MPI the communication overhead and other overheads when running the algorithm in parallel is minimal. Since the same parallel algorithm is implemented in Spark and Flink, this computing time provides a baseline for them as well. The large increase in time in Flink is caused by overheads introduced by the frameworks.

Fig. 3.17 Shown running the MDS algorithm with 32000 points on different number of nodes each having 20 parallel tasks. In MPI the running time decreases as expected while increasing the parallelism, and in Spark and Flink the running time increases with more nodes.

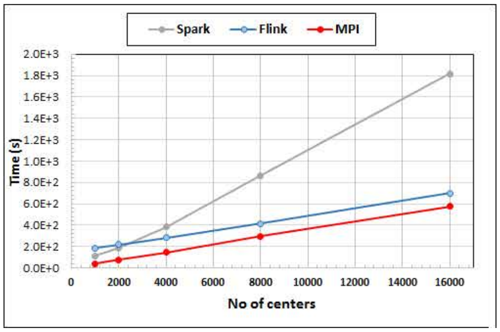


FIGURE 3.18. K-Means execution time on 16 nodes with 20 parallel tasks in each node with 10 million points and varying number of centroids. Each point has 100 attributes.

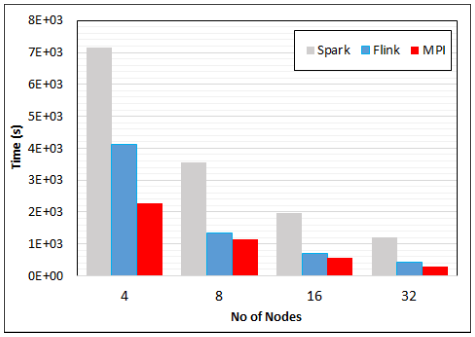


FIGURE 3.19. K-Means execution time on varying number of nodes with 20 processes in each node with 10 million points and 16000 centroids. Each point has 100 attributes.

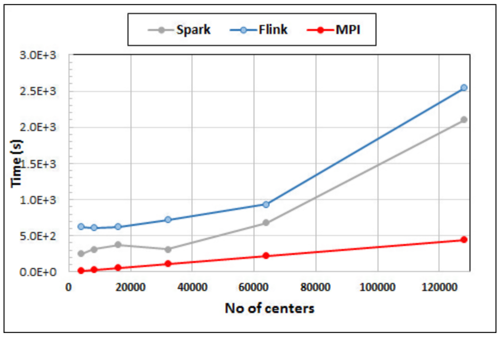


Figure 3.20. K-Means execution time on 8 nodes with 20 processes in each node with 1 million points and varying number of centroids. Each point has 2 attributes.

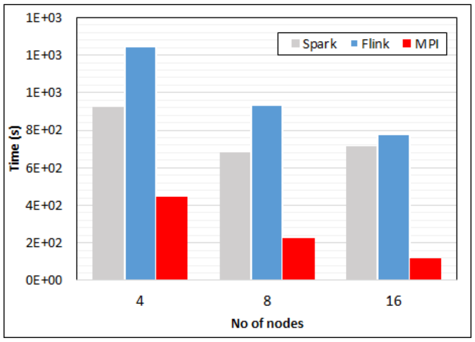


Figure 3.21. K-Means execution time on varying number of nodes with 20 processes in each node with 1 million points and 64000 centroids. Each point has 2 attributes.

We conducted two sets of experiments for the K-Means algorithm. In one experiment we used 10 million points each having 100 attributes (100 dimensions) and 10 iterations to calculate centers. In the next set we used 1 million points with 2 attributes and 100 iterations. The first test requires higher computation and lower communication time compared to the later experiment due to large number of points and small number of iterations.

Fig. 3.18 shows the results of the 10 million point experiment in 16 nodes with varying number of centers. Flink worked comparatively better than Spark in this case and even performed closer to MPI performance. Fig. 3.19 shows the results of having 10 million points with 16000 centers and varying the number of nodes. Unlike in the MDS case, we saw decreases in time in all three frameworks when the parallelism increased. Fig. 3.20 shows the results of the 1 million point experiment in 8 nodes. The performance gap between MPI and Flink is wider in this case. Fig. 3.20 shows the same experiment with 64000 centers and

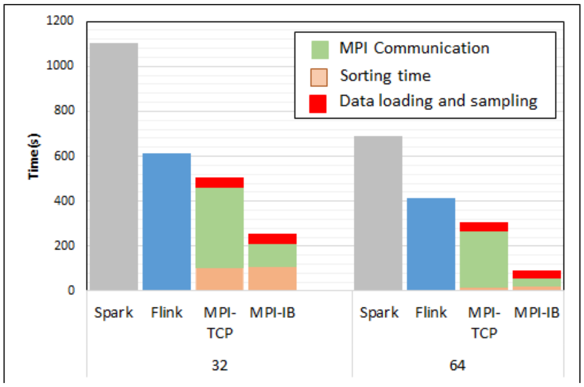


FIGURE 3.22. Terasort execution time in 64 and 32 nodes. Only MPI shows the sorting time and communication time as other two frameworks doesn't provide a viable method to accurately measure them. Sorting time includes data save time. MPI-IB - MPI with Infiniband

varying number of nodes. As the parallelism increased again, MPI performed better but Flink did not scale well.

Fig. 3.22 shows the run time of sorting 1 Terabyte of data in 64 nodes with all three frameworks. Because we are mainly comparing the in-memory performance of the algorithms, we used sufficient nodes so that we can do the sorting in-memory without using the disks. As such the largest time of the algorithm was spent on shuffling the data across multiple processes. In this case all the frameworks performed reasonably well and produced results closer to each other, although the MPI Infiniband results are significantly faster than other approaches. For 32 node test the memory was insufficient and the program had to use the disk to perform the sorting. The MPI-IB test shows that MPI with Infiniband performed best in transferring the data quickly. For 32 node MPI case we noticed that Java garbage collection was affecting the performance. This is an initial investigation, which we will extend later.

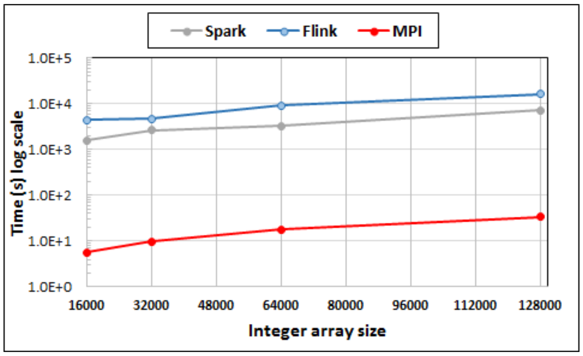


FIGURE 3.23. Reduce operation time with 640 parallel tasks in 32 nodes. Integer array with different sizes is used for the reduce operation.

A micro benchmark was conducted to measure the reduce operation communication time in the three frameworks and its results are shown in Fig. 3.23. The experiment was conducted in 32 nodes with 20 parallel tasks in each node having 640 parallel tasks. Integer array of varying sizes is used for the reduction operation, which was conducted several times in an iteration to calculate the average time. It is clear from the graph that there is a large difference in time between Flink and MPI.

### 3.6. Discussion

MDS is the most complex algorithm among the three algorithms considered here. We have encountered many inefficiencies of Spark and Flink while implementing the algorithm. For Flink the biggest inefficiency was its inability to support nested loops. This leads to a very laborious implementation where we save the intermediate data to file system at each iteration in the outer two loops. Also the way Flink is designed means it needs to read the input files each time it does the iterations, adding to the overhead. The main inefficiency in the Spark MDS implementation was caused due to the lack of all-to-all collective operations.

Using a reduce operations followed by a broadcast operation added couple of overheads to the algorithm.

K-Means showed some interesting characteristics with the three frameworks we used. In parallel K-Means, the communication cost is a direct function of the number of centroids involved and it doesn't depend on the number of points. With increased number of points, the computation time increases, but the communication time remains the same. When using the 10 million data set it was evident that Flink performed close to MPI, and when using the 1 million data set with 100 iterations, the performance gap widened. We concluded that Flink-like frameworks need improvements in communication algorithms used for transferring data to scale to larger nodes when communication requirements are high. Most of the practical clustering problems do not have hundreds of thousands of clusters. This means K-Means can perform equally better in Flink or spark for practical data analytics tasks.

For Terasort, both Flink and MPI displayed comparable performance results. The communication in Terasort involves transferring large amounts of data among the nodes. Ring-like topologies produce the best results for throughput in such bulk transfers since they effectively use the networks in all the nodes at the same time as evident by the good performance in MPI algorithm. Since Flink and Spark do the asynchronous point to point communications, they both saw the similar performance to MPI. Writing the MPI algorithm required time and effort and for such tasks involving transferring large amount of data Flink and Spark can be better choices given the ease of use.

The experiments showed an interesting observation where big data frameworks doesn't scale well for algorithms with high frequency of communicate and compute regions. Also it was evident that Spark performed better than Flink when there was high frequency of communication and computation regions. Algorithms with longer computation and communication times performed well in Flink compared to Spark.

The micro-benchmarks show there is a big difference in communication times for collective communications between MPI and the big data platforms. The primary reason is that, MPI has implemented very efficient collective communication algorithms while Flink and Spark rely on point to point connections.

## 3.7. Summary

This chapter discussed various open source big data frameworks and the performance of three algorithms on MPI, Spark and Flink. The results showed that MPI is best suitable for complex parallel algorithms while lacking abstractions for data processing. Spark and Flink provides the usable APIs for data processing while MPI can outperform them with lot of code developments for each applications.

CHAPTER 4

# Batch Applications

This chapter describes an end-to-end machine learning application where HPC techniques are used to analyze financial markets. In particular, it shows initial results of a study concerning the structure of financial markets viewed as collections of securities generating a time series of 3D values and visualized using WebPlotViz. The study should be viewed as an examination of technologies and approaches and not a definitive study of stock structures. For example, this study looks at only one set of U.S. securities over a 13-year time period with daily values defining the time series. A version of the WebPlotViz visualization software loaded with plots described in this chapter, along with many other plots, is available publicly for reference[1]. WebPlotViz is available either as open source software or as ”Software as a Service” supporting user upload and display of data.

## 4.1. Time series data visualization

TSmap3D is a software toolkit for generating and visualizing high dimensional time series data in 3D space as a sequence of 3D point plots. It consists of a web portal, WebPlotViz for viewing the points in 3D, a highly efficient parallel implementation of Multidimensional Scaling (MDS) [53] for mapping high dimensional data to 3D from large data sets, and a generic workflow with tools to prepare data for processing and visualization. The input to the system is a set of data points taking sequence of values at each time step. The output is

[1] https://spidal-gw.dsc.soic.indiana.edu/

a sequence of 3D point plots that can be portrayed by the visualization software as a continuous moving plot with user interactions enabled for the plots. The timestamp associated with a data point is used to segment the data into time windows, which can be calculated by a method such as sliding window approach. This sequence of data segments define the time steps of the time series. A distance metric between each pair of data points in a data segment is chosen, and the points are projected to 3D by the MDS algorithm such that the distances between the items in new 3D space matches the original distances as closely as possible.

The MDS projection of points to 3D is ambiguous up to an overall rotation, translation and reflection. When MDS is applied to each segment in a series of consecutive data segments for a time series data set, the resulting 3D points in adjacent plots are not aligned by default. We introduce an approach called âĂIJMDS alignmentsâĂİ to find the transformation which best aligns mappings that are neighboring in time. The MDS and alignment techniques are well established, but they are non-trivial to apply in this particular instance with so many details regarding the implementation. Variants of least squares optimization are involved in both the MDS and alignment stages, and in each case, we weight the terms in objective functions by a user defined function. The result of MDS followed by MDS alignments is a time series consisting of a set of vectors, one for each data entity, in 3D (MDS can project to any dimension but 3D is natural for visualization).

The visualization software is an HTML5 viewer (based on Three.js [50]) for general 3D time series, which greatly helps in understanding the behavior of time series data. The user is capable of viewing the time series as a moving plot, while interacting with the plots via 3D rotations, pan and zoom. This allows them to visually inspect changes in the high dimensional relationships between the items over time. The web-based viewer works on any platform with a web browser including handheld devices such as mobile phones.

### 4.2. Data Processing Workflow

The TSmap3D framework executes all the steps, including pre-processing, data analytics and post-processing, in a scripted workflow as shown in Figure 4.1. The initial workflow was developed with MPI(Message Passing Interface) as the key technology to implement the data processing and used a HPC cluster to run the workflow. Later we implemented the same data processing steps in Apache Hadoop [144] for better scalability, data management and fault tolerance. All the programs in the workflow are written using Java, and the integration is achieved using bash scripts. The MDS and Levenberg-Marquardt MDS alignment algorithms are Java-based MPI applications running efficiently in parallel.

Pre-processing steps of TSmap3D mainly focus on data cleansing and preparing the data to suit the input for the MDS algorithm. These data are then run through the MDS algorithm to produce 3D point sets at each time value. There can be high numbers of data segments for a data set depending on the data segmentation approach. For each of these segments, we first put the data into a vector form where $i^{th}$ element represents the value at $i^{th}$ time step in that window. For each vector file, a distance matrix file and weight file are calculated according to user defined functions. The $(i, j)^{th}$ entry of these files contains the pairwise distance or weight between $i$ and $j$ vectors. At the next step MDS is run on these data segments individually which produces the 3D point files.

Each MDS Projection is ambiguous to an overall rotation, reflection and translation, which was addressed by a least squares fit to find the best transformation between all consecutive pairs of projections. This transformation minimizes the distances between the same data points in the two plots. Next the plots are prepared for visualization by TSmap3D. The points in the plots are assigned to clusters according to user defined classifications. Colors and symbols are assigned to clusters, and finally the files are converted to the input format of the visualiser.

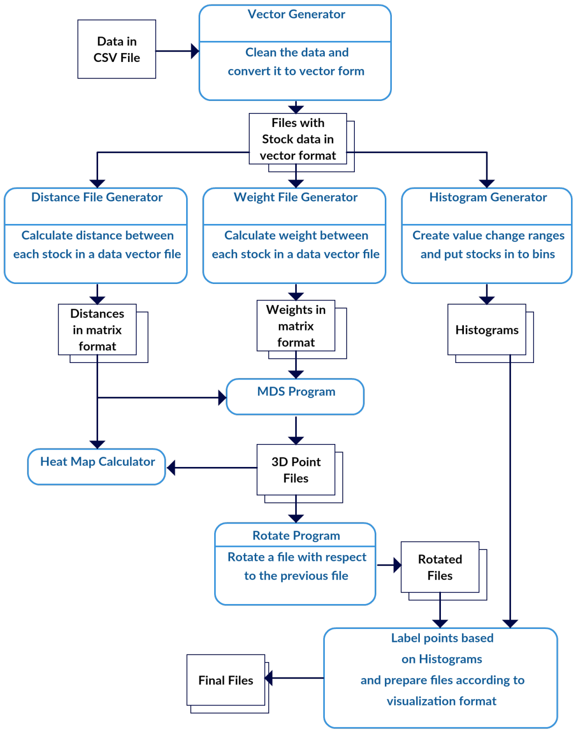

Data in CSV File
Vector Generator
Clean the data and convert it to vector form
Files with Stock data in vector format
Distance File Generator
Calculate distance between each stock in a data vector file
Weight File Generator
Calculate weight between each stock in a data vector file
Histogram Generator
Create value change ranges and put stocks in to bins
Distances in matrix format
Weights in matrix format
Histograms
MDS Program
Heat Map Calculator
3D Point Files
Rotate Program
Rotate a file with respect to the previous file
Rotated Files
Label points based on Histograms and prepare files according to visualization format
Final Files

FIGURE 4.1. Data processing workflow

**4.2.1. Segmenting time series.** TSmap3D mainly uses a sliding window approach to segment the data. Users can choose a time window and sliding time period to generate data sets. The data segmenting starts with time $t_0, s$ and adds a fixed time window $t_w$ to get the first data segment with end time $t_0, e$. Adding a shifting time $t_s$ to the previous start time and end time produces a sliding data segmentation. Adding a shifting time only to the end time produces an accumulating segmentation.

The sliding time approach with a 1 year $t_w$ and 7 day $t_s$ produces overlapping data segments, i.e. 2001-Jan-01/2002-Jan-01 shift to 2001-Jan-08/2002-Jan-08. Data cleansing functions need to run on each data segment because different segments can have different item sets and the data of an item in a given time segment can be incomplete.

**4.2.2. Distance & Weight Calculation.** The MDS algorithm expects a distance matrix $D$, which has the distance between each pair of items in a given data segment. It also expects a weight matrix $W$ with the weights between each pair of items within that data segment. $D = |d_{i,j}|; d_{i,j} = distance(i, j)$ $W = |w_{i,j}|; w_{i,j} = weight(i, j)$ Both distance matrix and weight matrix are stored as files with values converted to a short integer from its natural normalized double format (value between 0 and 1) for compact storage purposes.

To calculate both distance matrix and weight matrix, we put each data item into a vector format from the original input file. A row in this vector file contains values for each time step of the time window. When creating these vector files the data items are cleaned (i.e. remove items without sufficient values for the time period, and fill values that are missing). After this process, the data is now clean and in vector form.

**4.2.3. MDS Program.** The MDS algorithm we use [131, 132] is an efficient weighted implementation of SMACOF [46]. Its runtime complexity is effectively reduced from cubic to quadratic with the use of the iterative conjugate gradient method for dense matrix inversion. Also it implements robust Deterministic Annealing (DA) process to optimize the cost function without being trapped in local optima. Weighting allows us to give more importance to some data items in the final fit. Note our pipeline is set up for general cases where points are not defined in a vector space and it only uses distances and not scalar products. In earlier work, we have learned to prefer MDS to PCA as MDS does optimal non-linear transformations and PCA is only superior over all linear point transformations to the 3D space. Despite this PCA remains a powerful approach and one could use this visualization package with it as well as MDS presented here.

The objective of the MDS algorithm is to minimize the stress $\sigma(X)$ defined in Eq. 1 where $d_{i,j}$ is the distance between points $i$ and $j$ in the original space; $\delta_{i,j}$ is the distance between the two points in mapped space and $w_{i,j}$ is the weight between them.

$$\sigma(X) = \sum_{i,i \leq N} w_{i,j}(d_{i,j} - \delta_{i,j})^2 \tag{1}$$

The mathematical foundation of the parallel algorithm along with deterministic annealing optimization approach is described in great detail in [131]. The parallel MDS implementation uses Java and message passing. OpenMPI provides a Java binding over its native message passing implementation, which is further optimized in MDS to gain significant speedup and scalability on large HPC clusters [53]. The code for MDS is public and is also available at SPIDAL source repository [2].

**4.2.4. MDS alignments.** Because MDS solutions are ambiguous to an overall rotation, reflection and translation, the algorithm produces visually unaligned results for consequent

[2] https://github.com/DSC-SPIDAL/damds.git

windows in time. We experimented with two approaches to rotate and translate the MDS results among such different windows so that they can be viewed as a sequence of images with smooth transitions from one image to other.

In the first approach we generate a common dataset across all the data available and use this as a base to rotate and translate each of the datasets. In the second approach we rotate with respect to the result of the previous time window. This leads to a continuous alignment of the results but requires the data processing to run with overlapping times in a sliding window fashion. The first method is viable for time windows with less overlapping data, because the results of two consecutive time windows can be very different.

The algorithm reflects, rotates, translates and scales one data set to best match the second. It is implemented as a least squares fit to find the best transformation that minimizes sum of squares of difference in positions between the MDS transformed datasets at adjacent time values. We use the well-known reasonably robust LevenbergâĂŞMarquardt [116] minimization technique using multiple starting choices; this increases robustness and allows us to see if a reflection (improper transformation) is needed. Currently the algorithm only uses the reflection, rotation and translation in the final transformation as it needs to preserve the original scale of the generated points. In the future we intend to simplify the algorithm to remove the fitted scaling factor.

### 4.3. WebPlotViz Visualization Software

WebPlotViz is a HTML5-based viewer for large-scale 3D point plot visualizations. It uses Three.js [3] JavaScript library for rendering 3D plots in the browser. Three.js is built

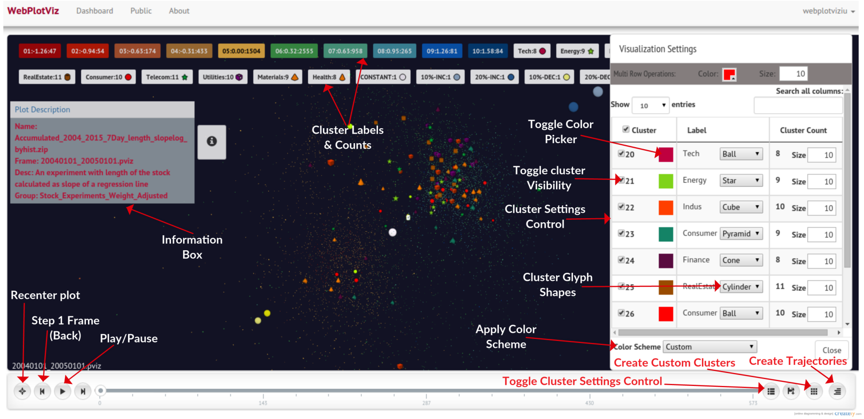
[3] http://threejs.org

using WebGL technology, which allows GPU-accelerated graphics using JavaScript. It enables WebPlotViz to visualize 3D plots consisting of millions of data points seamlessly. WebPlotViz is designed to visualize sequences of time series 3D data frame by frame as a moving plot. The 3D point files are stored along with the metadata in a NoSQL database which allows scalable plot data processing on the server side. The user can use the mouse to interact with the plots displayed, i.e. zoom, rotate and pan. They can also edit and save the loaded plots by changing point sizes and assigning colors and special shapes to points for better visualization. WebplotViz provides features to create custom clusters and define trajectories, as well as supporting single 3D plots such as point plots and trees. A sample plot with 100k points and a tree are shown in Fig 4.3. The online version has many such example plots preloaded.

WebPlotViz also serves as a data repository for storing and sharing the plots along with their metadata, including functions to search and categorize plots stored. The source code of WebPlotViz, data analysis workflow and MDS algorithm are available in the DSC-SPIDAL github repository[4].

**4.3.1. Input.** WebPlotViz has an XML-based input format, as well as text file-based and JSON-based versions. The XML and JSON file formats contain the points and other metadata such as cluster information of a plot. The text file input is a simple input format with only the point information for quick visualization of data sets. A time series input file is a collection of individual plots and an index file specifying the order of the plots to be displayed in the time series.

[4] https://github.com/DSC-SPIDAL



Plot Description

Name: Accumulated_2004_2015_7Day_length_slopelog_byhist.zip
Frame: 20040101_20050101.pviz
Desc: An experiment with length of the stock calculated as slope of a regression line
Group: Stock_Experiments_Weight_Adjusted

Visualization Settings

Multi Row Operations: Color: 🟥 Size: 10

Show 10 entries  Search all columns:

| ☑ Cluster | | Label | | Cluster Count | | |
|---|---|---|---|---|---|---|
| ☑ 20 | 🟥 | Tech | Ball | 8 | Size | 10 |
| ☑ 21 | 🟩 | Energy | Star | 9 | Size | 10 |
| ☑ 22 | 🟧 | Indus | Cube | 10 | Size | 10 |
| ☑ 23 | 🟩 | Consumer | Pyramid | 9 | Size | 10 |
| ☑ 24 | 🟪 | Finance | Cone | 8 | Size | 10 |
| ☑ 25 | 🟫 | RealEstate | Cylinder | 11 | Size | 10 |
| ☑ 26 | 🟥 | Consumer | Ball | 10 | Size | 10 |

Color Scheme: Custom

Close

FIGURE 4.2. WebPlotViz view area with various options

**4.3.2. Grouping & Special Shapes.** Points are grouped into user defined clusters and separate colors are used to distinguish them visually. WebPlotViz offers functions to manage these groups. For example, special shapes can be assigned to the items in a plot through the UI as well as in the plot input files, and the items can also be hidden. Additionally users are provided with the functionality to define custom clusters through the UI, which allows them to revise the clusters provided through the input file. This is an important functionality since when analyzing data through visualization, domain knowledge often allows users to determine groupings and clusters that might not be obvious prior to visualization. Special shapes are assigned to points to clearly highlight them in the plot.

WebPlotViz also provides several color schemes that can be applied to plots in order to mark clusters with unique colors. This can be of help if the original plot data does not contain any color information for clusters or if the existing color scheme does not properly highlight the key features of the plot.

**4.3.3. Trajectories.** Trajectories allow the user to track the movement of a point through time when the time series is played. A trajectory is displayed as a set of line segments

(A) 100k points and clusters

(B) Tree

FIGURE 4.3. Cluster of points and a tree visualized with WebPlotViz

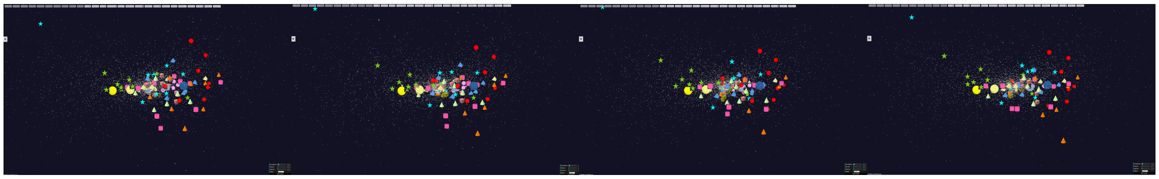


FIGURE 4.4. Four Consecutive Plots of L given by Eq. 4

connecting the movement of an item in time where the points are marked with labels to identify their place.

**4.3.4. Versioning.** WebPlotViz provides version support for plots to allow users to manage multiple perspectives of the same data set. Users can edit the original plot by changing various features such as its rotation, zoom level, point size, cluster color, and custom clusters. They can then save the modified plot as a separate version. Each version is recorded as a change log in the back-end system; when a version is requested, the relevant changes are applied to the original data set and presented to the users. This allows the system to support large quantities of plot data versions without increasing the storage space requirements.

**4.3.5. Data Management.** The software system supports management of uploaded plots by grouping plots to collections and tagging them with user defined keywords. The ability to associate metadata with each plot, which may include experiment descriptions and various settings used during the experiment, enables them to associate important information for future reference and to share their work easily and effectively. The system offers the option to share plots publicly and add comments to existing plots, which permits discussion with others regarding plot details and findings related to the experiments. WebplotViz also provides tag-based and content-based search functionality to enhance usability of the system.

## 4.4. Stock Data Analysis

**4.4.1. Source of Data.** Stock market data are obtained from the âĂIJCenter for Research in Security Prices (CRSP)âĂİ[5] database through the Wharton Research Data Services (WRDS) [6] web interface, which makes daily security prices available to Indiana University students for research. The data can be downloaded as a 'csv' file containing records of each day for each security over a given time period. We have chosen the CRSP data because it is being actively used by the research community, is readily available and free to use. The data includes roughly 6000 to 7000 securities for each annual period after cleaning. The number is not an exact one for all data segments because securities are added/removed from the stock exchange.

This study considered daily stock prices from 2004 January $1^{st}$ to 2015 December $31^{st}$. The paper discusses the examination of changes over one-year windows (the velocity of the stock) and the change over the full time period (the position of the stock). The data

[5] Calculated (or Derived) based on data from Security Files@2016 Center for Research in Security Prices (CRSP), The University of Chicago Booth School of Business.

[6] https://wrds-web.wharton.upenn.edu/wrds

can be considered as high dimensional vectors, in a space – the Security Position Space – with roughly 250 times the number of years of components. We map this space to a new three dimensional space using dimensional reduction for visualization. With a one-year period and a 1-day shift sliding window approach, are 2770 data segments, each generating a separate 3D plot. The Pearson Correlation between the stock vectors is primarily used to calculate distances between securities in the Security Position Space. A single record of the data contains the following attributes and information about a security for a given time (in our case a day): "ID, Date, Symbol, Factor to adjust volume, Factor to adjust price, End of day price, Outstanding stocks".

Price is the closing price or the negative bid/ask average for a trading day. If closing price or the bid/ask average is not available, the price is set to zero. Outstanding stocks is the number of publicly held shares. This study uses the outstanding shares and the price to calculate the market capitalization of the stock. In addition we take two other attributes called âĂIJfactor to adjust priceâĂİ and âĂIJfactor to adjust volumeâĂİ. These are used for determining a stock split as described by the CRSP.

**4.4.2. Data Cleansing.** Here are the data anomalies we found in the data and the steps taken to correct them.

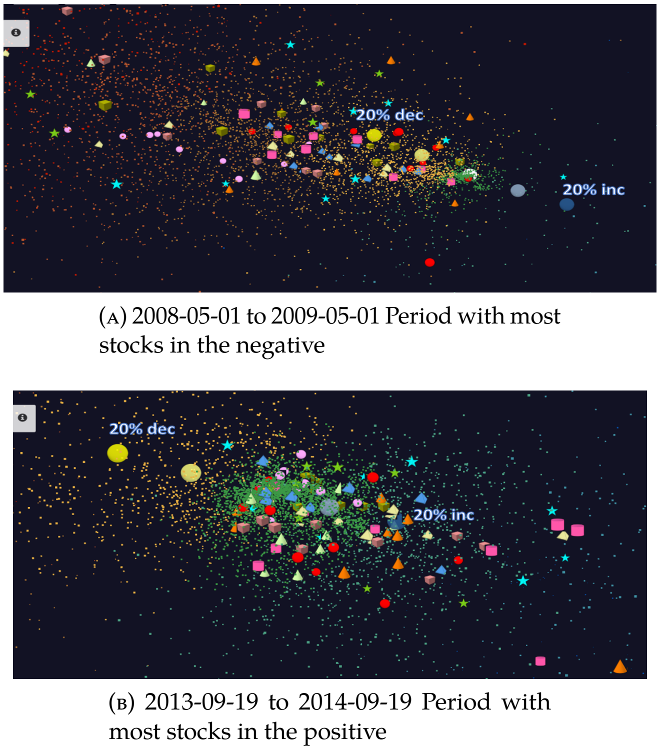
4.4.2.1. *Negative Price Values.* According to CRSP, if the closing price is not available for any given period, the number in the price field is replaced with a bid/ask average. Bid/ask averages have dashes placed in front of them (which we read as negative values). These serve simply to distinguish bid/ask averages from actual closing prices. If neither price nor bid/ask average is available, Price or Bid/Ask Average is set to zero. For values with a dash in front of them, we read this as a negative value and multiply by -1.

4.4.2.2. *Missing Values.* Missing values are indicated by empty attribute values in the data file and are replaced with the previous day's value. If there are more than 5% missing values for a given period we drop the stock from consideration. There are about 900 stocks with more than 5% of missing values per year from 2004 to the end of 2015.

4.4.2.3. *Stock splits.* In the 2004 to end of 2015 period there were 2456 stock splits. The CRSP data provides the split information in the form of two variables called âĂIJFactor to Adjust PriceâĂİ and âĂIJFactor to Adjust VolumeâĂİ. Factor to adjust price is defined as $(s(t) - s(t'))/s(t') = (s(t)/s(t')) - 1$ where $s(t)$ is the number of shares outstanding, $t$ is a date after or on the exit for the split, and $t'$ is a date before the split. We use this variable to adjust the prices of a stock to a uniform scale during the time period we consider by multiplying the stock price after the split with (Factor to Adjust Price + 1). When a stock split happens both Factor to Adjust Volume and Factor to Adjust Price are the same and we can adjust the price with the above method. In very rare cases these two can be different and we ignore such instances. We had only 1 instance of a record where these two factors had different values over the whole period, which we feel justifies our decision to ignore that case.

4.4.2.4. *Duplicated Values.* Although uncommon, there are duplicate values in the data records. The work flow removes the duplicates and uses the earliest record as the correct value.

**4.4.3. Data Segmentation.** The results of this paper are primarily based on two data segmentation approaches: 1. Sliding window with time interval of 1 Year and 7 day or



(A) 2008-05-01 to 2009-05-01 Period with most stocks in the negative

(B) 2013-09-19 to 2014-09-19 Period with most stocks in the positive

FIGURE 4.5. Two sample plots from Stock velocities

1 day shifts (we call the sliding windows the 'velocity approach'); 2. Accumulating times with 1 Year starting window and 1 or 7 day additions. At initial stages we conducted experiments with time window of 1 year and sliding time of one month and 1 year; this gave bad transitions for consecutive plots. For the accumulating approach we start with 1 year period and add 7 days to the previous period to create new segments and gradually expand the time to the period end.

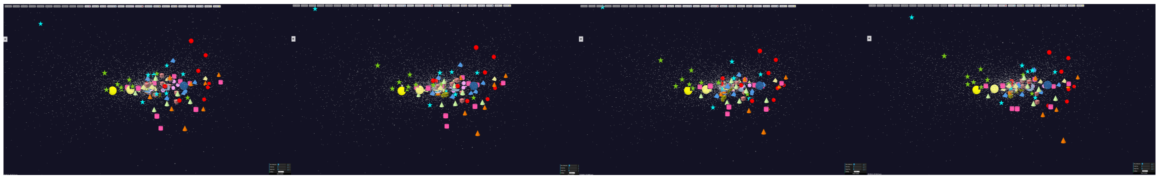


FIGURE 4.6. Four Consecutive Plots of L given by Eq. 4

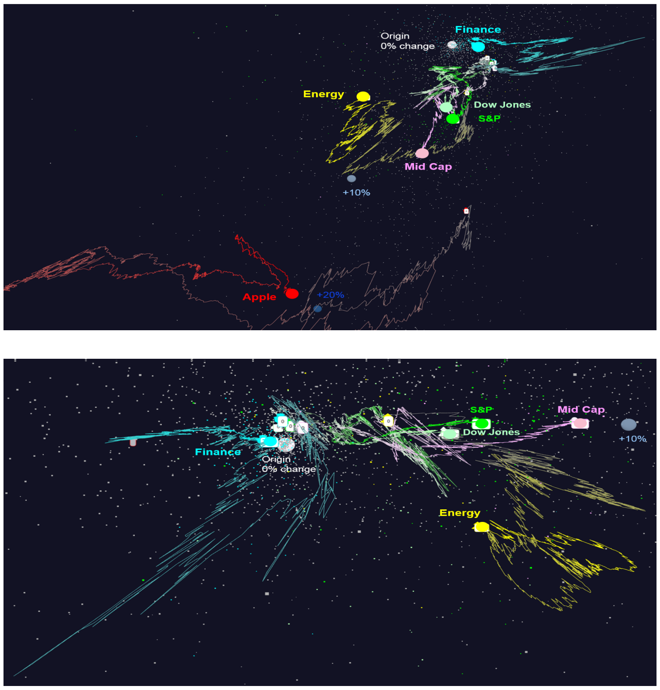


FIGURE 4.7. Two views of leading finance stocks final position trajectories for L given by Eq. 4 with accumulating data segments

**4.4.4. Distance Calculation.** The stocks study uses the Pearson correlation to measure the distance between two stocks. Pearson correlation is a measure of how related two vectors are with values ranging between -1 and 1, -1 indicating a linear opposite relation, 1 indicating a linear positive relation and 0 indicating no linear relation. To incorporate the change of price of a stock over a given time period we include a concept called length. The general formula we use for calculating distance between stocks $S_i$ and $S_j$ is shown below. Each stock has $n$ price points in its vector and $S_{i,k}$ is the $k^{th}$ price value of that stock. $S_j$ is the mean price of the stock during the period.

$$d_{i,j} = \sqrt{(L(i)^2 + L(j)^2 - 2L(i)L(j)c_{i,j})} \tag{2}$$

$$c_{i,j} = \frac{\sum_{k=1}^{n}(s_{i,k} - \bar{s}_i)(s_{j,k} - \bar{s}_j)}{\sqrt{\sum_{k=1}^{n}(s_{i,k} - \bar{s}_i)^2 \sum_{k=1}^{n}(s_{j,k} - \bar{s}_j)^2}} \tag{3}$$

In Eq. 2, $L$ is a function indicating the percentage change of a stock during a given period and $c_{i,j}$ is the correlation between two stocks for a given time window as shown in equation 3. For one set of experiments we have taken $L(S_i) = 1$ for all stocks. For another set of experiments Eq. 4 is used for $L$.

$$L(S_i) = 10|\log(S_{i,n}/S_{i,1})| \tag{4}$$

When calculating the ratio in Eq. 4, we impose a rather arbitrary cut that stock price ratios lie in the region 0.1 to 10. The two different distance calculations with $L = 1$ and $L$ as specified in Eq. 4 lead to different point maps in the final result.

**4.4.5. Weight Calculation.** We calculate weight using the market capitalization of stocks. Limiting the range of market capitalization is done to avoid large companies from dominating the weights and small companies from having negligible importance. We take the dynamic weight of a stock $i$ as specified in Eq. 5 where $M$ is a set with market capitalization of all stocks.

$$w_i = max(\sqrt[4]{max(M) \times .05}, \sqrt[4]{marketcap_i}) \tag{5}$$

Weight $w_{i,j}$ between two stocks $i$ and $j$ is defined by $w_{i,j} = w_i w_j$, where $marketcap$ of a stock is the average market capitalization of that stock during the considered period.

**4.4.6. Reference Stocks.** The system adds five reference or fiducial stocks with known stock values to all the data segments to better visualize and interpret the results.

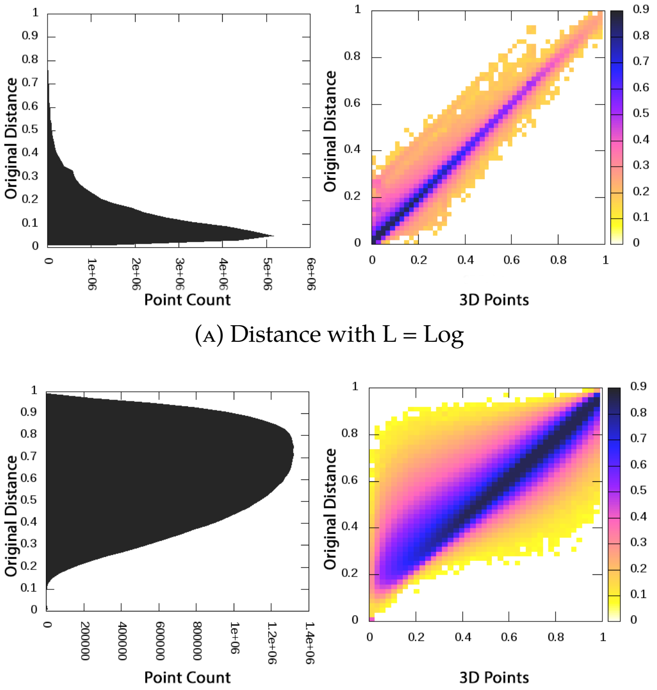
1. A constant stock.
2. Two increasing stocks with different increasing rates of $\alpha = 0.1, 0.2$ with Eq. 6 are used to calculate the price of *i*th day
3. Two decreasing stocks with different decreasing rates of $\alpha = 0.1, 0.2$ with Eq. 7 are used to calculate the price of *i*th day

For the uniform variation in price fiducial stocks, they are taken to have a constant fractional change each day so as to give annual value. For the choice $L = 1$ for all stocks, the two choices of annual change lead to identical structures, so in that case there are essentially just three "fiducialsâĂİ: constant, uniform increase, and uniform decrease. To calculate these fiducial values we use the formula 6 for increasing case and 7 for decreasing case where $\alpha_d = \alpha/250$ and $x_0 = 1$ for $\alpha = .1$ or $\alpha = .2$.

$$x_i = (1 + \alpha d) \times x_{i-1} \tag{6}$$

$$x_i = x_{i-1}/(1 + \alpha d) \tag{7}$$

**4.4.7. Stock Classifications.** We use sector-based classification of stocks and relative price change-based classification to color the stock point in the final plots. This allows us to visualize how stocks behave as groups.



(A) Distance with L = Log

(B) Distance with L = 1

FIGURE 4.8. Heatmap(right) of original distance to projected distance and original distance distribution(left)

4.4.7.1. *Relative Change in Value.* To visualize the distribution of stocks, a histogram-based coloring approach is used. Stocks are put into bins according to the relative change of prices of the corresponding time window. Then for each bin we assign a color and use that to paint the stocks which fall into that bin.

4.4.7.2. *Special Symbols for Chosen Stocks.* On top of the above two classifications, we took the top 10 stocks in each major sector in etf.org [7] and created special groups consisting of 10 stocks each. These stocks consist of large companies dominating in each sector. ETF top stocks and our special fiducial stocks are marked using special symbols (glyphs) which

[7] http://www.etf.com/

allow the user to visualize the stocks clearly in the plot and track their movements through time.

**4.4.8. Stock Experiments.** Several experiments with stock data were conducted with different time shifts and distance functions. This was done on a dedicated cluster using 30 nodes, each having 4 Intel Xeon E7450 CPUs at 2.40GHz with 6 cores, totaling 24 cores per node and 48GB main memory running Red Hat Enterprise Linux Server release 5.11 (Tikanga) OS. For all the experiments the starting time window $t_w$ is a 1-Year period.

(1) 4 experiments with distance calculation from Eq. 4 with time shift of 1 Day and 7 Days for velocity and accumulating segmentation.

(2) Distance calculation with $L(S_i) = 1$ and time shift of 7 Days with velocity segmentation.

The resulting plots of these experiments are publicly available in WebPlotViz[8].

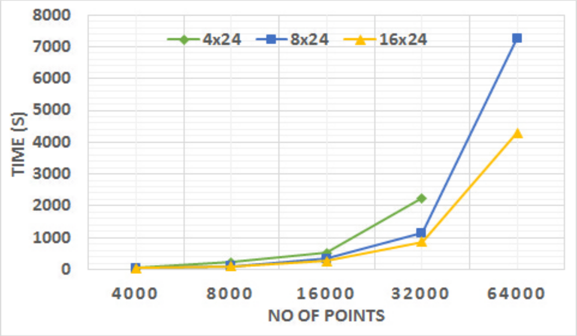
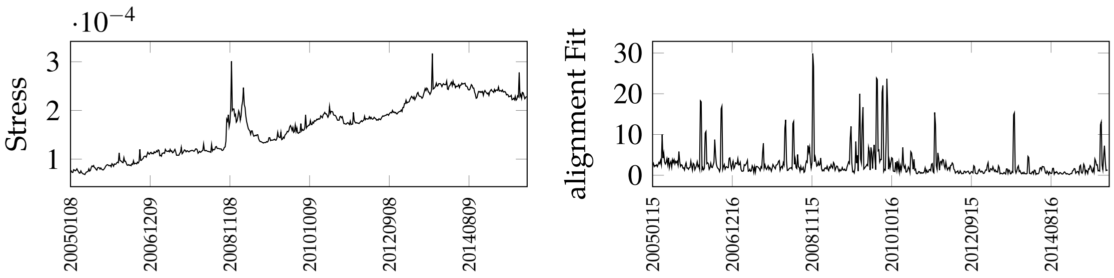


FIGURE 4.9. MDS performance on 4, 8 and 16 nodes with points ranging from 1000 to 64000. Each node has 24 MPI processes. Different lines shows the different parallelisms used.

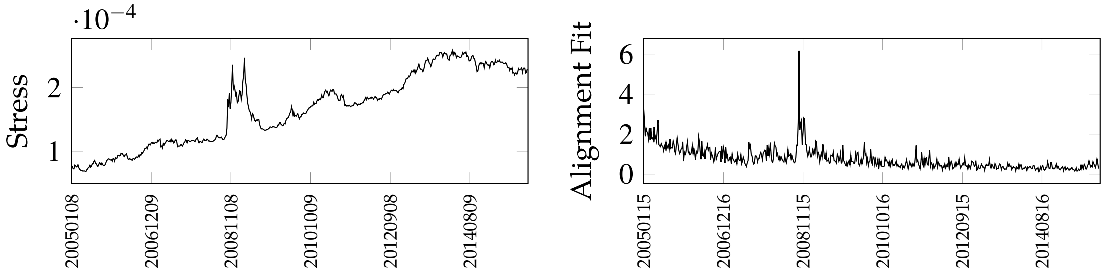
[8] https://spidal-gw.dsc.soic.indiana.edu/public/groupdashboard/Stocks



(A) MDS Final Stress

(B) MDS alignment Fit

Figure 4.10. MDS final Stress and alignment fit when applying independent MDS for consecutive plots, x axis shows the time series



(A) MDS Final Stress

(B) MDS alignment Fit

Figure 4.11. MDS final Stress and alignment fit when initializing MDS with the previous data segment's solution, x axis shows the time series

### 4.5. Discussion

Running time of the MDS algorithm for different number of points utilizing 4, 8 and 16 nodes is shown in Fig 4.9. A node has 24 cores and hence we were running 24 MPI processes in each node. The total parallelism of each test is number of nodes times number of processes. It is evident from the graph that the running time decreases when the parallelism increases at larger point counts hence achieving strong scaling. For four nodes test, it was not possible to run the 64000 point test due to memory limitations. A detailed study of the performance of the algorithm can be found in [53, 54].

| MDS | Pros | Cons |
|---|---|---|
| Run MDS independently for each segment | Can run in parallel for different data segments | Produce local optimal solutions for some data segments randomly |
| Initialize MDS with previous solution | Produces optimal solutions and runs quickly because the algorithm starts near solution | Needs to run sequentially and best suitable for online processing |

Table 4.1. MDS Choices

| Alignment | Pros | Cons |
|---|---|---|
| Respect to common data points for all segments | Can run easily in parallel for different data segments | Doesn't produce the best rotation |
| Respect to previous data points | Produce the best rotations when there are overlapping data and shift is small | Needs to run sequentially and best suitable for online processing |

Table 4.2. MDS Rotation Choices

Four sample consecutive plots from the stock analysis are shown in Figure 4.6 with distance calculated from Eq. 2 with $L$ from 4. Figure 4.5 shows two sample plots with velocity segmentation for different periods in time. The glyphs show those stocks the study has chosen to highlight. The different colors are decided by the histogram coloring approach. Figure 4.7 shows trajectories of a few stocks over the whole time period considered. The trajectories are useful to inspect the relative movement of stocks over time.

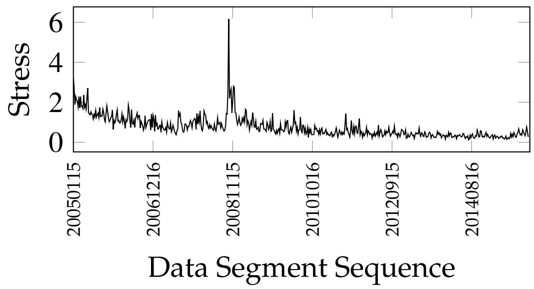


Figure 4.12. MDS alignment fit for 1 Day shift with Accumulating segmentation and Log distances

The accuracy of the MDS projection of original distances calculated using Eq. 2 to 3D is primarily visualized using the heat maps. Fig 4.8b is a sample heat map between 3D points and original distances for $L(S_i) = 1$ and Fig 4.8b is a heat map for $L$ calculated using Eq. 4 case. When the distances between the majority of the 3D points are closer to original distances they create a dense area along the diagonal of the heat map plot; this can be seen as the dark colors along the diagonal of Fig 4.8a and 4.8b. It is evident from sample heat maps 4.8a and 4.8b that the majority of 3D projections are accurate.

We first applied the MDS algorithm for each data segment independently. With this approach, we have observed high alignment errors in some data segments. MDS final stress and alignment error are shown in the Fig 4.10. There were some data segments with high stress values compared to their neighbors. When MDS was run again on these segments the stress reduced to the range of the neighbors. By this experiment, it was evident that even with the deterministic annealing and SMACOF approaches for MDS, it can produce local optimum solutions in rare cases. One way to reduce this effect is to initialize the MDS for a data segment with the solution of the previous data segment. This way the MDS algorithm starts with a solution closer to the global optimum and produced comparably good results as shown in Fig 4.11.

In Fig 4.10 and 4.11, high alignment errors seen in the 2008 stock crash period are primarily due to stock volatility. When two data segments are different from each other, it is expected to create an alignment error. Having a small time shift can definitely reduce these large changes from one plot to another. The alignment errors for a 1-Day shift are shown in Fig 4.12 which are shown to be much less than the 7-Day data. When the time shift is small, data segments increase dramatically and require more computing power. On the other hand, large time shifts produce larger volatility in the sequence of plots.

There were various experiments done with different choices for MDS alignment techniques and MDS algorithm before we came up with an approach that produces good visualizations. Table 4.1 summarizes the options for MDS algorithm and table 4.2 summarizes the options for alignments. We concluded that initializing MDS with the previous solution and using continuous alignments as the best approach among the choices we explored.

Developing a browser-based big data visualization framework is challenging due to several factors. One of the biggest challenges was efficiently transferring data between the server and the browser. WebPlotViz has adopted a highly optimized JSON format for transferring data between browser and server. For very large plots it still takes a considerable amount of time for data transfer. Having large amounts of data in JavaScript is a burden to the browser when buffering the data for time series visualization. Careful handling of data at the browser is required to avoid excessive use of memory.

## 4.6. Summary

With the user-centered rich set of functions for visualization software, as well as the scalable and efficient MDS algorithm and big data work flow, the TSmap3D system provides a comprehensive tool set for generating 3D projections of high dimensional data and analyzing them visually. We discussed how TSmap3D framework is being utilized to apply Multidimensional Scaling to financial data and visualized the changes in relationships among stocks over time. The stock data observed smooth transitions between consecutive plots when using the smallest time shift possible. So we can conclude that using finer-grained time shifts produces the best time series plots in general. The tools and workflow used are generic and can be applied to discover interesting relationships in other time series data sets.

Because the machine learning algorithms are written using MPI, it is hard to integrate the data processing with such algorithm. This is primarily because of lack of integration between these two technologies. HPC is not good at data processing and big data tools are not good at parallel applications. Having big data techniques included in HPC will make such applications much more easy to develop.

CHAPTER 5

# Streaming applications

In this chapter we look at a cloud based robotics application developed using Apache Storm streaming engine. This application look at the challenges in developing cloud based real time applications and how we overcome some of them.

## 5.1. Cloud-based Parallel Implementation of SLAM for Mobile Robots

Many potential IoT applications involve analyzing the rich, large-scale datasets that these devices produce, but analytics algorithms are often computationally expensive, especially when they have to scale to support vast numbers of devices. Cloud services are thus attractive for doing large scale offline and real-time analytics on the data produced in IoT applications. This chapter investigates a computationally-expensive robotics application to showcase a means of achieving complex parallelism for real-time applications in the cloud.

Parallel implementations of real-time robotics algorithms mostly run on multicore machines using threads as the primary parallelization mechanism, but this bounds parallelism by the number of CPU cores and the amount of memory in a single machine. This degree of parallelism is often not enough for computationally expensive algorithms to provide real-time responses. Parallel computations in a distributed environment could give a cost-effective option to provide low latency while also scaling up or down depending on the processing requirements.

Simultaneous Localization and Mapping (SLAM) is an important problem in which a mobile robot tries to estimate both a map of an unknown environment and its position in that environment, given imperfect sensors with measurement error. This problem is computationally challenging and has been studied extensively in the literature. Here we consider a popular algorithm called GMapping, which builds a grid map by combining (noisy) distance measurements from a laser range finder and robot odometer using Rao-Blackwellized Particle Filtering (RBPF)[71, 72]. It is known to work well in practice and has been integrated into robots like TurtleBot [147]. The algorithm is computationally expensive and produces better results if more computational resources are available.

We have implemented GMapping to work in the cloud on top of the IoTCloud platform [90], a framework for transfering data from devices to a cloud computing environment for real-time, scalable data processing. IoTCloud encapsulates data from devices into events and sends them to the cloud, where they are processed using a distributed stream processing framework (DSPF) [91]. In our GMapping implementation, laser scans and odometer readings are sent from the robot as a stream of events to an in-house cloud, where they are processed by SLAM and results are returned to the robot immediately. The algorithm runs in a fully distributed environment where different parts can be run on different machines, taking advantage of parallelism to split up the expensive computations. Our main contribution is to propose a novel framework to compute particle filtering-based algorithms, specifically RBPF SLAM, in a cloud environment to achieve high computation time efficiency.

## 5.2. Background

**5.2.1. IoTCloud framework.** IoTCloud[90][1] is an open source framework developed at Indiana University to connect IoT devices to cloud services. As shown in Figure 5.1, it consists of a set of distributed nodes running close to the devices to gather data, a set of publish-subscribe brokers to relay information to the cloud services, and a distributed stream processing framework (DSPF) coupled with batch processing engines in the cloud to process the data and return (control) information to the IoT devices. Applications execute data analytics at the DSPF layer, achieving streaming real-time processing. The IoTCloud platform uses Apache Storm[23] as the DSPF, RabbitMQ[143] or Kafka[98] as the message broker, and an OpenStack academic cloud[59] (or bare-metal cluster) as the platform. We use a coordination and discovery service based on ZooKeeper[83] to scale the number of devices.

In general, a real-time application running in a DSPF can be modeled as a directed graph with streams defining the edges and processing tasks defining the nodes. A stream is an unbounded sequence of events flowing through the edges of the graph and each such event consists of a chunk of data. The processing tasks at the nodes consume input streams and produce output streams. A DSPF provides the necessary API and infrastructure to develop and execute applications on a cluster of nodes. In Storm these tasks are called Spouts and Bolts. To connect a device to the cloud services, a user develops a gateway application that connects to the device's data stream. Once an application is deployed in an IoTCloud

[1] https://github.com/iotcloud

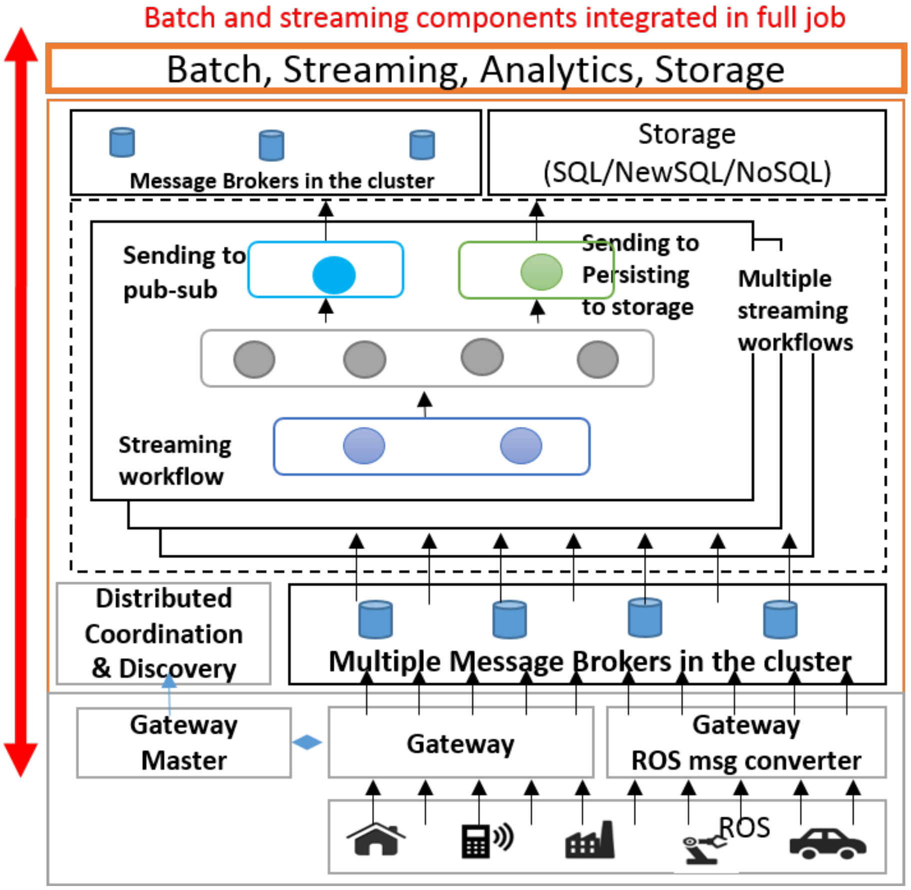


FIGURE 5.1. IoTCloud Architecture

gateway, the cloud applications discover those applications and connect to them for data processing using the discovery service.

**5.2.2. Design of robot applications.** We designed a cloud-based implementation of GMapping for a real robot, the TurtleBot [147] by Willow Garage, using the IOTCloud platform. TurtleBot is an off-the-shelf differential drive robot equipped with a Microsoft Kinect sensor. An overview of the implementation is shown in Figure 5.2. The application that connects to the ROS-based [124] Turtlebot API is deployed in an IoTCloud Gateway running on a desktop machine, where it subscribes to the TurtleBot's laser scans and odometer readings. It converts the ROS messages to a format that suits the cloud application and sends transformed data to the application running in the FutureSystems OpenStack [59] VMs using the message brokering layer. The cloud application generates a map and sends this back to the workstation running the gateway, which saves and publishes it back to ROS for viewing.

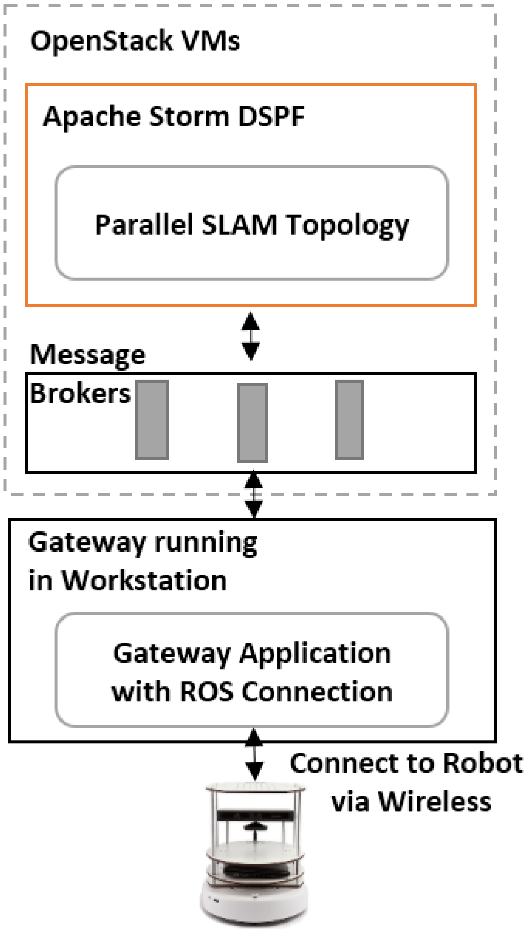

OpenStack VMs
Apache Storm DSPF
Parallel SLAM Topology
Message Brokers
Gateway running in Workstation
Gateway Application with ROS Connection
Connect to Robot via Wireless

FIGURE 5.2. GMapping Robotics application

### 5.2.3. RBPF SLAM Algorithm.

A detailed description of the Rao-Blackwellized particle filter for SLAM is given in [71, 72], but we give a brief overview here. Suppose we have a series of laser readings $z_{1:t} = (z_1, ..., z_t)$ over time, as well as a set of odometer measurements

$u_{1:t-1} = (u_1, ..., u_{t-1})$ from the robot. Our goal is to estimate both a map $m$ of the environment and the trajectory of the robot, $x_{1:t} = (x_1, ..., x_t)$. For any time $t$, we can sample from the posterior probability,

$$p(x_{1:t}, m | z_{1:t}, u_{1:t-1}) = p(x_{1:t} | z_{1:t}, u_{1:t-1}) p(m | x_{1:t}, z_{1:t})$$

by sampling from the first term on the right hand side to produce an estimate of the robot's trajectory given just the observable variables, and then sample from the second term to produce an estimate of the map using that sampled trajectory. The particle filter maintains a set of particles, each including a possible map of the environment and a possible trajectory of the robot, along with a weight which can be thought of as a confidence measure. A standard implementation of the algorithm executes the following steps for each particle $i$ as follows:

(1) Make an initial estimate of the position of the robot at time $t$, using the estimated position at time $t-1$ and odometry measurements, i.e. $x_t^{'i} = x_{t-1}^{'i} \oplus u_{t-1}$ where $\oplus$ is a pose compounding operator. The algorithm also incorporates the motion model of the robot when computing this estimate.

(2) Use the ScanMatching algorithm shown in Algorithm 4 with cutoff of $\infty$ to refine $x_t^{'i}$ using the map $m_{t-1}^i$ and laser reading $z_t$. If the ScanMatching fails, use the previous estimate.

(3) Update the weight of the particle.

(4) The map $m_t^i$ of the particle is updated with the new position $x_t^i$ and $z_t$.

```
input  : pose $u$ and laser reading $z$
output : $bestPose$ and $l$
$steps \leftarrow 0$; $l \leftarrow -\infty$; $bestPose \leftarrow u$; $delta \leftarrow InitDelta$;
$currentL \leftarrow$ likelihood($u$,$z$);
for $i \leftarrow 1$ to $nRefinements$ do
    $delta \leftarrow delta/2$;
    repeat
        $pose \leftarrow bestPose$; $l \leftarrow currentL$;
        for $d \leftarrow 1$ to $K$ do
            $xd \leftarrow$ deterministicsample($pose$,$delta$);
            $localL \leftarrow$ likelihood($xd$,$z$);
            $steps+=1$;
            if $currentL < localL$ then
                $currentL \leftarrow localL$; $bestPose \leftarrow xd$;
            end
        end
    until $l < currentL$ and $steps < cutoff$;
end
```

**Algorithm 4:** Scan Matching

After updating each particle, the algorithm normalizes the weights of all particles based on the total sum of squared weights, and then resamples by drawing particles with replacement with probability proportional to the weights. Resampled particles are used with the next reading. At each reading the algorithm takes the map associated with the particle of highest weight as the correct map. The computation time of the algorithm depends on the number of particles and the number of points in the distance reading. In general the accuracy of the algorithm improves if more particles are used.

### 5.3. Streaming parallel algorithm

We found that RBPF SLAM spends nearly 98% of its computation time on Scan Matching. Because Scan Matching is done for each particle independently, in a distributed environment the particles can be partitioned into different computation nodes and computed in parallel. However, the resampling step requires information about all particles so it needs to be executed serially, after gathering results from the parallel computations. Resampling also removes and duplicates some particles, which means that some particles have to be redistributed to different nodes after resampling.

Our stream workflow of the algorithm is shown in Figure 5.3, implemented as an Apache Storm topology. The topology defines the data flow graph of the application with Java-based task implementations at the nodes and communication links defining the edges. The different components of this workflow run in a cluster of nodes in the cloud. The main tasks of the algorithm are divided into ScanMatcherBolt and ReSamplingBolt. The LaserScanBolt receives data from the robot and sends it to the rest of the application. After computation, results are passed to SendOut bolts which send it back to the robot. If required, data can be saved to persistent storage as well.

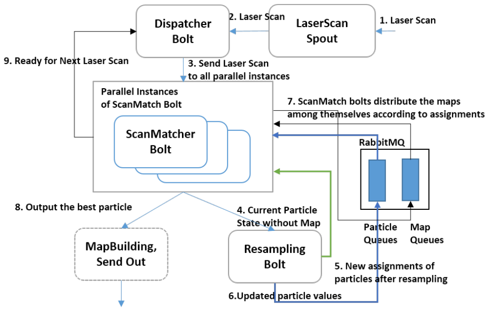

Dispatcher Bolt
2. Laser Scan
LaserScan Spout
1. Laser Scan
9. Ready for Next Laser Scan
3. Send Laser Scan to all parallel instances
Parallel Instances of ScanMatch Bolt
ScanMatcher Bolt
7. ScanMatch bolts distribute the maps among themselves according to assignments
RabbitMQ
8. Output the best particle
4. Current Particle State without Map
Particle Queues
Map Queues
MapBuilding, Send Out
Resampling Bolt
5. New assignments of particles after resampling
6.Updated particle values

FIGURE 5.3. Storm streaming workflow for parallel RBPF SLAM.

A key idea behind our implementation is to distribute the particles across a set of tasks running in parallel. This particle-specific code is encapsulated in the ScanMatcher bolt, so we can control the parallelism of the algorithm by changing the number of ScanMatcher bolt instances. The Resampling bolt must wait until it receives the results of the ScanMatcher bolts. After a resampling happens, the algorithm removes some existing particles and duplicate others, so the assignments of particles to ScanMatcher tasks have to be rearranged. The directed communication required among the parallel ScanMatcher tasks to do the reassignment is not well supported by Apache Storm, so we use an external RabbitMQ message broker. All the data flowing through the various communication channels are in a byte format serialized by Kryo. The steps for a single reading as shown in Figure 5.3 are:

(1) LaserScan spout receives laser and odometer readings via the message broker.

(2) The reading is sent to a Dispatcher, which broadcasts it to the parallel tasks.

(3) Each ScanMatcher task receives the laser reading, updates its assigned particles, and sends the updated values to the Resampling bolt.

(4) After resampling, the Resampling bolt calculates new particle assignments for the ScanMatchers, using the Hungarian algorithm to consider relocation costs. The new particle assignment is broadcast to all the ScanMatchers.

(5) In parallel to Step 4, the Resampling bolt sends the resampled particle values to their new destinations according to the assignment.

(6) After ScanMatchers receive new assignments, they distribute the maps associated with the resampled particles to the correct destinations, using RabbitMQ queues to send messages directly to tasks.

(7) The ScanMatcher with the best particle outputs its values and the map.

(8) ScanMatcher bolts send messages to the dispatcher, indicating their willingness to accept the next reading.

Our implementation exploits the algorithm's ability to lose readings by dropping messages that arrive at a Dispatcher bolt while a computation is ongoing, to avoid memory overflow. Owing to the design of the GMapping algorithm, only a few resampling steps are needed during map building. If resampling does not happen then steps 5 and 6 are omitted. Although an open source serial version of the algorithm in C++ is available through OpenSlam[2], it is not suitable for our platform which requires Java. The algorithm described above was implemented in Java and is available in github[3] along with steps to run the experiments.

### 5.4. Results & Discussion

The goal of our experiments was to verify the correctness and practical feasibility of our approach, as well as to measure its scalability. All experiments ran in FutureSystems [59] OpenStack VMs each having 8GB memory and 4 CPU cores running at 2.8GHz. Our setup had 5 VMs for Apache Storm Workers, 1 VM for RabbitMQ and 1 VM for ZooKeeper and Storm master (Nimbus) node. For all the tests the gateway node was running in another VM in FutureSystems. Each Storm worker instance ran 4 Storm worker processes with 1.5GB of memory allocated. We used the ACES building data set [101] and a small environment generated by the Simbad [82] robot simulator for our experiments. ACES has 180 distance measurements per laser reading and Simbad has 640 measurements per laser reading. For the ACES dataset we used a map of size 80x80m with 0.05 resolution, and for Simbad the map was 30x30m with 0.05 resolution.

A sample result from ACES is shown in Figure 5.10. Since GMapping is a well-studied algorithm, we did not extensively test its accuracy on different datasets, but instead focused

[2] https://www.openslam.org/

[3] https://github.com/iotcloud/iotrobots

TABLE 5.1. Serial average time (in ms) for different datasets and numbers of particles.

| | Particle count | | |
|---|---|---|---|
| Data set | 20 | 60 | 100 |
| Simbad | 987.8 | 2778.7 | 4633.84 |
| Simbad, Cutoff = 140 | 792.86 | 2391.4 | 4008.7 |
| ACES | 180 | 537 | 927.2 |

on the parallel behavior of our implementation. We measured parallel speedup (defined as serial time over parallel time, i.e. $T_s/T_p$) by recording the time required to compute each laser reading and then taking an average. We tested the algorithm with 20, 60 and 100 particles for each dataset. The serial time was measured on a single FutureSystems machine, as shown in Table 5.1, and the parallel times were measured with 4, 8, 12, 16 and 20 parallel tasks.

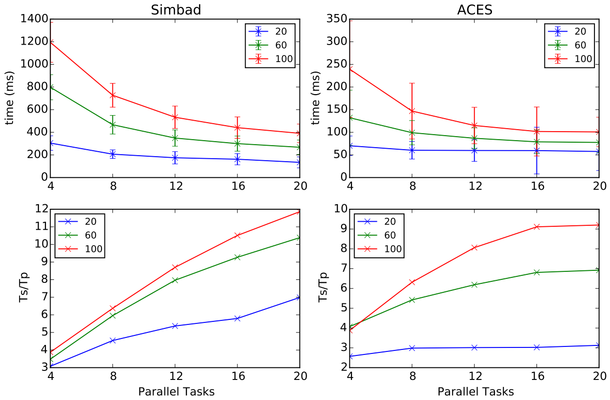
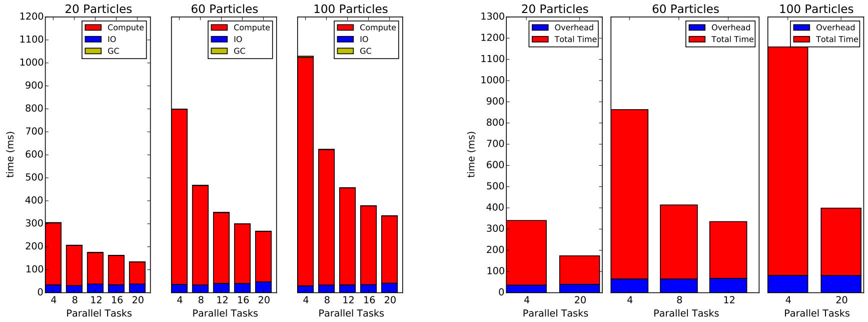


FIGURE 5.4. Parallel behavior of the algorithm for the Simbad dataset with 640 laser readings (left) and the ACES dataset with 180 readings (right). For each dataset, the top graph shows mean times with standard deviations and the bottom graph shows the speedup.

The parallel speedups gained for the ACES and Simbad datasets are shown in Figure 5.4. For ACES, the number of points per reading is relatively low, requiring relatively little computation at the the ScanMatcher bolts, which results in only a modest parallel speedup after 12 particles. On the other hand, Simbad has about 4 times more distance measurements per reading and produces higher speed gains of about a factor of 12 for 20 parallel tasks with 100 particles. While not directly comparable because we test on different datasets with different resources, our parallel speed-ups are significantly higher than those of [69], e.g. up to 12x compared with up to about 2.6x, illustrating the advantage of implementing in a distributed memory cloud versus the single node of [69].



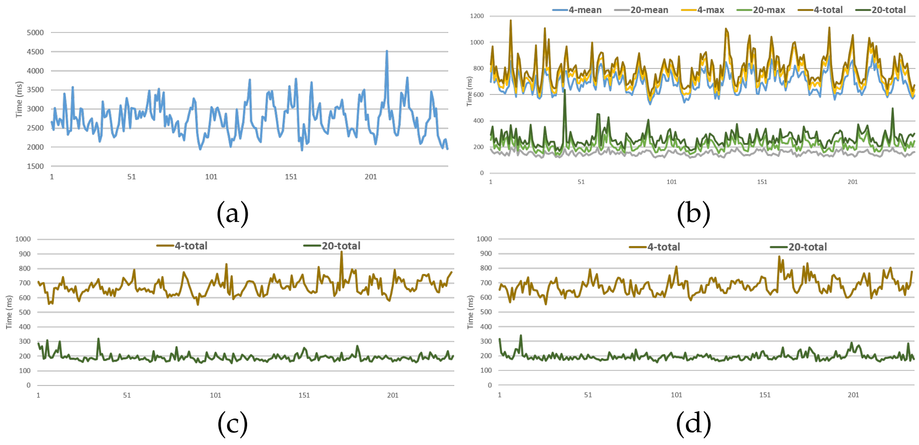
(a)

(b)

Figure 5.5. Overhead on the Simbad dataset: (a) I/O, garbage collection, and Compute time. (b) Overhead of imbalanced parallel computation.

However, ideally the parallel speedup should be close to 20 with 20 parallel tasks, so we investigated factors that could be limiting the speedup. Figure 5.5a shows I/O, garbage collection, and computation times for different parallel tasks and particle sizes. The main culprit for limiting the parallel speedup appears to be I/O: when the number of parallel tasks increases, the compute time decreases, but because of I/O overhead the speedup also decreases. The average garbage collection time was negligible, although we have seen instances where it increases the individual computation times. Additionally, the resampling step of the algorithm is done serially, although since this is relatively inexpensive compared with Scan Matching, it is not a significant source of speedup loss.

Another factor that affects parallel speedup is the difference in computation times among parallel tasks. Assume we have $n$ ScanMatcher tasks taking $t_1, \ldots, t_m, \ldots, t_n$ seconds, where $t_m$ is the maximum among these times. In the serial case, the total time for ScanMatching is $T_s = t_1 + \ldots + t_m + \ldots + t_n$, while for the parallel case it is at least $t_m$ because Resampling has to wait for all parallel tasks to complete. The overhead introduced because of this time imbalance is $t_{overhead} = t_m - T_s/n$, which is 0 in the ideal case when all tasks take the same time. Figure 5.5b shows the average overhead for the Simbad dataset compared with the total time. The average overhead remained almost constant while the total time decreased with more parallel tasks, restricting the speedup.



(a)

(b)

(c)

(d)

FIGURE 5.6. Computation time across individual readings, for Simbad and 60 particles. (a) Serial time, (b) Parallel without cutoff, (c, d) Two trials of parallel with cutoff at 140 steps

To further investigate the behavior of the algorithm, we plotted the computation times for each reading, as shown in Figures 5.6 (a) and (b). There are high peaks in the individual times in both serial and parallel algorithms. This is caused by the while loop ending in line 5 of Algorithm 4, which can execute an arbitrary number of times if the cutoff is ∞. We have observed a mean of about 150 and standard deviation around 50 for number steps executed by the ScanMatching algorithm for Simbad, although sometimes it is as much as 2 to 3 times the average. This is especially problematic for the parallel case because even one particle can significantly increase the response time. An advantage of nondeterministic particle-based algorithms is that if there are a sufficiently large number of particles, cutting off the optimization for a few of them prematurely does not typically affect the results, and we can easily increase the number of particles if needed to compensate for these premature cutoffs. We also observed that these large numbers of steps typically occur at later refinements with small delta values, where the corrections gained by executing many iterations is usually small.

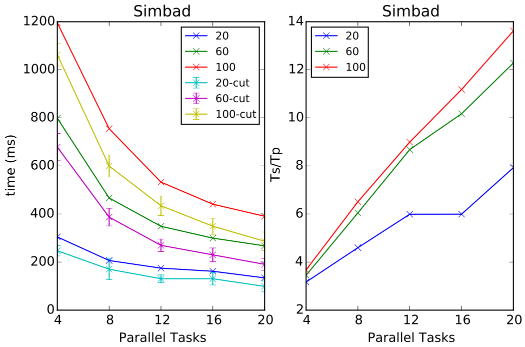


FIGURE 5.7. Average time and speedup for Simbad with cutoff at 140

We thus changed the original algorithm shown in Algorithm 4 to have a configurable cutoff for the number of steps and performed experiments by setting the maximum number of steps to 140, which is close to the empirical average. We found that maps built by this modified algorithm were of comparable quality to the originals. The resulting time variations for two tests are shown in Figure 5.6 (c,d). Here we no longer see as many large peaks as in Figure 5.6 (a), and the remaining peaks are due to minor garbage collection. Figure 5.7 shows the average time reduction and speedup after the cutoff. As expected, we see an improvement in speedup as well, because the parallel overhead is now reduced as shown in Figure 5.9 after cutoff at 140 steps. This shows that cutoff is an important configuration parameter that can be tuned to balance performance and correctness.

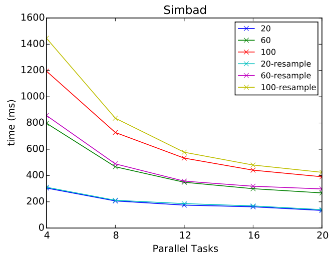


FIGURE 5.8. Resampling overhead with Simbad

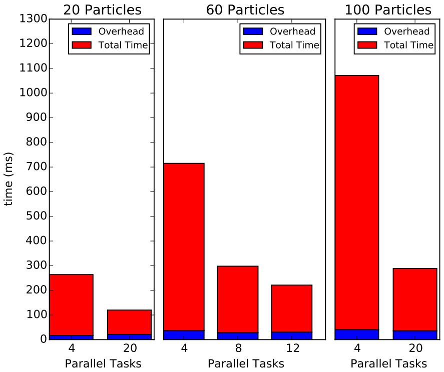


FIGURE 5.9. Parallel overhead with cutoff at 140

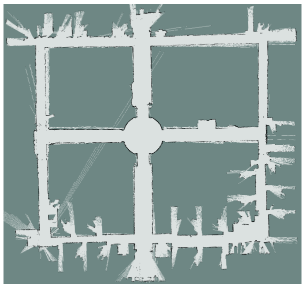


FIGURE 5.10. Map of ACES

Figure 5.8 presents the difference in calculations when we conducted the resampling step for every reading with the Simbad dataset. When the number of particles is high, the overhead is large. When we have more parallel workers, the map distribution happens simultaneously using more nodes, which reduces the I/O time. The original serial algorithm for Turtlebot runs every 5 seconds. Because the parallel algorithm runs much faster than the serial version, it can be used to build a map for a fast-moving robot. In particle filtering-based methods, the time required for the computation increases with the number of particles. By distributing the particles across machines, an application can utilize a high number of particles, improving the accuracy of the algorithm.

## 5.5. Summary

We have shown how to offload robotics data processing to the cloud through a generic real-time parallel computation framework, and our results show significant performance gains. Because the algorithm runs on a distributed cloud, it has access to potentially unbounded memory and CPU power. This allows the system to scale well, and for example could build maps in large, complex environments needing a large number of particles or dense laser readings.

There are many possible enhancements to our system. We addressed the problem of imbalances in particle computation times by simply discarding particles that exceed a hard computation limit, which works well for this particular algorithm but may not for others. Also we have observed fluctuations in processing time caused by virtualization, multi-stream interference and garbage collection. In the future we would like to address these fluctuations with a generic approach such as running duplicate computation tasks. Also, we have observed that result broadcast and gathering in the streaming tasks takes considerable time, so reducing I/O would also significantly improve performance.

Reducing programming complexity is also an interesting direction. Modern distributed stream processing engines expose low-level APIs, making developing intricate IoT applications quite complex. Future work should propose higher-level APIs to handle complex interactions by abstracting out the details. Distributing state between parallel workers currently requires a third node, such as an external broker or a streaming task acting as an intermediary. A group communication API between the parallel tasks would be a worthy addition to DSPF. Extending our work to abstract out a generic API to quickly develop any particle filtering algorithm would also be interesting.

CHAPTER 6

# HPC integration to Big Data

This chapter describes two efforts to integrate HPC techniques into Apache Streaming engines to enhance their performance.

## 6.1. HPC techniques with Apache Storm

Real-time data processing at scale in cloud-based large data centers is challenging due to their strict latency requirements and distributed nature of applications. Modern distributed stream processing frameworks (DSPF) such as Apache Storm [140] provide an effective platform for real-time large scale applications. A parallel real-time distributed streaming algorithm implementing simultaneous localization and mapping (SLAM) [93] for mobile robots in the cloud is a good example of an application requiring low latency parallel processing with strict guarantees. Current DSPFs [92] is designed to cater to traditional event processing tasks such as extract transformation and load (ETL) pipelines, counting, frequent itemset mining, windowed aggregations and joins or pattern detection. The above-mentioned novel applications with strict real-time guarantees demand low-latency synchronous and asynchronous parallel processing of events, which is not a fully explored area in DSPFs.

The work in real-time applications in robotics [77, 93] and research into high performance computing on streaming [92] highlighted the fact that there are opportunities for further enhancements in distributed streaming systems. Particularly in the areas such as low-latency and efficient communication, scheduling of streaming tasks for predictable performance, and high-level programming abstractions. This paper focuses on efficient communication in a DSPF and looks at how communication infrastructure of a DSPF can be improved to achieve low latency. Previous work has found that inefficient communication for data distribution operations, such as broadcasting in current DSPF implementations, are limiting the performance of parallel applications when the parallelism of the processing increases. Also, it was identified that communications among processes inside a node can be improved significantly using shared memory approaches.

A Distributed streaming application is generally represented as a graph where nodes are streaming operators and edges are communication links. Data flow occurs through the edges of the graph as streams of events. The operators at the nodes consume these event streams and produce output streams. With naive implementation, a collective operation such as broadcast is done through separate communication links (edges) from the source to each target serially. As shown in various areas such as MPI [123] and computer networks [146] these communications can be made efficient by modeling them based on data structures such as trees.

Apache Storm [140] is an open source distributed stream processing engine developed for large-scale stream processing. Its processing model closely resembles the graph-based data flow model that was described earlier. It is capable of low latency stream processing and has been used for real-time applications [77, 90, 93]. This paper presents the results of the improvements that were made to Apache Storm by implementing several broadcasting algorithms, as well as reducing communication overhead using shared memory. The underlying algorithms used for broadcasting are the flat tree, binary tree, and ring. Storm utilizes both process-based and thread-based parallel execution of stream operators. The tasks of a Storm streaming application run in different processes in different nodes. Tasks running in the same process can use the memory to communicate while those running in separate processes utilize networks. The communication algorithms are optimized in consideration of the task locality to improve network and inter-process communications. To test the system, a simple stream processing application is used with minimal processing at the parallel tasks for evaluating the behavior of the broadcasting algorithms with different data sizes and tasks.

In MPI, operations requiring more than simple P2P communications are termed collective operations. The collective communications are optimized for HPC applications [21, 123] in technologies such as MPI. There are many such collective operations available including *Gather*, *AllGather*, *Reduce*, *AllReduce*, *Broadcast*, *Scatter*, etc. Multiple communication algorithms apply to each of these, and their suitability depends on the message size, cluster size, number of processes and networks. There has been much discussion of various techniques to choose the best algorithm for a given application and hardware configuration [56, 57]. Recently collective communications are being introduced to batch processing big data solutions requiring rich communication [74, 155]. These improvements have greatly enhanced the performance of applications implemented on top of these platforms. High performance interconnects like RDMA [103] are being studied for MPI applications as well as big data platforms such as Hadoop [108] and Spark [110] to further reduce the communication among the processes. Also shared memory communications [70] are being used for interprocess communications in MPI applications. This paper investigates how to bring these improvements to Distributed Stream Processing applications.

**6.1.1. Communication.** The communication strategy between two nodes in the user graph is called stream grouping. A stream grouping defines how the messages are distributed from one set of parallel tasks of the first processing element to another set of parallel tasks of the second processing element. As an example, in Shuffle grouping, messages are sent from each task of the first component to all the other tasks of the second component in a round-robin fashion, thereby achieving load balancing of the events for the distributed tasks. Other communication strategies include patterns like key-based distributions, broadcasting and direct message sending.

The default implementation of Apache Storm uses a TCP-based binary protocol for sending messages between tasks. Connections are established between worker processes which carry messages with a destination task ID. Workers then use this in determining the correct task to deliver their message. Storm uses Kryo Java object serialization to create byte messages from the user objects that need to be transferred. Communication between the tasks running in the same worker happens through the process memory via object references; no serialization or deserialization is involved.

This design implies that there is a single TCP port at which every worker is listening for incoming messages. These ports are known across the cluster and workers connect to each other using these ports. The design reduces the number of connections required for an application. With the default implementation, the tasks within the nodes also communicate using TCP, which can be efficient because of the loopback adapter but can be further improved using shared memory-based communications.

### 6.2. Broadcast

Broadcasting is a widely used message distribution strategy in Apache Storm. Broadcasting involves a task instance sending a message to all the tasks of another node in the user graph. When this is applied to a continuous stream of messages, the broadcasting happens continuously for each message. Let's assume propagation delay of $l_t$ for TCP and $l_s$ for processes inside a node; transmission delay of $m_t$ for TCP and $m_s$ for processes inside a node and there are n nodes participating in broadcast each having w workers. The default implementation of Storm serially sends the same message to each task as a separate message and this method is inefficient due to the following reasons: 1. Max latency is at least $m_s \times n + l_t$. 2. If the message size is $M$ it takes at least $M \times n$ network bandwidth of the broadcasting node. 3. A worker can run multiple tasks and the same message is sent to the worker multiple times.

The three algorithms developed reduce deficiencies 1 and 2 mentioned above to varying degrees and eliminate 3 completely. They use a tree model to arrange the edges of the broadcast part of execution graph and take advantage of the fact that communication among the processes in a single computer is less expensive compared to inter machine communications. The workers are mapped to nodes of the tree instead of individual tasks because communication cost is zero between the tasks running in the same worker due to in-memory message transfers. To preserve the worker locality within a node, a machine participating in the broadcast operation uses at most one incoming message stream and one outgoing message stream for the broadcasting operation. This rule reduces the network communication drastically because it minimizes inter-machine communications and maximizes intra-node communications. Figure 6.1 shows an example message distribution for the broadcast operation with the three algorithms with 2 machines each running 4 workers.

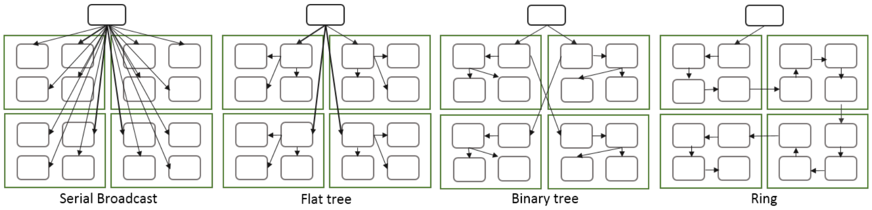


FIGURE 6.1. Example broadcasting communications for each algorithm in a 4 node cluster with each machine having 4 workers. The outer green boxes show cluster machines and inner small boxes show workers. The top box displays the broadcasting worker and arrows illustrate the communication among the workers

**6.2.1. Flat tree.** Flat tree algorithm broadcasting has a root level branching factor equal to the number of nodes with active topology workers participating in the broadcasting operation. This means the broadcast nodes first send the message to a worker in each node. After the designated worker in the node receives the message, it is distributed to workers running in that node sequentially or using a binary tree. The max latency observed by the workers will be $n \times m_t + l_t + (w-1)m_s + l_s$ for flat tree distribution inside the node.

**6.2.2. Binary Tree.** Binary tree algorithms broadcast to two workers in the first level and these two workers broadcast to another four workers. When picking the first two workers the algorithm always tries to use two workers in two nodes. The worker receiving the messages from upper machine broadcast to a worker inside its machine and another worker on another machine if there are such workers and machines available. The algorithm gives high priority to tasks in the same worker of the broadcast task and tasks in other workers in the same node in that order. So the tree always expands through those workers in the first levels if there are such workers. The max latency will be of the order $log_2 n \times (l_t + m_t) + log_2 w(l_s + 2m_s)$.

**6.2.3. Ring.** As shown in Figure 6.1, the ring starts at the broadcast worker and goes through all the workers participating in the broadcast. It always connects the workers of a single node first before reaching to the next node. Two variants of the ring algorithm are used. In the first variant, the ring starts from one node and ends at the last node connecting all the tasks. In the second case, two communications are started from the root and each of these connects half of the nodes in the broadcast creating a bidirectional ring. The broadcast takes the task locality into account and always starts the ring from tasks running in the same worker as broadcast tasks and then connects the workers in the same machine. The ring topology used with a stream of messages becomes a communication pipeline as incoming messages are routed through the workers while other messages transmitted before are still going through the worker. The max latency will be $(w-1)(l_s + m_s)n + (n-1)(l_t + m_t)n$ for full ring and half of this for bidirectional ring.

### 6.3. Shared Memory Communications

DSPFs use TCP messages to communicate within processes of a machine. Apache Storm, for example, may create up to $(w-1) * w$ TCP connections to communicate among tasks running with $w$ workers in a node. Such communication poses a significant bottleneck considering the fact that they occur within a single node. As an efficient alternative, this work implements a Java shared memory maps-based communication between intra-node workers.

While Java has built-in support for memory maps, it does not have consistency guarantees to be used as an inter-process communication technique. Therefore, we implement a custom multiple writer-singer-reader styled memory map-based queuing system. This implementation is safe to use across multiple platforms and it is possible to use either the file system or main memory to persist messages. In Linux systems, the special $tmpfs$ directory, usually mounted as */dev/shm*, points to the main memory, which is more efficient than using a regular file in this queue implementation.

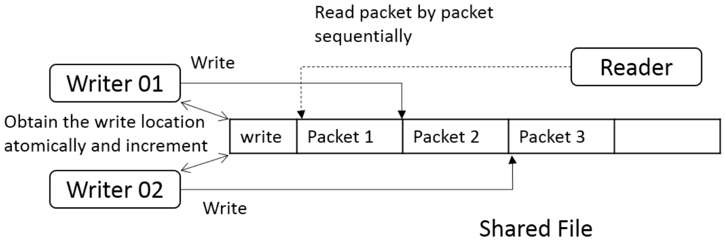


Figure 6.2. The structure of the shared memory file

| Fields | ID | No of Packets | Packet No | Dest Task | Content Length | Source Task | Stream Length | Stream | Content |
|---|---|---|---|---|---|---|---|---|---|
| Bytes | 16 | 4 | 4 | 4 | 4 | 4 | 4 | Variable | Variable |

Packet Structure

Figure 6.3. The structure of the data packet sent through shared memory

In the shared memory implementation each Storm worker has a memory mapped file allocated to it with a single reader. This reader continuously reads its file for new messages written by workers. The file is written and read using fixed size message chunks, meaning a message will be broken into multiple parts during communication. Also, these chunks include a unique ID and a sequence number to guarantee correctness and to match messages with the corresponding writer as it is possible for messages to arrive in a mixed order due to multiple writers writing to the same file. Figure 6.3 shows the structure of a packet. Each packet contains a UUID to identify which message it belongs to, the source task, destination task, total number of packets and the current packet number. A multiple parts messaging is used because the file size is fixed and the message sizes do not always fit the remaining file size. Also, multiple writers can write to the file without waiting for a single writer to finish with a large message. The implementation was started with open source shared memory bus [1] and deviated from it because of the continuous streaming requirements.

The structure of a single file is shown in Figure 6.2. To achieve multiple writers, a shared long integer is stored at the beginning of the file to mark the used space. When a writer intends to write a packet, it incrementally alters this field by the packet size atomically and gets the new address. Then it writes the packet to the allocated space. Because of the atomic increment of the value, multiple writers can write to the same file at the same time. When the file limit is reached the writer allocates a new file and starts writing to that. While allocating a new file, the writers acquire a lock using another shared file to prevent multiple writers from allocating the same file. With the current design of Storm, a separate reader has to poll the file for messages and these are put into a queue where the processing threads pick the message out. In the future, these two can be combined into one thread for reading and processing. The source code of the improvements is available in GitHub repository [2].

[1] https://github.com/caplogic/Mappedbus

[2] https://github.com/iotcloud/jstorm

## 6.4. Experiment Setup

The experiments are conducted on a dedicated cluster using 10 nodes, each having 4 Intel Xeon E7450 CPUs at 2.40GHz with 6 cores, totaling 24 cores per node; and 48GB main memory running Red Hat Enterprise Linux Server release 5.11 (Tikanga) OS. The nodes are connected using a 1GB/s standard Ethernet connection. The Nimbus and Zookeeper were running in one node, 8 Supervisors were running in 8 nodes each with 4 workers, and RabbitMQ [143] was running on another node. Each worker was configured to have 2GB of memory. This configuration creates 32 workers in the 8 nodes. Our experiments used 32 workers in the system.

Figure 6.4 shows the topology used for experiments, which includes a receive spout (S), broadcasting bolt (B), set of worker bolts (W) and a gather bolt (G). The experiments were set up to measure the latency and throughput of the application when message size and number of parallel tasks change. To measure the latency, a client sends a message to the spout R using RabbitMQ including the message generation time. Then this message is broadcast to N worker bolts W by broadcast bolt B and they send these messages to Gather bolt G. Finally the message is transmitted back to the client with the original timestamp and round trip time is calculated. Using this setup we avoid time measurements across different machines which can be inaccurate due to time skew. To measure the number of messages transferred per second using the broadcast operation, bolt B generated a set of messages and sent them through W to G at which point G measured the time it takes to completely receive the messages.

Two sets of experiments are conducted with TCP-based and shared memory-based messaging. All the latency results are for round trip latency, and theoretically a constant including the message broker overhead should be subtracted from latency to get the true time, but since this is constant for all the experiments and because we are comparing the results of previous and new implementations, we did not consider this cost.

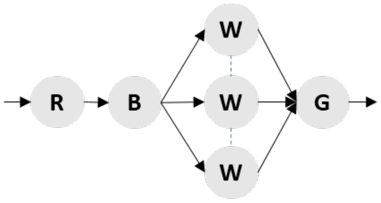


FIGURE 6.4. Storm application graph

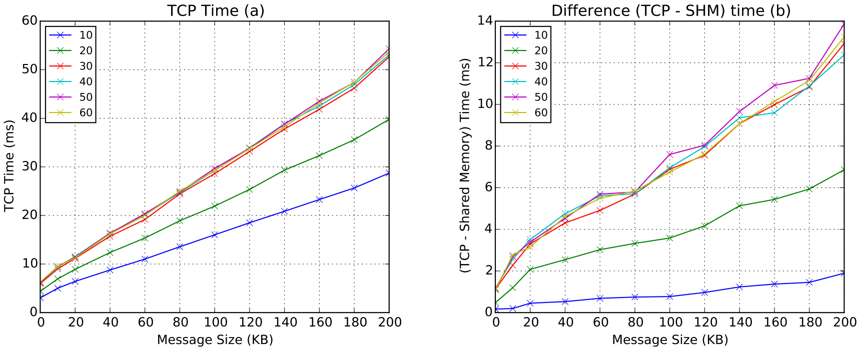


FIGURE 6.5. Relative Importance of Shared Memory Communication compared to TCP in a broadcasting ring, (a) The time for TCP communications. (b) Y-axis shows the difference in latency for TCP implementation and shared memory implementation (TCP - SHM)

## 6.5. Results & Discussion

The first experiment was conducted to measure the effect of memory mapped communications compared to the default TCP communications among the worker processes in a node. A topology with a communication going from worker to worker sequentially

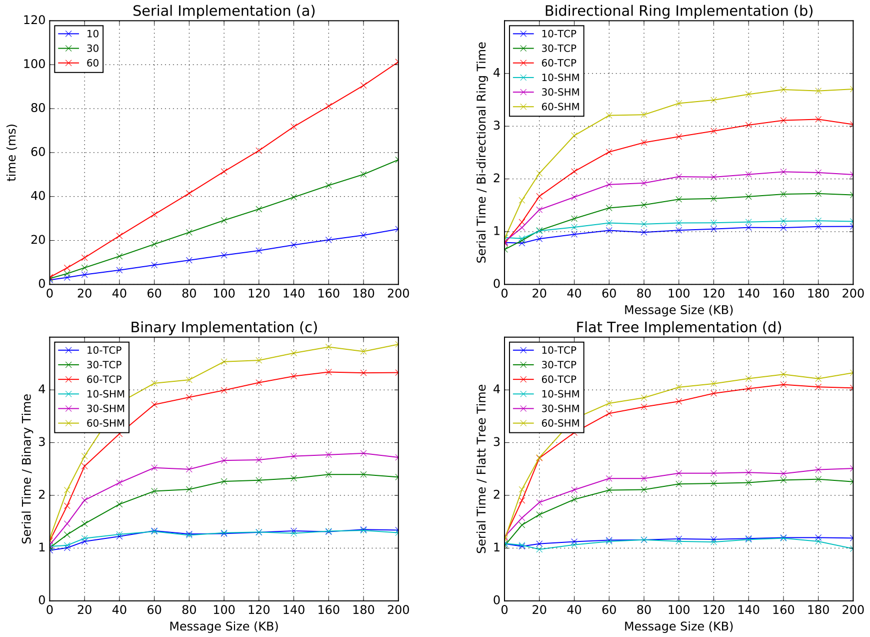


Figure 6.6. Latency of serial, binary tree, flat tree and bi-directional ring implementations. Different lines show varying parallel tasks with TCP communications and shared memory communications(SHM).

was used to measure the difference between the two. The communication connects the workers in one node and then connects to a worker in another node. All the workers in the Storm cluster were used for the experiment. The difference between the TCP latency and memory mapped latency is shown in Figure 6.5. Figure 6.5 shows the latency observed with the default TCP implementation. With only 10 tasks running in parallel and 32 workers available in 8 nodes, 6 nodes will have 1 task each and 2 nodes will have 2 tasks apiece. This means the use of memory mapped communication is minimal with only 10 tasks. Because of this, the difference between the two communications is practically zero. When increasing the number of tasks and the message size beyond 10K, it is clear from Figure 6.5 that we are gaining significant latency improvement by using memory mapped files, especially when

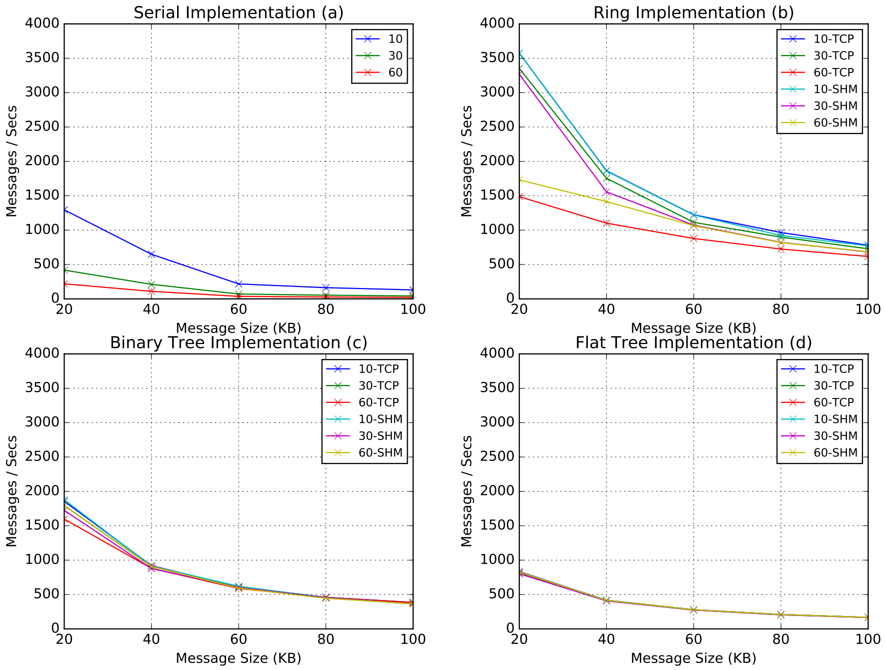


FIGURE 6.7. Throughput of serial, binary tree, flat tree and ring implementations. Different lines show varying parallel tasks with TCP communications and shared memory communications (SHM)

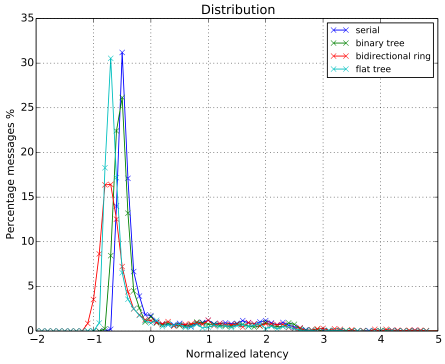


FIGURE 6.8. The distribution of message latencies for 100K messages with 60 tasks. The X-axis shows normalized latencies and Y-axis shows the percentage of messages observed for a latency

the number of tasks increases. Beyond 30 parallel tasks all the workers in the cluster are used by the topology, and because in-memory messaging is used between the tasks inside a worker there is practically no difference between latency for 30 and 60 tasks.

Figure 9.6b, 9.6c and 9.6d shows the gain in latency with bidirectional ring, binary tree and flat tree-based broadcasting algorithms, compared to the default serial implementation latency shown in Figure 9.6a. The Y-axis of the graphs shows the improvements made compared to serial time, i.e. $Serialtime/Improvedtime$. The binary tree algorithm performs the best among the three with about 5 times latency gain compared to the default algorithm for small message sizes which are most common in distributed streaming processing applications. Shared memory implementations show a further decrease in latency compared to the default TCP implementation for communications in a single node. The ring topology has the least latency decrease and flat tree falls between binary tree and ring. The effect of shared memory improvements are less for a binary tree and flat tree compared to ring because they are dominated mostly by the TCP communication among the nodes.

Figure 9.6 shows the arithmetic average latency of the topology. By looking at the distribution of individual latencies observed in the serial implementation and improved versions as shown in Figure 6.8, we concluded that there is no significant change in the latency distributions after applying the algorithms. It was observed that the variations in latency are mostly due to Java garbage collections and the original and improved results do not show any significant deviation in distribution due to that fact.

Figure 9.6 shows a micro-benchmark for broadcasting operation with default throughput 6.7a and ring 6.7b, binary tree 6.7c and flat tree 6.7d throughput. As expected the ring topology performs the best compared to the other algorithms. The shared memory latency has minimal effect on the throughput because it is dominated by the TCP connections between the nodes. In the ring implementation, the throughput for 60 parallel tasks is about half of 30 and 10 parallelism. With 0 parallelism, one worker hosts two worker tasks. These two worker tasks need to send the same message they receive to the gather bolts, resulting in the low throughput. This effect is not seen in the other algorithms because the network is saturated at the broadcast worker.

### 6.6. Infiniband & Omni-Path for Apache Heron

With ever increasing data production by users and machines alike, the amount of data that needs to be processed has increased dramatically. This must be achieved both in real time and as batches to satisfy different use cases. Additionally, with the adoption of devices into Internet of Things setups, the amount of real time data are exploding, and must be processed with reasonable time constraints. In distributed stream analytics, the large data streams are partitioned and processed in distributed sets of machines to keep up with the high volume data rates. By definition of large-scale streaming data processing, networks are a crucial component in transmitting messages between the processing units for achieving efficient data processing.

There are many hardware environments in which big data systems can be deployed including High performance computing (HPC) clusters. HPC clusters are designed to perform large computations with advanced processors, memory, IO systems, and high performance interconnects. High performance interconnects in HPC clusters feature microsecond latencies and large bandwidths. Thanks to recent advancements in hardware, some of these high performance networks have become cheaper to set up than their Ethernet counterparts. With multi-core and many-core systems having large numbers of CPUs in a single node, the demand for high performance networking is increasing as well.

Advanced hardware features such as high performance interconnects are not fully utilized in the big data computing frameworks, mostly because they are accessible to low level programming languages and most big data systems are written on Java platform. In recent years we have seen efforts to utilize high performance interconnects into big data frameworks such as Spark [110] and Hadoop [108]. Big data frameworks such as Spark and Hadoop focus on large batch data processing and hence their communication requirements are different compared to streaming systems which are more latency sensitive.

There are many distributed streaming frameworks available today for processing large amounts of streaming data in real time. Such systems are largely designed and optimized for commodity hardware and clouds. Apache Storm [140] was one of the popular early systems developed for processing streaming data. Apache Heron [3] [100] is similar to Storm with a new architecture for streaming data processing. It features a hybrid design with some of the performance-critical parts written in C++ and others written in Java. This architecture allows the integration of high performance enhancements directly rather than going through native wrappers such as Java Native Interface(JNI). When these systems are deployed on clusters that include high performance interconnects, they need to use an TCP interface to high performance interconnect which doesn't perform as well as a native implementation. To utilize these hardware features, we have integrated InfiniBand and Intel Omni-Path

[3] http://incubator.apache.org/projects/heron.html

interconnects to Apache Heron to accelerate its communications. InfiniBand [25] is an open standard protocol for high performance interconnects that is widely used in today's high performance clusters. Omni-Path [30] is a proprietary interconnect developed by Intel and is available with the latest Knights Landing architecture-based (KNL) [133] many-core processors. With this implementation, we have observed significantly lower latencies and improved throughput in Heron. The main contribution in this work is to showcase the benefits of using high performance interconnects for distributed stream processing. There are many differences in hardware available for communications with different bandwidths, latencies, and processing models. Ethernet has comparable hardware available to some of the high performance interconnects; it is not our goal to show that one particular technology is superior to others, as different environments may have alternate sets of these technologies.

The remainder of the paper is organized as follows. Section 6.7 presents the background information on InfiniBand and Omni-Path. Section 6.8 describes the Heron architecture in detail and section 6.9 the implementation details. Next, the experiments conducted are described in sections 6.10 and results are presented and discussed in section 6.11. Section **??** presents related work. The paper concludes with a look at future work.

## 6.7. Background

**6.7.1. InfiniBand.** InfiniBand is one of the most widely used high performance fabrics. It provides a variety of capabilities including message channel semantics, remote memory access, and remote atomic memory operations, supporting both connection-oriented and connectionless endpoints. InfiniBand is programmed using the Verbs API, which is available on all major platforms. The current hardware is capable of achieving up to 100Gbps speeds with microsecond latencies. InfiniBand does not require the OS Kernel intervention to transfer packets from user space to the hardware. Unlike in TCP, its protocol aspects are handled by the hardware. These features mean less CPU time spent on the network compared to TCP for transferring the same amount of data. Because the OS Kernel is bypassed by the communications, the memory for transferring data has to be registered in the hardware.

**6.7.2. Intel Omni-Path.** Omni-Path is a high performance fabric developed by Intel. Omni-Path fabric is relatively new compared to InfiniBand and there are fundamental differences between the two. Omni-Path does not offload the protocol handling to network hardware and it doesn't have the connection oriented channels as in InfiniBand. Unlike in InfiniBand, the Omni-Path network chip can be built into the latest Intel Knights Landing (KNL) processors. Omni-Path supports tagged messaging with a 96-bit tag in each message. A Tag can carry any type of data and this information can be used at the application to distinguish between different messages. Omni-Path is designed and optimized for small high frequency messaging.

**6.7.3. Channel & Memory Semantics.** High performance interconnects generally supports two modes of operations called channel and memory semantics [86]. With channel semantics, queues are used for communication. In memory semantics, a process can read from or write directly to the memory of a remote machine. In channel mode, two queue pairs for transmission and receive operations are used. To transfer a message, a descriptor is posted to the transfer queue, which includes the address of the memory buffer to transfer.

For receiving a message, a descriptor needs to be submitted along with a pre-allocated receive buffer. The user program queries the completion queue associated with a transmission or a receiving queue to determine the success or failure of a work request. Once a message arrives, the hardware puts the message into the posted receive buffer and the user program can determine this event through the completion queue. Note that this mode requires the receiving buffers to be pre-posted before the transmission can happen successfully.

With memory semantics, Remote Direct Memory Access(RDMA) operations are used. Two processes preparing to communicate register memory and share the details with each other. Read and write operations are used instead of send and receive operations. These are one-sided and do not need any software intervention from the other side. If a process wishes to write to remote memory, it can post a write operation with the local addresses of the data. The completion of the write operation can be detected using the completion queue associated. The receiving side is not notified about the write operation and has to use out-of-band mechanisms to figure out the write. The same is true for remote reads as well. RDMA is more suitable for large message transfers while channel mode is suitable for small messages. In general, RDMA has $1-2\mu s$ latency advantage over channel semantics for InfiniBand and this is not significant for our work.

**6.7.4. Openfabrics API.** Openfabrics [4] provides a library called libfabric [73] that hides the details of common high performance fabric APIs behind a uniform API. Because of the advantage of such an API, we chose to use libfabric as our programming library for implementing the high performance communications for Heron. Libfabric is a thin wrapper

[4] https://www.openfabrics.org/

API and it supports different providers including Verbs, Aries interconnect from Cray through GNI, Intel Omni-Path, and Sockets.

**6.7.5. TCP & High performance Interconnects.** TCP is one of the most successful protocols developed. It provides a simple yet powerful API for transferring data reliably across the Internet using unreliable links and protocols underneath. One of the biggest advantages of TCP is its wide adoption and simplicity to use. Virtually every computer has access to a TCP-capable adapter and the API is well supported across different platforms. TCP provides a streaming API for messaging where the fabric does not maintain message boundaries. The messages are written as a stream of bytes to the TCP and the application has to define mechanisms such as placing markers in between messages to indicate the boundaries. On the other hand, InfiniBand and Omni-Path both support message boundaries.

High performance interconnects have drivers that make them available through the TCP protocol stack. The biggest advantage of such implementation is that an existing application written using the TCP stack can use high performance interconnect without any modifications to the code. It is worth noting that the native use of the interconnect through its API always yields better performance than using it through TCP/IP stack. A typical TCP application allocates memory in user space and the TCP stack needs to copy data between user space and Kernel space. Also, each TCP invocation involves a system call which does a context switch of the application. Recent advancements like Netmap [128] and DPDK [63] removes these costs and increases the maximum number of packets that can be processed per second. Also, it is possible to achieve direct packet copy from user space to hardware. High performance fabrics usually do not go through OS kernel for network operations and

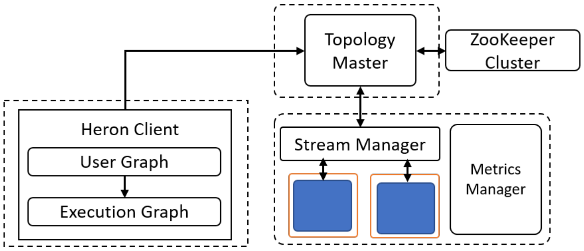


FIGURE 6.9. High level architecture of Heron. Each outer box shows a resource container allocated by a resource scheduler like Mesos or Slurm. The arrows show the communication links between different components.

the hardware is capable of copying data directly to user space buffers. InfiniBand offloads the protocol processing aspects to hardware while Omni-Path still involves the CPU for protocol processing.

## 6.8. Apache Heron

Heron is a distributed stream processing framework developed at Twitter and now available as an open source project in Apache Incubator [5]. Heron is similar to Apache Storm in its API with many differences in underlying engine architecture. It retains the same Storm API, allowing applications written in Storm to be deployed with no or minimal code changes.

**6.8.1. Heron Data Model.** A stream is an unbounded sequence of high level objects named events or messages. The streaming computation in Heron is referred to as a Topology. A topology is a graph of nodes and edges. The nodes represent the processing units executing the user defined code and the edges between the nodes indicate how the data (or stream) flows between them. There are two types of nodes: spouts and bolts. Spouts are the sources of streams. For example, a Kafka [99] spout can read from a Kafka queue and

[5] https://github.com/twitter/heron

emit it as a stream. Bolts consume messages from their input stream(s), apply its processing logic and emit new messages in their outgoing streams.

Heron has the concept of a user defined graph and an execution graph. The user defined graph specifies how the processing units are connected together in terms of message distributions. On the other hand, the execution graph is the layout of this graph in actual nodes with network connections and computing resources allocated to the topology to execute. Nodes in the execution graph can have multiple parallel instances (or tasks) running to scale the computations. The user defined graph and the execution graph are referred to as logical plan and physical plan respectively.

**6.8.2. Heron Architecture.** The components of the Heron architecture are shown in Fig. 6.9. Each Heron topology is a standalone long-running job that never terminates due to the unbounded nature of streams. Each topology is self contained and executes in a distributed sandbox environment in isolation without any interference from other topologies. A Heron topology consists of multiple containers allocated by the scheduler. These can be Linux containers, physical nodes or sandboxes created by the scheduler. The first container, referred to as the master always runs the Topology Master that manages the topology. Each of the subsequent containers have the following processes: a set of processes executing the spout/bolt tasks of the topology called Heron instances, a process called a stream manager that manages the data routing and the connections to the outside containers, and a metrics manager to collect information about the instances running in that container.

Each Heron instance executes a single task of the topology. The instances are connected to the stream manager running inside the container through TCP loop-back connection.

It is worth noting that Heron instances always connect to other instances through the stream manager and they do not communicate with each other directly even if they are in the same container. The stream manager acts as a bridge between Heron instances. It forwards the messages to the correct instances by consulting the routing tables it maintains. A message between two instances in different containers goes through two stream managers. Containers can have many Heron instances running in them and they all communicate through the stream manager. Because of this design, it is important to have highly efficient data transfers at the stream manager to support the communication requirements of the instances.

Heron is designed from the ground up to be extensible, and important parts of the core engine are written in C++ rather than Java, the default language of choice for Big Data frameworks. The rationale for the use of C++ is to leverage the advanced features offered by the OS and hardware. Heron instances and schedulers are written in Java while stream manager and topology master are written in C++.

6.8.2.1. *Acknowledgements.* Heron uses an acknowledgment mechanism to provide at least once message processing semantics that ensures the message is always processed in the presence of process/machine failures. In order to achieve at least once, the stream manager tracks the tuple tree generated while a message progresses through the topology. When a bolt emits a message, it anchors the new message to the parent message and this information is sent to originating stream manager (in the same container) as a separate message. When every new message finishes its processing, a separate message is again sent to the originating stream manager. Upon receiving such control messages for every emits in the message tree, the stream manager marks the message as fully processed.

6.8.2.2. *Processing pipeline.* Heron has a concept called max messages pending with spouts. When a spout emits messages to a topology, this number dictates the amount of in-flight messages that are not fully processed yet. The spout is called to emit messages only when the current in-flight message count is less than the max spout pending messages.

**6.8.3. Heron Stream Manager.** Stream manager is responsible for routing messages between instances inside a container and across containers. It employs a single thread that uses event-driven programming using non-blocking socket API. A stream manager receives messages from instances running in the same container and other stream managers. These messages are Google protocol buffer [141] serialized messages packed into binary form and transmitted through the wire. If a stream manager receives a message from a spout, it keeps track of the details of the message until all the acknowledgements are received from the message tree. Stream manager features an in-memory, store and forward architecture for forwarding messages and can batch multiple messages into single message for efficient transfers. Because messages are temporarily stored, there is a draining function that drains the store at a user defined rate.

## 6.9. Implementation

Even though high performance interconnects are widely used by HPC applications and frameworks, they are not often utilized in big data systems. Furthermore, experiences in using these interconnects in big data systems are lacking in the public domain. In

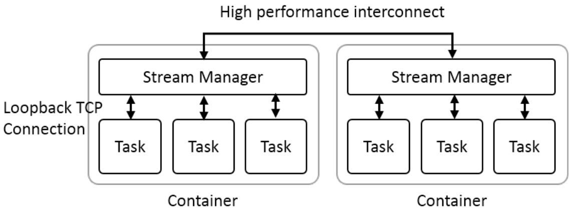


FIGURE 6.10. Heron high performance interconnects are between the stream managers

this implementation, InfiniBand and Omni-Path interfaces with Heron through its stream manager, as shown in Fig. 6.10. InfiniBand or Omni-Path message channels are created between each stream manager in the topology. These then carry the data messages going between the stream managers. The control messages that are sent between stream manager and topology master still use the TCP connections. They are not frequent and do not affect the performance of the data flow. The TCP loop-back connections from the instances to the stream manager are not altered in this implementation as loop back connection is much more efficient than the network. Both InfiniBand and Omni-Path implementations use channel semantics for communication. A separate thread is used for polling the completion queues associated with the channels. A credit based flow control mechanism is used for each channel along with a configurable buffer pool.

**6.9.1. Bootstrapping.** InfiniBand and Omni-Path require information about the communication parties to be sent out-of-band through other mechanisms like TCP. InfiniBand uses the RDMA(Remote direct memory access) Connection manager to transfer the required information and establish the connections. RDMA connection manager provides a socket-like API for connection management, which is exposed to the user in a similar fashion through Libfabric API. The connection manager also uses the IP over InfiniBand network adapter to discover and transfer the bootstrap information. Omni-Path has a built-in TCP server for discovering the endpoints. Because Omni-Path does not involve connection management, only the destination address is needed for communications. This information can be sent using an out-of-band TCP connection.

**6.9.2. Buffer management.** Each side of the communication uses buffer pools with equal size buffers to communicate. Two such pools are used for sending and receiving data for each channel. For receiving operations, all the buffers are posted at the beginning to the fabric. For transmitting messages, the buffers are filled with messages and posted to the fabric for transmission. After the transmission is complete the buffer is added back to the pool. The message receive and transmission completions are discovered using the completion queues. Individual buffer sizes are kept relatively large to accommodate the largest messages expected. If the buffer size is not enough for a single message, the message is divided in to pieces and put into multiple buffers. Every network message carries the length of the total message and this information can be used to assemble the pieces.

The stream manager de-serializes the protocol buffer message in order to determine the routing for the message and handling the acknowledgements. The TCP implementation first copies the incoming data into a buffer and then use this buffer to build the protocol buffer structures. This implementation can directly use the buffer allocated for receiving to build the protocol message.

**6.9.3. Flow control at communication level.** Neither InfiniBand nor Omni-Path implement flow control between the communication parties, and it is up to the application developer to implement the much higher level functions [156]. This implementation uses a standard credit-based approach for flow control. The credit available for sender to communicate is equal to the number of buffers posted into the fabric by the receiver. Credit information is passed to the other side as part of data transmissions, or by using separate messages in case there are no data transmissions to send it. Each data message carries the current credit of the communication party as a 4-byte integer value. The credit messages do not take into account the credit available to avoid deadlocks, otherwise there may be situations where there is no credit available to send credit messages.

**6.9.4. Interconnects.** Infiniband implementation uses connection-oriented endpoints with channel semantics to transfer the messages. The messages are transferred reliably by the fabric and the message ordering is guaranteed. The completions are also in order of the work request submissions.

Intel Omni-Path does not support connection-oriented message transfers employed in the Infiniband implementation. The application uses reliable datagram message transfer with tag-based messaging. Communication channels between stream managers are overlaid on a single receive queue and a single send queue. Messages coming from different stream managers are distinguished based on the tag information they carry. The tag used in the implementation is a 64-bit integer which carries the source stream manager ID and the destination stream manager ID. Even though all the stream managers connecting to a single stream manager are sharing a single queue, they carry their own flow control by assigning a fixed amount of buffers to each channel. Unlike in Inifiniband, the work request completions are not in any order of their submission to the work queue. Because of this, the application keeps track of the submitted buffers and their completion order explicitly.

IP over Fabric or IP over InfiniBand(IPoIB) is a mechanism to allow a regular TCP application to access the underlying high performance interconnects through the TCP Socket API. For using IPoIB heron stream manager TCP sockets are bound to the IPoIB network interface explicitly without changing the existing TCP processing logic.

## 6.10. Experiments

An Intel Haswell HPC cluster was used for the InfiniBand experiments. The CPUs are Intel Xeon E5-2670 running at 2.30GHz. Each node has 24 cores (2 sockets x 12 cores each) with 128GB of main memory, 56Gbps InfiniBand interconnect and 1Gbps dedicated Ethernet connection to other nodes. Intel Knights Landing(KNL) cluster was used for Omni-Path tests. Each node in KNL cluster has 72 cores (Intel Xeon Phi CPU 7250F, 1.40GHz) and is connected to a 100Gbps Omni-Path fabric and 1Gbps Ethernet connection. There are many variations of Ethernet, InfiniBand and Omni-Path performing at different message rates and latencies. We conducted the experiments in the best available resources to us.

We conducted several micro-benchmarks to measure the latency and throughput of the system. In these experiments the primary focus was given to communications and no computation was conducted in the bolts. The tasks in each experiment were configured with 4GB of memory. A single Heron stream manager was run in each node. We measured the IP over Fabric (IPoIB) latency to showcase the latency possible in case no direct implementation of InfiniBand or Omni-Path.

**6.10.1. Experiment Topologies.** To measure the behavior of the system, two topologies shown in Fig. 6.12 and Fig. 6.11 is used. Topology A in Fig. 6.11 is a deep topology with

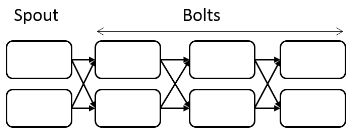


FIGURE 6.11. Topology A: Deep topology with a spout and multiple bolts arranged in a chain. The spout and bolt run multiple parallel instances.

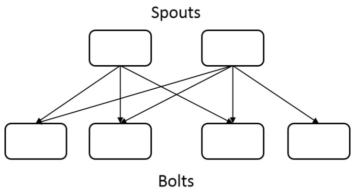


FIGURE 6.12. Topology B: Shallow topology with a spout and bolt connected in a shuffle grouping. The spout and bolts run multiple parallel instances.

multiple bolts arranged in a chain. The parallelism of the topology determines the number of parallel task for bolts and spout in the topology. Each adjacent component pair is connected by a shuffle grouping. Topology B in Fig. 6.12 is a two-component topology with a spout and a bolt. Spouts and bolts are arranged in a shuffle grouping so that the spouts load balance the messages among the bolts.

In both topologies the spouts generated messages at the highest sustainable speed with acknowledgements. The acknowledgements acted as a flow control mechanism for the topology. The latency is measured as the time it takes for a tuple to go through the topology and its corresponding acknowledgement to reach the spout. Since tuples generate more tuples when they go through the topology, it takes multiple acknowledgements to complete a tuple. In topology A, it needs control tuples equal to the number of Bolts in the chain to complete a single tuple. For example if the length of the topology is 8, it takes 7 control tuples to complete the original tuple. Tests were run with 10 in flight messages through the topology.

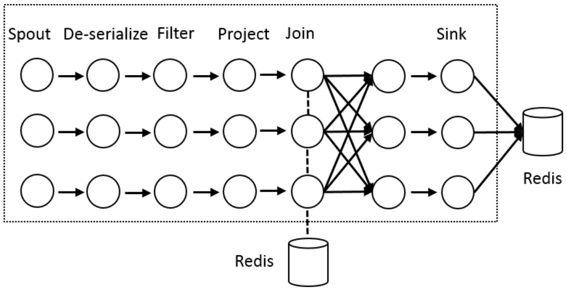


FIGURE 6.13. Yahoo Streaming benchmark with 7 stages. The join bolts and sink bolts communicate with a Redis server.

**6.10.2. Yahoo streaming benchmark.** For evaluating the behavior with a more practical streaming application, we used a benchmark developed at Yahoo![6] to test streaming frameworks. We modified the original streaming benchmark to support Heron and added additional features to support our case. The modified benchmark is available open source in Github [7]. It focuses on an advertisement application where ad events are processed using a streaming topology as shown in Fig. 6.18. We changed the original benchmark to use a self-message-generating spout instead of reading messages through Kakfa [99]. This was done to remove any bottlenecks and variations imposed by Kafka.

The benchmark employs a multiple stage topology with different processing units. The data is generated as a JSON object and sent through the processing units, which do de-serialization, filter, projection and join operations on each tuple. At the joining stage, it uses a Redis [40] database to query data about the tuples. Additionally, the last bolt saves information to the Redis database. At the filter step about 66% of the tuples are dropped. We use an acking enabled topology and measured the latency as the time it takes for a tuple to go through the topology and its ack to return back to the spout. For our tests we

[6] https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

[7] https://github.com/iotcloud/streaming-benchmarks.git

used 16 nodes with 8 parallelism at each step, totaling 56 tasks. Each spout was sending 100,000 messages per second, which gave 800,000 messages per second in total. Note that this topology accesses the Redis database system for 33% messages it receives.

## 6.11. Results & Discussion

Fig. 6.14 shows the latency of the Topology A with varying message sizes and parallelism for IB, TCP and IPoIB. InfiniBand performed the best while Ethernet showed the highest latency. Fig. 6.15 shows the latency of the Topology B with varying message sizes and parallelism as in Fig. 6.14. For small messages, we see both IPoIB and Ethernet performing at a similar level while InfiniBand was performing better. When increasing the parallelism, the latency increased as expected and InfiniBand showed a smaller increase than IPoIB and Ethernet. As expected Topology B showed less improvements compared to Topology A. For most practical applications, the minimum latency possible will be within the results observed in these two topologies as these represent the minimum possible topology and a deep topology for most practical applications.

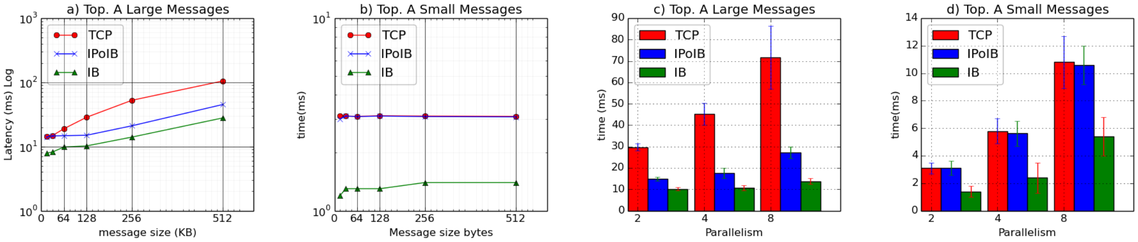


Figure 6.14. Latency of the Topology A with 1 spout and 7 bolt instances arranged in a chain with varying parallelism and message sizes. a) and b) are with 2 parallelism and c) and d) are with 128k and 128bytes messages. The results are on Haswell cluster with IB.

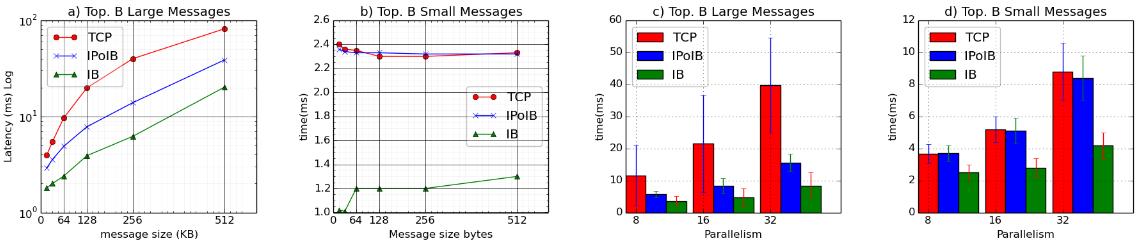


Figure 6.15. Latency of the Topology B with 32 parallel bolt instances and varying number of spouts and message sizes. a) and b) are with 16 spouts and c) and d) are with 128k and 128bytes messages. The results are on Haswell cluster with IB.

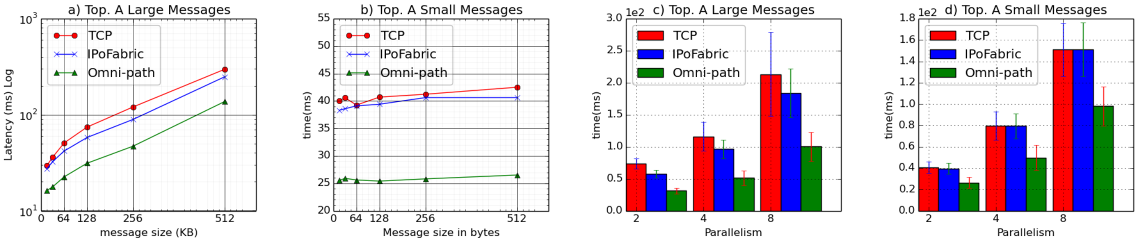


Figure 6.16. Latency of the Topology A with 1 spout and 7 bolt instances arranged in a chain with varying parallelism and message sizes. a) and b) are with 2 parallelism and c) and d) are with 128k and 128bytes messages. The results are on KNL cluster.

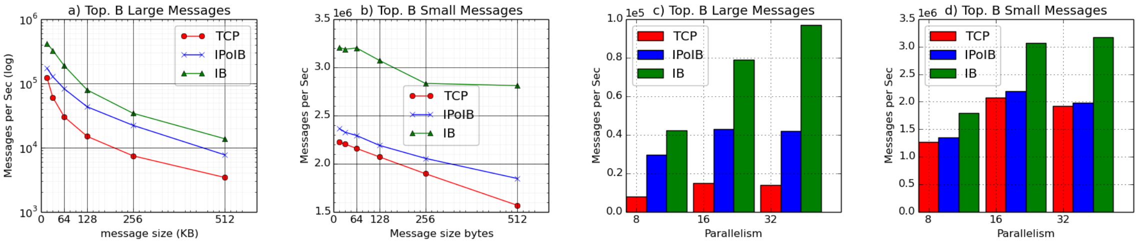


Figure 6.17. Throughput of the Topology B with 32 bolt instances and varying message sizes and spout instances. The message size varies from 16K to 512K bytes. The spouts are changed from 8 to 32.

Fig. 6.16 shows the latency results of Omni-Path implementation with Topology A, in the KNL cluster for large and small messages. IPoFabric is the IP driver for Omni-Path. The KNL machine has less powerful CPU cores and hence the latency was higher compared to InfiniBand in the tests. Fig 6.18 shows the latency distribution of the Topology A with

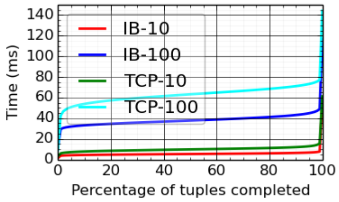


FIGURE 6.18. Percent of messages completed within a given latency for the Topology A with in-flight messages at 100 and 10 with 128K messages and 8 parallelism

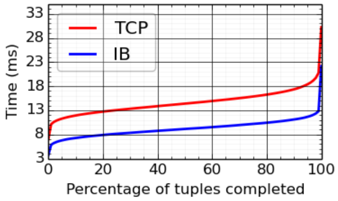


FIGURE 6.19. Percent of messages completed within a given latency for the Yahoo! streaming benchmark with InfiniBand network. The experiment was conducted in Haswell cluster with 8 nodes and each stage of topology have 8 parallel tasks.

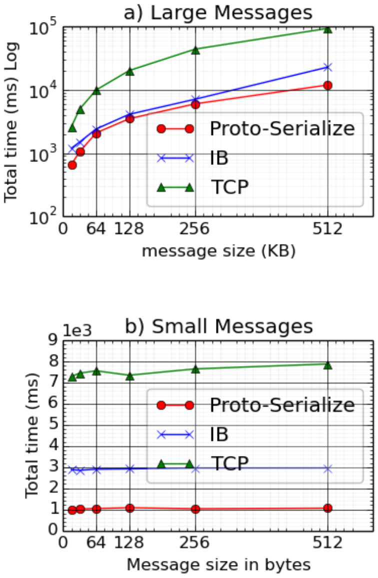


FIGURE 6.20. The total time to finish the messages vs the total time to serialize the messages using protobuf and Kryo. Top. B with 8 parallelism for bolts and spouts used. Times are for 20000 large messages and 200000 small messages. The experiment is conducted on Haswell cluster.

100 and 10 inflight messages with 128k message size for TCP and IB. The graphs show that the network latencies are according to a normal distribution with high latencies at the 99th percentile. Fig. 6.19 shows the latency distribution seen by the Yahoo stream benchmark

with InfiniBand fabric. For all three networking modes, we have seen high spikes at the 99th percentile. This was primarily due to Java garbage collections at the tasks which are unavoidable in JVM-based streaming applications. The store and forward functionality of the stream manager contributes to the distribution of latencies as a single message can be delayed up to 1ms randomly at stream manager.

Fig. 6.17 present the message rates observed with Topology B. The experiment was conducted with 32 parallel bolt instances and varying numbers of spout instances and message sizes. The graph shows that InfiniBand had the best throughput, while IPoIB achieved second-best and Ethernet came in last. Fig. 6.17 c) and d) show the throughput for 128K and 128 bytes messages with varying number of spouts. When the number of parallel spouts increases, the IPoIB and Ethernet maxed out at 16 parallel spouts for 128k messages, while InfiniBand kept on increasing. Fig. 6.20 shows the total time required to serialize protocol buffers and time it took to complete the same number of messages with TCP and IB implementations. It is evident that IB implementation overhead is much less compared to TCP.

The results showed good overall results for InfiniBand and Omni-Path compared to the TCP and IPoIB communication modes. The throughput of the system is bounded by the CPU usage of stream managers. For TCP connections, the CPU is used for message processing as well as the network protocol. InfiniBand, on the other hand, uses the CPU only for message processing at the stream manager, yielding better performance.

The results showed much higher difference between TCP and InfiniBand for large message sizes. For small messages, the bandwidth utilization is much lower than large messages. This is primarily due to the fact that CPU is needed to process every message. For larger messages, because of the low number of messages transferred per second, the CPU usage is low for that aspect. For smaller messages, because of the large number of messages per second, the CPU usage is much higher. Because of this, for small messages, the stream managers saturate the CPU without saturating the communication channel. For practical applications that require large throughput, Heron can bundle small messages into a large message in-order to avoid some of the processing overheads and transfer overheads. This makes it essentially a large message for the stream manager and large message results can be observed with elevated latencies for individual messages. We used 10 buffers with 1 megabyte allocated to each for sending and receiving messages. Protocol buffers are optimized for small messages and it is unlikely to get more than 1 megabyte messages for streaming applications targeting Heron.

The KNL system used for testing Omni-Path has a large number of processes with fewer CPU frequencies compared to Haswell cluster. In order to fully utilize such a system, multiple threads need to be used. For our implementation we did not explore such features specific to KNL and tried to first optimize for the Omni-Path interconnect. The results show that Omni-Path performed considerably better than the other two options. In this work, the authors did not try to pick between Omni-Path or InfiniBand as a better interconnect as they are tested in two completely different systems under varying circumstances. The objective of the work is to show the potential benefits of using interconnects to accelerate stream processing.

CHAPTER 7

# Next generation distributed and parallel frameworks

The ever increasing data has changed the way we think about analytic in general. The rich set of applications demand systems that are configurable and adaptable to different applications. We believe the approaches taken by current systems are not flexible enough for the diverse set of applications. A typical data analytic workflow deal with data gathering, pre-processing, machine learning, post processing and visualization. It is important to note the differences between data processing stages and executing machine learning algorithms.

The data is gathered through various sources. These include streams of events coming from users and devices, machine generated data, instruments, log file and many more. They are stored in data lakes such as HDFS, and databases including SQL and NoSQL. Some form of data schema is followed and data is cleaned to some extent. Data lakes are schema-less and store raw data.

Machine learning and other analytic algorithms require data in certain formats and needs data to be cleaned before hand. Also not every data point in a data set is used by analytic algorithms. So data has to be filtered, enriched and modified before it can be used. This process is usually not compute intensive and mostly depend on I/O performance. Because of the I/O dominance data locality is of utmost importance. With streaming data, the data has to be processed before storing to disks. Otherwise the same principles like data cleaning, filtering are applied to data while they stream through. Both streaming and batch analytics need to work in same dataflow to accomplish end-to-end applications.

## 7.1. Choice of tools

There are many big data tools available that claim they support every type of application including ML, data querying and streaming. A closer look at their performance reveal that they mostly work best for one type of application, even though technically they can support others. For example Flink is good at streaming but it lacks basic structures such a nested parallel iterations required by ML and batch programs. Spark is good at data querying but not at streaming or ML as shown in 3. Heron and Storm are developed targeting streaming applications and do not support batch applications.

Because these systems are technically capable of supporting different types of applications, users are mostly confused as which platform to use for a given application. Adding to the confusion is that, there are subsets of applications from a particular application class that work comparably good in these systems even though they cannot support the broad application set. A very good example is Flink K-Means and MDS as shown in chapter 3. Flink K-Means algorithm performs best compared to Spark and a user who is looking only at such a result can conclude that Flink works better than Spark and comparable to MPI. But MDS algorithm 3 shows that this is not true for a more complex algorithm that requires nested parallel iterations.

Also we need to note that MPI is not designed for data processing applications and lacks the abstractions required to do so. With MPI one can build any type of application because it is a communication standard and user is allowed to control every other aspect of the program. This doesn't mean that MPI abstractions are correct for all types of applications and one needs to build different abstractions to support different types of applications.

## 7.2. Multiple abstractions

There are three main aspects that can be abstracted to provide a programming API to develop a parallel program. At the bottom is the communication between the parallel tasks which is essential in any distributed and parallel application. At the second level we have the thread and process management for execution of parallel tasks. Third abstraction is how to mange the data including both input and intermediate data. Higher level abstractions are in general easy to program but lacks the performance of a low level abstraction. It is important to choose the correct level of abstraction depending on the performance requirements of the applications and the software engineering aspects such as availability of software programmers, testing, validation, fault tolerance and maintainability.

We have shown that BSP style communications as implemented by MPI and Harp are best for machine learning algorithms outperforming big data framework based implementations by large margins. This fact is substantiated by many others who have used big data tools to implement ML algorithms [51, 68, 127]. We believe only a handful of developers will implement high performance machine learning algorithms and majority will look to apply these algorithms to their data sets. Machine learning algorithms will be available in highly optimized libraries developed by specialized programmers as in Intel DAAL [6] or TensorFlow [16]. We believe it is acceptable to have higher programming complexity for better performance of ML algorithm libraries.

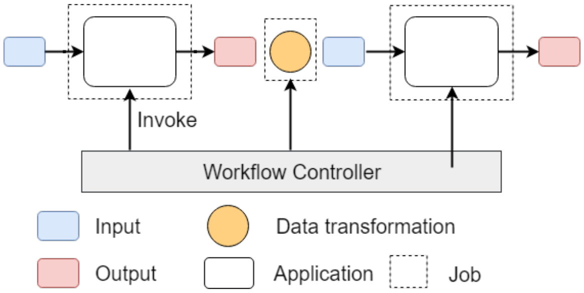


FIGURE 7.1. HPC workflow for application orchestration

The data processing algorithms such as ETL (Extract, load, transform) operations as seen in streaming and data pipeline applications are more suitable to be developed as data abstractions and task abstractions due to their dominance in I/O operations. With communication abstractions, a programmer needs to take into account data locality of different data sources and thread management with emphasis on I/O to achieve the best performance. As in ML algorithms that perform well when programmed in the data abstractions layer (Flink K-Means), one can find low level requirements of ETL and streaming use cases that performs best in the communication level. But in general it is safe to say that these applications are better suited at higher levels.

Because of these factors, one abstraction may not work across all types of applications. Even for one application class there can be applications that are better suited at different levels of abstractions. A well designed system will allow a user to pick the correct abstraction based on needs.

### 7.3. Dataflow vs Workflow

For over decades workflow systems were the focal point of integration for HPC applications. Fig. 7.1 shows an example of set of applications controlled by a workflow. Workflow

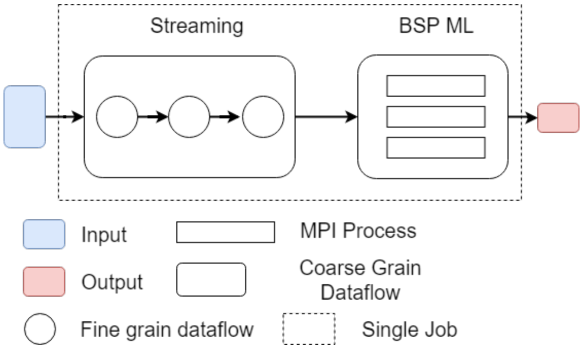


FIGURE 7.2. Coarse grain dataflow graph with fine grain graphs as in streaming and BSP application

controller can be a script or a separate entity as in Kepler [111] or Tevarna [148]. In general, workflow controller executes an application, takes its output in disk, transforms output to suite the next application and invokes the next one.

Data driven applications are modeled as dataflow graphs as shown in Fig. 7.3 in contrast to workflows. This model allows the integration of different applications with large data sets in a seamless manner without going to disk in-between applications. Depending on the application, granularity of the graph can be different. For example, we can model a ML application as a fine grained dataflow graph. When multiple applications are connected to each other using dataflow edges, we can view it as a hierarchical set of dataflow graphs.

## 7.4. Next generation frameworks

We list key requirements of next generation systems for parallel and distributed computing.

(1) Data driven applications with seamless integration

(2) Using correct abstractions

(3) Ability to work on Cloud and HPC environments

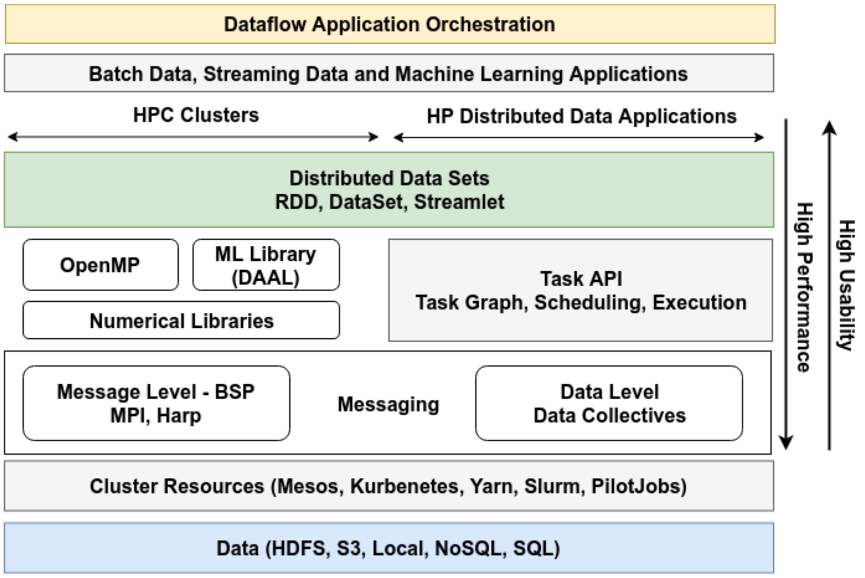

Dataflow Application Orchestration
Batch Data, Streaming Data and Machine Learning Applications
HPC Clusters
HP Distributed Data Applications
Distributed Data Sets
RDD, DataSet, Streamlet
OpenMP
ML Library
(DAAL)
Numerical Libraries
Task API
Task Graph, Scheduling, Execution
Message Level - BSP
MPI, Harp
Messaging
Data Level
Data Collectives
Cluster Resources (Mesos, Kurbenetes, Yarn, Slurm, PilotJobs)
Data (HDFS, S3, Local, NoSQL, SQL)
High Performance
High Usability

FIGURE 7.3. Overall system view of future systems, each layer will have different API's

By satisfying the first and third requirement, a system should be able to integrate streaming, ETL and ML applications both in clouds as well as in HPC environments. The second requirement allows applications to be developed with best possible abstractions to achieve performance, usability and maintainability desired. For example the current systems only allow applications to be developed using coarse grained dataflow graphs and we intend to allow MPI style applications to be implemented as nodes of dataflow graph. Also it is important to build different layers as standalone components with configurable and extendable architectures which will allow future applications to be built on top of these foundations.

A simplified view of a future system supporting different systems is shown in Fig. **??**. It shows the key abstractions of a general framework along with different choices at layers.

CHAPTER 8

# Twister2: Big data toolkit

## 8.1. Introduction

Systems such as Spark [151] and Hadoop primarily focus on batch data, while Heron [100], Flink [37] and Storm target streaming data. As opposed to these systems, the high performance computing (HPC) community uses Message Passing Interface (MPI) and its implementations as their framework of choice for large-scale parallel applications.

From an execution perspective, we can identify four key aspects of a parallel program whether it is designed for data processing or HPC: 1. Acquiring computing resources, 2. Spawning processes/threads and managing them on the allocated resources to execute the user program, 3. Communication layer between the parallel processes, and 4. Managing the data including both static and intermediate data. The HPC community has developed different technologies to abstract out these layers including resource schedulers such as Slurm [149], MPI, and OpenMP [45] for process and thread management, communication using MPI, and in-memory distributed data management using PGAS [43]. These systems are mostly independent and allow a user to pick and choose depending on application requirements. For example, one can use only MPI where the application manages both threads and data of the program. In another setting, MPI plus OpenMP can be used where OpenMP manages the threads within an MPI process.

The big data systems are mostly designed in a monolithic approach with the above mentioned functions developed in a single project with tight integration between them. With the advent of Mesos [78] and Yarn [142], the resource scheduling layer is being separated from most big data systems, but other layers mostly remain within the same framework. Such designs make it harder to evolve functionality independently and adhere to standards. Also, we note that these systems are designed with assumptions at different components making them suitable for a limited set of applications [14].

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. 8.1 shows the runtime architecture of Twister2 with various components. Even though Fig. 8.1 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. 8.1 gives a summary of various components, APIs, and implementation choices.

## 8.2. Toolkit Components

**8.2.1. Architecture Specification.** System specification captures the essentials of a parallel application that will determine the configuration of the components. We identify

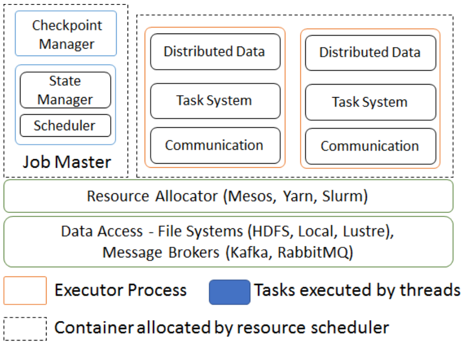

Checkpoint Manager
State Manager
Scheduler
Job Master
Distributed Data
Task System
Communication
Distributed Data
Task System
Communication
Resource Allocator (Mesos, Yarn, Slurm)
Data Access - File Systems (HDFS, Local, Lustre), Message Brokers (Kafka, RabbitMQ)
Executor Process
Tasks executed by threads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

FIGURE 8.1. Runtime architecture of Twister2

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 is 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

Table 8.1. 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 |

finer-grained than workflow and similar to those in HPC systems where computing phases move to communication phases.

**Execution semantics:** Execution semantics of an application defines 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.

**8.2.2. 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.

## 8.3. 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 [136]. Some of the heavily used primitives are Broadcast, Gather, Reduce, AllGather and AllReduce [136]. 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. 8.2 summarizes some of the operations available in BSP and dataflow communications.

Table 8.2. 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 |

8.3.0.1. *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 [137] and the current implementations are optimized to an extremely high extent. A comprehensive summary of MPI collective operations and possible algorithms is found in [145]. BSP communications can be used as in MPI, or by the task system. Harp [153] is a machine learning-focused collective library that supports the standard MPI collectives as well as some other operations like rotate, push and pull.

8.3.0.2. *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 [27], 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.8.2 and Fig.8.3 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 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.

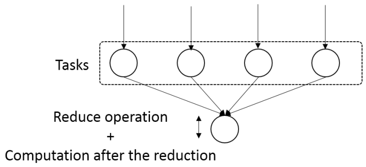


Figure 8.2. Default implementation of a dataflow reduce

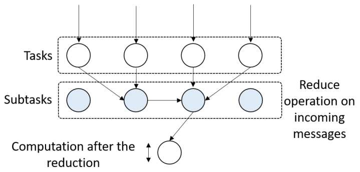

Tasks
Subtasks
Reduce operation on incoming messages
Computation after the reduction

Figure 8.3. Optimized dataflow reduce operation with sub-tasks arranged in a tree

(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 library 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.

8.3.0.3. *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 [86], Hadoop [108] and Spark [107]. 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 [73] and Photon [96], this is no longer the case.

### 8.4. Task System

In order to develop an application at the communication layer, one needs a deep understanding of 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.

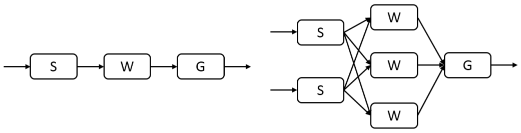


FIGURE 8.4. Left: User graph, Right: execution graph of a data flow

**8.4.1. 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.

**8.4.2. 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 number 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. 8.4 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.

**8.4.3. 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 [5]. 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.

8.4.3.1. *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 [34] 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.

**8.4.4. 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. 8.5. 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. 8.6 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 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

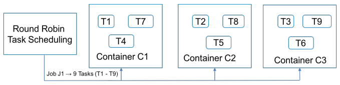


FIGURE 8.5. RR Task Scheduling (Homogeneous Containers and Heterogeneous Task Instances)

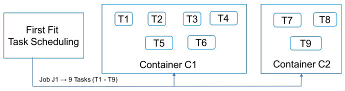


FIGURE 8.6. FF Task Scheduling (Heterogeneous Containers and Heterogeneous Task Instances)

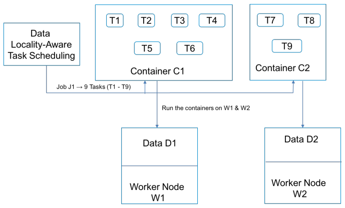


FIGURE 8.7. DLA Task Scheduling (Execution on the Data Nodes)

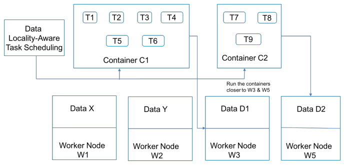


FIGURE 8.8. DLA Task Scheduling (Execution Closer to the Data Nodes)

case while scheduling the tasks. Fig. 8.7 and Fig. 8.8 show the data locality-aware task scheduling scenarios handled in the Twister2 framework.

**8.4.5. 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.

**8.4.6. 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 [20].

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 [129] is important for achieving optimum performance. It has been observed that using a single task executor for both these applications would bring inferior performance [41].

8.4.6.1. *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 [134]. 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 [134].

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 the number of processes per node must be maintained.

**8.4.7. Data Access.** Data access abstracts from 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.

### 8.5. 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 allows 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 are 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 on 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. 8.9 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.

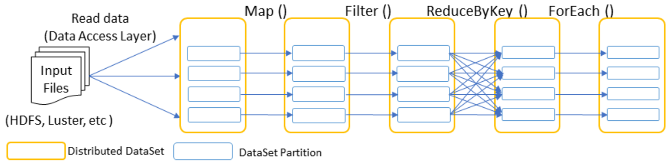


Figure 8.9. Example execution graph for $logFile.map(...).filter(...).reduceByKey(...).forEach(...)$

## 8.6. 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 projects such as [55] and [84] 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.

8.6.0.1. *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 [38]. 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.

8.6.0.2. *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 an 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.

**8.6.1. State Management.** State management is an important aspect of 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 need to take the state into consideration when creating checkpoints of the system. Job level state is managed by a separate process that make 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.

**8.6.2. 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. 8.10 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.

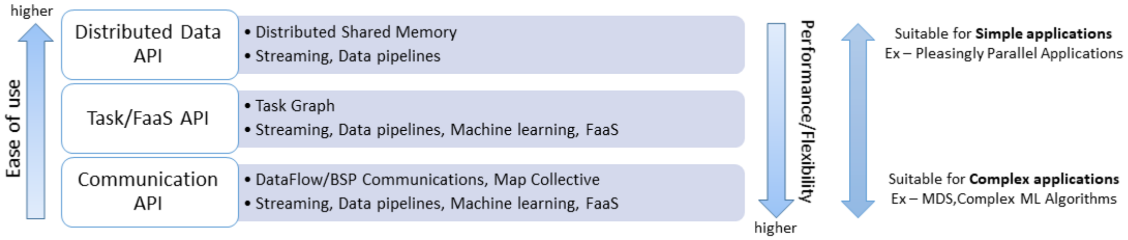

higher
Ease of use
Distributed Data API
• Distributed Shared Memory
• Streaming, Data pipelines
Task/FaaS API
• Task Graph
• Streaming, Data pipelines, Machine learning, FaaS
Communication API
• DataFlow/BSP Communications, Map Collective
• Streaming, Data pipelines, Machine learning, FaaS
Performance/Flexibility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

FIGURE 8.10. Twister2 Big Data Toolkit API levels

Table 8.3. 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 |

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 8.3 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.

## 8.7. Summary

In this chapter, 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 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.

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 cross 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.

CHAPTER 9

# Twister:Net

This chapter presents the communication layer of Twister2, which is named Twister:Net. Specifically it focuses on the dataflow style communications as it is a novel library for data processing.

## 9.1. Introduction

MPI is the dominant standard for HPC applications whereas big data adopted the dataflow model. Every parallel program including big data applications and HPC applications can be modeled as a graph with nodes doing computations and edges representing the communication. Dataflow programming model has become popular in data analytics due to its simplicity and ease of use. With this model, big data frameworks represent a computation as a generic graph where nodes of the graph can be executed on different machines depending on the requirements of the application. This generic graph structure adopted by big data systems allows one to model both streaming and batch applications.

Big data systems do not view parallel operations in terms of communications but rather as higher level API's on data sets. By doing so they can hide the communications under higher level APIs such as task and data transformation APIs. As a result, such communications are in general loosely defined by separate implementations without a standard specification. Even though every big data framework is designed according to the same dataflow model, different systems have their own primitives with slightly varying semantics depending on the implementations. The communication implementations found in the current frameworks are tightly integrated and not exposed as APIs to be used as libraries without relying on the entire framework. In this paper, authors recognize the need to have a separate communication library supporting dataflow style big data communications.

Twister:Net is the communication component of Twister2, which has been developed as a stand-alone library for big data applications supporting dataflow communications. Bulk synchronous parallel(BSP) communication as implemented by MPI can be used in Twister2 as well. Twister2 plans to incorporate Harp [154], which is a BSP implementation for big data systems. The dataflow communications in Twister:Net are implemented on top of TCP and MPI communications, allowing it to be deployed in both HPC and cloud environments. In this paper we present, dataflow communications of Twister:Net and compare it to the existing big data frameworks in order to show it can achieve equivalent performance or better with existing frameworks. Also, we look at MPI communications and big data requirements and compare different applications in those settings. This chapter highlights two major contributions.

(1) Define a dataflow communication model for big data to include both streaming and batch data

(2) Present Twister:Net as an implementation of this model and show that it can achieve better or equal performance compared to streaming and batch systems

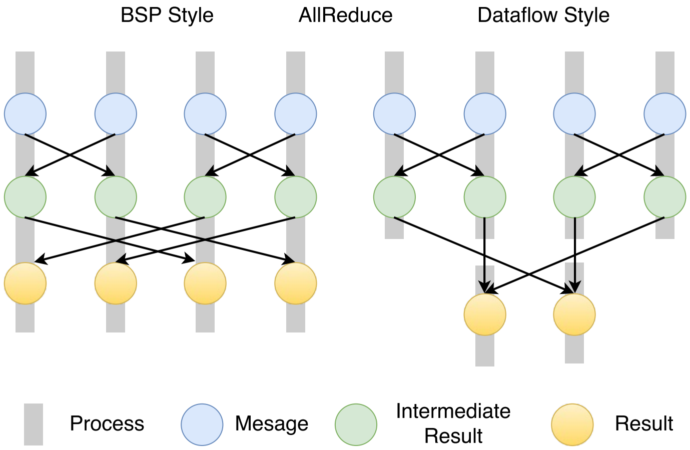


Figure 9.1. Dataflow vs. BSP graph structure

## 9.2. Dataflow for Big Data

Dataflow has been recognized and accepted as the preferred mechanism for processing large data sets. A dataflow program models a computation as a graph with nodes of the graph doing user-defined computations and edges representing the communication links between the nodes. The data flowing through this graph is termed as events or messages. It is important to note that even though by definition dataflow programming means data is flowing through a graph, it may not necessarily be the case physically, especially in batch applications. Big data systems employ different APIs for creating the dataflow graph. For example, Flink and Spark provide distributed data set-based APIs for creating the graph while systems such as Storm and Hadoop provide task-level APIs.

For a dataflow program (DFP) or a bulk synchronous program (BSP) executing in parallel, peer-to-peer communications and collective communications are used for sharing data between parallel tasks. Collective communications define data transfer between a set of tasks in contrast to one-to-one communications as in the case of peer-to-peer. The collective communication patterns as identified by the parallel computing communities are available through MPI [62] implementations. Heavily used collective operations include Broadcast, Scatter, Gather, Reduce, AllGather, and AllReduce [137]. The parallel computing community found that these communications can be optimized for latency and throughput using special algorithms that send the messages among the tasks and determine the routing of messages. These algorithms are termed collective algorithms and are available through MPI implementations.

MPI has been the standard communication API for high performance computing for the last two decades. It boasts a solid API with a mathematical foundation to support highly scalable parallel applications. There are many implementations of the MPI standard available and the mainstream MPI implementations enable applications to scale to hundreds of thousands of cores due to their superior collective communication algorithms, efficient use of memory and support of different networking hardware.

The graph structure of a parallel program is defined by the communications and execution of tasks. One can build every parallel execution graph using basic communication operations such as send and receive. When developing higher level communication patterns as in collective communications, a certain structure of the graph can be assumed in order to make the communication operations efficient. MPI collective operations assume that the tasks producing the data and the receiving tasks are in the same communicator. Big data relaxes these requirements and allows collective operations between any set of tasks.

A dataflow communication pattern defines the edges of an execution graph. For instance, a single node can broadcast a message to multiple nodes in the graph. This means that there is an edge from a source node to every receiving node. Reduce is the opposite of a Broadcast operation where multiple nodes link to a single node. Apart from these operations, Gather, Join and Union are used extensively. Each of these operations can accept keys and group messages and they are termed keyed operations.

## 9.3. Dataflow Communication Requirements

Batch processing deals with complete data sets as opposed to partial data sets as in stream processing. Stream processing does not necessarily imply real-time data analytics and can work on stored data. Batch data processing pipelines for big data are executed stage by stage where whole cluster resources are used by one part of the computation graph at any given time. On the other hand, stream processing executes every part of the graph on a stream of events as needed. To facilitate the execution of the complete graph, different parts of the dataflow graph need to be deployed across machines and the communications are done from one set of tasks to another.

A big data application requires the data to be partitioned in a hierarchical manner due to memory limitations. In general, data is first partitioned according to the number of parallel tasks and then each partition is again split into smaller partitions. This hierarchical approach is implicit in streaming applications, as only a small portion of the data is available at any given time.

Because of the data locality and processing requirements, the computation graph of a dataflow program can contain a different number of parallel tasks at different stages.

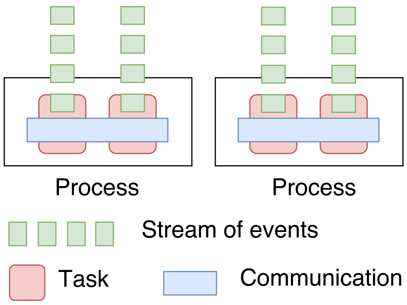


FIGURE 9.2. Model of a communication operator

Furthermore, streaming applications require the nodes of the graph to be deployed in different CPUs in order to handle a higher rate of messages. Big data applications mostly deal with unstructured data like text records. These records do not have specific data sizes defined. Because of this, the communication operations cannot assume the data sizes across the participating tasks. When transferring data, keys are used to group the data.

The communication requirements of dataflow programs for big data are summarized below. Twister:Net is designed according to these requirements.

(1) The communications are between a set of tasks in an arbitrary task graph.

(2) Handle communications larger than available memory.

(3) Dynamic data sizes across communicating tasks.

(4) Batch is modeled as a special case of streaming.

(5) Keys are part of the abstraction.

### 9.4. Twister2:Net

The abstractions of Twister2 include: 1. Data Access, 2. Resource, 3. Communication, 4. Task, 5. Data Management. Each layer is generic and can have multiple implementations to support the needs of specific applications. For example, Twister2 supports different

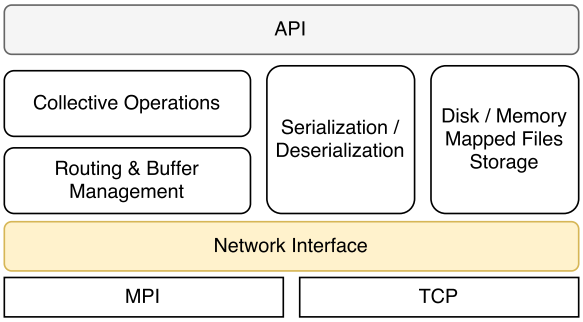

API
Collective Operations
Routing & Buffer Management
Serialization / Deserialization
Disk / Memory Mapped Files Storage
Network Interface
MPI
TCP

FIGURE 9.3. Twister:Net Architecture

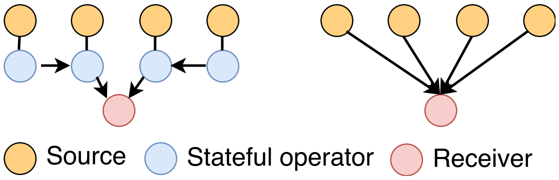


FIGURE 9.4. Reduce Operation as a Graph Extension, **left:** Optimized graph **right:** Default graph

resource schedulers including HPC schedulers like Slurm and big data schedulers such as Aurora [49], Mesos [78] and Kurbenetes [29].

Twister:Net is an open source library [1] implemented using Java language. Its base operators are non-blocking and one can build blocking communications by using the non-blocking semantics. The architecture of Twister:Net is shown in Figure 9.3. The bottom layer is the network access API, which allows one to plug in different networking providers such as OpenMPI [62] and TCP. This layer assumes a reliable channel by the underlying provider. The MPI implementation uses the ISend/IRecv operations to build the dataflow operations and by default we use OpenMPI [62]. TCP implementation uses Java NIO and creates MPI ISend/IRecv semantics to be used by Twister:Net.

[1] https://github.com/DSC-SPIDAL/twister2

On top of this layer the buffer management and data serialization are implemented. The data serialization frameworks are pluggable and we use Kryo as the default serializer for Java objects. Also various communication operations are built on these layers. The disk-based storage can be used by operations when they need to transfer data between disk and memory in order to handle larger data transfers. The top level provides the API layer for the base abstractions.

## 9.5. Communication Model

The dataflow communications are modeled as an extension to the dataflow graph. The generic form of a dataflow communication as modeled by Twister:Net is as follows.

[] Operation(S, D, E, M, T, Optional Op..., C)

KeyedOperation(S, D, E, M, T, K, KT, Optional Op..., C)

S = Source tasks, D = Destination tasks, E = Edges, M = Message, T = Data Type, Op = Optional Stateful operators C = Callback K = Key KT = Key data type

For dataflow style collectives, the source tasks and destinations tasks can be mutually exclusive. Each operation needs an edge identifier (Integer) to distinguish from other operators happening simultaneously. Some operators may require more than one edge number. We support Java Objects and primitive array types as message types and will expand to other formats such as protocol buffers. If keyed operations are used, each message can have a key.

The communication model of Twister:Net is shown in Fig. 9.2. Each operation created in a single process can be shared by multiple tasks in that process and can accept a stream of messages. For streaming cases, the stream is unbounded, while for batch cases bounded stream is assumed with the last message marked as the end of the stream. For the streaming mode of communications, the operators do not buffer data and forward them accordingly. For batch operations, the operators can buffer data to increase the throughput. With batch communication, a special flag is used by sources to signal the end of the flow.

(1) Streaming mode - Each operation only considers a single datum as an entity and an infinite stream of events

(2) Batch mode - Multiple data items are considered into a single operation with a finite stream of events

Furthermore, each of these communication modes can operate with keys. If required, a communication can use the disks in order to handle data sets that do not fit the available memory.

Twister:Net has implemented the following dataflow collective operations; 1. Reduce, 2. AllReduce, 3. KeyedReduce, 4. Gather, 5. AllGather, 6. KeyedGather, 7. Partition, and 8. Broadcast. Fig. 9.4 shows the dataflow graph of reduce and how it is optimized using a tree structure and stateful operators. For operations like Reduce and Gather messages flow through this optimized graph. We implemented the AllReduce operation as a reduce + broadcast and AllGather operation as gather + broadcast. For keyed operations, we create an optimized routing to each destination of the operation. Messages are routed according to the correct destination using these underlying structures. For example for a keyed reduce, we create multiple trees each pointing to a single destination.

**Stateful operations** Communication operators can keep state about streams of messages passing through them. For example, with batch operations, messages can be gathered and presented once the operation completes. For example such stateful operators can be used to create a combiner for Hadoop like gathers.

**Thread Safety** Different threads can progress the communication as well as perform message packing and callback handling. Depending on the underlying channel implementation, the network side of the communication can be thread-safe or not. For example, MPI can be compiled with different levels of thread safety, and if such a compilation is used the channel can be thread-safe.

**Dynamic messages** Unlike, in MPI programs where mostly structured data is used, the dataflow communications deal with unstructured data such as text records. With MPI a user has to take additional steps like knowing the data sizes (which may require additional operations), serialization before using operations like Reduce, Gather on such data. Twister:Net by default support such data and hides the details from the user. Because of this realization, the framework has to allocate memory to receive the incoming messages as user is not aware of the size of the messages the operation receives.

**Buffer Management & Back Pressure** The buffers are managed internally by the framework. When a higher level message is submitted, it is serialized and put into an available buffer. For parallel operations, it is important to balance the communications such that one source cannot overproduce data. At each stage of the communication, the library buffers the data up to a configurable amount. Once these buffers are filled the operations will not accept messages from sources or the network. Buffering can be used to increase the throughput of the operations at the expense of latency. Because of the relaxed constraints of the message sizes, fixed size buffers are employed by sending and receiving sides. If a submitted message does not fit the buffer size used, it is packed into multiple buffers before being sent out. The framework deploys a message length-based protocol to unpack such messages at the receiver.

**Initialization** Since we are targeting for deployment on different environments, Twister:Net can use a pluggable architecture to bootstrap. When used with MPI, the MPI handles the connection management and the framework delegates to MPI. For TCP and other potential transports, a TCP-based bootstrapping can be used. Every process needs to know a master TCP process and its address. This can be a client that submits the job or a master process of the job. The workers of the communication use a simple handshake protocol with the master to send it port numbers which the master distributes among the workers in order for them to know the port numbers and network addresses and thus establish the communication.

**Key** A message can contain a key of any data type supported by Twister:Net. When keys are used, an actual message will contain the key and the content as separate bytes. Each message contains the overall length of the message and the key length if a variable length key is used. If a fixed size key such as a primitive type is used, the key length is not included in the message.

### 9.6. Communication Spilling to Disk

Big data and HPC applications deal with large data sets that sometimes cannot be processed and analyzed in-memory even with large clusters. As a solution, Twister:Net sends data to disk when a configurable amount of in-memory messages are accumulated. We implemented direct file based version and memory mapped files based version using LmdbJava[7] for achieving this feature. When messages are received, they are put into a queue and when this queue becomes full, it is saved into the disk. If key based communication is used, the values are sorted according to keys before saving to disk. Each such buffer is saved to a separate file. Sorting and saving is done by a separate thread. After all the data is received, the saved values are retrieved from the disk or memory mapped files and served to the user operator. For example, a gather operation would collect all the results sent from participating tasks and store them. Further, we found that the LmdbJava implementation doesn't scale to very large data sets in the gigabytes range and used large amounts of memory. Filesystem based approach performed better for some such large operations.

TABLE 9.1. 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 | |

### 9.7. Evaluation

We conducted several micro-benchmarks and developed two applications to compare the performance of Twister:Net with existing frameworks. We compared Twister:Net performance with Apache Spark, Flink, and Heron in different situations as they are designed to process specific workloads. In results, DFW and BSP indicate dataflow results and MPI respectively.

The experiments were conducted in two clusters. One cluster had 16 nodes of Intel Platinum processors with 48 cores in each node and 56Gbps Infiniband and 10Gbps network connections. The other cluster had Intel Haswell processors with 24 cores in each node with 128GB memory, 56Gbps Infiniband, and 1Gpbs Ethernet connections. 32 nodes of this cluster were used for the experiments.

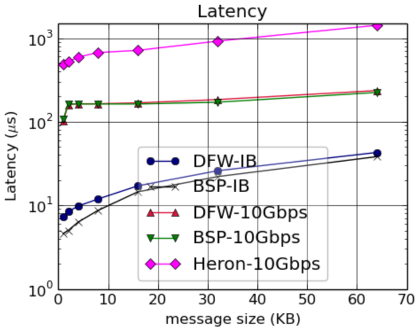


FIGURE 9.5. Latency of MPI and Twister:Net with different message sizes on a two-node setup

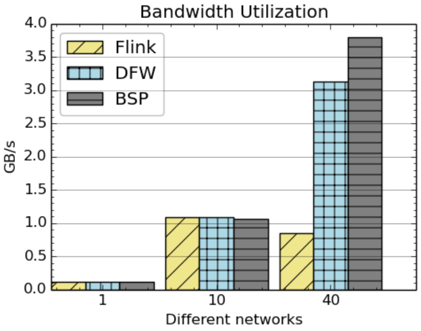


FIGURE 9.6. Bandwidth utilization of Flink, Twister2 and OpenMPI over 1Gbps, 10Gbps and IB

**9.7.1. Micro-benchmarks.** Fig. 9.5 and Fig. 9.6 shows the bandwidth utilization and latency of Twister:Net in a two-node setting where one task sends messages to another task in a different node. Even though we use MPI underneath with other layers added, the latency shows that the overhead is minimal. The bandwidth utilization of Twister:Net is less compared to MPI because a new byte buffer is created for each receiving message. For the MPI test, we did not create a new buffer for every message received, thus increasing the bandwidth.

9.7.1.1. *Streaming Benchmarks.* We have conducted several micro-benchmarks with Apache Heron streaming engine to observe how Twister2 performs in a streaming setting. The first experiment used a data re-balance communication where a set of *N* tasks distributed the data received from another set of *N* tasks. In the next benchmark, we used a more involved reduce communication where a set of tasks sends messages to a single task in a reduce operation. For the last benchmark, the broadcast operation is used. A feedback loop is established from destination tasks to the source tasks to control the flow of the data and to facilitate the latency measurements. Without such a loop, it is harder to measure latency as the tasks are deployed on different machines. The experiments were conducted on 16 nodes with 256 parallel tasks on the Haswell cluster. The feedback loop carries a constant size message with the original message ID.

Fig. 9.7 shows that Twister:Net communication times are well below those of Heron. There are many reasons to explain these results: 1. Heron does not implement optimized communication algorithms, 2. Heron uses stream managers as message routers, meaning the messages generated by parallel instances in a single node go through a single process, 3. It uses both protocol message serializations and Kryo-based object serialization for a single message.

9.7.1.2. *MPI Benchmarks.* We implemented experiments with OpenMPI Java binding to compare Twister:Net performance with direct OpenMPI performance. The experiments were conducted in order to observe whether there are any drastic performance drops compared to OpenMPI operations in a one-to-one mapping setting. We used two tests with one using Reduce operation and another with Gather operation. In one test we use a fixed size integer message while for other we simulate and dynamic object with different sizes. The results show that Twister:Net Reduce operation is slightly slower than that of MPI for fixed size messages. The gather operation of Twister:Net is slower than MPI due to its relaxed buffer management compared to MPI. As a first implementation we believe these are acceptable results as big data systems perform much slower.

9.7.1.3. *Flink Benchmarks.* We compared the performance of Twister:Net with Flink for measuring the total time. Fig. 9.8 shows the results of two streaming operations with Flink and Twister2. This test was conducted on 32 nodes with 640 parallelism. For Reduce operation, each parallel task generates 1000 messages, and for Partition operation each task generates 1 million messages. The total time to finish these messages was measured. The results show that the partition operation of Flink has equal performance to Twister:Net for Ethernet but Twister:Net Reduce operation has far superior performance. Flink doesn't implement optimized reduce operations as in this paper hence the performance is less.

### 9.7.2. Applications.

9.7.2.1. *Terasort.* Terasort is a popular benchmark to measure the performance of data processing systems. With Terasort, the data is partitioned into equal-sized chunks so that each task in the job gets the same amount of data. Next, the algorithm collects a set of sample records per task and sorts this sample set to determine an ordered partitioning for the complete data set. In the third step, the generated global partitioning is used to send records to the correct task. This phase is generally known as a shuffling phase. Finally, after all the records are collected, each task will sort the local set of records and write the results to disk. This will result in a globally sorted data set across all the tasks. The parallel version of Terasort is described in Algorithm [95]. The authors detailed and discussed Terasort implementations with Spark, Flink, and MPI in [95].

The Twister2 implementation of Terasort uses Gather, Broadcast, and Partition operations. Initially, a task reads the data partition assigned to it. Then a Gather operation is performed to collect the sample set of records from every task. This collected sample data set is used to create the ordered partitioning, which is a set of keys of size N-1 where N is the number of tasks. Afterwards, this key set is broadcast to all tasks. In order to shuffle the data to the correct tasks, the Partition operation is used. The data is read part by part from disk and send over the dataflow operation. The receiving records are sorted by the framework using the disks. After all the records are received, every task reads the records and writes the results to the disk. Fig. 9.11 shows the total time of Terasort on 16 nodes with BSP-style implementation, dataflow implementation and Flink. The BSP implementation was performing slightly better than the DFW implementation due to the algorithm used to shuffle the data, which is a rotating shuffle algorithm [95]. Also, it shows the running time of terasort on a 32 node cluster with 1 terabytes and 500 Gigabytes data sets and the dataflow implementation performed equally well compared to BSP implementation.

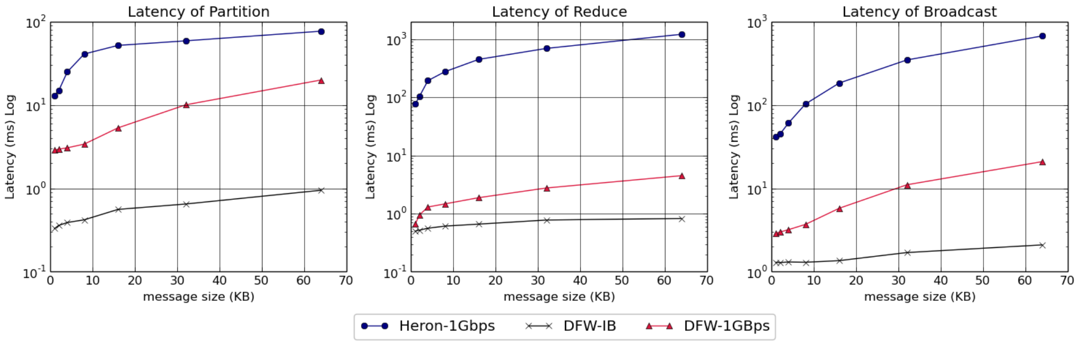


Figure 9.7. Latency of Heron and Twister:Net for Reduce, Broadcast and Partition operations in 16 nodes with 256-way parallelism

9.7.2.2. *K-means.* We implemented the K-means algorithm using Twister:Net dataflow style and compared its performance with a K-means implementation of Spark and a BSP-style MPI implementation. We used the K-means algorithm implemented in Mllib of Spark 2.1.3. The dataflow style uses a task which is invoked by the threads of the process. This tasks computes and outputs the centers which are sent using the AllReduce operation. Once the results of the AllReduce operation available the thread updates the new centers of the task and executes the task again until the given number of iterations are reached.

Fig 9.10 shows the total time for running K-means on a 16-node cluster with Spark, Twister:Net dataflow and BSP-style operations with IB and 10Gbps networks. We use 320 parallel tasks for execution. The tests were conducted with a data set of 2 million points with 2 features and varying number of centers. The algorithm ran for 100 iterations for each test. The dataflow style program is written using Twister:Net performed equal to the BSP-style K-means program. Spark did not perform well for this algorithm because of the rapid creation of tasks for each iteration and the heightened communications.

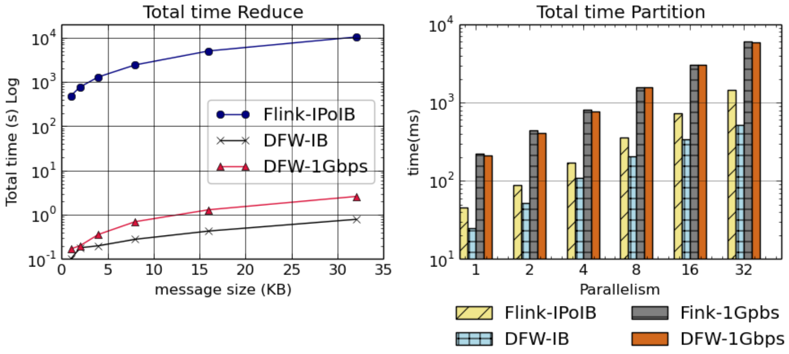


FIGURE 9.8. Total time for Flink and Twister:Net for Reduce and Partition operations in 32 nodes with 640-way parallelism. The time is for 1 million messages in each parallel unit, with the given message size

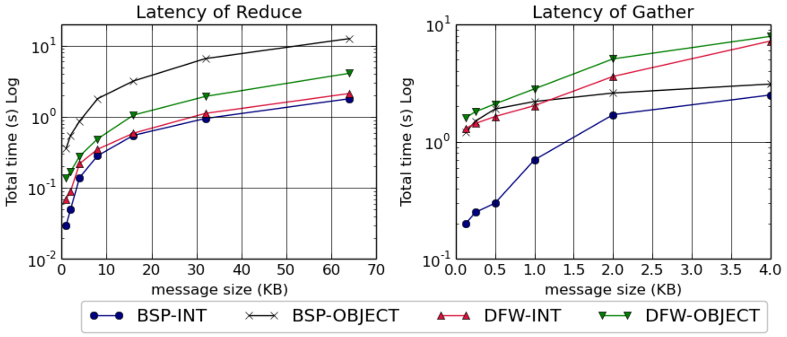


FIGURE 9.9. Latency for OpenMPI and Twister:Net for Reduce and Gather operations in 32 nodes with 256-way parallelism. The time is for 1 million messages in each parallel unit, with the given message size. Infiniband network is used.

## 9.8. SLAM Algorithm with Twister:Net

In this section, we present the implementation of SLAM algorithm described in Chapter 5 with Twister2. The algorithm implementation is shown in Fig. 9.12 where we combine both BSP and Dataflow operations to create a streaming application.

The parallel computation is triggered when events are dispatched from the dispatcher task using a broadcast dataflow communication. Once the events are received byte the computation tasks, they communicate with each other using different types of communications

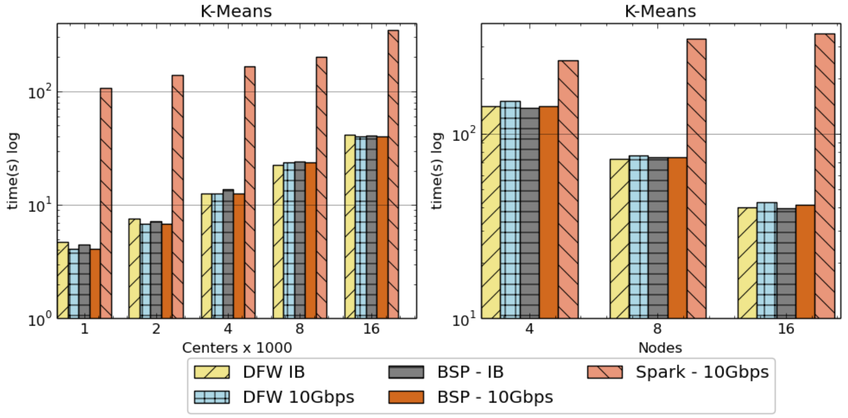


Figure 9.10. Left: K-means job execution time on 16 nodes with varying centers, 2 million points with 320-way parallelism. Right: K-Means wth 4,8 and 16 nodes where each node having 20 tasks. 2 million points with 16000 centers used.

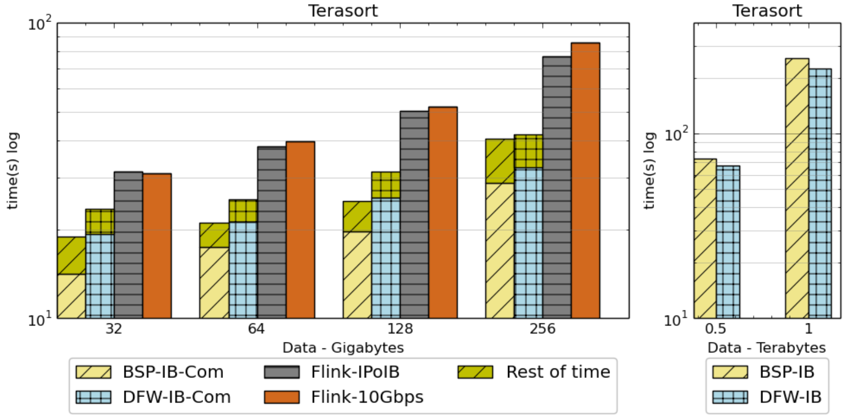


Figure 9.11. Left: Terasort time on a 16 node cluster with 384 parallelism. BSP and DFW shows the communication time. Right: Terasort on 32 nodes with .5 TB and 1TB datasets. Parallelism of 320.

to do the parallel computation. Note that the Storm implementation cannot use such communications and had to go outside of tasks to do this calculation. This is primarily because Storm and other systems don't expose the communications to the task level programming. Fig. 9.13 and Fig. 9.14 shows speedup observed with this new application. The experiments

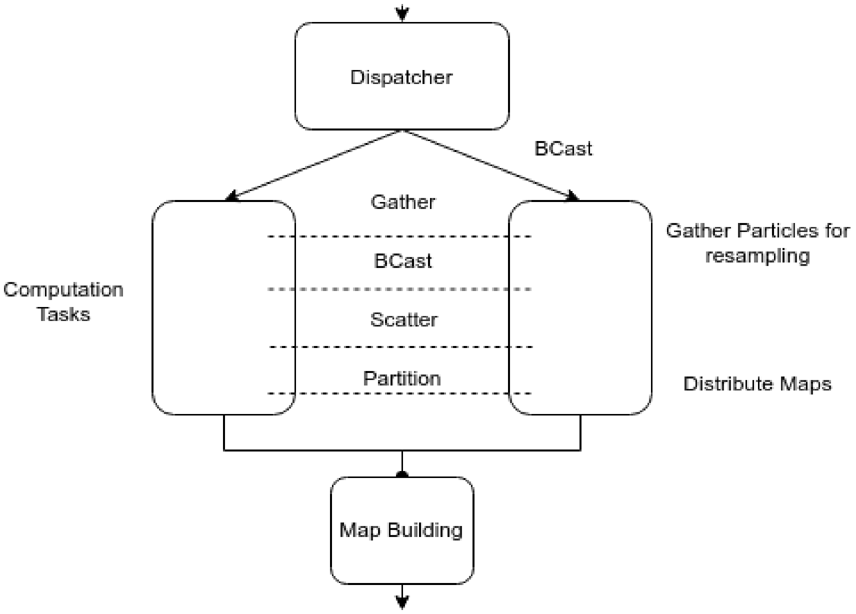


FIGURE 9.12. SLAM Algorithm implemented with Twister2. The parallel computing happens using BSP style communications. The stream of events are handled by the dispatcher and broadcast to the BSP tasks using dataflow communications.

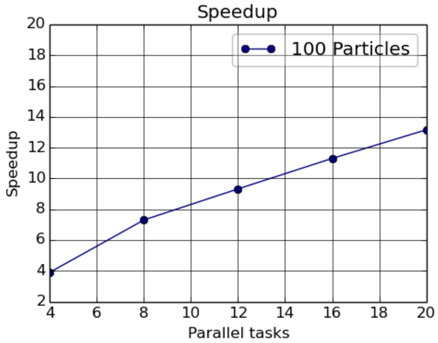


FIGURE 9.13. SLAM speedup with Twister:Net for 180 laser readings and 100 particles.

were conducted on 5 nodes with InfiniBand. The speedup was increased from 14 to about 18 using the Twister2 approach for 640 laser reading case.

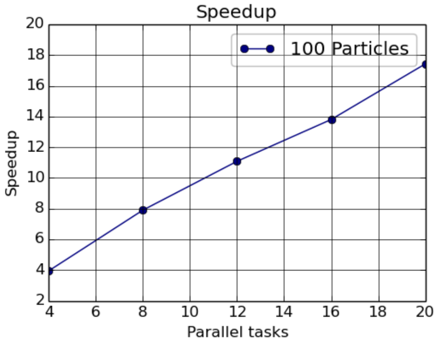


Figure 9.14. SLAM speedup with Twister:Net for 640 laser readings and 100 particles.

## 9.9. Summary

Twister:Net is an optimized big data communication library for both cloud and HPC technologies. With Twister:Net we acknowledge the need to have different types of communications for different applications, particularly streaming, data pipelines and complex algorithms including machine learning. In addition, Twister:Net defines the communication requirements of big data in a separate library without integrating to any particular big data framework. The results indicate that by using Twister:Net as a pure communication library, one can build highly efficient applications. In Twister:Net, we do not consider fault tolerance and leave that to the frameworks using the communication library.

CHAPTER 10

# Resource Scheduling and Task System

Twister2 task system consists of two important components namely resource scheduler and task system. The resource scheduler is responsible for acquiring resources in different cluster environments. This chapter introduces these components briefly as they are been actively developed.

## 10.1. Resource Allocation

The resource allocation layer abstracts the resource scheduling requirements into a common API to allow the integration of different cluster resource managers. Twister2 identifies several key requirements when allocating resources and staging a job on a set of cluster resources. These requirements are based on different cluster environments and resource managers.

(1) Requesting the resources from a cluster manager

(2) Uploading the artifacts of a job into the worker nodes

(3) Setting up logging on different worker nodes

(4) Configuring persistence storage for worker processes

(5) Binding worker processes to CPU's and sockets

(6) Worker discovery service

Each of these requirements is satisfied in different ways by different resource managers and each plugin needs to implement these API's in order for Twister2 to leverage these functions. Some functions such as binding workers to CPU's are optional while others such as uploading, logging are mandatory. By abstracting out the resource scheduler requirements Twister2 provides a rich environment to integrate with future resource managers. At the moment it supports Mesos, Aurora, Slurm, and Nomad scheduler.

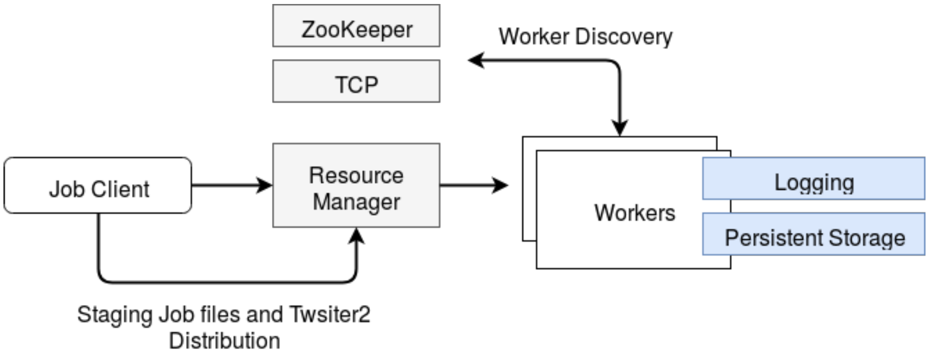


FIGURE 10.1. Runtime architecture of Twister2

The resource scheduler is responsible for starting the base processes in allocated cluster resources and these processes are termed as workers. After the workers come online, they can discover each other using the worker discovery service. Worker discovery can be implemented with different ways and by default Twister2 provides ZooKeeper and TCP based discovery. At this point the workers can use the Twister:Net to establish connections between the processes.

## 10.2. Task System

The task system consists of task graph, task scheduler, and executor. User is responsible for developing the task graph and scheduler allocates its nodes into the workers. Then executor can use threads to execute them with different strategies.

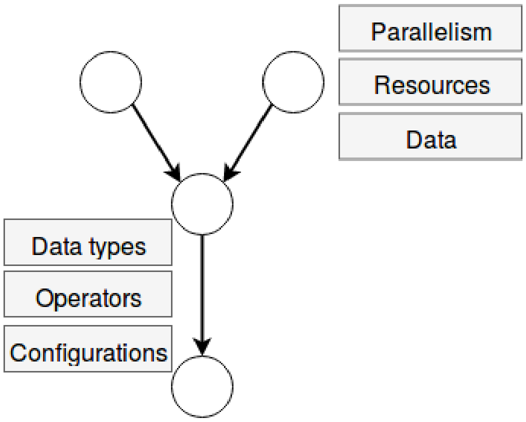


Figure 10.2. The dataflow graph annotated with different requirements

### 10.2.1. Task Graph.

The task graph is generated as a generic graph with vertices representing user code and edges representing communications. Each edge of the graph contains information about the type of communication used. Also a user can give a communication operator as a function to the edge in case of operations like reduce.

Each vertex of the graph contains user code and auxiliary information to support scheduling. The parallelism of the task is an important information while CPU, memory and disk information can be provided. Some tasks allow additional information such as data dependencies to be specified in order to achieve data locality aware scheduling. An example graph with configurations for tasks and edges is shown in Fig. 10.2.

**10.2.2. Task Scheduler.** The task scheduler takes a task graph, available set of resources and schedules the individual tasks of this graph into different workers. Task scheduling is designed to take into account the auxiliary information about a task such as its memory requirements, CPU requirements and data requirements. The architecture allows different task schedulers to be used depending on the application requirements. We support round-robin and first fit scheduling at the moment and working on data locality aware scheduling.

At the moment the task scheduler is running in each worker node and we are planning to support central task schedulers in the future. This will allow us to run hierarchical scheduling with graphs containing coarse grained and fine grained dataflows.

**10.2.3. Task Execution.** The task execution of Twister2 takes a task graph and task schedule and creates an execution plan. The execution plan consists of task instances that are created from the task vertices and the communication links. This execution plan is executed by each of the worker using a configurable execution strategy. Task execution uses threads to execute tasks inside a worker.

**10.2.4. Summary.** As per the design principles of the Twister2 the resource scheduler and task system are modular and have pluggable components. For resource scheduler, every aspect such as logging, staging files, and worker discovery are developed as plugins. The task system supports different annotations at the node level and edge level for the task scheduler and communications to take decisions. Task scheduling algorithms are implemented as plugins to the system.

CHAPTER 11

# Conclusion

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 thesis, 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 component 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 enables 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 [4].

The components of the Twister2 are developed as an open source project and the thesis described the details of these components and how they interact and integrate. The thesis especially focused on Twister:Net which is an optimized big data communication library for both cloud and HPC technologies. With Twister:Net we acknowledge the need to have different types of communications for different applications, particularly streaming, data pipelines and complex algorithms including machine learning. In addition, Twister:Net defines the communication requirements of big data in a separate library without integrating to any particular big data framework. The results indicate that by using Twister:Net as a pure communication library, one can build highly efficient applications. The experimental results of Twister:Net indicates that big data systems need to implement optimized communications. Also, it presents a communication model that can be implemented by HPC systems for data intensive operations.

Twister2 has extensive support for different deployment strategies including docker, Mesos, Nomad and HPC schedulers like Slurm. This pluggable architecture will allow it to be used in both cloud and HPC environments seemlesly.

Twister2 provides the functionality to view different parts of an application separately and use different abstraction in different parts of the same dataflow. We believe the flexibility and the robust functionality offered by Twister2 at each layer will enable the users to develop rich applications without constraining to frameworks with fixed APIs and functions.

## 11.1. Future Work

Operations such as reduction require information inside a complex message. This can be costly for high level object-based messages where we need to de-serialize the complete message. Having an option to look at only the parts of the messages without de-serialization can decrease the latency and increase the throughput. It is possible to build such a framework with binary protocols like Google Protocol Buffers. The current big data communications and operations do not have standard semantics available, meaning different frameworks have slightly different APIs. It would be better to have standard APIs so that users can work with uniform APIs.

We are working on incorporating more collective algorithms into the library that can perform well in different circumstances, like throughput critical applications. At the moment Twister:Net supports only MPI-based and TCP-based communications, but we intend to include further networks. There are other dataflow operations such as Unions and Joins that we are working to integrate with the library. Twister:Net relies on flow control at the network layer for handling back-pressure at the application level. When multiple communications share the same underlying network channel between nodes, one operation can slow down the others. The communication library needs a back-pressure mechanism as an add-on feature that can be used as required. We would like to integrate the dataflow-style communications directly into an MPI implementation like OpenMPI.

Having the communication implemented directly on top of an RDMA library such as Photon [96] would be useful and remove the dependency to MPI. The communication library can be used as a standalone library with other frameworks. Twister:Net can be used to replace the communication fabric of existing frameworks such as Heron.

We are positioning resource scheduling as a standalone component that can be used by any distributed system to launch and manage tasks using its clean set of APIs. The APIs are still evolving and will be ready in few iterations of improvements. Task layer will implement different scheduling algorithms and support both central and distributed scheduling.

We are working on the fault tolerance of the system with different approaches to specific applications. For example, streaming fault tolerance will be implemented using barrier injection and batch case will be handled with check-pointing based mechanism.

# Bibliography

[1] Apache Flink. `https://flink.apache.org/`. Accessed: 2016.

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

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

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

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

[6] IntelÂő Data Analytics Acceleration Library (IntelÂő DAAL). Accessed: January 02 2017.

[7] LMDB Java.

[8] Spark Communication. Accessed: March 02 2018.

[9] Spark Improvements 1.1. Accessed: March 02 2018.

[10] Storm and Heron Communication Concepts. Accessed: March 02 2018.

[11] The Design of the Borealis Stream Processing Engine., author=Abadi, Daniel J and Ahmad, Yanif and Balazinska, Magdalena and Cetintemel, Ugur and Cherniack, Mitch and Hwang, Jeong-Hyon and Lindner, Wolfgang and Maskey, Anurag and Rasin, Alex and Ryvkina, Esther and others, booktitle=CIDR, volume=5, pages=277–289,

year=2005.

[12] MPI: A Message-Passing Interface Standard Version 3.0, 2012. Technical Report.

[13] Apache Apex: Enterprise-grade unified stream and batch processing engine, 2017.

[14] Twister2:Design of aBig Data Toolkit, 2017. Technical Report.

[15] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases*, 12(2):120–139, 2003.

[16] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

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

[18] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[19] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam 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, August 2015.

[20] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The spraylist: A scalable relaxed priority queue. *SIGPLAN Not.*, 50(8):11–20, January 2015.

[21] George Almási, Philip Heidelberger, Charles J Archer, Xavier Martorell, C Chris Erway, José E Moreira, B Steinmacher-Burow, and Yili Zheng. Optimization of MPI collective communication on BlueGene/L systems. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 253–262. ACM, 2005.

[22] Michael Anderson, Shaden Smith, Narayanan Sundaram, Mihai Capotă, Zheguang Zhao, Subramanya Dulloor, Nadathur Satish, and Theodore L. Willke. bridging the gap between hpc and big data frameworks.

[23] Quinton Anderson. *Storm Real-time Processing Cookbook*. Packt Publishing Ltd, 2013.

[24] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pages 207–218. ACM, 2013.

[25] InfiniBand Trade Association et al. *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association, 2000.

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

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

January 2017.

[28] Motti Beck and Michael Kagan. Performance Evaluation of the RDMA over Ethernet (RoCE) Standard in Enterprise Data Centers Infrastructure. In *Proceedings of the 3rd Workshop on Data Center - Converged and Virtual Ethernet Switching*, DC-CaVES '11, pages 9–15. International Teletraffic Congress, 2011.

[29] D. Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, Sept 2014.

[30] Mark S Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D Underwood, and Robert C Zak. Intel® Omni-path Architecture: Enabling Scalable, High Performance Fabrics. In *High-Performance Interconnects (HOTI), 2015 IEEE 23rd Annual Symposium on*, pages 1–9. IEEE, 2015.

[31] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali 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.

[32] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. fog computing and its role in the internet of things.

[33] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack 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.

[34] Peng Boyang, Hosseini Mohammad, and Hong Zhihao. R-storm: Resource-aware scheduling in storm. Annual Middleware Conference. ACM, 2015.

[35] 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.

[36] Thilina Buddhika and Shrideep Pallickara. NEPTUNE: Real Time Stream Processing for Internet of Things and Sensing Environments.

[37] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Data Engineering*, page 28, 2015.

[38] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *arXiv preprint arXiv:1506.08603*, 2015.

[39] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Distributed QoS-aware scheduling in storm. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 344–347. ACM, 2015.

[40] Josiah L Carlson. *Redis in Action*. Manning Publications Co., 2013.

[41] C. T. Chen, L. J. Hung, S. Y. Hsieh, R. Buyya, and A. Y. Zomaya. Heterogeneous job allocation scheduler for hadoop mapreduce using dynamic grouping integrated neighboring search. *IEEE Transactions on Cloud Computing*, PP(99):1–1, 2017.

[42] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y. Ng, and Kunle 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.

[43] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An Evaluation of Global Address Space Languages: Co-array Fortran and Unified Parallel C. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 36–47, New York, NY, USA, 2005. ACM.

[44] Javier Conejero, Sandra Corella, Rosa M Badia, and Jesus Labarta. task-based programming in compss to converge from hpc to big data.

[45] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, Jan 1998.

[46] Jan De Leeuw and Patrick Mair. Multidimensional scaling using majorization: SMACOF in R. *Department of Statistics, UCLA*, 2011.

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

[48] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G Bruce Berriman, John Good, et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[49] R. DelValle, G. Rattihalli, A. Beltre, M. Govindaraju, and M. J. Lewis. Exploring the Design Space for Optimizations with Apache Aurora and Mesos. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 537–544, June 2016.

[50] Jos Dirksen. *Learning Three. js: the JavaScript 3D library for WebGL*. Packt Publishing Ltd, 2013.

[51] Celestine Dünner, Thomas Parnell, Kubilay Atasu, Manolis Sifalakis, and Haralampos Pozidis. Understanding and Optimizing the Performance of Distributed Machine Learning Applications on Apache Spark. *arXiv preprint arXiv:1612.01437*, 2016.

[52] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey 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, 2010. ACM.

[53] Saliya Ekanayake, Supun Kamburugamuve, and Geoffrey 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.

[54] Saliya Ekanayake, Supun Kamburugamuve, Pulasthi Wickramasinghe, and Geoffrey C Fox. Java thread and process performance for parallel machine learning on multicore hpc clusters. In *Big Data (Big Data), 2016 IEEE International Conference on*, pages 347–354. IEEE, 2016.

[55] Graham Fagg and Jack Dongarra. Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world. *Recent advances in parallel virtual machine and message*

*passing interface*, pages 346–353, 2000.

[56] Ahmad Faraj and Xin Yuan. Automatic generation and tuning of MPI collective communication routines. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 393–402. ACM, 2005.

[57] Ahmad Faraj, Xin Yuan, and David Lowenthal. STAR-MPI: self tuned adaptive routines for MPI collective operations. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 199–208. ACM, 2006.

[58] G.C. 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, May 2015.

[59] Geoffrey Fox et al. FutureGrid: A reconfigurable testbed for Cloud, HPC and Grid Computing. *Contemporary High Performance Computing: From Petascale toward Exascale*, 2013.

[60] Geoffrey Fox, Judy Qiu, Shantenu Jha, Saliya Ekanayake, and Supun Kamburugamuve. Big data, simulations and hpc convergence. In *Workshop on Big Data Benchmarks*, pages 3–17. Springer, 2015.

[61] Maosong Fu, Ashvin Agrawal, Avrilia Floratou, Graham Bill, Andrew Jorgensen, Mark Li, Neng Lu, Karthik Ramasamy, Sriram Rao, and Cong Wang. Twitter heron: Towards extensible streaming engines. *2017 IEEE International Conference on Data Engineering*, Apr 2017.

[62] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *European Parallel Virtual Machine/Message Passing Interface UsersâĂŹ Group Meeting*, pages 97–104. Springer, 2004.

[63] S. GallenmÃijller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle. Comparison of frameworks for high-performance packet IO. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 29–38, May 2015.

[64] Dennis Gannon, Roger Barga, and Neel Sundaresan. Cloud Native Applications. *IEEE Cloud Computing Magazine special issue on cloud native computing*, 3001to be published.

[65] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol 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.

[66] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S Yu, and Myungcheol Doo. SPADE: the system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1123–1134. ACM, 2008.

[67] 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.

[68] Alex Gittens, Aditya Devarakonda, Evan Racah, Michael Ringenburg, Lisa Gerhardt, Jey Kottalam, Jialin Liu, Kristyn Maschhoff, Shane Canon, Jatin Chhugani, et al. Matrix factorizations at scale: A comparison of scientific data analytics in Spark and C+ MPI using three case studies. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 204–213. IEEE, 2016.

[69] Bruno D Gouveia, David Portugal, and Lino Marques. Speeding up Rao-blackwellized particle filter SLAM with a multithreaded architecture. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1583–1588, 2014.

[70] Richard L Graham and Galen Shipman. MPI support for multi-core architectures: Optimized shared memory collectives. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 130–140. Springer, 2008.

[71] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. Improving grid-based SLAM with Rao-Blackwellized particle filters by adaptive proposals and selective resampling. In *IEEE International Conference on Robotics and Automation*, pages 2432–2437, 2005.

[72] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. Improved techniques for grid mapping with Rao-Blackwellized particle filters. *IEEE Transactions on Robotics*, 23(1):34–46, 2007.

[73] 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.

[74] Thilina Gunarathne, Jian Qiu, and Dennis Gannon. Towards a Collective Layer in the Big Data Stack. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 236–245. IEEE, 2014.

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

[76] Jing Han, Haihong E, Guan Le, and Jian Du. Survey on nosql database. In *2011 6th International Conference on Pervasive Computing and Applications*, pages 363–366, Oct 2011.

[77] Hengjing He, Supun Kamburugamuve, and Geoffrey C Fox. Cloud based real-time multi-robot collision avoidance for swarm robotics.

[78] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[79] 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.

[80] Torsten Hoefler, Timo Schneider, and Andrew 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.

[81] David Hollman, Jonathon Lifflander, Jeremiah Wilke, Nicole Slattengren, Aram Markosyan, Hemanth Kolla, and Francesco Rizzi. Darma v. beta 0.5, Mar 2017.

[82] Louis Hugues and Nicolas Bredeche. Simbad: an autonomous robot simulation package for education and research. In *From Animals to Animats 9*, pages 831–842. 2006.

[83] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, 2010.

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

[85] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Uğur Çetintemel, Michael Stonebraker, and Stan Zdonik. High-availability algorithms for distributed stream processing. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 779–790. IEEE, 2005.

[86] 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.

[87] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 407–420, New York, NY, USA, 2015. ACM.

[88] Laxmikant V. Kale and Sanjeev 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.

[89] S. Kamburugamuve, S. Ekanayake, M. Pathirage, and G. Fox. Towards High Performance Processing of Streaming Data in Large Data Centers. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1637–1644, May 2016.

[90] Supun Kamburugamuve, Leif Christiansen, and Geoffrey Fox. A Framework for Real Time Processing of Sensor Data in the Cloud. *Journal of Sensors*, 2015, 2015.

[91] Supun Kamburugamuve, Geoffrey Fox, David Leake, and Judy Qiu. Survey of distributed stream processing for large stream sources. Technical report, Indiana University, 2013.

[92] Supun Kamburugamuve, Geoffrey Fox, David Leake, and Judy Qiu. Survey of distributed stream processing for large stream sources, 2016.

[93] Supun Kamburugamuve, Hengjing He, Geoffrey Fox, and David Crandall. Cloud-based Parallel Implementation of SLAM for Mobile Robots.

[94] Supun Kamburugamuve, Karthik Ramasamy, Martin Swany, and Geoffrey Fox. Low latency stream processing: Apache heron with infiniband & intel omni-path. 2017.

[95] Supun Kamburugamuve, Pulasthi Wickramasinghe, Saliya Ekanayake, and Geoffrey C Fox. Anatomy of machine learning algorithm implementations in MPI, Spark, and Flink. *The International Journal of High Performance Computing Applications*, 0(0):1094342017712976, 2017.

[96] 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.

[97] Adrian Klein, Fuyuki Ishikawa, and Shinichi Honiden. Towards Network-aware Service Composition in the Cloud. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 959–968, New York, NY, USA, 2012. ACM.

[98] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases*, 2011.

[99] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[100] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth 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.

[101] Rainer Kümmerle, Bastian Steder, Christian Dornhege, Michael Ruhnke, Giorgio Grisetti, Cyrill Stachniss, and Alexander Kleiner. On measuring the accuracy of SLAM algorithms. *Autonomous Robots*, 27(4):387–407, 2009.

[102] Fan Liang, Chen Feng, Xiaoyi Lu, and Zhiwei Xu. Performance benefits of DataMPI: a case study with BigDataBench. In *Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware*, pages 111–123. Springer, 2014.

[103] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K Panda. High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming*, 32(3):167–198, 2004.

[104] Ruirui Lu, Gang Wu, Bin Xie, and Jingtong Hu. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*, pages 69–78. IEEE, 2014.

[105] 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, Oct 2013.

[106] X. Lu, F. Liang, B. Wang, L. Zha, and Z. Xu. Datampi: Extending mpi to hadoop-like big data computing. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 829–838, May 2014.

[107] 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.

[108] Xiaoyi Lu, Nusrat S Islam, Md Wasi-Ur-Rahman, Jithin Jose, Hari Subramoni, Hao Wang, and Dhabaleswar K Panda. High-performance design of hadoop rpc with rdma over infiniband. In *2013 42nd International Conference on Parallel Processing*, pages 641–650. IEEE, 2013.

[109] Xiaoyi Lu, Fan Liang, Bing Wang, Li Zha, and Zhiwei Xu. DataMPI: extending MPI to hadoop-like big data computing. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 829–838. IEEE, 2014.

[110] Xiaoyi Lu, Md Wasi Ur Rahman, Nahina Islam, Dipti Shankar, and Dhabaleswar K Panda. Accelerating spark with RDMA for big data processing: Early experiences. In *High-Performance Interconnects (HOTI), 2014 IEEE 22nd Annual Symposium on*, pages 9–16. IEEE, 2014.

[111] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[112] Bertram LudÃd'scher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[113] Suresh Marru, Lahiru Gunathilake, Chathura Herath, Patanachai Tangchaisin, Marlon Pierce, Chris Mattmann, Raminder Singh, Thilina Gunarathne, Eran Chinthaka, Ross

Gardler, et al. Apache airavata: a framework for distributed applications and computational workflows. In *Proceedings of the 2011 ACM workshop on Gateway Computing Environments*, pages 21–28. ACM, 2011.

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

[115] T. G. Mattson, R. Cledat, V. CavÃĺ, V. Sarkar, Z. BudimliÄĞ, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, Min 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.

[116] Jorge J Moré. The Levenberg-Marquardt algorithm: implementation and theory. In *Numerical analysis*, pages 105–116. Springer, 1978.

[117] Brandon L. Morris and Anthony Skjellum. Mpignite: An mpi-like language and prototype implementation for apache spark. *CoRR*, abs/1707.04788, 2017.

[118] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. naiad: A timely dataflow system.

[119] M. A. U. Nasir, G. De Francisci 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.

[120] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, David García-Soriano, Nicolas Kourtellis, and Marco Serafini. The Power of Both Choices: Practical Load Balancing for Distributed Stream Processing Engines. *arXiv preprint arXiv:1504.00788*, 2015.

[121] 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.

[122] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177. IEEE, 2010.

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

[124] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, volume 3, 2009.

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

[126] M. J. Rashti and A. Afsahi. 10-gigabit iwarp ethernet: Comparative performance analysis with infiniband and myrinet-10g. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, March 2007.

[127] Jorge L. Reyes-Ortiz, Luca Oneto, and Davide Anguita. Big Data Analytics in the Cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf. *Procedia Computer Science*,

53:121 – 130, 2015. INNS Conference on Big Data 2015 Program San Francisco, CA, USA 8-10 August 2015.

[128] Luigi Rizzo. Netmap: a novel framework for fast packet I/O. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 101–112, 2012.

[129] Emilia Rosti, Giuseppe Serazzi, Evgenia Smirni, and Mark S. Squillante. Models of parallel applications with large computation and i/o requirements. *IEEE Trans. Softw. Eng.*, 28(3):286–307, March 2002.

[130] P. C. Roth, D. C. Arnold, and B. P. Miller. Mrnet: A software-based multicast/reduction network for scalable tools. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 21–21, Nov 2003.

[131] Yang Ruan and Geoffrey Fox. A Robust and Scalable Solution for Interpolative Multidimensional Scaling with Weighting. In *Proceedings of the 2013 IEEE 9th International Conference on e-Science*, ESCIENCE '13, pages 61–69, Washington, DC, USA, 2013. IEEE Computer Society.

[132] Yang Ruan, Geoffrey L House, Saliya Ekanayake, Ursel Schutte, James D Bever, Haixu Tang, and Geoffrey Fox. Integration of clustering and multidimensional scaling to determine phylogenetic trees as spherical phylograms visualized in 3 dimensions. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 720–729. IEEE, 2014.

[133] A. Sodani. Knights landing (KNL): 2nd Generation Intel xAE; Xeon Phi processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–24, Aug 2015.

[134] Per Stenström, Truman Joe, and Anoop Gupta. Comparative performance evaluation of cache-coherent numa and coma architectures. *SIGARCH Comput. Archit. News*, 20(2):80–91, April 1992.

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

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

[137] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

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

[139] Brian Tierney, Ezra Kissel, Martin Swany, and Eric Pouyoul. Efficient data transfer protocols for big data. In *E-Science (e-Science), 2012 IEEE 8th International Conference on*, pages 1–9. IEEE, 2012.

[140] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[141] Kenton Varda. Protocol buffers: GoogleâĂŹs data interchange format. *Google Open Source Blog, Available at least as early as Jul*, 2008.

[142] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[143] Alvaro Videla and Jason JW Williams. *RabbitMQ in action: distributed messaging for everyone*. Manning, 2012.

[144] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

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

[146] Jeffrey E Wieselthier, Gam D Nguyen, and Anthony Ephremides. On the construction of energy-efficient broadcast and multicast trees in wireless networks. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 585–594. IEEE, 2000.

[147] Willow Garage. Turtlebot. *http://turtlebot.com/*.

[148] Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, et al. The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. *Nucleic acids research*, page gkt328, 2013.

[149] Andy B. Yoo, Morris A. Jette, and Mark Grondona. ”SLURM: Simple Linux Utility for Resource Management”,.

[150] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion 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*, pages 2–2. USENIX Association, 2012.

[151] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion 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.

[152] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.

[153] 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.

[154] B. Zhang, Y. Ruan, and J. Qiu. Harp: Collective communication on hadoop. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pages 228–233, March 2015.

[155] B. Zhang, Y. Ruan, and J. Qiu. Harp: Collective communication on hadoop. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pages 228–233, March 2015.

[156] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 523–536, New York, NY, USA, 2015. ACM.