Storm on Multi-core

Mark Nemec

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2015

Abstract

In recent years the number of real-time data stream processing applications has been growing. However, data stream processing has been a province of distributed systems such as Apache Storm. This is because commodity hardware clusters offer good scale-out properties.

On the other hand, chip makers have been able to pack more and more transistors into a single processor chip. This resulted in a steady increase in the number of cores in a single processor chip. Furthermore, it is possible that scale-out processors with high number of processor cores will be used for future throughput-based applications.

In this report we present Storm-MC, an API-compatible port of Apache Storm for multi-core. This system combines an easy to use API with implementation tailored to multi-core environments. This allows us to take existing applications written with Storm in mind and run them efficiently on a single server.

Through a series of benchmarks we show that running applications on Storm-MC can provide substantial improvement in throughput (up to 3.3x) compared to running them on Apache Storm in local mode.

Acknowledgements

I would like to thank my supervisor, Dr. Stratis Viglas, for providing guidance every step of the way.

Furthermore, I would like to thank my parents who supported me during my studies at the University of Edinburgh. Without them this project would not be possible.

Finally, I would like to thank Daniela Kellerova who kept me sane while I worked on this project.

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Motivation

In recent years, there has been an explosion of cloud computing software. After Google published their paper on MapReduce [1], many new open-source systems for distributed computation have emerged, most notably Apache Hadoop [2] for batch processing and Apache Storm [3] for real-time data stream processing.

These systems split the work that needs to be carried out and distribute it across nodes of a commodity hardware cluster. Commercial companies and researchers have been able to utilise these systems and create distributed systems which can accomplish things that would not be otherwise possible [4], [5]. This has mostly been allowed by low price and good scale-out properties of commodity hardware.

At the same time, chip makers have been increasing the number of cores in processors and now we are at a point where servers with 10-core processors are considered standard. Moreover, most high-end servers have multiple processor sockets thus increasing the parallelisation possible with a single machine even further.

The number of cores is increasing but the price of highly parallel machines has gone down. In 2008, a typical Hadoop node had two dual-core processors and 4 GB of random access memory (RAM) [6]. Nowadays, a server with two eight-core processors and 256 GB of RAM can be purchased for roughly $10,000 USD [6]. Hence a single server today might have better processing power than a small cluster from a few years ago. If this trend continues there will be processors with even more cores in the near future with higher processing power than most clusters today.

Moreover, tiled processors have emerged as competitors to traditional processors in throughput-based computations [7]. These processors use a large number of tiles connected by an on-chip network and even though the single-thread performance of each tile is lower than the performance of a conventional core, the increased parallelism yields higher throughput [8].

Seeing these trends, we believe there is a place for a real-time data stream processing system running on a single multi-core machine.

It is generally believed that writing parallel software is hard. The traditional techniques of message passing and shared memory require the programmer to manage concurrency at a fairly low level. On the other hand, Apache Storm has become the *de facto* tool used in stream processing on clusters because of its simple application programming interface (API). Furthermore, according to Storm's "Powered By" page there are tens of companies already using Storm to process their real-time data streams [9]. We think that Storm's popularity and easy to use API makes it the ideal candidate for porting to multi-core.

1.2 Project Contributions

The main idea of this project is to take the existing Apache Storm project and port it to multi-core. This is implemented in Storm-MC - a library with an API compatible with Apache Storm. This compatibility enables programmers to take an existing application written with Apache Storm in mind and run it on a single multi-core server using Storm-MC. This way, they can avoid network latency and enjoy the substantial performance improvements of a shared-memory environment.

Through a series of benchmarks we show that with its simpler design, Storm-MC offers substantial improvement in throughput for data stream processing applications over Apache Storm running in local mode.

1.3 Structure of the Report

The remainder of the report is structured as follows:

- Chapter 2 presents an overview of related literature and gives background on data stream processing and multi-core architectures.
- Chapter 3 explains the concepts used in Apache Storm as well as the architecture of a Storm cluster.
- Chapter 4 describes how Apache Storm was ported to multi-core and explains the design and implementation of Storm-MC.
- Chapter 5 presents results of benchmarking Storm-MC against Apache Storm running in local mode.
- Chapter 6 summarises this report, presents considerations for future work, and lists challenges encountered while building Storm-MC.

Chapter 2

Literature Review

The following chapter explains the concept of a data stream (2.1), discusses advantages and disadvantages of using multi-core over clusters (2.2), gives an overview of previous work on Apache Storm (2.3), and discusses similar efforts for porting distributed systems to multi-core (2.4).

2.1 Data Stream

Golab and Özsu [10] define data stream as as "a real-time, continuous, ordered (implicitly by arrival time or explicitly by timestamp) sequence of items. It is impossible to control the order in which items arrive, nor is it feasible to locally store a stream in its entirety."

A data stream processing system allows real-time analysis of data streams. Hence, users of the system can be quick to react to changes and are able to follow trends as they happen in real time.

2.1.1 Comparison to a Database Management System

Historically, data has been stored in database management systems (DBMS) where it was later analysed, the assumption being that there would be enough disk space to contain the data. This approach fits many purposes but real-time applications have recently started feeling the need to analyse huge amounts of rapidly changing data on the fly.

This has brought on the advent of data stream processing. Several stream-processing systems have emerged such as Apache Storm [3], Apache Spark [11], and Apache S4 [12]. These systems, ran on a cluster, provide the programmer with abstractions which greatly simplify writing real-time data stream processing applications.

Whereas DBMSs excel at getting an exact answer to a query, data streams usually provide an approximate answer. The answer is approximate because it is usually

correct only within a certain window of time, because the query is simplified so it can be ran in one pass, or because sampling is used which does not include all events. A typical data stream analysis using windows and sampling is shown in Figure 2.1.

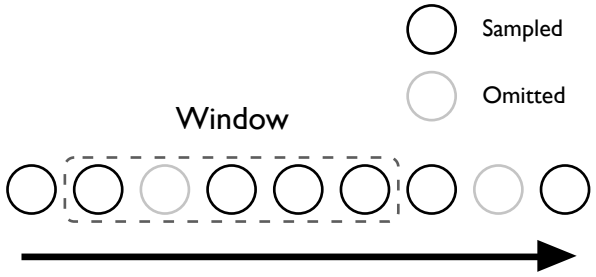


Figure 2.1: Stream querying.

2.1.2 Querying a Data Stream

The assumption behind using a time window is that users of a real-time system are most likely interested in the most recent events. Sampling, on the other hand, is used to reduce the number of events used to answer a query [13]. This way the system can keep up with the data and stay real-time.

Even though an answer to a query might be only approximate it can have great value because the query is answered at the right time. Furthermore, even though the query may only run on a subset of the data it is still possible to detect trends or system failures. For example, Twitter is using Apache Storm to perform real-time analysis on millions of events per second with their analytics product Answers [4].

In research, several techniques have been developed to enable real-time data stream mining. For example, the MOA environment presented in [5] which enables running online machine learning algorithms using the WEKA machine learning workbench [14]. In [15] authors present a real-time Opinion Mining (sentiment analysis) system built with Apache Storm and the Twitter Streaming API.

2.2 Multi-core

Akhter and Roberts [16] define multi-core processor as a processor that has two or more independent execution cores. This means that every thread has a hardware execution environment entirely to itself which enables threads to run in a truly parallel manner.

Running an application on multi-core can in the best case produce a speedup equivalent to the number of cores. The best case is when an application is embarrassingly parallel i.e. there is no inter-thread communication. Even though data stream processing is not an embarrassingly parallel problem, running a producer-consumer application on multiple cores can produce significant speedup, as shown in [17].

2.2.1 Advantages Over Clusters

There are several reasons why someone might prefer to deploy their data stream application to a single multi-core machine rather than a cluster:

Communication Overhead
The latency of over-the-network communication is significantly higher than that of processor cores communicating within a single machine.

Lower Cost than a Data Centre
To run a distributed system on a cluster one would usually first need to own a data centre. This comes with a high capital cost and increased maintenance costs compared to owning a single server.

More Control than with a Cloud Provider
Alternatively, one could rent out nodes on cloud computing services such as Amazon AWS or Rackspace. While the cost of such services is acceptable the user does not have full control over their system. This might be essential for high-security applications.

2.2.2 Disadvantages Over Clusters

On the other hand, there are certain disadvantages in deploying a data stream application to a single multi-core machine rather than a cluster:

Horizontal Scaling
Commodity hardware clusters offer better horizontal scaling than a single multicore server. If one needs to add nodes to the cluster it is as easy as purchasing more commodity hardware. On a multi-core machine it is not that simple. For example, a top of the line Intel® Xeon® E5-2699 processor has support for 36 threads. Beyond that one would need to add another socket which essentially doubles the price. Moreover, the server would need to support multiple processor sockets.

Higher Short-term Cost than with a Cloud Provider
In short-term the cost of purchasing a server may be significantly higher than renting out a cluster from a cloud computing service. Thus, it makes most sense to run an application on multi-core as a medium- to long-term investment.

More Maintenance than with a Cloud Provider
Owning a server requires more maintenance than simply renting it out from a cloud provider. A cloud provider usually does all the necessary maintenance and can easily provision a new machine. Hence it is advantageous to use a multi-core server only if one can afford to maintain it as well.

2.3 Apache Storm

Apache Storm is an open source distributed real-time computation system. Storm was originally created by Nathan Marz while working at BackType [18]. BackType was later acquired by Twitter and Storm became open source. Storm was incubated into the Apache Software Foundation with version 0.9.1 and became a top-level Apache project in September 2014.

Storm was developed to run on top of a cluster where nodes execute components of a computation in parallel. Running Storm on a cluster of commodity hardware gives it good horizontal scaling properties. Running separate components in parallel allows the system to fully utilise available hardware and execute in real time.

2.3.1 Dependencies

Storm has five major dependencies:

Apache Zookeeper
: Apache Zookeeper [19] is an open source server that allows reliable distributed coordination. Storm uses Apache Zookeeper to maintain state which is then read and written to by nodes of a Storm cluster. More details on how Storm uses Zookeeper are provided in section 3.4.3.

Apache Thrift
: Apache Thrift [20] is a cross-language framework for network services. It allows you to write a definition file for services and data types required by your application and it then automatically generates source code which supports remote procedure calls and serialisation of the data types.

Kryo
: Kryo [21] is a serialisation library. It is used to serialise objects before they are sent between nodes of a Storm cluster.

Netty
: Netty [22] is an asynchronous event-driven network application framework. Storm utilises Netty to send intra-cluster messages. Thus when a node produces a result to be consumed by another node of the cluster it sends a message over the network using the TCP protocol implemented in Netty.

LMAX Disruptor
: LMAX Disruptor [23] is a high-performance data structure used to exchange data between concurrent threads. It uses an optionally lock-free implementation of a ring buffer which Storm components running on the same node of a cluster use to exchange messages.

2.3.2 Usage of Apache Storm

Storm works particularly well with sister Apache projects such as Apache Kafka [24] and Apache HBase [25]. Apache Kafka is a messaging broker that is often used as the missing link between producers and consumers of a cluster. For example, web API endpoints might send data to a Kafka queue and a Storm application can then read the data from the queue. Apache HBase is a big data-store that allows real-time random reads and writes. It is modelled after Google's Bigtable project [26] and can be used to store data in a distributed way.

Storm is reportedly used by 81 companies listed on their "Powered By" website [9] and possibly many others. Storm's popularity is one of the reasons why it was chosen for this project. We also believe that the concepts used in Storm (explained in Section 3.2) apply to a large number of different situations and many applications can be easily adapted to work on Storm.

There has been significant research into optimising computations running on Apache Storm. For example, [27] looked at how to reconfigure a Storm job by reallocating component tasks to minimise communication cost, [28] proposed a domain specific language for defining Storm jobs, and finally [29] looked at how to provide exactly-once semantics in a Haskell port of Storm.

2.4 Similar Efforts

Currently data stream processing is a province of distributed systems such as the ones mentioned in the previous section. Most programming languages support parallel execution and while there are many open-source libraries that ease the process of writing parallel programs on a single multi-core machine such as OpenMPI [30] and OpenMP [31] , none of them are tailored to data stream processing. Furthermore, these libraries usually require the programmer to do some of the heavy lifting rather than abstract it away.

Apache Hadoop is another open-source distributed system. However, it is used in offline batch processing rather than in online real-time processing. There has been effort to port Hadoop to multi-core in [6] (Hone) as well as in [32] (Phoenix). This research suggests that borrowing ideas from distributed systems and applying them in the context of multi-core has potential. However, to our knowledge there has not been effort to port Storm or any other distributed real-time data stream processing system to multi-core.

2.5 Summary

Distributed real-time computation systems such as Apache Storm provide programmers with abstractions that make it easy to implement data stream processing

applications on top of a cluster. However, in case of single multi-core machines there are not any obvious open-source software choices. While there exist several libraries that allow programmers to parallelise their computations, they are not tailored to data stream processing.

Additionally, running these systems on a single multi-core server might have potential benefits as opposed to executing them on a cluster of commodity hardware.

Chapter 3

Background on Storm

In this chapter we give background information on Storm. This information is necessary to understand the design of Storm-MC. We give a quick overview of Apache Storm (3.1), explain the concepts used in Storm (3.2), show an example Storm program (3.3), give details about the underlying architecture of Storm (3.4), and finally describe the serialisation used in Storm (3.5).

3.1 Storm Overview

Apache Storm was developed in a mix of Java and Clojure. As mentioned by the author of Storm in [33], writing the Storm interfaces in Java ensured large potential user-base while writing the implementation in Clojure increased productivity.

To ensure API compatibility with Storm, Storm-MC was developed using the same set of languages. This allowed for code reuse and not having to re-implement functionality already present in Storm. Hence, in the following sections we describe Storm in greater detail in hope that this will later clarify design choices made for Storm-MC.

3.2 Storm Concepts

3.2.1 Core Concepts

There are several core concepts used by Storm and hence by extension Storm-MC as well. These concepts are put together to form a simple API that allows the programmer to break down a computation into separate components and define how these components interact with each other. The three core concepts of Storm are:

Spout
A spout is a component that represents the source of a data-stream. Typically, a spout reads from a message broker such as RabbitMQ [34] or Apache Kafka but can also generate its own stream or read from somewhere like the Twitter streaming API [35].

Bolt
A bolt is a component that transforms tuples from its input data stream and emits them to its output data stream. A bolt can perform a range of functions e.g. filter out tuples based on some criteria or perform a join of two different input streams.

Topology
The programmer connects spouts and bolts in a directed graph called topology which describes how the components interact with each other. The topology is then submitted to Storm for execution.

Every component is represented by a set of tasks. These tasks can run in parallel and execute the function of the component.

It should also be noted that spouts and bolts can be implemented not only in Java but other languages such as Python and Ruby.

3.2.2 Additional Concepts

There are several additional concepts which describe how components of a topology interact:

Stream
A stream is defined as an unbounded sequence of tuples. Streams can be thought of as edges of a topology connecting bolts and spouts (vertices).

Tuple
A tuple wraps named fields and their values. The values of the fields can be of different types. When a component emits a tuple to a stream it sends that tuple to every bolt subscribed to the stream.

Stream Grouping
Every bolt needs to have a type of stream grouping associated with it. This grouping decides the means of distributing the tuples coming from a bolt's input stream among task instances of the bolt. Following are the possible types of stream grouping:

Shuffle Randomly partition the tuples among all the bolt tasks.

Fields Hash on a subset of the tuple fields. All tuples with same values of those fields will go to same bolt task.

All Replicate the entire stream to all the bolt tasks.

Direct The producer of the tuple decides which task of the bolt will receive this tuple.

Global Send the entire stream to a single bolt task.

Local or Shuffle Prefer sending to tasks in the same node, if that is not possible use same strategy as Shuffle.

Users are also able to specify their own custom grouping by implementing the `CustomStreamGrouping` interface.

All the components of a Storm topology execute in parallel. The user can specify how much parallelism he wants associated with every component and Storm spawns the necessary number of threads. This is done through a configuration file, defined in YAML, which is then submitted along with the topology.

There are two additional bolts running for every topology:

Acker
The Acker bolt guarantees fault tolerance for the topology. It tracks every tuple that was produced and ensures that the tuple has been acknowledged by every bolt of the stream.

System Bolt
The System bolt is useful in two ways:

Metrics System bolt collects metrics on the local Java Virtual Machine (JVM). Other components can subscribe to these metrics and receive their values at regular intervals.

Ticks Components of a topology can subscribe to receive tick tuples in regular intervals. These tuples can be used to trigger events at subscribed components.

3.3 Example Topology

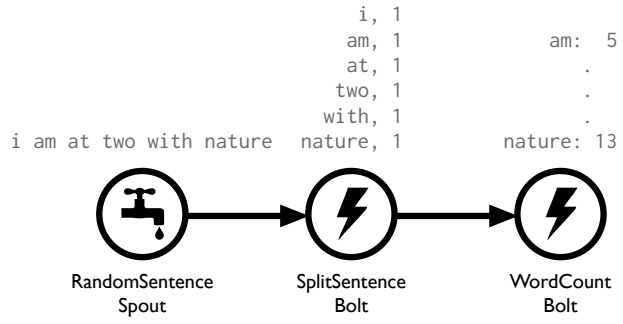


Figure 3.1: WordCount topology.

A classic example used to explain Storm topologies is the WordCount topology. In this topology, there is a spout generating random sentences, a bolt splitting the

sentences on white space, and a bolt counting occurrences of every word. Figure 3.1 shows how we could represent this topology graphically.

This may seem as a simplistic example but it is useful when demonstrating how easy it is to implement a working topology using the Storm API.

Listing 1 shows how the topology is put together in Storm to form a graph of components. Storm uses the Builder design pattern [36] to build the topology which is then submitted to an emulated cluster for execution. The last argument to the `setBolt`/`setSpout` method is the number of parallel tasks we want Storm to execute for the respective component. For implementation of the spout and the bolts used in this topology, refer to Appendix A.

```
public class WordCountTopology {
    public static void main(String[] args) throws Exception {
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("spout", new RandomSentenceSpout(), 5);
        builder.setBolt("split",
                new SplitSentence(), 8).shuffleGrouping("spout");
        builder.setBolt("count",
                new WordCount(), 12).fieldsGrouping("split", new Fields("word"));
        LocalCluster cluster = new LocalCluster();
        cluster.submitTopology("word-count", conf, builder.createTopology());
    }
}
```

Listing 1: WordCountTopology.java

3.4 Storm Architecture

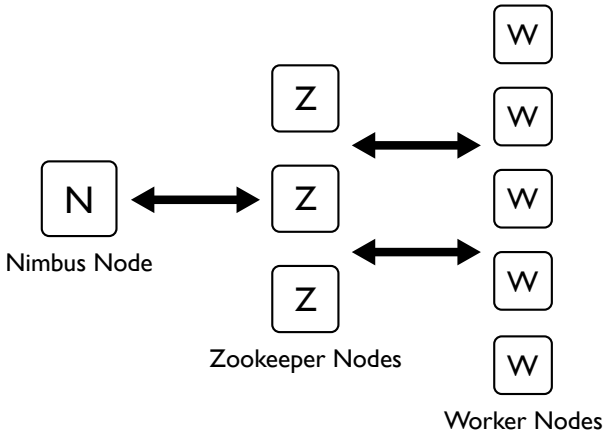


Figure 3.2: Apache Storm architecture.

A Storm cluster adopts the Master-Worker pattern. To set up a Storm topology, the user launches daemon processes on nodes of the cluster and submits the topology to

the master node, also called Nimbus. The worker nodes receive task assignments from the master and execute the tasks assigned to them. The coordination between the master node and the worker nodes is handled by nodes running Apache Zookeeper. Figure 3.2 shows a graphical representation of Storm architecture.

3.4.1 Nimbus Node

The master node runs a server daemon called Nimbus. The main role of Nimbus is to receive topology submissions from clients. Upon receiving a topology submission, Nimbus takes the following steps:

Validate the topology
The topology is validated using a validator to ensure the submitted topology is valid before trying to execute it. Programmers using Storm can write their own validator by implementing the `ITopologyValidator` interface.

Distribute the topology source code
Nimbus ensures that the workers involved in the topology computation have the source code by sending it to all nodes of the cluster.

Schedule the topology
Nimbus runs a scheduler that distributes the work among workers of the cluster. Similarly to validation, the user can use his own scheduler by implementing the `IScheduler` interface or use the default scheduler provided by Storm. The default scheduler uses a simple Round-robin strategy [37].

Activate the topology
Nimbus transitions the topology to active state which tells the worker nodes to start executing it.

Monitor the topology
Nimbus continues to monitor the topology by reading heartbeats sent by the worker nodes to ensure that the topology is executing as expected and worker nodes have not failed.

Nimbus is an Apache Thrift [20] service (more on Thrift in section 3.5) that listens to commands submitted by clients and modifies the state of a cluster accordingly. Following are the commands supported by Nimbus:

Submit a topology
Clients can submit a topology defined in a Java Archive (JAR) file. The Nimbus service then ensures that the topology configuration and resources are distributed across the cluster and starts executing the topology as previously described.

Kill a topology
Nimbus can stop running a topology and remove it from the cluster. However, the cluster can still continue executing other topologies.

Activate/deactivate a topology
Topologies can be deactivated and reactivated by Nimbus. This could be useful if the spout temporarily cannot produce a stream and the user does not want the cluster to execute idly.

Rebalance a topology
Nimbus can rebalance a topology across more nodes. Thus if the number of nodes in the cluster ever changes the user can increase or decrease the number of nodes involved in the topology computation.

3.4.2 Worker Nodes

The worker nodes run a daemon called Supervisor. There are 4 layers of abstraction which control the parallelism of a worker node.

Supervisor
A supervisor is a daemon process the user runs on a worker node to make it part of the cluster. It launches worker processes and assigns them a port they can receive messages on. Furthermore, it monitors the worker processes and restarts them if they fail. A worker node runs only one supervisor process.

Worker
A worker process is assigned a port and listens to tuple messages on a socket associated with the port. A worker launches executor threads as required by the topology. Whenever it receives a tuple, it puts it on a receive queue of the target executor.

Furthermore, the worker has a transfer queue where its executors enqueue tuples ready to be sent downstream. There can be multiple worker processes running inside one supervisor.

Executor
An executor controls the parallelism within a worker process. Every executor runs in a separate thread. An executor's job is to pick up tuples from its receive queue, perform the task of a component it represents, and put the transformed tuples on the transfer queue of the worker. There can be many executors running inside one worker and an executor performs one (the usual case) or more tasks.

Task
A task represents the actual tuple processing function. However, within an executor thread all the tasks are executed sequentially. The main reason for having tasks is that the number of tasks stays the same throughout the lifetime of a topology but the number of executors can change (by rebalancing). Thus if some worker nodes in the cluster go down, the topology can continue executing with the same number of tasks as before.

3.4.3 Zookeeper Nodes

A Storm cluster contains a number of Zookeeper nodes which coordinate the communication between Nimbus and the worker nodes. Storm does this by storing the state of the cluster on the Zookeper nodes where both Nimbus and worker nodes can access it.

The cluster state contains worker assignments, information about topologies, and heartbeats sent by the worker nodes to be read by Nimbus. Apart from the cluster state, Storm is completely stateless. Hence, if the master node or a worker node fail the cluster continues executing and the node will get restarted if possible. The only time the cluster stops executing completely is if all the Zookeper nodes die.

3.5 Serialisation

Since Storm topologies execute on a cluster all components need to be serialisable. This is achieved with Apache Thrift. Components are defined as Thrift objects and Thrift generates all the Java serialisation code automatically.

Furthermore, since Nimbus is a Thrift service Thrift generates all the code required for remote procedure call (RPC) support. This allows defining topologies in any of the languages supported by Thrift and easy cross-language communication with the Nimbus service.

Chapter 4

Bringing Storm to Multi-core

The following chapter explains how Storm was ported to multi-core. We describe how Apache Storm behaves in a multi-core environment (4.1), discuss the design of Storm-MC (4.2), outline how Storm-MC was implemented (4.3), and list feature differences between Apache Storm and Storm-MC (4.4).

4.1 Apache Storm on Multi-core

To begin, we discuss why Apache Storm does not perform optimally in a multi-core environment. Storm can be ran in local mode where it emulates execution on a cluster. This mode exists so that it is possible to develop and debug topologies without needing access to a cluster. However, there are several reasons why the local mode is not as performant as it could be.

4.1.1 Tuple Processing Overhead

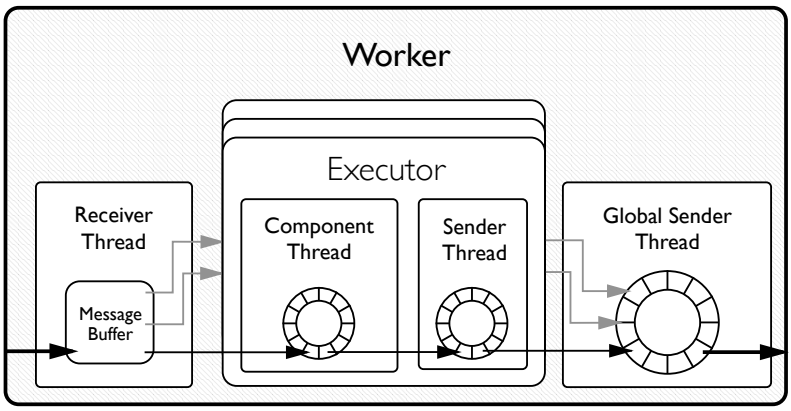


Figure 4.1: Tuple processing in Apache Storm.

Figure 4.1 shows how tuple processing is implemented inside a Storm worker process. The tuple is read from a message buffer by the receiver thread of the worker and put on a receive queue of the target executor. The tuple is then picked up by the component thread of the executor for task execution.

After the component thread has executed the task it optionally puts a newly created tuple on the executor send queue. There, it is picked up by the executor sender thread which puts the tuple on the global send queue of the worker. Finally, the global sender thread of the worker serialises the tuple and sends it downstream over the network.

Alternatively, if the tuple is targeted to an executor in the same worker process it is put on the receive queue of the corresponding executor directly after task execution.

The queues used in tuple processing are implemented as ring buffers using the Disruptor library [23]. Detailed background on how Disruptor works and its performance benchmarks can be found in [38]. In summary, due to less write contention, lower concurrency overhead, and being more cache-friendly the Disruptor pattern can offer latency of inter-thread messages lower than 50 nanoseconds and a throughput of over 25 million messages per second [38].

There is significant overhead required to emulate sending tuples to executors in other worker nodes. For one, there is the overhead from the tuple passing through the three queues of a worker. The authors of LMAX Disruptor showed that a three step pipeline can have half the throughput of a single consumer-producer pipeline [39].

Furthermore, to emulate over-the-network messages Storm uses a map of `LinkedBlockingQueues` which according to [38] have several orders of magnitude lower performance than Disruptor queues.

4.1.2 Thread Overhead

Another major overhead stems from Apache Storm running too many threads even for simple topologies. As we show most of these threads are not necessary in a multi-core environment.

Acker Bolt

The Acker bolt provides so-called at-least-once semantics by ensuring that tuples propagate through the topology even if a failure in processing occurs. In Storm it is included in every topology. It can be disabled via the configuration file in which case it is mostly idle not receiving any tuples. However, it can still use up resources especially if it waits for tuples using a busy waiting strategy (more on waiting strategies in Section 4.3.3.1). Furthermore, since hardware fault tolerance cannot be guaranteed on a single multi-core machine, the Acker bolt is not necessary.

Heartbeats & Timers

Every worker has a heartbeat thread that simulates sending heartbeat messages to the Nimbus node. It does this by writing to a local cache which is persisted to a file by a write on every heartbeat. Since the write is implemented using the `java.io` package the write is blocking - the thread cannot continue until the write is completed. While heartbeats are essential in cluster mode to signal the node being alive, there is no need for them in local mode.

Zookeeper Emulation

More overhead is produced by a local Zookeper server which emulates the Zookeeper nodes of a cluster. Running the Zookeeper server is a massive addition to the list of overheads as shown in the following paragraphs. The purpose of Zookeeper is to maintain state of running topologies and nodes of the cluster. As we will show in the following sections maintaining this state on multi-core is not necessary.

During profiling we found that a topology with three components running with one worker and one executor per component was being executed with 55 threads (not including system JVM threads and threads created by the profiler). Table 4.1 shows a breakdown of what the individual threads were used for.

Spout Parallelism	# of Threads
Main Thread	1
Worker Sender & Receiver Threads	2
Acker & System Component Threads	2
Executor Component Threads	3
Executor Sender Threads	5
Various Timers & Event Loops	14
Zookeper Server	28

Table 4.1: Storm thread usage: topology with three executors.

To find out what state the threads were actually in at any given time the topology was executed for three minutes and a JVM thread dump was recorded every second. The average results of this experiment can be observed in Table 4.2 and the state distribution over time can be seen in Figure 4.2.

Spout Parallelism	# of Threads
RUNNABLE	8
TIMED WAITING	22
WAITING	25

Table 4.2: Storm thread states: topology with three executors.

Even though three minutes may seem to be a short amount of time the fact that there is almost no variation shows that it is sufficient. As can be seen from the table,

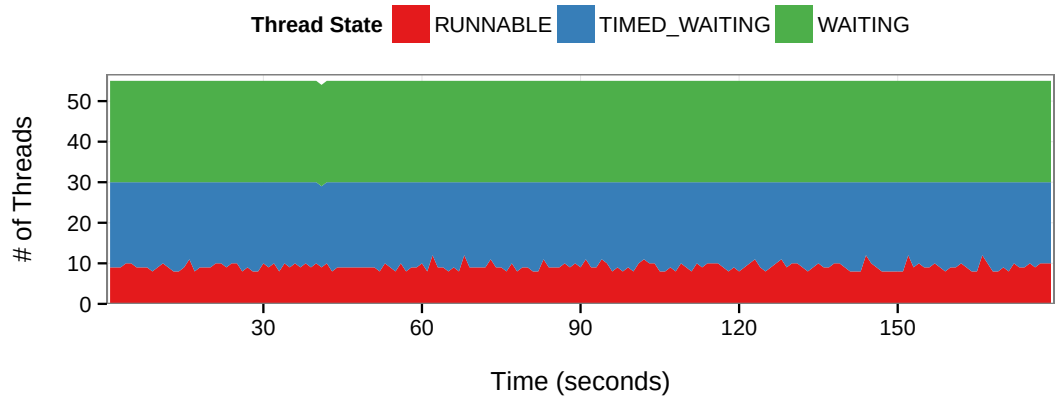


Figure 4.2: Thread state distribution over time.

most of the threads were either in state `WAITING` or `TIMED WAITING`. According to the Java documentation on thread states [40] these two states are used for threads that are waiting for an action from a different thread and cannot be scheduled by the scheduler until that action is executed.

However, on average there were eight threads in state `RUNNABLE` which JVM uses to mark threads which are executing on the JVM and are possibly waiting for resources from the operating system (OS) such as processor [40]. Hence, these threads are directly competing to be scheduled by the OS. This means that for three components with one executor each running in parallel there are five threads doing potentially unnecessary work.

In the subsequent sections we show that these threads were in fact unnecessary and we will discuss how the total number of threads was reduced. In fact, to execute the same topology on Storm-MC requires only 5 threads.

4.2 Storm-MC Design

The design we adopted for porting worker nodes is to only have one worker executing all the executors of a topology. This design simplified the communication model and allowed removal of unnecessary abstractions.

Additionally, the source code for the Nimbus service was merged with the worker source code. This was done because there is no need to run Nimbus and worker specific code in parallel. Once the Nimbus source code sets up the topology, all the work is done by the worker source code. Hence they can be executed serially.

4.2.1 Porting Nimbus

Nimbus in Apache Storm performs as a server that clients can send topologies to for execution. In Storm-MC, we opted for a different design. Storm-MC is designed as a standard Java library that applications can import and use to create and execute topologies in the main method of their application.

This means that Storm-MC does not support running multiple topologies at the same time. However, to do that one only needs to run every topology in a separate process. This is because, unlike when executing on a cluster, different topologies do not need to share any state and it is also more natural to execute them as separate processes. This design decision has the added benefit of each process having its own part of main memory thus reducing cache conflicts as shown in [41] and providing higher security by not having different topologies share memory space.

The interaction with the Nimbus service in Storm is usually through a shell script with a path to a JAR file of the topology and the main class to execute. This shell script was ported over to Storm-MC but instead of communicating with a service it spawns a new separate Java process that executes the topology.

Unlike Apache Storm, Storm-MC does not support topology scheduling. Since within one process there is always only one topology running at a time and the hardware configuration of the machine does not change during execution, the parallelism is clearly defined by the number of executors per component specified in the topology configuration.

One way to implement scheduling could be to pin threads to specific cores. Unfortunately, Java does not provide support for CPU affinity; the assignments are handled automatically by JVM. Potentially, this could be achieved by using C or C++, both of which support CPU affinity, but this was not implemented in Storm-MC.

The role of Nimbus in Storm-MC has effectively been reduced to validating the topology and its configuration and passing the topology along to the worker source code which handles topology execution.

4.2.2 Porting Worker Nodes

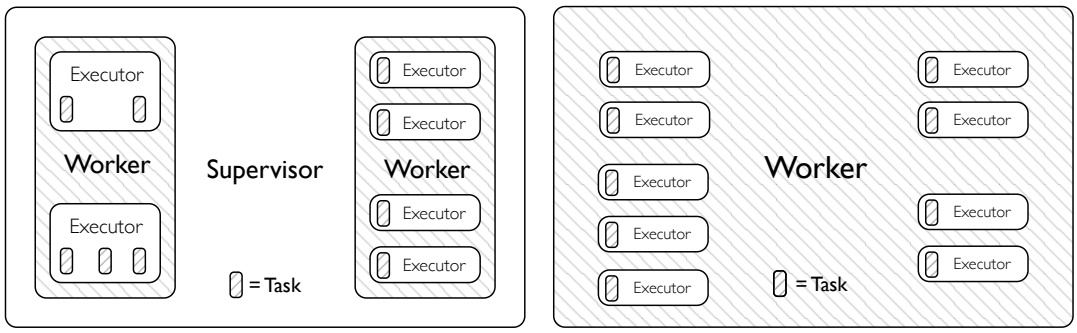
In Apache Storm, a worker node runs a supervisor daemon, which in turns launches worker processes which run executors which execute tasks. There are thus three ways to control the parallelism of a component in Apache Storm: setting the number of workers, setting the number of executors per component, and setting the number of tasks per executor.

In Storm-MC, however, there is only one worker wrapper which runs all executors and their tasks. Furthermore, only one task executes within an executor. This limit is due to the fact that tasks execute serially and hence there is no speedup to be gained by one executor running more than one task. Hence, the parallelism of

a component is controlled by only one variable: the number of executors per component. This represents the number of threads that will function as the component within a topology. This design has several benefits:

- All the communication occurs within one worker wrapper.
- The supervisor daemon can be removed as there is no need to synchronise or monitor workers.
- There is no need to simulate over-the-network message passing.
- Tuple passing between executor threads within a worker stays the same as in Apache Storm.

A comparison of an Apache Storm worker node and its Storm-MC equivalent is shown in Figure 4.3.



(a) Worker node in Apache Storm. (b) Worker wrapper in Storm-MC.

Figure 4.3: Comparison of a worker in Storm and Storm-MC.

The role of the worker wrapper is to launch executors and provide them with a shared context through which they can communicate. This is done with a map of Disruptor queues which the executors use as receive queues to pick tuples from.

Moreover, the worker wrapper contains a map of components and streams. This map specifies which bolts subscribe to a stream a component produces. Executors use this map to figure out which components they should send tuples to.

Additionally, the worker wrapper has a timer which components can use to get tick tuples at regular intervals. As mentioned before, bolts can use tick tuples to trigger events at regular intervals. For example, one might want to sort a window of tuples based on some criteria every five minutes. This timer can also collect component metrics at regular intervals and send them as tuples to subscribed instances of a `MetricsConsumerBolt`.

Finally, the worker wrapper provides access to a fixed-size thread pool through a Java `ExecutorService` object. Storm-MC executors can use this service to launch background tasks on a shared thread pool.

4.2.3 Removing State

Storm-MC is completely stateless. The cluster state that was managed by Zookeeper in Apache Storm was completely stripped away. In Storm, workers use the Zookeeper cluster state to communicate with Nimbus and vice versa. For example, when Nimbus creates topology assignments it informs workers via the cluster state. In Storm-MC, we adopted a more functional approach where worker is just a function invoked by the Nimbus part of the source code.

While this is not something that is visible to a user of Storm-MC, it required a great effort as all the code that referenced the Zookeeper library had to be refactored.

4.2.4 Removing Serialisation

Similarly to removing the Zookeeper state, great amount of work was put into removing the dependency of Storm-MC on Apache Thrift. This was mostly done to reduce code bloat and remove an unnecessary dependency since there is no serialisation required in a multi-core environment.

Moreover, code generated by Thrift does not use standard Java camelCase naming conventions but instead uses underscore_case. For example, Thrift generates method names such as `get_component_common` which make the API a little less elegant.

Removing Thrift required refactoring all the data types generated automatically by Thrift. This not only reduced the size of the codebase significantly but also made the code more readable and self-documenting than the code generated by Thrift.

There is some serialisation required to enable communication between shell components and the outside processes they represent. This was left in place so that Storm-MC can support components defined in other languages as well.

4.3 Storm-MC Implementation

Most of the implementation of Storm-MC was ported over from Apache Storm with adjustments made where necessary. The problem with describing implementation of ported software is that there is a lot of functionality that needed to be changed but the changes usually did not require a complete overhaul or a completely new algorithm. This is the case with Storm-MC as well.

4.3.1 Topology Submission

Topologies are built using an instance of the `TopologyBuilder` class which uses the builder pattern - same as in Apache Storm. While the basis of this class was reused

from Storm, the internals had to be refactored so they work with the non-Thrift data types used by Storm-MC. Once a topology is built, it is submitted to an instance of the `LocalCluster` class. This class is used in Storm for emulating the cluster on a local machine and Storm-MC adapted the class for backwards compatibility. This way, code created for Storm needs minimal adaptation to work on Storm-MC. A topology is submitted for execution via the `submitTopology` method which takes three arguments: the name of the topology, a Java Map with configuration, and a topology built by `TopologyBuilder`.

Along with a topology, users of Storm-MC can submit a configuration file written in YAML. This is done by setting a JVM property called `storm.conf.file`. This file can define the capacity of the Disruptor queues, the waiting strategy used by components when there are no tuples to pick up, and hooks they want executed every time a tuple is processed. Additionally, this file can define which implementation of `ITopologyValidator` to use for topology validation.

4.3.2 Spouts and Bolts

There are two core interfaces for implementing spouts and bolts in Storm-MC: `ISpout` and `IBolt`, respectively. These interfaces provide a set of standard methods that get executed during the lifetime of a topology:

`open/prepare`
: These methods get called for every spout and bolt, respectively, before a topology starts executing.

  This method provides the component with the following parameters: a map of the topology configuration, a worker context which provides access to the aforementioned resources of a worker, and an `OutputCollector` object. The component can use this method and its parameters to prepare for execution.

`nextTuple/execute`
: The `nextTuple` method is called when Storm requests the next tuple from the spout. It should be non-blocking, that is if there is no tuple to produce it should simply return. Similarly, the `execute` method supplies a new tuple to the bolt as a parameter of this method.

  Both spouts and bolts can emit new tuples using the `OutputCollector` provided by the `open/prepare` methods.

`close/cleanup`
: These methods are called right before a Storm-MC application is either killed with a signal or programatically. A component can close or clean up any resources used during execution.

Alternatively, one can implement spouts and bolts in other languages such as Python or Ruby by subclassing the `ShellSpout` and `ShellBolt` classes. These classes implement the aforementioned interfaces but also include code that handles communication with external processes.

4.3.3 Tuple Processing

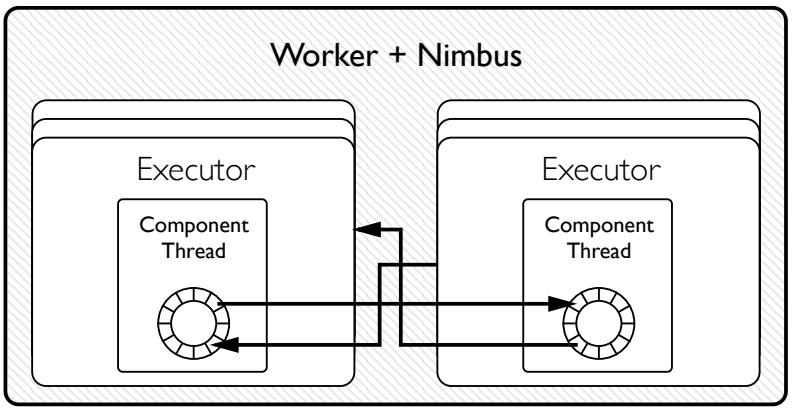


Figure 4.4: Tuple processing in Storm-MC.

The implementation of tuple processing in Storm-MC is depicted in Figure 4.4. As can be seen from the figure, the queues used for remote message sending present in Storm were stripped away and there is only one Disruptor queue for every executor. Once an executor is done processing a tuple it puts it directly on the receive queue of its downstream bolts.

In Apache Storm, every executor runs two threads: a component thread for tuple processing and a sender thread for tuple sending. Thus the component thread is not slowed down by competing with other executors trying to access the worker send queue; the sender thread takes care of that. In Storm-MC, an executor runs only one thread. This design works because executors do not have to compete for access to the worker send queue which is not needed. Hence, the number of queues a tuple needs to pass through in a component is lowered as compared to Apache Storm.

Tuple processing in Storm-MC is a variant of multiple producer single consumer problem. In general, multiple producer single consumer problems are hard to optimise since there needs to be some form of synchronisation between multiple producers trying to produce an entry at the same time. In Disruptor queues this is implemented using the atomic Compare-and-Swap (CAS) operation. This operation ensures that even if multiple threads attempt to modify a variable only one of them succeeds and all threads involved are able to tell whether it was them that succeeded. Hence, one thread will succeed at claiming the next entry of a Disruptor queue and others will have to retry.

Alternatively, locks can be used used to synchronise access but lock-free queues using CAS are considered to be more efficient than locks because they do not require a kernel context switch [38]. However, even with CAS a processor must lock its instruction pipeline to ensure atomicity and employ a memory barrier to make the changes visible to other threads.

We investigated other data structures besides Disruptor queues that could be used for tuple exchange in Storm-MC but Disruptor queues are considered state of the

art in low-latency parallel systems. This is one area that Apache Storm got right and we were not able to further optimise.

Other options we considered were `ArrayBlockingQueue` and `LinkedBlockingQueue` both of which are in the Java standard library. However, the Disruptor shows superior throughput and latency compared to these options as shown in [39].

4.3.3.1 Waiting Strategies

There are four different waiting strategies an executor can employ while waiting for a tuple to become available:

BlockingWaitStrategy
: This strategy uses a lock and a condition variable. The thread waits on the condition variable and is signalled once a new tuple becomes available. This strategy wastes the minimum number of CPU cycles when an executor is waiting.

SleepingWaitStrategy
: This strategy initially spins for hundred iterations, then uses `Thread.yield()`, and finally uses `LockSupport.parkNanos(1L)` to sleep. Thus after quiet periods this strategy might introduce slight latency spikes.

YieldingWaitStrategy
: This strategy initially spins for hundred iterations and then uses `Thread.yield()`. It provides good compromise between CPU utilisation and performance.

BusySpinWaitStrategy
: With this strategy the thread is in a so-called tight loop, where it checks whether a new entry is available every iteration of the loop and only breaks out of the loop if there is a new entry. This strategy guarantees the best performance but since it maximises CPU utilisation it works well only if the number of CPU cores is higher than the number of active threads.

The default strategy used in Storm-MC is `BlockingWaitStrategy` but users can change the strategy in the configuration file. Using other waiting strategies can lead to improved performance but since this can also be done in Apache Storm we do not further differentiate between them.

4.3.3.2 Tuple Pools

Once a component wants to send a new tuple to its downstream components it needs to initialise a Java tuple object. Here, we saw room for improvement since this might need to be done at very high rates, possibly million times per second.

Hence, a tuple pool was implemented where executors could place tuples after they were done with them so the tuples could be reused by other executors. However,

access to this pool also had to be synchronised and hence accessing the pool introduced higher latency than simply initialising new tuples. Moreover, Java garbage collector actually does a good job of re-claiming unused memory. For these reasons, the idea of using a tuple pool was abandoned.

This problem could potentially be circumvented by using different constructs to implement tuple passing but backwards compatibility with applications written for Apache Storm was deemed more important than the slight potential gain in performance.

4.3.4 Executor Algorithm

The implementation of an executor processing a tuple is as follows:

1. The executor tries to pick a tuple from its receive queue. If there are no tuples to be picked up it employs a waiting strategy as per the configuration of the topology.
2. Once a new tuple becomes available, the executor tries to process as many tuples as possible (more tuples could have been added to the queue while the executor was "waking up"). The executor then processes this batch of tuples as per the component it represents.
3. If the executor emits a new tuple it attempts to send it downstream by repeatedly trying to claim an entry of the downstream receive queue.
4. Once the executor successfully claims an entry it publishes the tuple and goes back to step 1.

4.4 Differences between Apache Storm and Storm-MC

The codebase of Apache Storm is fairly large - 54,985 lines of code as reported by `cloc` [42]. Thus we had to prioritise features that were ported over to Storm-MC. Table 4.3 presents a list of Storm features and shows which were ported over to Storm-MC and which were not.

A feature we deemed very important is the ability to define topology components in other languages. Thus, Storm-MC allows defining components in other languages such as Python or Ruby and can automatically connect them with a Java topology. An example of a component defined in Python can be seen in Listing 4.

Storm-MC has support for task hooks just like Storm. Task hooks allow users of Storm-MC to capture a number of events and execute custom code when the event occurs at a registered component. Hooks can be created by implementing the `ITaskHook` interface. They can be used to, for example, update a web server with the latest performance metrics.

Feature	Apache Storm	Storm-MC
Multi-language Topologies	✓	✓
Hooks	✓	✓
Metrics	✓	✓
Tick Tuples	✓	✓
System Bolt	✓	✓(optional)
Multiple Topologies	✓	✗
Topology Scheduling	✓	✗
Trident API	✓	✗
Built-in Metrics	✓	✗
Nimbus as a Server	✓	✗

Table 4.3: Feature comparison of Apache Storm and Storm-MC.

Additionally, Storm-MC has support for topology metrics. This way, components can record metrics such as number of tuples processed or a count of event occurrences. These metrics can then be automatically consumed by a bolt that subclasses the `MetricsConsumerBolt` class.

As mentioned before, Storm-MC supports tick tuples which can be used to trigger component-local events at regular intervals.

Apache Storm supports an alternative high-level API called Trident. Trident functions get converted to spouts and bolts by the Storm library. Trident was omitted from Storm-MC but its implementation on top of the current API is possible in the future.

Moreover, Apache Storm collects JVM metrics with a bolt called `SystemBolt`. In Apache Storm, this bolt is added automatically to all Storm topologies. Storm-MC topologies do not include this bolt by default, however, users can choose to add this bolt on their own. This way overhead is reduced if this bolt is not required but the bolt can still be added to a topology if deemed necessary.

Chapter 5

Evaluation

This chapter evaluates Storm-MC. We describe the metrics used to evaluate performance of Storm-MC (5.1), list the configuration used for benchmarking (5.2) and compare Storm-MC to Apache Storm executing in local mode on a set of different topologies (5.3).

5.1 Evaluation Metrics

The performance of Apache Storm and Storm-MC was evaluated on the following three metrics:

Throughput
: The tuple throughput per component was computed over a period of five minutes. This metric provides an insight into implementation performance of the evaluated systems.

CPU Utilisation
: The average CPU utilisation was computed by sampling the instant CPU utilisation every second throughout program execution. This metric shows how utilised the machine was while running a topology with the same configuration on both systems.

Resident Memory Size
: The average resident size was computed by sampling the instant resident size every second throughout program execution. This metric provides an insight into how much main memory both systems consume to run a topology with the same configuration.

5.2 System Configuration

5.2.1 Software Setup

All performance benchmarks in this chapter were executed on the following versions of Apache Storm and Storm-MC:

- Apache Storm version 0.9.2
- Storm-MC version 0.1.6

The Apache Storm source code had to be adapted to include a workaround for a deadlock bug present in version 0.9.2. This bug caused a topology to exit with threads left in Zombie state under certain conditions. This prevented Storm from logging the benchmark metrics after execution. Hence a workaround was added so the metrics were logged.

Version 0.1.6 is the latest version of Storm-MC as of this moment. The first release was version 0.1.0 which was production-ready but since then there were 6 minor versions fixing bugs as they were discovered during testing.

Furthermore, the following versions of benchmark libraries were used to implement the performance benchmarks described in this chapter:

- A fork of IBM Storm Email Benchmarks version 0.1.12
- Storm-benchmark version 0.1.0

IBM open sourced a project with a suite of benchmarks which they used to compare Apache Storm to their real-time data stream system IBM Infosphere Streams [43]. We forked this project and used the Enron topology to benchmark Storm-MC against Apache Storm. The current version of the fork is 0.1.12.

Apache Storm developers use a project called storm-benchmark when testing the performance of Apache Storm. The WordCount and RollingSort topologies from this project were adapted and used to benchmark Storm-MC against Apache Storm.

Since Storm-MC reuses package names from Apache Storm, the same benchmark is directly executable by both libraries. This way there was no need to maintain two different benchmarks suites and it can be easily shown that the same application code was executed for both systems. An example topology submission to Storm-MC and Apache Storm, respectively, would look as follows:

```
java -cp storm-multicore-0.1.6.jar:storm-email-benchmark.jar:...
    com.ibm.streamsx.storm.email.benchmark.FileReadWordCount wordcount
```

```
java -cp storm-core-0.9.2-incubating.jar:storm-email-benchmark.jar:...
    com.ibm.streamsx.storm.email.benchmark.FileReadWordCount wordcount
```

There are several things being done in this one command:

- The JAR file of the corresponding library is added to the Java classpath.

- The JAR file of the benchmark library is added to the Java classpath.
- The ellipsis is used in place of listing JAR files of Storm and Storm-MC dependencies.
- The main class is specified.
- The name of the topology is defined as an argument.

Note the difference between these commands is only the library that is included on the Java classpath.

5.2.2 Hardware Setup

The machine used for benchmarking is the University of Edinburgh Informatics Student Compute server (`student.compute.inf.ed.ac.uk`). This server has the following hardware components:

Processor: Intel® Xeon® E5-2690 v2 @ 3.00 GHz
: The machine has two sockets with the same processor each. The processor has 10 physical cores with Hyper-Threading Technology which means each processor can handle up to 20 threads in parallel. Thus with two sockets, there is potential to execute up to 40 threads in parallel.

Main Memory
: The machine has 378 GB of main memory. Since data stream processing uses windows to store only up to a certain amount of memory this was more than enough to conduct the benchmarks.

5.2.3 Storm Configuration

As mentioned before, when submitting a topology the programmer can submit a configuration file as well. To ensure that the performance difference between Apache Storm and Storm-MC was not caused by different configuration, the default configuration file from Storm 0.9.2 was used to benchmark both projects. Most notably, the size of the ring buffer used by executors was set to 1024 and the wait strategy employed by executors when there are no tuples to pick up was set to `BlockingWaitStrategy`.

Furthermore, to avoid penalising Apache Storm for simulating sending messages over the network, the number of workers used in Storm was set to one.

5.3 Results

To compare performance of Storm-MC and Apache Storm 3 different benchmarks were executed, each with a different focus. The benchmarks were executed with

each system for five minutes. After that the system was killed and metrics were collected. To avoid any performance differences caused by varying amounts of heap memory required by the tested systems, the programs were run with the following flag: `-Xmx10240M`. This flag sets the maximum amount of heap memory used by JVM to 10 GB which was more than enough for all the benchmarks. This also shows that stream processing applications can often be ran on systems that do not have massive amount of RAM.

The parallelism of all components was varied from one to ten. It should be noted that having same amount of parallelism in every component does not guarantee the best global throughput but it is a good heuristic. The average CPU utilisation and resident memory size were recorded by the Unix `top` program [44]. Maximum CPU utilisation with 40 threads is 4,000%. Resident memory size is the amount of non-swapped physical memory a task has used. This metric can be deceiving as it depends on how the OS manages memory but it is the only fairly reliable memory metric reported by `top` that can be used for Java programs. We employ this metric only to show proportional difference in memory usage between Storm and Storm-MC.

5.3.1 WordCount Topology

The first topology tested for performance is a variant of the aforementioned WordCount topology. This topology has a spout `FileReadSpout` generating random sentences, which sends messages to a `SplitSentenceBolt` bolt which splits the sentences on whitespace and sends individual words to a `CountBolt` which counts word frequencies. Recall, that this topology is shown graphically in Figure 3.1. Since the components do not store any data in memory or make any I/O calls this topology is mostly CPU-bound.

The number of tuples processed by each component in Storm-MC and Apache Storm is shown in tables 5.1 and 5.2, respectively. As can be seen from the tables, Storm-MC often processed more than twice as many tuples per component than Apache Storm, while having lower CPU utilisation. The number of tuples processed by `CountBolt`, the last component of the topology, is also show in Figure 5.1. Since this topology is serial, the number of tuples processed by `CountBolt` is a good indicator of the total throughput.

Furthermore, it can be seen that after the parallelism is increased beyond three the throughput of Apache Storm tails off and even starts going down. This can be attributed to the number of threads ran by Apache Storm. For Storm-MC this tailing off occurs with parallelism of six where the overhead of multiple producers possibly trying to publish to the same queue becomes apparent. Moreover, with parallelism set to 6, Storm-MC executes with 20 threads which is close to the number of physical cores of the machine. However, it should be noted that even with parallelism equal to 10, Storm-MC still processes more than three times as many tuples as Storm.

Parallelism	FileReadSpout	SplitSentenceBolt	CountBolt	CPU Utilisation	Resident Size
1	25,767,502	25,767,502	225,815,174	217.9%	690.8M
2	34,403,678	34,403,127	301,493,247	414.6%	759.1M
3	45,731,188	45,732,988	400,767,999	611.5%	798.4M
4	52,285,327	52,283,540	458,187,555	805.5%	804.1M
5	55,326,941	55,325,167	484,844,652	998.7%	806.0M
6	56,747,319	56,744,629	497,285,149	1,195.3%	824.8M
7	51,048,092	51,044,774	447,330,207	1,314.8%	686.5M
8	51,892,349	51,889,982	454,739,674	1,502.0%	687.1M
9	55,312,654	55,308,442	484,696,841	1,656.8%	694.9M
10	54,377,002	54,374,432	476,512,544	1,822.5%	702.0M

Table 5.1: Storm-MC: Component throughput in WordCount topology.

Parallelism	FileReadSpout	SplitSentenceBolt	CountBolt	CPU Utilisation	Resident Size
1	12,583,377	12,579,132	110,233,966	294.5%	2.2G
2	16,800,475	16,796,695	147,194,709	481.7%	2.8G
3	22,120,695	22,107,696	193,735,106	687.1%	2.6G
4	20,720,637	20,711,756	181,500,586	895.3%	2.6G
5	17,177,688	17,164,209	150,412,037	1,129.3%	2.5G
6	17,402,418	17,388,691	152,374,303	1,342.1%	2.3G
7	17,702,568	17,689,940	155,009,826	1,532.6%	2.4G
8	19,161,174	19,143,254	167,749,429	1,697.3%	2.6G
9	17,833,631	17,803,323	156,006,619	1,903.4%	2.7G
10	17,488,484	17,442,562	152,843,992	2,136.7%	2.8G

Table 5.2: Apache Storm: Component throughput in WordCount topology.

The number of threads required to execute a topology is a linear function of the parallelism for both Storm and Storm-MC. However, as shown in Figure 5.2, the number of threads required by Storm increases more rapidly than Storm-MC. For example, with parallelism set to 10, Storm creates 109 threads whereas Storm-MC creates only 32. More formally, the number of threads required by both systems can be expressed by formulae shown in Figure 5.3. These are general formulae that apply to all topologies, not just WordCount.

Of note, the resident size used by Storm-MC is also less than half the resident size used by Apache Storm.

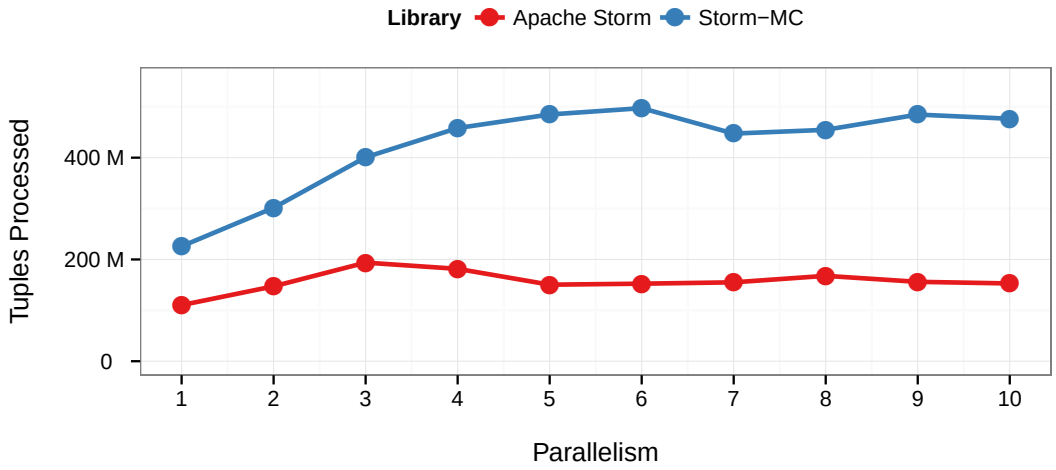


Figure 5.1: CountBolt throughput in Apache Storm and Storm-MC

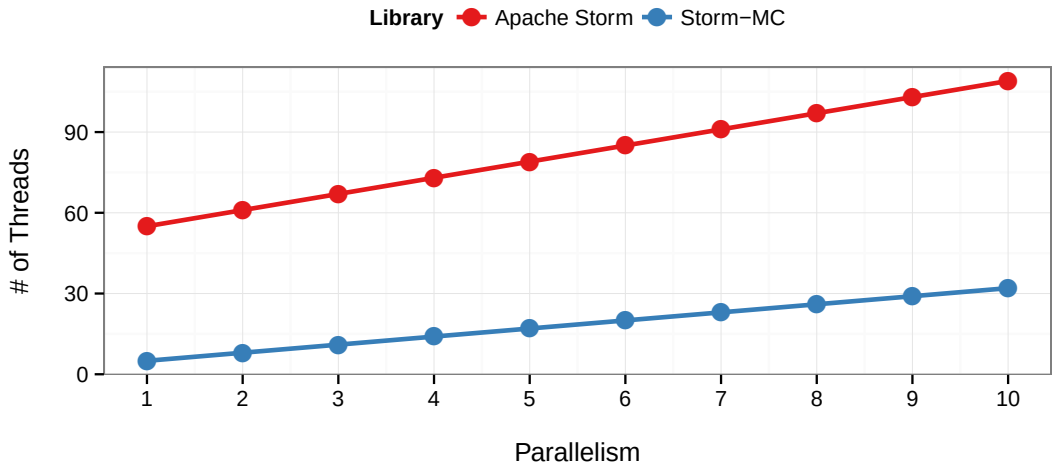


Figure 5.2: Number of threads used by Apache Storm and Storm-MC

$$49 + 2 \times \sum_{c}^{components} parallelism(c) \qquad \text{for Apache Storm.}$$

$$2 + \sum_{c}^{components} parallelism(c) \qquad \text{for Storm-MC.}$$

Figure 5.3: Number of threads used by Apache Storm and Storm-MC.

5.3.2 Enron Topology

Next, Enron topology from the IBM benchmarks was tested for performance. In this topology, serialised emails from the Enron email dataset are read from a file by a `ReadEmailsDecompressSpout` spout. They are then deserialised by a `AvroDeserializeBolt` bolt, filtered by a `NewFilterBolt` bolt, modified by a `ModifyBolt` bolt, and finally metrics are recorded by a `NewMetricsBolt` bolt. Additionally, every instance of the `NewMetricsBolt` bolt sends its local average email throughput to a global (excluded from the parallelism setting) `GlobalMetricsBolt` bolt every four seconds. This bolt then records the global average email throughput. Figure 5.4 shows this topology graphically.

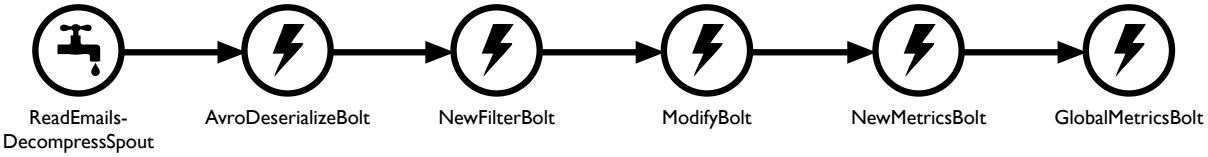


Figure 5.4: Enron topology.

Similarly to the WordCount topology, this topology is serial in nature. However, whereas the spout in WordCount topology keeps a small number of sentences in memory, the Enron topology has a spout that produces tuples by reading from a file. Thus, this benchmark is mostly I/O-bound. The average email throughput in Storm-MC and Apache Storm is shown in tables 5.3 and 5.4, respectively.

Parallelism	Emails Processed	CPU Utilisation	Resident Size
1	3,285,742	297.7%	806.8M
2	6,696,612	482.1%	756.1M
3	8,493,772	729.5%	641.4M
4	11,102,969	1036.9%	726.0M
5	12,630,475	1311.0%	860.8M
6	14,082,501	1590.3%	934.0M
7	11,261,275	1625.8%	1.3G
8	11,026,977	1689.5%	1.2G
9	13,413,492	1888.0%	1.3G
10	12,682,028	2513.8%	1.3G

Table 5.3: Storm-MC: Email throughput in Enron topology.

As can be seen from the tables, the difference in throughput in Enron Topology is less staggering than in WordCount. This is due to the fact that the throughput is limited by the file reads of the spout. However, as the parallelism increases the improvement in throughput of Storm-MC becomes more apparent as shown in Figure 5.5. This figure also shows that the throughput is fairly volatile. This is due to the fact that the file is loaded into main memory in chunks and hence the

Parallelism	Emails Processed	CPU Utilisation	Resident Size
1	2,943,709	406.6%	1.94G
2	4,832,874	945.1%	2.93G
3	5,623,028	1,427.4%	3.32G
4	6,238,395	1,891.2%	3.56G
5	6,105,155	2167.4%	3.65G
6	7,242,298	2388.6%	4.09G
7	7,756,729	2626.6%	2.65G
8	8,785,455	2815.7%	2.68G
9	9,379,807	2943.6%	2.72G
10	9,118,728	3051.2%	2.74G

Table 5.4: Apache Storm: Email throughput in Enron topology.

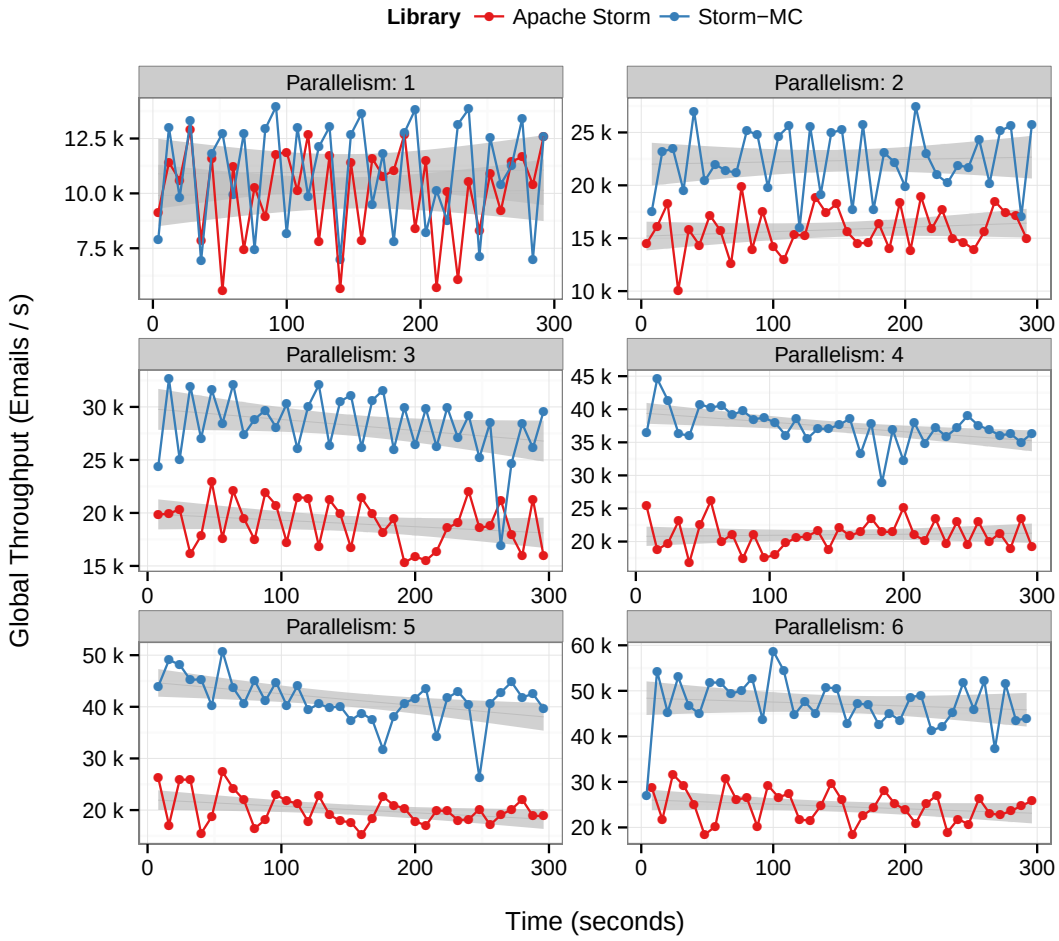


Figure 5.5: Global email throughput over time with standard error.

throughput drops when the spout is trying to read from a file in between the loads. As before, the resident size used by Storm-MC is significantly lower than that of Apache Storm.

5.3.3 RollingSort Topology

The RollingSort topology was ported over from the aforementioned storm-benchmark project. This topology only includes one spout sending tuples to one bolt. The `RandomMessageSpout` spout produces hundred character long strings of random digits from zero to eight. The `SortBolt` bolt then stores a rolling window of hundred of such messages and sorts them every 10 seconds upon receiving a tick tuple. The graphic representation of this topology can be seen in Figure 5.6.

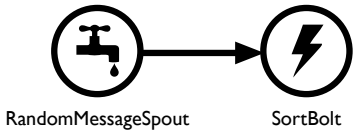


Figure 5.6: RollingSort topology.

This benchmark is considered to be memory-bound: the bolt stores a window of tuples in memory and performs a non-linear time sort. The results of running this benchmark on Storm-MC and Apache Storm can be seen in tables 5.5 and 5.6, respectively.

Parallelism	RandomMessageSpout	SortBolt	CPU Utilisation	Memory Usage
1	249,143,444	249,142,400	186.2%	504.3M
2	444,261,351	444,259,400	352.0%	401.7M
3	350,861,061	350,859,800	514.7%	382.9M
4	412,429,850	412,428,600	675.2%	314.2M
5	470,813,184	470,811,300	835.8%	423.2M
6	498,957,255	498,954,600	989.6%	235.1M
7	519,744,352	519,741,700	1,149.0%	637.9M
8	532,285,376	532,283,800	1,302.9%	618.1M
9	501,519,539	501,517,700	1,430.4%	579.4M
10	555,468,830	555,467,000	1,651.6%	564.7M

Table 5.5: Storm-MC: Component throughput in RollingSort topology.

While Storm-MC still outperforms Apache Storm, the difference in performance is not as high as in previous benchmarks. This is due to the fact that the topology only has two components and the application is mostly memory bound. Storm-MC provides maximum speed improvements for topologies that are mostly CPU-bound and have several components working serially such as WordCount topology.

Parallelism	RandomMessageSpout	SortBolt	CPU Utilisation	Memory Usage
1	173,906,935	173,900,300	267.3%	3.0G
2	226,583,924	226,579,200	468.3%	3.0G
3	310,949,455	310,943,000	634.6%	2.9G
4	362,675,336	362,663,600	815.2%	2.8G
5	409,470,032	409,462,100	969.4%	2.7G
6	435,471,042	435,459,600	1,139.6%	2.6G
7	395,386,336	395,309,900	1,327.7%	2.6G
8	366,680,402	366,553,300	1,509.1%	2.7G
9	359,091,633	358,942,200	1,689.5%	2.7G
10	313,912,451	313,811,300	1,889.5%	2.7G

Table 5.6: Apache Storm: Component throughput in RollingSort topology.

Even in this benchmark, however, Storm-MC beats Storm significantly when the parallelism is high. Furthermore, the tuple throughput increases with parallelism for Storm-MC whereas for Apache Storm it tails off with parallelism set to 6.

5.4 Summary

In summary, we have shown that Storm-MC provides higher tuple throughput per component and globally as compared to Apache Storm running in local mode. Moreover, we have shown that this is especially true for CPU-bound applications with at least several components executing in series. Finally, we have shown that whereas the throughput of Apache Storm starts tailing off at a certain point, the throughput of Storm-MC grows in proportion to component parallelism.

Chapter 6

Conclusion

This final chapter concludes with a discussion on future work that could stem from this project (6.3), describes challenges encountered while building Storm-MC (6.2), and presents a summary of contributions of this project (6.3).

6.1 Future Work

Storm-MC could be improved in a number of ways. Following are ideas that were out of scope of this project but we would like to see get implemented in the future:

Storm-MC as a Server
: Storm-MC could be updated to allow server-like execution. This could have several benefits such as being able to execute multiple topologies at the same time with a thin wrapper that could control their execution just like the Nimbus service in Apache Storm. This was not implemented as part of this project as we assumed most of the time users would be executing one topology at a time.

Higher Level Abstractions
: Defining components of a Storm-MC topology is fairly simple. Users of the library only need to define how components are connected and how they consume and produce tuples. However, this could be taken even further with the user specifying high-level functions and the Storm-MC library figuring out how to parallelise the work that needs to be carried out. In Apache Storm this is implemented with the Trident API which was not ported as part of this project.

Automatic Parallelism
: Sometimes when configuring a topology it may be difficult to predict the rate at which spouts are going to produce tuples. If the rate is underestimated consumers could be lagging behind producers. On the other hand, if the rate is overestimated consumers could be idle, not doing any useful work. Thus it

could be advantageous to have an automatic parallelism setting which could add or remove consumers based on the current tuple rate.

It may seem that this would be trivial to implement with a pool of threads representing one component. However, there are several problems that need to be considered. For example, fields grouping guarantees that tuples with the same field values go to the same executor. Changing the parallelism at runtime breaks this guarantee.

Alternatively each executor could use a pool of threads. This comes with its own set of problems: the executor object would have to provide synchronised access to the pool which would only increase tuple latency.

Performance Comparison with Distributed Storm

The benchmarks in this report compared Storm-MC to Apache Storm running in local mode. It would be interesting to see how Storm-MC compares to Apache Storm running on a cluster. One could compare the number of nodes required in a cluster to the number of cores required in a multi-core server to achieve certain throughput for a given topology. Benchmarks like this could provide insight into when it becomes advantageous to deploy the topology to a cluster. These benchmarks were not included in this project because we did not have access to a cluster.

6.2 Challenges

In this section we discuss challenges we encountered while porting Apache Storm to multi-core. We also try to provide a critical analysis of the project.

Storm API Compatibility

A big challenge while working on this project was ensuring that the final system is backwards compatible with the Apache Storm API. Doing this ensured that existing applications developed for Apache Storm can be executed with Storm-MC. On the other hand, this was sometimes limiting the possible performance improvements. An example of this is the automatic parallelism scaling mentioned in the previous section.

Unfamiliarity with Clojure

One of the main challenges while working on this project was learning a new programming language - Clojure. Since most of the implementation of Apache Storm is written in Clojure, this language had to be studied and its concepts well understood for us to be able to write code that worked with the existing codebase. By the end of the project writing Clojure has become second nature to us but initially progress was slow.

Lack of Documentation

Even though Apache Storm is a popular project documentation is available only for the high level concepts used within Storm. The implementation

details are often obscured away in hard to understand functions. Since the documentation is lacking our knowledge of Storm had to be obtained by reading the source code of an initially unfamiliar language. By the end of the project the Storm-MC codebase became well documented and we might attempt back-porting parts of it to Apache Storm.

6.3 Summary of Contributions

The primary contribution of this project is Storm-MC - a library aimed at data stream processing applications. The benefits of using Storm-MC are twofold:

- It offers the same easy-to-use API as Apache Storm.
- It is tailored to multi-core environments.

Since Storm-MC uses the same API as Apache Storm, applications written with Storm in mind can be ported to use Storm-MC with minimum amount of effort. Thus if an application requires parallelism satisfiable by a single multi-core machine, it can be executed efficiently on one machine instead of a cluster.

Moreover, the Storm API allows programmers to create data stream processing applications on multi-core with an unprecedented ease. All of this comes with the superior performance Storm-MC offers compared to running Apache Storm in local mode, as shown in Section 5.3.

Bibliography

[1] J. Dean and S. Ghemawat, "Mapreduce: A flexible data processing tool," *Commun. ACM*, vol. 53, no. 1, pp. 72–77, 2010. DOI: 10.1145/1629175.1629198. [Online]. Available: http://doi.acm.org/10.1145/1629175.1629198.

[2] (2015). Apache Hadoop, [Online]. Available: https://hadoop.apache.org (visited on 03/25/2015).

[3] (2015). Apache Storm, [Online]. Available: https://storm.apache.org (visited on 03/20/2015).

[4] E. Solovey. (2015). Handling five billion sessions a day – in real time, [Online]. Available: https://blog.twitter.com/2015/handling-five-billion-sessions-a-day-in-real-time (visited on 03/20/2015).

[5] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer, "Moa: Massive online analysis," *J. Mach. Learn. Res.*, vol. 11, pp. 1601–1604, Aug. 2010, ISSN: 1532-4435. [Online]. Available: http://dl.acm.org/citation.cfm?id=1756006.1859903.

[6] K. A. Kumar, J. Gluck, A. Deshpande, and J. Lin, "Hone: "scaling down" hadoop on shared-memory systems," *PVLDB*, vol. 6, no. 12, pp. 1354–1357, 2013. [Online]. Available: http://www.vldb.org/pvldb/vol6/p1354-kumar.pdf.

[7] B. Wheeler. (Jul. 2011). Tilera sees opening in clouds, [Online]. Available: http://www.linleygroup.com/newsletters/newsletter_detail.php?num=4732 (visited on 03/26/2015).

[8] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, Y. O. Koçberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Özer, and B. Falsafi, "Scale-out processors," in *39th International Symposium on Computer Architecture (ISCA 2012), June 9-13, 2012, Portland, OR, USA*, IEEE Computer Society, 2012, pp. 500–511, ISBN: 978-1-4673-0475-7. DOI: 10.1109/ISCA.2012.6237043. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2012.6237043.

[9] (2014). Apache Storm: Powered By, [Online]. Available: https://storm.apache.org/documentation/Powered-By.html (visited on 03/25/2015).

[10] L. Golab and M. T. Özsu, "Issues in data stream management," *SIGMOD Record*, vol. 32, no. 2, pp. 5–14, 2003. DOI: 10.1145/776985.776986. [Online]. Available: http://doi.acm.org/10.1145/776985.776986.

[11] (2015). Apache Spark, [Online]. Available: https://spark.apache.org (visited on 03/20/2015).

[12] (2015). Apache S4, [Online]. Available: http://incubator.apache.org/s4 (visited on 03/20/2015).

[13] M. M. Gaber, A. Zaslavsky, and S. Krishnaswamy, "Mining data streams: A review," *SIGMOD Rec.*, vol. 34, no. 2, pp. 18–26, Jun. 2005, ISSN: 0163-5808. DOI: 10.1145/1083784.1083789. [Online]. Available: http://doi.acm.org/10.1145/1083784.1083789.

[14] G. Holmes, A. Donkin, and I. Witten, "Weka: A machine learning workbench," in *Proc Second Australia and New Zealand Conference on Intelligent Information Systems*, Brisbane, Australia, 1994.

[15] I. Raina, S. Gujar, P. Shah, A. Desai, and B. Bodkhe, "Twitter sentiment analysis using apache storm," *International Journal of Recent Technology and Engineering(TM)*, vol. 3, pp. 23–26, 2014, ISSN: 2277-3878.

[16] S. Akhter and J. Roberts, "Multi-core programming," in. Intel press Hillsboro, 2006, vol. 33, p. 11.

[17] A. Prat-Pérez, D. Dominguez-Sal, J.-L. Larriba-Pey, and P. Trancoso, "Producer-consumer: The programming model for future many-core processors," in *Proceedings of the 26th International Conference on Architecture of Computing Systems*, ser. ARCS'13, Prague, Czech Republic: Springer-Verlag, 2013, pp. 110–121, ISBN: 978-3-642-36423-5. DOI: 10.1007/978-3-642-36424-2_10. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36424-2_10.

[18] N. Marz. (2015). About Me, [Online]. Available: http://nathanmarz.com/about/ (visited on 03/20/2015).

[19] (2015). Apache Zookeeper, [Online]. Available: http://zookeeper.apache.org (visited on 03/15/2015).

[20] (2015). Apache Thrift, [Online]. Available: https://thrift.apache.org (visited on 03/15/2015).

[21] (2015). Esoteric Software Kryo, [Online]. Available: https://github.com/EsotericSoftware/kryo (visited on 03/20/2015).

[22] (2015). Netty, [Online]. Available: http://netty.io (visited on 03/15/2015).

[23] (2015). LMAX Disruptor, [Online]. Available: https://lmax-exchange.github.io/disruptor/ (visited on 03/20/2015).

[24] (2015). Apache Kafka, [Online]. Available: http://kafka.apache.org (visited on 03/15/2015).

[25] (2015). Apache HBase, [Online]. Available: http://hbase.apache.org (visited on 03/27/2015).

[26] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, 4:1–4:26, Jun. 2008, ISSN: 0734-2071. DOI: 10.1145/1365815.1365816. [Online]. Available: http://doi.acm.org/10.1145/1365815.1365816.

[27] A. Chatzistergiou and S. D. Viglas, "Fast heuristics for near-optimal task allocation in data stream processing over clusters," in *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, ser. CIKM '14, Shanghai, China: ACM, 2014, pp. 1579–1588, ISBN: 978-1-4503-2598-1. DOI: 10.1145/2661829.2661882. [Online]. Available: http://doi.acm.org/10.1145/2661829.2661882.

[28] K. Chandrasekaran, S. Santurkar, and A. Arora, "Stormgen - A domain specific language to create ad-hoc storm topologies," in *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems, Warsaw, Poland, September 7-10, 2014.*, M. Ganzha, L. A. Maciaszek, and M. Paprzycki, Eds., 2014, pp. 1621–1628, ISBN: 978-83-60810-58-3. DOI: 10.15439/2014F278. [Online]. Available: http://dx.doi.org/10.15439/2014F278.

[29] T. Dimson and M. Ganjoo, "Hailstorm: Distributed stream processing with exactly once semantics," 2014. [Online]. Available: http://www.scs.stanford.edu/14sp-cs240h/projects/dimson_ganjoo.pdf (visited on 03/25/2015).

[30] (2015). Openmpi, [Online]. Available: http://www.open-mpi.org (visited on 03/25/2015).

[31] (2015). Openmp, [Online]. Available: http://openmp.org (visited on 03/25/2015).

[32] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, Ieee, 2007, pp. 13–24.

[33] N. Marz. (Oct. 2014). History of apache storm and lessons learned - thoughts from the red planet - thoughts from the red planet, [Online]. Available: http://nathanmarz.com/blog/history-of-apache-storm-and-lessons-learned.html (visited on 03/15/2015).

[34] (2015). Rabbit MQ, [Online]. Available: http://www.rabbitmq.com (visited on 03/15/2015).

[35] (2015). The Streaming APIs, [Online]. Available: https://dev.twitter.com/streaming/overview (visited on 03/24/2015).

[36] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of reusable object-oriented software*. Pearson Education, 1994, pp. 97–106.

[37] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in *The 7th ACM International Conference on Distributed Event-Based Systems, DEBS '13, Arlington, TX, USA - June 29 - July 03, 2013*, S. Chakravarthy, S. D. Urban, P. Pietzuch, and E. A. Rundensteiner, Eds., ACM, 2013, pp. 207–218, ISBN: 978-1-4503-1758-0. DOI: 10.1145/2488222.2488267. [Online]. Available: http://doi.acm.org/10.1145/2488222.2488267.

[38] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart, "Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads," May 2011. [Online]. Available: http://lmax-exchange.github.io/disruptor/files/Disruptor-1.0.pdf (visited on 03/25/2015).

[39] (2015). LMAX Disruptor Wiki, [Online]. Available: https://github.com/LMAX-Exchange/disruptor/wiki/Performance-Results (visited on 03/20/2015).

[40] (2015). Java Thread Documentation, [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html (visited on 03/22/2015).

[41] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, ser. HPCA '05, Washington, DC, USA: IEEE Computer Society, 2005, pp. 340–351,

ISBN: 0-7695-2275-0. DOI: 10.1109/HPCA.2005.27. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2005.27.

[42] A. Danial, *Cloc*, http://cloc.sourceforge.net, 2014. [Online]. Available: http://cloc.sourceforge.net (visited on 03/26/2015).

[43] (2015). IBM - InfoSphere Streams, [Online]. Available: http://www.ibm.com/software/products/en/infosphere-streams (visited on 03/24/2015).

[44] J. Warner, *Top manual page*, http://linux.die.net/man/1/top, 2015. [Online]. Available: http://linux.die.net/man/1/top (visited on 03/24/2015).

Appendices

Appendix A

Listings

```
public class RandomSentenceSpout extends BaseRichSpout {
    SpoutOutputCollector _collector;
    Random _rand;

    public void open(Map conf, TopologyContext context,
            SpoutOutputCollector collector) {
        _collector = collector;
        _rand = new Random();
    }

    public void nextTuple() {
        Utils.sleep(100);
        String[] sentences = new String[]{
            "the cow jumped over the moon",
            "an apple a day keeps the doctor away",
            "four score and seven years ago",
            "snow white and the seven dwarfs",
            "i am at two with nature" };
        String sentence = sentences[_rand.nextInt(sentences.length)];
        _collector.emit(new Values(sentence));
    }

    public void ack(Object id) {}

    public void fail(Object id) {}

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }

}
```

Listing 2: RandomSentenceSpout.java: Definition of a spout that emits a randomly chosen sentence from a predefined collection of sentences.

```
public static class SplitSentence extends ShellBolt
    implements IRichBolt {

    public SplitSentence() {
        super("python", "splitsentence.py");
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }

    public Map<String, Object> getComponentConfiguration() {
        return null;
    }
}
```

Listing 3: SplitSentence.java: Definition of a bolt that executes a Python script.

```
import storm


class SplitSentenceBolt(storm.BasicBolt):

    def process(self, tup):
        words = tup.values[0].split(" ")
        for word in words:
            storm.emit([word])


SplitSentenceBolt().run()
```

Listing 4: splitsentence.py: Definition of a bolt that splits sentences on whitespace in Python.

```
public static class WordCount extends BaseBasicBolt {
    Map<String, Integer> counts = new HashMap<String, Integer>();

    public void execute(Tuple tuple, BasicOutputCollector collector) {
        String word = tuple.getString(0);
        Integer count = counts.get(word);
        if (count == null)
            count = 0;
        count++;
        counts.put(word, count);
        collector.emit(new Values(word, count));
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word", "count"));
    }
}
```

Listing 5: WordCount.java: Definition of a bolt that counts word frequencies.