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.

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.

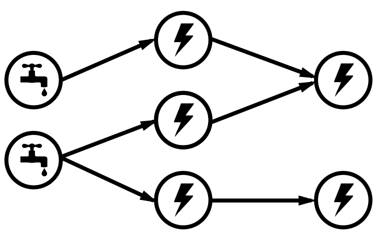

Figure 2.1: An example topology.

2.2 Additional Concepts

Stream A stream is defined as an unbounded sequence of tuples. Streams can be thought of as edges connecting the bolts and spouts which act as vertices.

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.

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 Storm Architecture

The Storm cluster uses a Master-Worker pattern. The master node runs a daemon called Nimbus which receives topologies submitted by clients, makes task assignments to worker nodes, and monitors them for failure. The worker nodes run a daemon called Supervisor which spawns worker processes which execute the tasks assigned to them. Lastly, the cluster contains a number of Zookeeper nodes which coordinate the communication between the master and the workers.

2.3.1 Nimbus Node

The master node (also called Nimbus) is responsible for the following tasks:

1. Receive a submission with a topology and configuration from programmer.
2. Distribute the work performed by different components to worker nodes.

The Nimbus node is fault tolerant

2.3.2 Worker Nodes

The innards of a Worker node are as follows:

Supervisor
A supervisor is a daemon process, run under supervision, that launches worker processes. This daemon is what the user runs on a worker node to make it part of the cluster.

Worker
: A worker is a process that is assigned a port and listens to tuple messages on a socket associated with the port. 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.

Executor
: An executor controls the parallelism within a worker process. Every executor runs in a separate thread. Its job is to pick up tuples from the receiving queue, perform the function of a component they represent, and put the transformed tuples on the transfer queue, ready to be sent along to the next component. An 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.

2.3.3 Zookeeper Nodes

The Zookeeper nodes can be thought of as the glue of the network. They ensure that messages are delivered.

2.4 Serialisation

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

Chapter 3

Design

The design of Storm-MC was ported over from Apache Storm. 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. When running on multi-core architectures, Storm runs in local mode. 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].

Redundant threads
: Storm runs many threads which are only useful in distributed context. Indeed, 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.

3.2 Storm-MC Architecture

The overall architecture of Storm-MC can be seen in figure.

3.2.1 Main Process

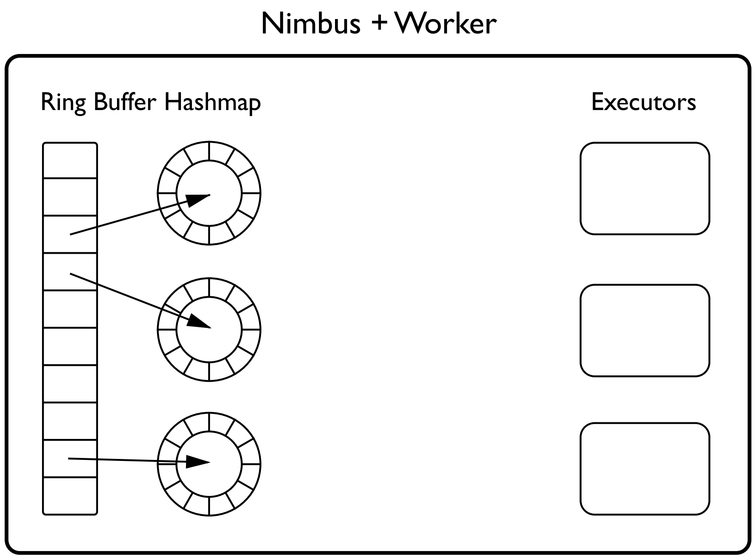

Figure 3.1: Storm-MC Architecture.

Chapter 4

Implementation

Chapter 5

Evaluation

Chapter 6

Conclusion

Of course you may want to use several chapters and much more text than here.

Bibliography

[1] K. Ashwin Kumar Jonathan Gluck, Amol Deshpande, and Jimmy Lin. Optimization techniques for “scaling down” hadoop on multi-core, shared-memory systems. page 14.