Storm on Multi-core

Mark Nemec

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

2015

Abstract

This is an example of `infthesis` style. The file `skeleton.tex` generates this document and can be used to get a "skeleton" for your thesis. The abstract should summarise your report and fit in the space on the first page. You may, of course, use any other software to write your report, as long as you follow the same style. That means: producing a title page as given here, and including a table of contents and bibliography.

Acknowledgements

Acknowledgements go here.

Table of Contents

Chapter 1

Introduction

In recent years, there has been an explosion of cloud computing software. After Google published their paper on MapReduce [?], many new open-source frameworks for distributed computation have emerged, most notably Apache Hadoop for batch processing and Apache Storm 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 multiple 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 possible on a single computer. This has mostly been allowed by the low price of commodity hardware and good horizontal scaling properties.

This project is about taking the ideas from the distributed system Apache Storm and applying them in the context of multi-core machines instead of clusters.

1.1 Motivation

While the cost of a commodity hardware cluster might be lower than the price of a single computer with equal power there are certain limitations:

- The nodes of a cluster communicate through network. This limits the speed of communication between processes that live on different nodes.
- Distributed systems waste resources by replicating data to ensure reliability.
- Running a distributed computation on commodity hardware usually requires a data centre or renting out instances on cloud computing services such as Amazon EC2 or Rackspace. This is not ideal for some use cases which require full control over the system or a heightened level of security.

On the other hand, even though Moore's law still holds true, processor makers now favour increasing the number of cores in CPU chips to increasing their frequency. This trend implies that the "free lunch" of getting better software performance by

upgrading the processor is over and programmers now have to design systems with parallel architectures in mind. However, there are some limitations to this 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.

1.2 Main Idea

The solution proposed in this paper is to take the existing Apache Storm project and port it for multi-core machines. This is implemented in a library Storm-MC with an API compatible with Apache Storm. This allows us to take an existing application written with Apache Storm in mind and run it in a multi-core setting. This way, we can avoid network latency and enjoy the significant performance improvements of a shared-memory environment.

- Prices of high-end server have decreased and one can get a 32-core machine for 10,000 USD.

1.3 Structure of the Report

In chapter 1, blah blah.

Chapter 2

Background

In this chapter we give background information necessary to understand the design of Storm-MC.

Mention Storm is in Java and Clojure?

2.1 Core Concepts

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

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 or Kafka but can also generate its own stream or read from somewhere like the Twitter streaming API.

Bolt A bolt is a component that takes a data-stream as an input and transforms it into a different stream for its output. 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 these components in a directed acyclic graph called topology which describes how the components interact with each other. The topology is then submitted to Storm.

2.2 Additional Concepts

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

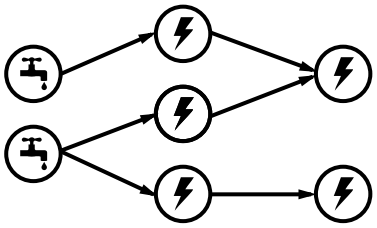

Figure 2.1: An example topology.

Tuple A tuple wraps named fields and their values. The values of 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 its input streams amongst the instances of the bolt task.

Maybe list Grouping types

All the components of a Storm topology execute in parallel. A user can specify how much parallelism he wants associated with every component and Storm spawns the necessary number of threads.

2.3 Example Topology

Maybe write a few paragraphs about some sample topology

2.4 Storm Architecture

Maybe highlight similarities to Hadoop

A Storm cluster adopts a Master-Worker pattern. To set up a Storm topology, user launches daemon processes on nodes of the cluster and submits the topology to the master node. The worker nodes receive task assignments from the master and execute on them.

2.4.1 Master (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:

Is listing the steps involved the right way to explain it?

1. Validates the topology.
2. Distributes the topology source code across the cluster.
3. Runs the scheduler and makes worker assignments.
4. Activates the topology.

Link to Apache Thrift here?

Nimbus is an Apache Thrift service (more on Thrift in section 2.5) that listens to commands submitted by clients and modifies the state of a cluster accordingly. Following are commands supported by Nimbus:

Is listing this really necessary?

Submit a topology
: Clients can submit a topology defined in a jar file. The Nimbus service then ensures that the topology configuration and resources are distributed across the cluster and starts executing the topology.

Kill a topology
: Nimbus can stop running a topology and remove it from the cluster. The cluster can continue executing other topologies.

Activate/deactivate a topology
: Topologies can be deactivated and reactivated by Nimbus.

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

2.4.2 Worker Nodes

The worker nodes run a daemon called Supervisor. This daemon launches worker processes which execute the tasks assigned to the node. A worker node runs the following processes:

Supervisor
: A supervisor is a daemon process, ran under supervision, that launches worker processes. This daemon is what 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 listen on.

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 queue where it is picked up by one or more executors of the worker process. Furthermore, the worker has a transfer queue where worker's executors enqueue tuples ready to be sent down the stream.

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 the receiver queue, perform the task of a component it represents, and put the transformed tuples on the transfer queue of its worker. Within its thread, executor runs one (the usual case) or more tasks.

Task
A task performs the actual data processing. 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.

2.4.3 Zookeeper Nodes

Lastly, the cluster contains a number of Zookeeper nodes which coordinate the communication between the master and the workers. The Zookeeper nodes can be thought of as the glue of the network. They ensure that messages are delivered.

2.5 Serialisation

Since Storm topologies execute on a cluster all the 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 RPC support. This allows for easy cross-language communication.

Chapter 3

Bringing Storm to Multi-core

The design of Storm-MC was ported over from Apache Storm. This enabled rapid progress while guaranteeing compatibility with Apache Storm API. Clearly, however, some differences had to be made to take advantage of a multi-core machine performance. This chapter explains the design of Storm-MC.

3.1 Apache Storm on Multi-core

To begin, we discuss why Apache Storm does not perform well on a single machine. Storm can be ran in local mode where it only simulates execution on a cluster. This mode was created for being able to develop and debug topologies without needing access to a cluster. There are several reasons why the local mode is not as performant as it could be.

Multi-process Nature of Storm
: In general, multi-process applications suffer overhead from inter-process communication and are generally slower than their multi-threaded counterparts [1].

Fault Tolerance
: Storm is made to be fault-tolerant. This means that it can guarantee a tuple being processed from its spout all the way to its final bolt. To do this it adds an additional acker bolt to every topology. This bolt acts as a root of a tree with the nodes being all the components the tuple "goes" through. If the tuple gets successfully processed by a component its node is marked as "acked". Hence once all nodes of the tree are marked as "acked" Storm can guarantee the tuple was processed.

  Change "goes" and "acked" to something else.

Redundant threads
: Storm runs many threads which are only useful in distributed context. In-

deed, during our experiments we found that a topology with 8 executors was being executed with 64 threads. This included threads which were used as timeout timers or to send heartbeats and were unnecessary in a multi-core setting. Obviously, not all of them were executing in parallel but there is clearly room for reduction.

Maybe include Thread Dump and a graph of thread counts

3.2 Storm-MC Architecture

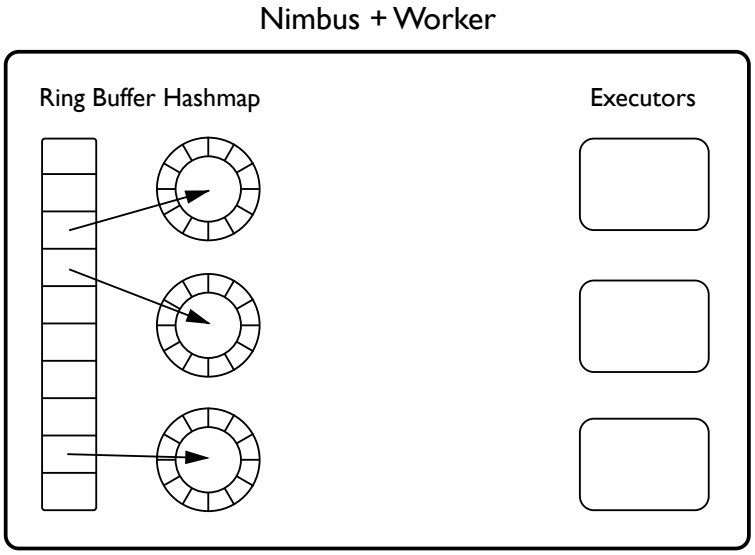

Figure 3.1: Storm-MC Architecture.

The overall architecture of Storm-MC can be seen in figure 3.1. The design we adopted for porting worker nodes is to only have one worker process running multiple executor threads. Additionally, the code for the Nimbus daemon was merged with the worker. This design has several benefits:

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

3.2.1 Nimbus

Rewrite paragraph below, too informal.

In Apache Storm, Nimbus runs as a server that communicates with possibly multiple clients. While appropriate for a system running on a cluster this is unnecessary on a single machine. Thus to run a topology on Storm-MC one only has to import the library into their project and use it as a standard Java library. To kill the topology, one only needs to send the kill signal as in a standard Java program. Furthermore, there is no need for activating and deactivating as the user can just kill and resubmit the topology. Lastly, rebalancing does not really transfer to a multi-core system and was thusly omitted.

As can be understood from the previous paragraph, Storm-MC does not support running multiple topologies at the same time. However, to do that one only needs to run a separate process. This is because unlike on the cluster different topologies do not need to share any memory and it is more natural to execute them as separate processes.

Mention things like cache lines in support of above argument.

Additionally, Storm-MC does not support any scheduling. Since 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.

In light of this Storm-MC is (almost) completely stateless. The state that was managed by Zookeeper in Apache Storm was completely stripped away.

3.2.2 Worker

3.2.3 Message Exchange

LMAX Disruptor is used for tuple passing between components of a stream. Detailed background of how Disruptor works and its performance benchmarks can be found in [2].

There were two types of tuple transfers in Apache Storm:

Inter-worker (remote) transfers
: When the tuple is sent to a different worker, it is put on a disruptor buffer by an executor thread and picked up by a separate sender thread which sends it across the network to the executors subscribed to the stream. The worker of these executors runs a receiver thread which listens to messages on its port and puts the tuples on disruptor buffers of the corresponding executors.

Intra-worker (local) transfers
: When the tuple is sent within the same worker, it is put directly on the disruptor buffer of the executor thus avoiding sending a network message to the same machine.

Since Storm-MC only has one worker process, all tuple transfers are local. Thus, a map of executor identifier to disruptor buffers is maintained. Hence, an executor thread picks up a tuple from the ring buffer corresponding to its identifier and after processing the tuple inserts it into a buffer of executors subscribed to its output stream.

3.3 Things Not Included in Storm-MC

This section name should be rephrased.

Maybe present this as a table of features.

Chapter 4

Evaluation

4.1 Performance

Chapter 5

Conclusion

5.1 Future Work

Bibliography

[1] 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.

[2] 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.