Storm on Multi-core

Mark Nemec

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

2015

Abstract

This is the abstract.

Acknowledgements

Acknowledgements go here.

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 frameworks for distributed computation have emerged, most notably Apache Hadoop [2] for batch processing and Apache Storm [3] for real-time data stream processing.

The main idea of these frameworks is to split the work that needs to be carried out and distribute it across nodes of a cluster. Commercial companies and researchers have been able to utilise these frameworks and create distributed systems which can accomplish things that would not be otherwise possible. This has mostly been allowed by the low price and good horizontal scaling 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 standard. Moreover, most high-end servers support multiple processor sockets thus furthering the parallelisation possible with a single machine even more.

The price of multi-core servers has been going down as well. In 2008, a typical Hadoop node had two dual-core processors and 4 GB of random access memory (RAM). Nowadays, a server with two eight-core processors and 256 GB of RAM can be purchased for roughly $10,000 USD [4]. Hence a single server today might have better processing power than a small cluster from a few years ago [4].

Even though the cost of a commodity hardware cluster might be lower than the price of a single computer with equal power there are certain limitations. These limitations are further explored in section 2.2.

1.2 Main Idea

The main idea of this report 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 allows programmers to take an existing application written with Apache Storm in mind and run it on a multi-core server. This way, we can avoid network latency and enjoy the significant performance improvements of a shared-memory environment.

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 over to Storm-MC.
- **Chapter 5** gives an overview the implementation details of Storm-MC.
- **Chapter 6** discusses the evaluation results of Storm-MC.
- **Chapter 7** presents the conclusion of this report.

Chapter 2

Literature Review

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

2.1 Data Stream

Data stream can be defined 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” [5].

2.1.1 Comparison to a Database Management System

Historically, data has been stored into 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 recently applications started ”feeling the need” to analyse rapidly changing data on-the-fly.

This has brought on an advent of data stream processing. Several stream-processing frameworks have emerged such as Apache Storm [3], Apache Spark [6], and Yahoo S4 [7]. These frameworks, usually ran on a cluster, provide the user with abstractions which greatly simplify writing a real-time data stream processing application.

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, the query is simplified because it can only be ran in one pass, or because it is used with a sampling rate which does not include all events. A typical data stream analysis using windows and sampling is depicted in figure 2.1.

2.1.2 Querying a Data Stream

The assumption behind using a time window is that users of the 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 for a query [8].

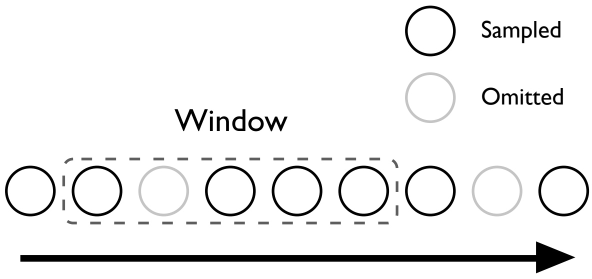


Figure 2.1: Stream Querying.

Even though the answer might only be approximate it can have great value because the query is answered at the right time. Furthermore, even though the query may run only on a subset of data it is still possible to detect trends or system failures. For example, Twitter are using Apache Storm to run real-time analysis on millions of events per second for their analytics product [9].

In research, several techniques have been developed to enable real-time data stream mining. For example, the MOA environment created by **(author?)** [10] which enables real-time machine learning using the WEKA machine learning workbench [11].

2.2 Multi-core

A multi-core processor is a processor that has two or more independent execution cores. “By providing multiple execution cores, each sequence of instructions, or thread, has a hardware execution environment entirely to itself. This enables each thread [to] run in a truly parallel manner” [12].

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 [13] .

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 over a cluster:

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

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

More Control than with a Cloud Provider
Alternatively, one could rent out nodes on cloud computing services such as Amazon EC2 or Rackspace. While the cost of such services is acceptable the user does not have full control over their system.

2.2.2 Disadvantages Over Clusters

On the other hand, there are certain disadvantages in running a computation on 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® Processor E5-2699 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 if it is a 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 provision a new machine very easily. Hence it is advantageous to use multi-core only if one can afford to maintain them as well.

It is generally believed that writing parallel software is hard. The traditional techniques of message passing and parallel threads sharing memory require the programmer to manage the concurrency at a fairly low level, either by using messages or locks.

Apache Storm has become the de facto tool used in stream processing on a cluster and according to their "Powered By" page there are tens of companies already using Storm to process their real-time streams. It would be nice if they could keep that code.

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. [14] BackType was later acquired by Twitter which is when Storm became open source. Storm was incubated into Apache 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 execute in real time.

2.3.1 Dependencies

Storm has five major dependencies:

Apache Zookeeper
Apache Zookeeper [15] 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 detail on how Storm uses Zookeeper is given in section 3.4.3.

Apache Thrift
Apache Thrift [16] is a cross-language framework for developing services. It allows you to write a definition file for services and data types required by your application and automatically generates interface code which supports remote procedure calls and serialisation of the data types.

Kryo
Kryo [17] is a serialisation library. It is used by Storm to serialise objects when sent over the network between nodes of a cluster.

Netty
Netty [18] 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 [19] is a high-performance data structure used to exchange data between concurrent threads. It uses a lock-free implementation of a ring buffer which components of a Storm program running on the same node a cluster use to exchange messages.

2.3.2 Usage of Apache Storm

Storm works well with sister Apache projects such as Apache Kafka [20] and Apache HBase [?]. Apache Kafka is a messaging broker that is often used as the missing link between producers and consumers of a cluster. Apache HBase is a big data-store that allows real-time random reads and writes modelled after Google's Bigtable project [21].

Storm is reportedly used by 81 companies listed on their website [?] and possibly many others. Storm's popularity is one of the reasons why it was chosen for this project. Moreover, we believe that the concepts used in Storm (explained further in section 3.2) apply to many different situations and many applications can be easily adapter to work on Storm.

There has been research into how to optimise computations running on a Storm cluster. [22] looked at how to reconfigure the job by reallocating component tasks to minimise communication cost.

2.4 Similar Efforts

Currently data stream processing is a province of distributed systems such as the ones mentioned in the previous section. Most languages support parallel execution and there are many libraries that ease the process of writing parallel programs on a single multi-core machine. However, they are not tailored to data stream processing and usually require the programmer to do the heavy lifting rather than abstract it away.

There has been effort to port Hadoop to multi-core in [4] (Hone) as well as port of Google's MapReduce in [23] (Phoenix). However, to our knowledge there has not been effort to port Storm or any other real-time data stream framework to multi-core.

2.5 Summary

Distributed real-time computation systems such as Apache Storm provide programmers with abstractions that make it very easy to implement a data stream processing applications on top of a cluster. However, in case of single multi-core machines there are not any obvious software choices. While several frameworks that allow the programmer to parallelise the computation exist, they are not really tailored to data stream processing.

The following chapters provide a closer analysis of Apache Storm as well as a port of Apache Storm for a single multi-core machine.

Chapter 3

Background

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

3.1 Storm Overview

Apache Storm was developed in a mix of Java and Clojure. As mentioned by the author of Storm in [24], 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 [25] or Apache Kafka but can also generate its own stream or read from somewhere like the Twitter streaming API [26].

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.

3.2.2 Additional Concepts

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 the bolt's input streams amongst the instances of the bolt task. 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 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 executors in the same worker process, 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 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 some event of a component.

3.3 Example Topology

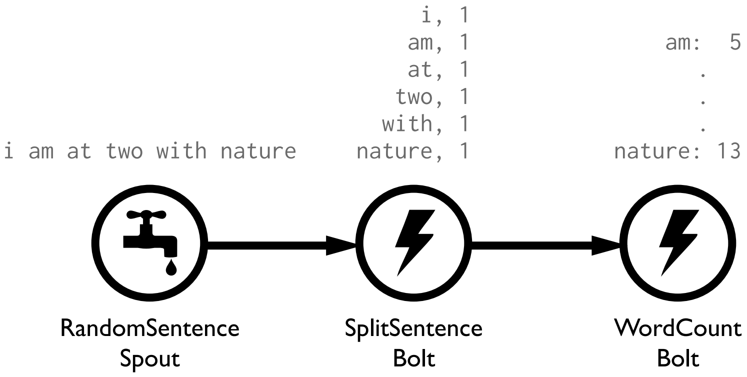


Figure 3.1: WordCount topology.

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 [27] to build up the topology which is then submitted to Storm 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 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

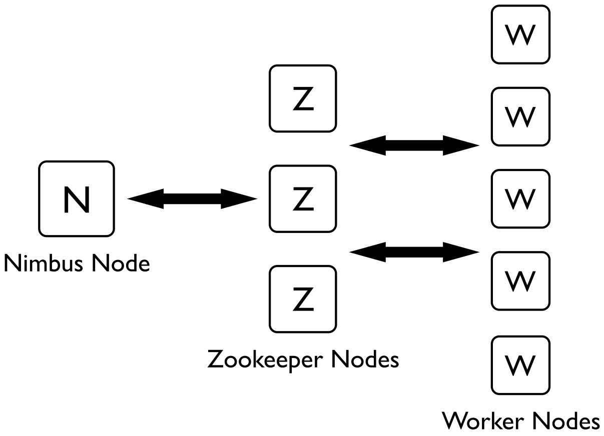


Figure 3.2: Apache Storm Architecture.

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 that the submitted topology is valid before trying to execute it. The user can use his own validator by implementing the `ITopologyValidator` interface or use the default validator provided by Storm.

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. [28]

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 [16] 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. 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

The 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 the design of Storm-MC. We describe how Apache Storm behaves on multi-core machines (4.1), how the Storm architecture was ported over to multi-core (4.2), and we list feature differences between Apache Storm and Storm-MC (4.3).

4.1 Apache Storm on Multi-core

To begin, we discuss why Apache Storm does not perform optimally on a single multi-core machine. Storm can be ran in local mode where it emulates execution on a cluster. This mode exists so that it is possible to debug and develop 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

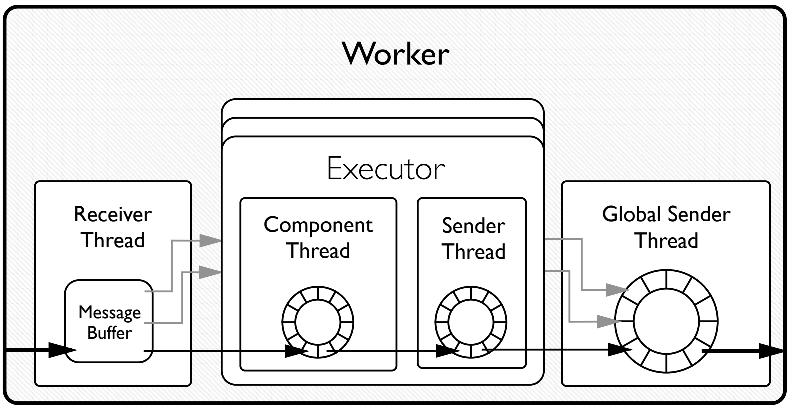


Figure 4.1: Tuple processing in Apache Storm.

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 puts the 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. Lastly, the global sender thread of the worker serialises the tuple and sends it downstream.

Alternatively, if the tuple is forwarded 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 Storm are implemented as ring buffers using the LMAX Disruptor library [19]. Detailed background on how Disruptor works and its performance benchmarks can be found in [29].

There is significant overhead required to simulate 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 has half the throughput of a single consumer-producer pipeline [30].

Furthermore, to emulate over-the-network messages Storm uses a `Hashmap` of `LinkedBlockingQueues` which according to [29] has several orders of magnitude lower performance than the Disruptor. 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.

4.1.2 Thread Overhead

Maybe mention waiting strategies?

Acker Bolt

The Acker is included in every topology. The Acker bolt can be disabled via the configuration file. In such a case it is mostly idle since it does not receive any messages but it can still use up resources especially if it waits for messages using a busy waiting strategy.

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 i.e. 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 states 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 one worker and three executors 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: Breakdown of threads used by Storm to execute a 3-component topology.

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: Average number of recorded thread states over a three minute period.

Even though three minutes may seem to be a very short amount of time the fact that there is almost no variation shows that it is sufficient. As can be seen from the table, most of the threads were either in state `WAITING` or `TIMED WAITING`. According to the Java documentation on thread states [31] 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.

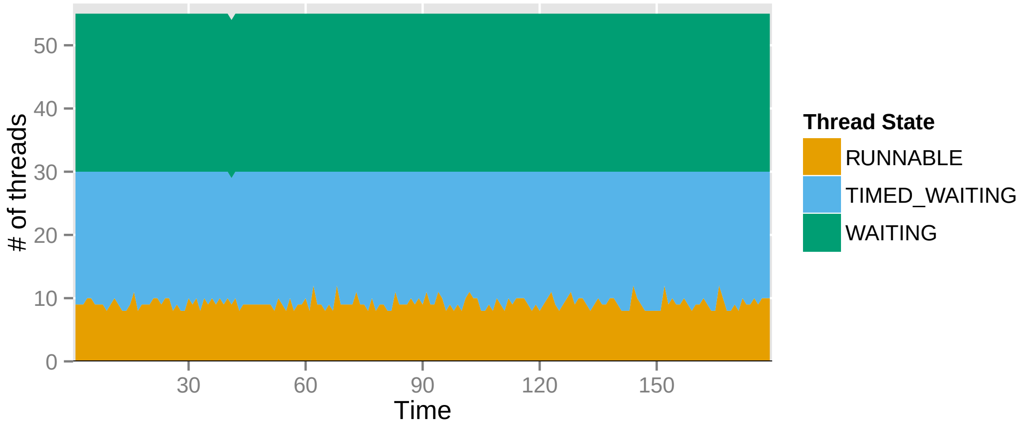


Figure 4.2: Thread state distribution over time.

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 [31]. Hence, these are threads directly competing to be scheduled by the OS. This means that for three components running in parallel there are five threads doing potentially unnecessary work.

In the subsequent sections we will show that these threads were in fact unnecessary and we will discuss how the 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 process running all the executor threads of a topology.

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

4.2.1 Nimbus

Unlike Nimbus executing on a Storm cluster, Nimbus in Storm-MC does not support running multiple topologies at the same time. However, to do that one only needs to run the topology in a separate process. This is because unlike when executing on the cluster different topologies do not need to share any state and it is more natural to execute them as separate processes.

This has the added benefit of each process having its own part of main memory thus reducing cache conflicts as shown in [32] and providing higher security by not having different topologies share memory space. Additionally, if a single thread of one topology is blocking it does not block other topologies.

Nimbus on Storm-MC does not support scheduling topologies. Since within one process there is only one topology running at a time and the hardware configuration of the machine does not change, 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 the 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 passing it along to the worker part of the process which handles the topology execution.

4.2.2 Worker

In Apache Storm, a worker node runs the supervisor daemon, which in turns launches worker processes which contain executor threads which contain tasks. In Storm-MC, however, there is only one worker process which contains all the executor threads and their tasks.

This design has several benefits:

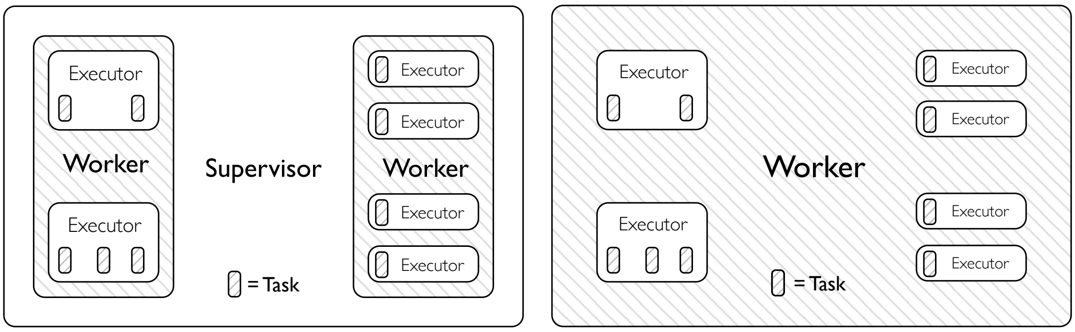
- All the inter-thread communication is occurring within one worker process.
- Supervisor can be removed as there is no need to synchronise multiple workers.
- There is no need to simulate over-the-network message passing.
- Message passing between executor threads within a worker stays the same as in Apache Storm.

The role of worker is to launch executors and provide them with a shared context through which they can communicate. This is done by via a map of executor identifiers to Disruptor queues which the executors use to pass tuples between each other.

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

4.2.3 State

As mentioned before, Storm-MC is completely stateless. The cluster state that was managed by Zookeeper in Apache Storm was completely stripped away. This state



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

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

was only relevant when multiple topologies were sharing resources.

4.2.4 Serialisation

Great amount of work was put into removing the dependency of Storm-MC on Apache Thrift. This was done not only for optimisation purposes but also to reduce code bloat and remove an unnecessary dependency since there is no serialisation required in multi-core communication.

Mention overhead by running SystemBolt vs just timer in Storm-MC

4.2.5 Tuple Processing

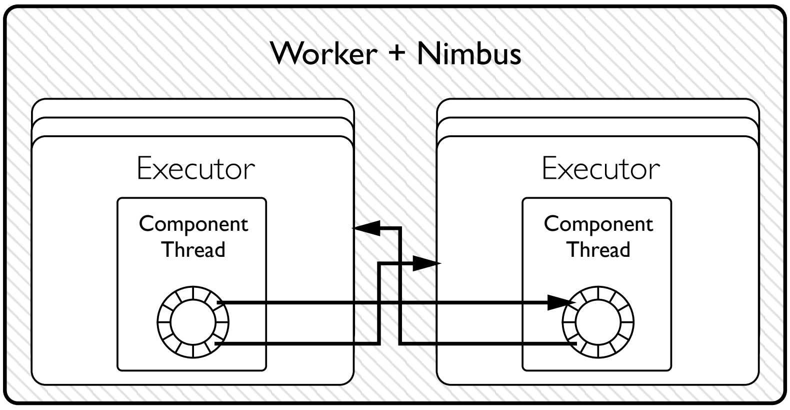


Figure 4.4: Tuple processing in Storm-MC.

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 were stripped away and there is only one Disruptor queue for every executor. Once an executor is done processing a tuple it simply puts it on the Disruptor queue of its downstream bolts.

Thus the tuple processing in Storm-MC is a variant of multiple producer single consumer problem. We considered several other options such as `ArrayBlockingQueue` when implementing the tuple processing mechanism. However, the Disruptor shows superior throughput and latency compared to alternative solutions [30].

4.3 Differences between Apache Storm and Storm-MC

Feature	Apache Storm	Storm-MC
Multiple Topologies	✓	✗
Trident Support	✓	✗
Built-in Metrics	✓	✗
Nimbus as a Server	✓	✗
Multi-language Topologies	✓	✓
Hooks	✓	✓
Metrics	✓	✓
Tick Tuples	✓	✓

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

Chapter 5

Storm-MC Implementation

The following chapter describes the implementation of Storm-MC.

5.1 Nimbus

5.2 Worker

As mentioned in previous chapter Storm-MC was implemented with only one worker per topology.

Chapter 6

Evaluation

In this chapter we evaluate Storm-MC. We describe the metrics used to evaluate performance of Storm-MC (6.1), list the configuration used for benchmarking (6.2), compare Storm-MC to Apache Storm executing in local mode on a set of different topologies (6.3), and finally talk about challenges encountered while designing Storm-MC (6.4).

6.1 Evaluation Metrics

The system was evaluated on the following metrics:

Throughput
The number of tuples processed by every component in the given time of the topology is recorded.

CPU utilisation
Usage of CPU is recorded every **x** seconds throughout execution and is then averaged.

Memory utilisation
Main memory usage is recorded every **x** seconds throught execution and is then averaged.

6.2 System Configuration

6.2.1 Software Setup

Change versions below as applicable. Link to GitHub for source?

All performance benchmarks were ran using the following software packages:

- Apache Storm version 0.9.2
- Storm-MC version 0.1.6
- A fork of IBM Storm Email Benchmarks version 0.1.10
- Storm-benchmark version 0.1.0

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 results 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.

IBM open sourced a suite of benchmarks which they used to compare Apache Storm to their real-time stream system IBM Infosphere Streams [33]. These benchmarks were adapted and used to benchmark Storm-MC against Apache Storm.

Lastly, a number of spout and bolt components were used from the storm-benchmark project which Apache Storm developers use to benchmark Storm.

Since Storm-MC reuses package names from Apache Storm, the same benchmark is directly executable by both libraries. This saved a lot of time and furthermore there is no need to maintain two benchmarks suites.

Go into more detail which components were re-used and where?

6.2.2 Hardware Setup

Processor

Following is the processor used in the multi-core machine used in benchmarks: Intel® Xeon® E5-2690 v2 @ 3.00 GHz. The machine has two sockets each with the same processor. This processor has 10 physical cores with Hyper-Threading Technology (maybe link here) which means it can handle up to 20 threads in parallel. Thus with two sockets, there is potential to execute 40 threads in parallel.

Main Memory

The machine has 378 GB of main memory.

I/O

The machine uses Andrew File System (AFS).

This is currently student.compute, find out what it is.

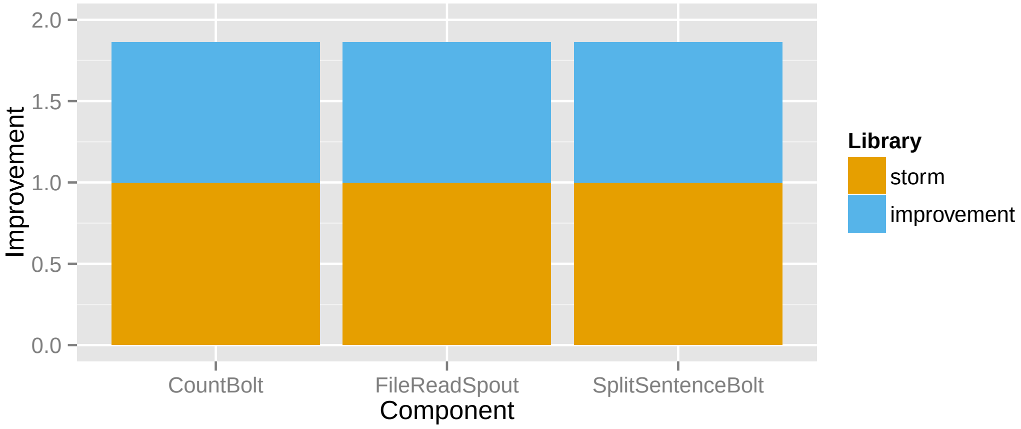


Figure 6.1: Improvement of Storm-MC over Apache Storm in number of tuples processed

6.3 Performance

To assess performance of Storm-MC, 4 different benchmarks were executed, each with a different focus. The benchmarks were executed for a constant period of time (five minutes) after which 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 the JVM to 10G which was more than enough for all benchmarks.

The parallelism of components was varied from one to six and average CPU utilisation and resident memory size were recorded by the Unix top program [34]. CPU Utilisation Maximum utilisation with 40 threads available 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 OS manages memory but it is the only fairly reliable memory metric reported by top that can be used for Java programs. This is because the amount of virtual memory used by a Java program can be skewed by the JVM.

6.3.1 WordCount Topology

The first topology we tested for performance is a variant of the aforementioned WordCount topology. Recall, that this topology is shown graphically on figure 3.1. The topology was ran with 3 executors running for every component.

This topology is considered to be CPU-intensive.

```
## Loading required package: methods
```

Parallelism	FileReadSpout	SplitSentenceBolt	CountBolt	CPU Utilisation	Memory Usage
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
10	40,341,798	40,336,962	353,490,567	1967.4%	
20	60,798,276	60,790,475	532,737,413	3161.5%	

Table 6.1: Storm-MC: Tuples processed per component in WordCount Topology.

Parallelism	FileReadSpout	SplitSentenceBolt	CountBolt	CPU Utilisation	Memory Usage
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
10				%	
20				%	

Table 6.2: Apache Storm: Tuples processed per component in WordCount Topology.

Could be slower because not on same socket => slower inter-thread comms.

When the topology was run on Apache Storm in local mode, the process executed with 55 threads. Compared to that, running it on Storm-MC required only 5 threads: the main thread (1), one thread for each component (3), and a user timer used for topology metrics and ticks (1).

6.3.2 Enron Topology

Next, we tested the Enron topology from the IBM benchmarks. In this topology, serialised emails from the Enron email database are read from a file by a spout. They are further deserialised by one bolt, filtered by another bolt, modified by yet another bolt and then finally metrics are recorded by another bolt.

Similarly, to the WordCount topology this topology is serial in nature. However, whereas the WordCount topology keeps the random sentences in memory, the Enron topology reads from a file. Thus, this benchmark is mostly I/O intensive.

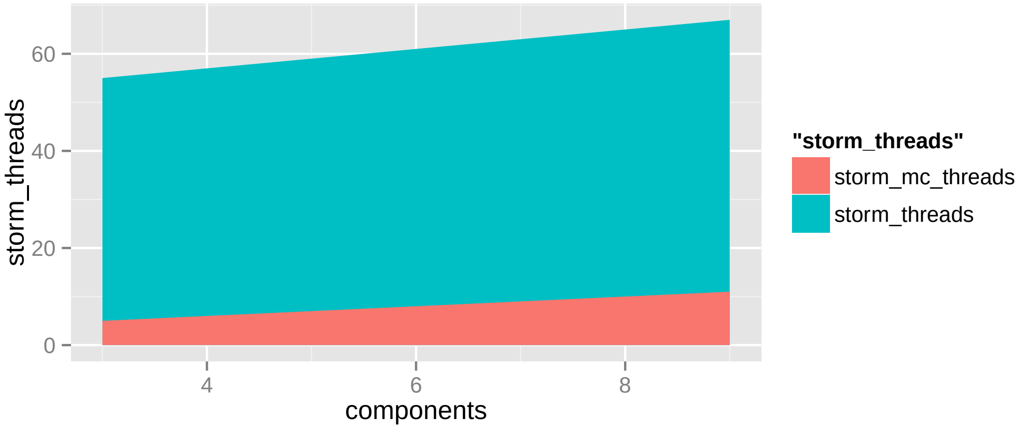


Figure 6.2: Number of threads used by Storm and Storm-MC

Parallelism	Email Throughput	CPU Utilisation	Memory Usage
1	10,868.89	297.7%	806.8M
2	22,150.44	482.1%	756.1M
3	28,079.70	729.5%	341.4M
4	36,717.50	1036.9%	326.0M
5	41,788.59	1311.0%	260.8M
6	46,595.78	1590.3%	334.0M

Table 6.3: Storm-MC: Email Throughput in Enron Topology.

Parallelism	Email Throughput	CPU Utilisation	Memory Usage
1	9,624.53	406.6%	1.94G
2	15,897.62	945.1%	2.93G
3	18,481.66	1,427.4%	3.32G
4	20,517.13	1,891.2%	3.56G
5	20,091.93	2167.4%	3.65G
6	23,769.80	2388.6%	4.09G

Table 6.4: Apache Storm: Email Throughput in Enron Topology.

6.3.3 RollingSort Topology

The RollingSort topology is ported over from the aforementioned storm-benchmark project. This topology includes one spout and one bolt. The spout produces hundred character-long strings of random digits from zero to eight. The bolt stores a rolling window of hundred of these messages and sorts them every x seconds.

This benchmark is included because it is considered memory-intensive.

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

Table 6.5: Storm-MC: Tuples processed per component in RollingSort 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	1139.6%	2.6G

Table 6.6: Apache Storm: Tuples processed per component in RollingSort Topology.

change x depending on the actual benchmark.

6.4 Challenges

In this section we are going to discuss challenges we encountered while porting Apache Storm to multi-core machines.

Unfamiliarity with Clojure

Lack of Documentation

Chapter 7

Conclusion

7.1 Future Work

Add Trident.

Make Storm-MC a server.

Appendices

Appendix A

Listings

Listing 2 shows the definition of a spout that emits a randomly chosen sentence from a predefined collection of sentences.

Listings 3 and 4 show how a bolt defined in Python can be part of this Java-defined topology.

Finally, listing 5 shows how a bolt that counts the number of word occurrences can be implemented.

```
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

```java
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

```python
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

```java
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

Bibliography

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[2] Apache Hadoop. `https://hadoop.apache.org`, 2015. [Online; accessed 25-March-2015].

[3] Apache Storm. `https://storm.apache.org`, 2015. [Online; accessed 20-March-2015].

[4] K. Ashwin Kumar, Jonathan Gluck, Amol Deshpande, and Jimmy Lin. Hone: "scaling down" hadoop on shared-memory systems. *Proc. VLDB Endow.*, 6(12):1354–1357, August 2013.

[5] Lukasz Golab and M Tamer Özsu. Issues in data stream management. *ACM Sigmod Record*, 32(2):5–14, 2003.

[6] Apache Spark. `https://spark.apache.org`, 2015. [Online; accessed 20-March-2015].

[7] Yahoo. Yahoo s4. http://incubator.apache.org/s4, 2015. [Online; accessed 20-March-2015].

[8] Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. Mining data streams: A review. *SIGMOD Rec.*, 34(2):18–26, June 2005.

[9] Ed Solovey. Handling five billion sessions a day – in real time, 2015.

[10] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. Moa: Massive online analysis. *J. Mach. Learn. Res.*, 11:1601–1604, August 2010.

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

[12] Shameen Akhter and Jason Roberts. *Multi-core programming*, volume 33. Intel press Hillsboro, 2006.

[13] Arnau Prat-Pérez, David Dominguez-Sal, Josep-Lluis Larriba-Pey, and Pedro Trancoso. Producer-consumer: The programming model for future many-core processors. In *Proceedings of the 26th International Conference on Architecture of Computing Systems*, ARCS'13, pages 110–121, Berlin, Heidelberg, 2013. Springer-Verlag.

[14] Nathan Marz. About me. http://nathanmarz.com/about/, 2015. [Online; accessed 20-March-2015].

[15] Apache. Apache Zookeeper. http://zookeeper.apache.org, 2015. [Online; accessed 15-March-2015].

[16] Apache. Apache Thrift. https://thrift.apache.org, 2015. [Online; accessed 15-March-2015].

[17] Esoteric Software. Esoteric software kryo. https://github.com/EsotericSoftware/kryo, 2015. [Online; accessed 20-March-2015].

[18] Netty. Netty. http://netty.io, 2015. [Online; accessed 15-March-2015].

[19] LMAX. Lmax disruptor. https://lmax-exchange.github.io/disruptor/, 2015. [Online; accessed 20-March-2015].

[20] Apache. Apache Kafka. http://kafka.apache.org, 2015. [Online; accessed 15-March-2015].

[21] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[22] Andreas Chatzistergiou and Stratis 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*, CIKM '14, pages 1579–1588, New York, NY, USA, 2014. ACM.

[23] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 13–24. Ieee, 2007.

[24] Nathan Marz. History of apache storm and lessons learned - thoughts from the red planet - thoughts from the red planet, Oct 2014. [Online; accessed 15-March-2015].

[25] RabbitMQ. Rabbit MQ. http://www.rabbitmq.com, 2015. [Online; accessed 15-March-2015].

[26] Twitter. The streaming apis, 2015. [Online; accessed 24-March-2015].

[27] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[28] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive online scheduling in storm. http://www.orgs.ttu.edu/debs2013/presentations/DEBS13-Paper88-Querzoni.pdf, Jul 2013.

[29] Martin Thompson, Dave Farley, Michael Barker, Patricia Gee, and Andrew Stewart. Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads. May 2011.

[30] LMAX. Lmax disruptor wiki. https://github.com/LMAX-Exchange/disruptor/wiki/Performance-Results, 2015. [Online; accessed 20-March-2015].

[31] Oracle. Java thread documentation. http://incubator.apache.org/s4/, 2014. [Online; accessed 22-March-2015].

[32] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 340–351, Washington, DC, USA, 2005. IEEE Computer Society.

[33] IBM - InfoSphere Streams. http://www-03.ibm.com/software/products/en/infosphere-streams. [Online; accessed 24-March-2015].

[34] Jim Warner. top manual page. `http://linux.die.net/man/1/top`, 2015. [Online; accessed 24-March-2015].