MODELING AND IMPLEMENTING DISTRIBUTED COMPUTING WITH APPLICATIONS IN MULTISENSOR-MULTITARGET TRACKING
MODELING AND IMPLEMENTING DISTRIBUTED COMPUTING WITH APPLICATIONS IN MULTISENSOR-MULTITARGET TRACKING

BY

FRANKLYN D’SOUZA, B.Sc.

A THESIS
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING
AND THE SCHOOL OF GRADUATE STUDIES
OF MCMASTER UNIVERSITY
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

© Copyright by Franklyn D’souza, August 2015
All Rights Reserved
TITLE: MODELING AND IMPLEMENTING DISTRIBUTED COMPUTING WITH APPLICATIONS IN MULTISENSOR-MULTITARGET TRACKING

AUTHOR: Franklyn D’souza
B.Sc., (Electrical Engineering)
McMaster University, Hamilton, ON, Canada

SUPERVISOR: Dr. T. Kirubarajan

NUMBER OF PAGES: 50
To my mother and father, without whom I would never have gotten to where I am today.
Abstract

Big data is a term used to describe quantities of data that are too large to process using traditional means of data processing. The rise of such quantities of data amongst virtually every industry has lead to changes in data processing paradigms that favour distributed architectures over centralized processing. However great care must be taken when designing such processing architectures as slight overhead in computing or communication can nullify gains achieved from distributing computations.

In this paper we discuss the theoretical elements involved in designing distributed systems and develop a heuristic for the performance of such a system. Our heuristic helps define when a problem requires distribution and informs designers in choosing the right topology to meet the needs of the problem and hardware involved.

Finally we present results from our own distributed computing architecture applied to a prediction problem in radar image processing.
Acknowledgements

I would like to thank all my friends and friends for encouraging me and supporting me during the period leading up to this thesis.

This thesis involved a lot of topics that are personally interesting to me and I would like express my sincerest gratitude to Dr. T. Kirubarajan for introducing me to the problem and challenges in this area, thus giving me a great opportunity to work on an interesting and challenging problem.

I thank my fellow graduate students in the ECE department for their help and support. I would also like to thank members of the ECE Faculty and Staff for their academic and research advice.
Notation and abbreviations

**Nodes, Machines, Servers** Used to refer to a computer that we are in control of.

**In-memory** Utilizing fast Random Access Memory as opposed to storing data on comparatively slower hard disks.

**Cloud** Third party hosting provider that allows for machines to be leased and networked on demand while being managed by the third party, usually accompanied by several other computing related services such as database and network management.

**DFS** Distributed File System, a file storage system that utilizes a network of computers to store and manipulate files.

**RDD** Resilient Distributed Datasets, Fast in-memory datastructure that manages distributed datasets in a fault tolerant manner.

**CAP** Consistency Availability Partition tolerance, A theorem that explains requirements of a distributed system and the tradeoffs involved in implementing one.

**DAG** Directed Acyclic Graph, A graph without edges or cycles used to describe a series of functional transformations of data.
JNI  Java Native Interface, a programmable interface to allow for data to be shared between native processes and the java virtual machine.
Contents

Abstract iv

Acknowledgements v

Notation and abbreviations vi

1 Introduction and Problem Statement 1
  1.1 Motivation and Contribution of the Thesis . . . . . . . . . . . . . . 1
  1.2 Organization of the Thesis . . . . . . . . . . . . . . . . . . . . . . . 5

2 Distributed Computing Concepts 7
  2.1 The CAP Theorem . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
  2.2 Consensus . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
  2.3 MapReduce . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

3 Problem Classification and Analysis 17
  3.1 Distributed Computing . . . . . . . . . . . . . . . . . . . . . . . . . 17
  3.2 Problem Classification . . . . . . . . . . . . . . . . . . . . . . . . . 18
    3.2.1 CPU-Bound, High data Low CPU throughput . . . . . . . . . 21
    3.2.2 IO-Bound, Low data High CPU throughput . . . . . . . . . . 24
4 Architecture

4.1 Overview of the Architecture ........................................... 28
4.2 Distributed Storage ....................................................... 31
4.3 Distributed Computation .................................................... 33
4.4 Service orchestration ....................................................... 35
  4.4.1 High Availability ...................................................... 39

5 Procedure and Results .................................................... 40

5.1 Problem ........................................................................... 40
5.2 Procedure ........................................................................ 42
5.3 Results ............................................................................ 43
5.4 Conclusion ........................................................................ 44
  5.4.1 Heuristic ..................................................................... 45
  5.4.2 Predictor ..................................................................... 46
5.5 Future work ...................................................................... 46
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Relationships in the CAP Theorem</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Example of state in a distributed system</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Log Replication to achieve state machine consensus</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>Example of a MapReduce DAG</td>
<td>14</td>
</tr>
<tr>
<td>2.5</td>
<td>DAG Nodes composed of three machines</td>
<td>15</td>
</tr>
<tr>
<td>2.6</td>
<td>Data transfer in a shuffle event.</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>High data Low CPU throughput</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>High data Low CPU throughput - with varying cores</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>Low data High CPU throughput</td>
<td>25</td>
</tr>
<tr>
<td>3.4</td>
<td>Low data High CPU throughput - with varying cores</td>
<td>26</td>
</tr>
<tr>
<td>3.5</td>
<td>Running time for various number of cores and machines</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>Network Diagram of our Architecture</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>File replication and distribution in a DFS</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>Block Diagram of a Distributed File System</td>
<td>33</td>
</tr>
<tr>
<td>4.4</td>
<td>Available RDD Transformations</td>
<td>34</td>
</tr>
<tr>
<td>4.5</td>
<td>Distributed registry for services</td>
<td>36</td>
</tr>
<tr>
<td>4.6</td>
<td>System diagram of the infrastructure</td>
<td>38</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction and Problem Statement

1.1 Motivation and Contribution of the Thesis

In the consumer sector the ubiquity of cheap publicly available sensor technology has caused a proliferation in the amount of data being generated and made available, leading to the advent of the so-called 'Big Data' era. Sensors such as GPS, Accelerometers, Temperature and cellular data can be encapsulated into consumer devices leading to complex networks of interconnected devices termed the 'Internet of Things'. These devices can generate vast amounts of event data both independently and while working in concert with each other as they react and sense their environment.

The exponential growth of data both in Military and Consumer applications has necessitated processing methodologies that are capable of ingesting and processing large quantities of data at a reasonable rate. Such a system would need to be robust
to errors in data and fault tolerant so that momentary interruptions do not cause data loss.

One such system is distributed computing and an extension to distributed called cloud computing where compute resources are managed by external third party cloud hosting services. In a cloud computing infrastructure data and processing is distributed amongst a network of computers, computers in the network are aware of each other and processing tasks are orchestrated through a master driver computer.

Starting in the information technology industry where cloud computing has been used to power a range of popular high traffic web services such as Google, YouTube, Facebook etc. Cloud computing has gained adoption in the fields of agriculture[1][2], healthcare[3][4], power transmission [5] and numerous others.

Research done at Fujitsu in [2] has utilized cloud computing infrastructure to increase efficiency for agriculture as an industry in Japan. Although Agriculture has long used cloud-based systems the focus has been on filing tax returns and documentation systems for maintaining traceability records. In [2] a cloud computing system has been developed to aid in data storage and data analysis.

Agricultural Data Storage involves storing information from sensor networks that analyze weather, soil, position and various land parameters. In addition to this a knowledge base has been constructed to record agricultural techniques and inform farmers on the best way to optimize crop yields given the parameters of their land. The Data Analysis system utilizes the cloud infrastructure to process stored sensor information and perform various data mining tasks to answer valuable questions such as profit loss calculations, power virtual models to simulate planting and predicting crop yields.
The value of cloud infrastructure to perform these distributed storage and analysis tasks comes from:

**Reduced Initial Investment**

The ability to create large networks without having to invest in physical components and space dramatically reduces the investment required in building powerful computing systems.

**Resource Allocation**

Cloud computing services allow for new resources to be easily allocated and upgraded on demand whenever they are needed. This allows for system to be easily scaled vertically by improving the performance of existing machines and scaled horizontally by increasing the number of machines.

**Management and Maintenance**

Unlike traditional physical machines, assets allocated in the cloud are managed and maintained by the cloud service. This ensures that the computers in the compute cluster are always available to serve requests without having to invest in maintenance.

In the field of healthcare [4] has discusses several areas where cloud computing has been utilized including Drug Discovery, DNA sequencing and Electronic Records. Computer Aided Drug Discovery (CADD) allows researchers to discover new drugs that target specific diseases, CADD allows researchers to utilize complex molecular models to determine if a proposed drug has the right structural properties in order to target a disease other models can also predict the toxicity and absorption rate of the drug. The models utilized in CADD use complex computations and have
combinatorial challenges that require high performance computational capabilities to simulate and optimize thus gaining from the use of Cloud infrastructure.

DNA sequencing produces amounts of data that far exceed the capabilities of single computers, analyzing this data necessitates the use of computational clusters to ingest and analyze the data using models to attempt to classify functions of different genes and predict the results of different combinations. Electronic Health Records aim to aid clinicians and researchers to access and share patient’s medical data, forming an invaluable base that can be used in combination with machine learning techniques to identify trends in diseases or predict patient outcomes.

The primary goal of Distributed Computing is to enable data retrieval and processing in the presence of network errors and hardware faults. Several methods have been devised to ensure data integrity in a distributed system. Network coding schemes have been developed to split input files into \( N \) segments and retrieve the original file using \( M \) out of \( N \) segments where \( M \leq N \). One such encoding scheme is the Information Dispersal Algorithm (IDA)[6], applications of IDA to distributed systems [7] have shown promising results however the algorithm is quite computationally costly for large data-sets.

Performance requirements of real-time stream processing and batch data processing of Terabyte and Gigabyte scale data would require storage techniques that are simpler to implement, test and run in a feasible amount of time.

The concern of our work is in applying Cloud Computing resources to perform spatial data processing tasks in applications related to Multisensor-Multitarget tracking. Modern tracking systems not only record and process Electromagnetic reflections but also detect and intercept adversarial communications using Electronic Support (ES)
systems which require sampling over large bandwidths. Progress in the field of sampling technology has lead to implementations of ES receivers that are able to record at high sampling rates to capture larger bandwidths. These improvements have lead to a large amount of data generation, So much so that it is becoming impractical to store, process and analyze ES data [8] on a single machine.

1.2 Organization of the Thesis

The thesis is organized as follows, Chapter 2 presents a summary of theory and technology that our architecture is composed of, we discuss fundamental limitations behind distributed computing with an introduction to the Consistency Availability Partition tolerance (CAP) theorem that relates important relationships and restrictions in distributed system design, we then present a brief discussion on the MapReduce execution architecture.

In Chapter 3 we discuss tools that aid in designing appropriate distributions for different types of problems, when designing a distributed architecture one must be able to develop a solution that meets the needs of particular problem sizes and complexities, a particularly important decision is determining the number and performance of machines in the system, we show that simple heuristics can help in selecting architecture parameters instead of relying on empirical measures.

Finally In Chapter 4 we describe the architecture of our distributed processing system which is broken down into 3 major layers: Storage that is capable of handling large datasets without loss of data, a Computation System that utilizes fast in-memory distributed computing to process data and an Orchestration layer that manages communication between layers and manages jobs.
In chapter 5 we present empirical results from using our architecture to process synthetic data and perform a prediction task, our heuristic is used to determine an appropriate level of parallelism and distribution for the problem and we compare system runtimes across a variety of system parameters.
Chapter 2

Distributed Computing Concepts

2.1 The CAP Theorem

No discussion of Distributed Computing would be complete without mention of the CAP theorem. The CAP theorem is a tool to understand the tradeoffs involved when designing distributed systems. Theorized by Eric Brewer, the CAP theorem states that there are three core requirements of a distributed system that have special relationships with each other [9][10]. The requirements are Consistency, Availability and Partition tolerance.

In [9] the CAP theorem is informally described in terms of safety and liveness in an unreliable environment. In an informal sense safety describes a system where nothing bad ever happens and every request receives the correct response, in addition to this liveness defines a system where eventually something good happens i.e: eventually every request receives a response. In this definition Brewer has shown that guaranteeing that a system is both safe and live in the face of unreliability is impossible.
Unreliability in distributed systems can be expressed in many different ways such as but not limited to the following:

**Partitions**

Physical network disconnections.

**Crashes**

Software and Hardware errors that cause a machine to enter a unresponsive state.

**Message Loss**

Network characteristics cause messages between computers to be lost.

**Malicious Attacks**

Malicious Attacks or Byzantine Failures cause machines in the network to either not respond or respond with false information.
Formally the CAP theorem which describes the relationships seen in Figure 2.1 is defined as:

**Consistency**
Consistency represents the overall atomicity of the system in terms of the transactions and state of its data. Consistency guarantees that all the nodes in the system see the same state of data at the same time, namely a read performed on the data from any node returns the most recent write on the data from any node.

**Availability**
The second requirement of the CAP theorem is that the system guarantee availability. Availability as the name suggests simply guarantees that the system be available, for each request the system must eventually have a correct response.

**Partition Tolerance**
Partition Tolerance guarantees that the system is resilient to changes in network performance. In a distributed system nodes can become partitioned if connections between the nodes become severed either due to physical disconnection events or network parameters. A system’s ability to perform under such partitions defines its Partition Tolerance.

However it has been proved in the theorem that a distributed system cannot simultaneously provide consistency, availability and partition tolerance, since partition tolerance is a necessity to operating in real world networks a tradeoff is made between consistency and availability. To understand how this tradeoff may appear in a system we can analyze a simplified example of a distributed system in Figure 2.1.

Figure 2.2: Example of state in a distributed system
In Figure 2.1 we see a simple system consisting of two machines $P_1$ and $P_2$ that service two clients $C_1$ and $C_2$ that access the state of a single variable through read and write requests. Initially client $C_1$ writes data into the variable through a write request this modifies the state of the variable on $P_1$ however since this state is only visible to $P_1$ when a read request is made to $P_2$ the client $C_2$ receives the old state of the variable causing the system to be inconsistent. $P_1$ and $P_2$ synchronize their states leading the system to be eventually consistent, however it demonstrates instances where systems can enter an inconsistent state due to the unreliability presented by asynchronous communication and network latency.

Although CAP represents characteristics that one would desire most in a distributed system, Since it is impossible to achieve both consistency and availability in an unreliable environment it is necessary in practice to sacrifice one of these two desired properties. This leads to systems that guarantee consistency and best effort availability i.e CP systems and systems that guarantee availability and best effort consistency i.e AP systems.

The choice of which system to use depends heavily on the needs of the problem being solved. Choosing AP systems that are always available allows clients to have a low latency experience even if the result isn’t always correct, whilst choosing CP systems favours data integrity and is suitable for applications where the correctness of the response is more important than timeliness.
2.2 Consensus

One of the problems encountered when building distributed systems is to have nodes in a network to agree on a certain value. This is the so-called problem of consensus. Consensus issues arise when systems need to agree on an important value such as the leader of a cluster or configuration data. In Figure 2.1 we saw how a system loses atomicity due to asynchronous communication, a system is atomic if for every operation there is a single instant between the request and response at which the operation appears to occur. There must be a total ordering of operations such that for each operation it appears to be completed at a single instance. This requirement is equivalent to requiring requests of a distributed shared memory system to act as if it were executing on a single machine. Consensus systems allow for nodes in a distributed architecture to synchronize state changes and make the system atomic.

However, consensus systems aren't just used to implement consistency; a system is considered available if it is continuously operational to service commands on demand. One way of guaranteeing high availability out of a system comprised of unreliable commodity hardware is through redundancy, by replicating the state of services across machines the system is made tolerant to failures or network interruptions.

Consensus algorithms allow nodes to keep these service states replicated across the cluster in lieu of race conditions and network failures. Each replicated process can be modelled as a deterministic state machine with a transition model that maps one (state,input) to a (new state,output). Therefore maintaining replicated processes across different nodes is a matter of all the state machines agreeing on the same set of inputs and their respective ordering, this is the heart of the consensus problem.
In the context of distributed consensus a state machine is any datastructure that is meant to be kept synchronized amongst several servers. In figure 2.2 we see several client machines accessing data from a state machine replicated amongst three servers. Clients can transform the data or state of the state machine by issuing read/write requests, these requests form a log and by utilizing the consensus module the log is synchronized amongst the servers in order to maintain the same state.

A popular algorithm used in real world system is the Raft consensus algorithm [11]. Raft provides a means for the logs on several servers to be synchronized with one another despite unreliability in the network and hardware. The Raft algorithm is comprised of three components leader election, log replication and safety.

During the Leader election phase each node in the cluster is initialized with a randomly seeded timer, when a node’s timer runs out it requests votes from the other nodes in the server, a majority of votes determines the leader of the round. This favours electing the node with the shortest timer length. In the unlikely event of two nodes having the same initialized timer a split vote is declared and the timers are reinitialized to allow for another vote. This simple mechanism allows for nodes in a
network to easily work together to determine a leader machine.

At the beginning of each round the leading machine processes incoming client requests, when a request is first received it is in an uncommitted state. Uncommitted requests are transmitted to every available machine and the leader waits for acknowledgements. When the leader has received a majority of acknowledgements it is finally safe to commit the request into the log on every machine without fear of data loss. The system proceeds in this manner to continuously process client requests.

### 2.3 MapReduce

MapReduce, a framework proposed by Google for distributing computation over a cluster works by implementing algorithms as a series of transformations on data[12]. The framework represents a computation by decomposing it into a Directed Acyclic Graph (DAG). Each node in the graph represents a transformation of the data from its incoming edges to its outgoing edges, nodes are connected together into a graph that represents a series of transformations on input data.

![Figure 2.4: Example of a MapReduce DAG](image)

Nodes in the graph are in turn composed of several machines in the cluster which perform the transformation on a portion of data independently of all other machines. This independent processing ensures that both computation and data is distributed over the cluster.
When necessary, nodes must connect across machines to perform an operation that requires data from more than one machine. This event is called a "shuffle". Shuffles are usually performed to compute aggregations or other computations that require data from machines external to the one the computation is being performed on.

The graph is then segmented topologically by extracting sequences of nodes whose outgoing edges do not cross over machine boundaries, each sequence of nodes that can be separated in this form are termed "Stages". Individual stages are grouped together to form Jobs and Jobs are then scheduled to run on each machine of the cluster.

Programming in the MapReduce framework is simple but very expressive. As the name implies the framework exposes two functions to the user \texttt{map()} and \texttt{reduce()}. Although simplistic these two functions can be used to compose very complex data analytics tasks with a high degree of parallelism, including machine learning, data mining and graph processing[13].
The `map()` and `reduce()` functions represent different data transformations and phases of execution. In the mapping phase where data is transformed from one or more input pairs into one or more output pairs, this is analogous to a mathematical function mapping variables from one domain to another. The reduction phase groups together mapped data according to identifiers such as ID’s or key values and is useful in aggregating values of a certain type. The reduce phase incurs a shuffle event due to the necessity of accessing and grouping data from every machine.
Chapter 3

Problem Classification and Analysis

3.1 Distributed Computing

System optimization can be roughly categorized into two strategies, scaling up and scaling out. Scaling up refers to optimizing the computing system within a single machine. Tuning the performance of operations, parallelizing tasks and upgrading hardware are all valid strategies when choosing to scale up a system. Scaling up performance works well for classes of problems that are said to be CPU-bound i.e: Problems that devote most of their execution time to computation as opposed to I/O. CPU-bound problems typically operate on small amounts of data and are potentially optimized by decomposing the application into tasks that can be efficiently parallelized on multicore CPUs.

Scaling out is a system optimization strategy that spreads a computation task over several computers that co-operate to solve the problem. Scaling up is analogous
to Compute-parallelism where a problem’s computation is split over multiple CPU cores, similarly scaling out can be thought of as analogous to Data-parallelism. Data-parallelism splits a problem’s data into separate partitions over several computers thus allowing computation to be performed independently on each data item of a set of data. Data-parallelism is best suited for classes of problems that are said to be I/O bound i.e: Problems whose performance characteristics are dominated by time spent reading and writing data to disk. I/O bound problems are typically data intensive and operate on large volumes in the order of tens of terabytes and petabytes.

3.2 Problem Classification

In order to understand how one might classify a problem as being I/O or CPU bound we need to develop a heuristic for the achievable speedup from Data-parallelism. A well known law in concurrency theory is Amdahl’s Law, the law characterizes the maximum achievable speedup in a CPU-Parallel system using \( n \) processors and a parallelizability factor \( F \) [14].

\[
T(N) = T(1)
\]

\[
(F + \frac{1 - F}{N})
\]

Amdahl’s law states that the total time to run an algorithm with \( n \) processors is the sum of the fraction of work that is parallelizable with \( n \) cores and a constant overhead of sequential processing.

from this Amdahl formulated the achievable gain \( S(n) \) of a system as

\[
S(N) = \frac{T(1)}{T(N)} = \frac{T(1)}{T(1)(F + \frac{1 - F}{N})} = \frac{1}{F + \frac{1}{N}(1 - F)}
\]
This gain equation helps quantify the tradeoffs involved in designing a parallel computing pipeline. A corollary to Amdahl’s law in a Distributed Data centric paradigm would depend upon

\( D \) the size of data being ingested in megabytes (Mb), \( M \) the number of machines in the cluster, \( I \) the I/O rate measured in Mb/s, \( C \) processing rate of a sequential solution measured in Mb/s.

Each machine in the cluster must at the very least perform two steps, read data from disk and process said data on the CPU. Therefore the total time spent is the sum of the time spent reading from disk \( T_i \) and the time spent processing the data \( T_c \) two operations, however the time spent processing is limited by the I/O rate giving the following:

\[
T_i = \frac{D}{I} \quad (3.3)
\]

\[
T_c = \frac{D}{\min(I, C)} \quad (3.4)
\]

The time spent in I/O also can not be parallelized amongst the processors leading to the following relation

\[
T(N) = T_i + T_c\left(F + \frac{1 - F}{N}\right) \quad (3.5)
\]

This can then be distributed across the cluster, as more computers are added the effective I/O and compute of the system increases thus leading to processing gains in the form of \( T(N)/M \). This doesn’t fully account for the complexities of distribution as there is overhead from moving data over the network and orchestrating the different stages of computation between computers, Furthermore there is also a fraction of
computation time that effectively can not be distributed, periods when all machines must aggregate their data into one result. These costs $\gamma(M, D)$ increase with the number of machines and the size of the data. This gives us the following:

$$T(M, N) = \frac{T(N)}{M} + \gamma(M, D)$$

$$= T_i + T_c(F + \frac{1-F}{N}) + \gamma(M, D)$$

The choice of gamma is dependent on the problem being solved and in a real setting would need to be determined empirically from network parameters and runtime analysis. However for our heuristic we can assume a constant network overhead $K$ proportional to the amount of data transmitted over the network in units of s/Mb to model the time spent in network communication.

$$T(M, N) = \frac{T_i + T_c(F + \frac{1-F}{N})}{M} + DK$$

The gain achieved from distribution is therefore

$$S(M, N) = \frac{T(1, N)}{T(M, N)}$$

$$= \frac{T(1, N)}{\frac{T(1, N)}{M} + DK}$$

$$= \frac{MT(1, N)}{T(1, N) + MDK}$$

(3.8)
Although this expression has a lot of variables to tune we can see some important characteristics of the problem classes appear. Here are a few of the interesting regimes of problems one might face.

### 3.2.1 CPU-Bound, High data Low CPU throughput

In this regime the I/O rate far exceeds the rate at which the CPU can process the data. Using a sample of One terabyte of data $D$ I have empirically measured the read speed $I$, The compute rate $C$, The network transfer latency $K$ and the fraction of sequential compute $F$

\[
D = 1000000
\]
\[
I = 500
\]
\[
C = 3
\]
\[
K = 0.01
\]
\[
F = 0.1
\]
\[
N = 4
\]

(3.11)

Holding the number of cores constant at 4, we can see the effect of adding additional computers to the cluster.
This graph demonstrates the diminishing returns of adding more machines to the cluster, as the number of machines increases the performance gain approaches an asymptote and further increasing the number of machines does not improve performance appreciably this is due to the increasing cost of transmitting data over the network.

Looking at the regime holding the number of machines at $M = 5$ shows us a different picture of the tradeoffs involved.

This graph demonstrates how different points in the performance function can
be obtained through different configurations. Although it may seem intuitive that adding more machines will increase efficiency, the graph shows that having more CPUs is a better strategy for this domain as the performance is only marginally better than having the same number of cores on a single machine. This validates our understanding of CPU bound problems, which are a good target for high CPU and low network utilization strategies.
3.2.2 IO-Bound, Low data High CPU throughput

In this regime the CPU rate is higher than the rate at which data is read from disk. The following parameters were sourced from data sheets on commodity hardware.

\[ D = 1000000 \]
\[ I = 60 \]
\[ C = 1000 \]
\[ K = 0.01 \]
\[ F = 0.1 \]
\[ N = 4 \]

(3.12)

With 4 cores and a varying number of machines, we get the following.
Figure 3.3: Low data High CPU throughput

From this graph we can see that there isn’t much to gain from distributing work loads in this domain. Even though adding more machines increases the effective I/O rate of the entire system the overhead paid in network traffic outweighs the gain from distribution. A similar relationship can be seen from increasing the number of cores.
Since the system is I/O bound, increasing the number of cores does not increase the performance of the system appreciably. The difference in performance between the distributed system and a single machine does not change significantly.

Figure 3.4: Low data High CPU throughput - with varying cores
Finally, by plotting actual run times of different cluster configurations in Figure 3.2.2, we can see that there is a range of feasible solutions depending on the problem type. The appropriate configuration can then be chosen based on other factors such as cost and the level of redundancy required. It is important to note that this heuristic is not an estimate of run times but simply offers a metric for comparison across configurations.
Chapter 4

Architecture

4.1 Overview of the Architecture

In [3] Doukas et al have developed a cloud based sensor network that manages sensor measurements from relatively simple and inexpensive sensors being processed on a centralized hub. In their work, cheap wearable sensors with very low computational power send their data to a server hosted on cloud infrastructure, this centralized approach has the benefits of being simple to implement while utilizing cheap low power sensors. Our approach applies distributed processing in a similar vein to process incoming data from sensor networks in a decentralized fashion, decentralized architectures introduce more complex sensors in order to improve computational and storage demands.

In a decentralized architecture sensors are accompanied by storage and computational resources that work in concert with each other to process data, It has the benefit of being able to handle more expensive computations and handle high storage demands without a single point of failure.
Our objective in building the system is to have a processing architecture that is easy to deploy, manage and scale up when new processing requirements arise. In order to accomplish these goals we have decided to use cloud computational resources namely storage and computation, this allows for the system to easily scale by simply leasing new computers from the cloud to work in the cluster. To ensure ease of deployment a service oriented design will orchestrate tasks in the network.

In the CAP sense our goal is to build a system that is Consistent and Partition tolerant, CP systems prioritize system state integrity over with a best effort guarantee of system availability. Designing our processing architecture as a CP system ensures that the system either returns a result or does not, there is no possibility of the system retaining old results or other mutated state which is a priority when implementing systems for scientific computations.

Figure 4.1: Network Diagram of our Architecture

The following is a discussion of the distributed architecture built to process sensor data, We discuss the considerations and tradeoffs involved in constructing such a distributed system. In Figure 4.1 we see a summary of the network diagram of our
architecture composed of a master driver node in charge of co-ordination amongst the machines, the driver is in charge of assigning work to each worker computer and keep track of data in the network. An in-memory caching layer on each computer allows computations to quickly access and store data.

Each worker in the network is composed of three major components, The discussion is broken down into these key areas:

**Storage**

Storage is an essential component of a distributed computing system. Traditional Centralized network storage would incur high cost in network latency as machines in the cluster will continuously request data over the network. Centralized storage also represents a single point of failure in the system. The storage needs of a distributed system need to be highly available, avoid data loss and be performant when handling large datasets.

**Computation**

Computation has a lot of the same goals as storage, the computation component of the system implements the MapReduce data processing paradigm with a few important extensions to ensure resiliency to failures and good performance characteristics.

**Orchestration**

The last and most important component of the system is the orchestration framework. As the size of the cluster increases it becomes harder to manage deploying and running software on the system. A suitable orchestration framework centralizes the management of the cluster, ensures that the computer are
able to communicate with one another and maintains the software running on each computer.

4.2 Distributed Storage

Distributed systems used to scale out processing on large datasets need to be designed with careful attention paid to data storage modelling. Data in a distributed architecture needs to be modelled in a manner that is aware of the network topology and resilient to hardware failures. In our architecture data to be processed is stored across several machines called DataNodes [15].

As a result of continuous usage, hardware in a datacenter eventually fails, data resiliency in distributed storage can be achieved through replication. Large data sets are first split into smaller blocks, blocks are replicated and then randomly distributed across the cluster making sure that no DataNode in the cluster has more than one copy of the block. A central master machine in the cluster called the NameNode monitors each slave and determines the health of the machine through a heartbeat signal sent at a constant interval, the receipt of a heartbeat signal indicates that the DataNode is functioning correctly. In the event of a missed heartbeat the NameNode tracks down the replicas of the lost blocks and schedule re-replication.
Storage orchestration responsibilities are divided into two parts, Resource management and Storage servicing. The Resource Manager also called the NameNode is in charge of tracking the directory tree across the entire distributed file system, it tracks which machine each block is stored on and aids machines in finding each other to access blocks in the cluster. Client applications can then perform file system operations read/write/move/delete by communicating with the NameNode.
Additionally the direct servicing of file system operations is done on storage servicing machines called DataNodes, these nodes contain the raw replicated blocks that build up the distributed file system. On startup Datanodes connects to the NameNode and sends a heartbeat signal to the NameNode in 10 second intervals, It then responds to the NameNode to perform the required file system operations as well as informing the NameNode of any changes to the file system.

4.3 Distributed Computation

Computations in the architecture are implemented using the MapReduce framework defined in chapter 2. Data is read from the distributed file system and converted into Resilient Distributed Datasets (RDD) [16]. RDDs are fault tolerant, parallel data structures that allow users to have explicit control over data persistence, partitioning and manipulation using an extensive set of data transformations.
The RDDs provide the transformations and actions listed in Figure 4.3. Data transformations are applied in batches called partitions making RDDs a good fit for many parallel applications where solutions involve applying the same operation to multiple data items. RDDs are a read-only data structure representing a group of records, RDDs can only be created directly from source data or as a result of a data transformation of an existing RDD. This gives RDDs the property of immutability.

The Immutability of RDDs means that every RDD is defined completely by the history of transformations applied to each RDD in its ancestry. RDDs ensure fault tolerance by storing the lineage of transformations. In the event of a failure, missing partitions can be re-computed from the data by applying the lineage of transformations. Re-computation allows for fault tolerance without having the complexity of having to store and synchronize the state of a mutable distributed data-structure. These re-computations can be done in parallel utilizing the computational resources of the network if necessary. Immutability also ensures that race conditions do not cause problems tracking in the state of the system as the data is not modified.
but a new version is computed for every transformation.

In addition to this RDDs expose an interface that allow for control over persistence and partitioning. Persisting RDD data allows for results from intermediate stages to be retained in-memory and reused by other computations. In-memory storage of results increases performance of iterative algorithms where the result of the previous iteration are used in the next iteration. Partitioning allows users to control how RDD records are partitioned across machines based on a key, this is useful in controlling data placement.

4.4 Service orchestration

Service oriented architecture is a design pattern that involves architecting large complex systems by decomposing them into small independent services that communicate with each other to implement the necessary features of the application[17].

Services are simple reusable applications that perform a single task through a well defined protocol. Services providers and consumers communicate through this protocol to combine their respective features and provide the functionality of a larger system.

One of the keys of service oriented architecture is that services operate in a loosely coupled fashion i.e services depend on each other to the least extent possible and do not have an understanding of the underlying implementations of each other.

In a distributed architecture with several nodes, services need to be implemented in a cluster aware fashion. This involves starting services on machines where they function best (machine affinity), Exposing a means for services to find each other (service discovery) and a means for services to talk to each other over the network.
For each of these tasks to be performed in a distributed setting the Services require a master to coordinate the services across the cluster, initialize them and provide a registry so that services can register themselves and find new services.

Master election is performed using the raft algorithm\cite{raft}\cite{raft2}, the master then starts a registration service that allows a service to register its name and address. The registration service is a fault tolerant distributed key/value store, where the service name is the key and the value is the address where the service is located. This registry is replicated and made available across the entire cluster so that services running anywhere can find and communicate with each other.

Figure 4.5: Distributed registry for services
For efficient routing the Master requires metadata units on each service. Units describe properties of the service such as:

**Dependencies**

Defines whether this service relies on the existence of another service and must therefore be run after said service e.g. the master service needs to be run before all slave services.

**Conflicts**

Occasionally we might want to restrict which services can be scheduled together and can use conflict directives to specify services that must not be co-located on the same machine e.g. The master and secondary master should not operate on the same machine.

**Machine Affinity**

Units can also instruct which particular machine to run a service on, this is useful when a service requires some specific resource such as running the DFS services on machines with large hard disks.

The elected master analyzes the Unit descriptions of all the services and then decides where to run the services and in what order. In our architecture each component of the infrastructure represents a different service. The following are the services exposed through the system:

**NameNode**

The main resource manager of the Distributed File System.

**DataNode**

A service that connects to the NameNode service and listens for data events.
DAG Master

This service co-ordinates DAG scheduling and computation.

DAG Slave

Connects to the DAG Master and listens for stage schedules and processes DAG jobs.

The final running state of these services are organized in the following hierarchy.

Figure 4.6: System diagram of the infrastructure
4.4.1 High Availability

To guarantee high availability multiple redundant Master nodes are employed. At startup each slave machine registers itself into the service registry as a Secondary Master. In the event of a master failure a secondary master is chosen from the registry and the slaves are switched over to the new Master.

In case of slave failure the work continues to be distributed amongst the remaining slaves and any lost RDD data is recomputed through its known lineage.
Chapter 5

Procedure and Results

5.1 Problem

An important problem in Multi-Target Tracking is that of prediction. Prediction is the estimation of a state at a point in the future given its current estimated state, measurements and a model of the state dynamics. In Multi-Target Tracking prediction algorithms are used to estimate future positions of Radar targets.

The prediction problem is very well suited to distribution especially when target positions are not dependent on the positions of other targets. This condition allows for target positions to be predicted independently of each other allowing for large gains from parallelism.

For the experiment a constant velocity model was assumed, measurements for each target are fed into a Kalman Filter which estimates the position and velocity of the given target.
\[ \hat{x}_{k|k-1} = F \hat{x}_{k-1|k-1} + Bu \]  
(5.1)

\[ P_{k|k-1} = FP_{k-1|k-1}F^T + Q_k \]  
(5.2)

\[ S_k = HP_{k|k-1}H^T + R_k \]  
(5.3)

\[ K_k = P_{k|k-1}H^T S_k^{-1} \]  
(5.4)

\[ \hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k(z_k - H\hat{x}_{k|k-1}) \]  
(5.5)

\[ P_{k|k} = (I - K_kH_k)P_{k|k-1} \]  
(5.6)

Estimation is split into two phases, In the first phase the previous estimate is used to predict the state at the current time step \( \hat{x}_{k|k-1} \) this predicted state is then merged with the measurement \( z_k \) using the Kalman gain \( K_k \) which corrects the prediction according to the uncertainty expressed in the measurement and prediction. The merged result is the estimated state for the current time step \( \hat{x}_{k|k} \). Estimated velocity and position \( \hat{x}_{k|k} \) is then used to predict target positions an hour into the future using the state transition model \( F \).
5.2 Procedure

The first step in testing out distributed architecture is to present it with data. A one terabyte dataset of target measurements was generated representing target locations and velocity. Target positions were uniformly randomly sampled across the earth.

A DAG for the prediction algorithm was then constructed where Raw target state information is first transformed into target data-structures, measurements for each target are aggregated together and then mapped into predicted target states. The DAG for this prediction algorithm utilizes the JNI binding to allow data to be passed to natively implemented prediction algorithms, while this method incurred a heavy memory overhead it allowed for consistent calculations and ensured backwards compatibility with existing systems.

Finally an N Node cluster was constructed using our Service Oriented Architecture framework which allowed us to start, stop and restart the cluster. All machines in the cluster were leased through Amazon Web Services a thirdparty cloud hosting provider.

The synthetically generated data was then loaded into the DFS and the DAG was run over the data. The DAG was constructed to randomly sample the input data and predict batches of predictions in increments of 100 Gb in order to benchmark the performance of the system over a range of input data sizes.

Benchmark measures were averaged over 3 runs of the graph.
5.3 Results

The procedure described above was run on three classes of systems, a single machine, a 5 node cluster with low RAM utilization and a 5 node cluster with full RAM utilization. Missing experiments were left out due to cost constraints (i.e. cost of runtimes over 24 hours).

The machines in the cluster have the following configuration.

<table>
<thead>
<tr>
<th>Machine Type</th>
<th>Instance Type</th>
<th>CPU Cores</th>
<th>RAM (GB)</th>
<th>Storage (GB)</th>
<th>Price ($/hr)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Node</td>
<td>r3.2xlarge</td>
<td>8</td>
<td>30</td>
<td>2000</td>
<td>0.7</td>
</tr>
<tr>
<td>Low RAM</td>
<td>c3.xlarge</td>
<td>4</td>
<td>0.683</td>
<td>400</td>
<td>0.23</td>
</tr>
<tr>
<td>Full RAM</td>
<td>c3.xlarge</td>
<td>4</td>
<td>4</td>
<td>400</td>
<td>0.23</td>
</tr>
</tbody>
</table>

Each machine was allotted twice the required amount of disk space to allow for ample on disk caching in the event that RAM utilization was insufficient for the problem. The single node machine has twice the number of cores and almost 7 times as many CPU Cores leading to a higher price point than the other machines.
Table 5.2: Runtimes in hours, for varying input data sizes

<table>
<thead>
<tr>
<th>Batch Size (GB)</th>
<th>1 Node</th>
<th>5 Nodes - Low RAM</th>
<th>5 Nodes - Full RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>4.6</td>
<td>11.5</td>
<td>3.7</td>
</tr>
<tr>
<td>200</td>
<td>5.6</td>
<td>-</td>
<td>4.6</td>
</tr>
<tr>
<td>300</td>
<td>24.6</td>
<td>24.3</td>
<td>5</td>
</tr>
<tr>
<td>400</td>
<td>-</td>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td>500</td>
<td>-</td>
<td>-</td>
<td>5.2</td>
</tr>
<tr>
<td>600</td>
<td>-</td>
<td>-</td>
<td>5.2</td>
</tr>
<tr>
<td>700</td>
<td>-</td>
<td>-</td>
<td>5.9</td>
</tr>
<tr>
<td>800</td>
<td>-</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>900</td>
<td>-</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>1000</td>
<td>-</td>
<td>-</td>
<td>6.1</td>
</tr>
</tbody>
</table>

As expected we see that low RAM utilization offers very poor performance compared to a single machine due to the effectively lower compute throughput $C$. Our defined heuristic predicts a gain of $S(5, 4) = 3.44$ for this high data low CPU regime which is comparable to the 4.8 measured gain of our architecture.

5.4 Conclusion

In summary we’ve seen how a simple heuristic can be used to compare cluster configurations and aide in designing an efficient cloud computing platform for spatial processing tasks. We’ve demonstrated that parallelism isn’t simply a matter of adding machines and cores to a problem as the gains from parallelism can quickly reach diminishing returns if careful attention isn’t paid to the class of problem, We’ve also seen
that the specific hardware parameters of the configuration are extremely important especially in terms of memory and network utilization. Following is a discussion of the limitations of our work.

5.4.1 Heuristic

In terms of modelling cluster performance there are many additional desirable features that one could model whilst maintaining the simplicity of a heuristic.

These include:

Computation Runtime

Currently our heuristic relies on an empirically measured compute throughput for a certain size of data. If instead a polynomial was used to represent $C$ the performance gain for different data sizes could be better estimated. This would help especially in problems of $O(n^2)$ and higher complexities.

Heterogeneous Architectures

For the sake of simplicity our heuristic assumes that every node in the cluster has the same performance characteristics and are simply multiple instances of the single node case. A matrix of performance characteristics could better model a system with a heterogeneous architecture.

Monetary cost and Power consumption

An extension to the heuristic could include monetary costs of running the network and the power consumption. This coupled with other metrics could help decide whether it would be more feasible to use more lower power machines vs one higher powered machine.
5.4.2 Predictor

A limitation of our framework is that it does not handle Spatio-Temporal data. Efficiently handling batch and streamed time series data which also represents geospatial locations is an open problem in Distributed Computing. Spatio-Temporal offers technical challenges in storage due to the volume of data and complex data structures to efficiently index in a multidimensional domain.

5.5 Future work

We have only covered one aspect of target tracking. Distributed Algorithms for Data Association and Sensor Fusion need to be developed in order to provide a complete distributed tracking solution.

Some good starting points to for the basis of a complete solution are:

Spatial Datastructures

Spatial Datastructures such as KD trees and Quadtrees can greatly aid in processing [19, 20, 21]. Spatial Datastructures provide $O(N\lg N)$ lookups for data that neighbors a given point in the spatial sense. These datastructures work by spatially clustering data to provide a spatial index. A simple example of a spatial datastructure would be to cluster data into a grid and index by grid cells, however for unevenly distributed data this wastes memory representing empty grid cells. more sophisticated datastructures offer multiresolution querying by branching and clustering only the areas where spatial data exists. A fast in memory Distributed Spatial Datastructure would help in co-ordination between nodes,performing queries and aggregations.
Partial Estimates and Approximation

Reduction operations in distributed architectures are very expensive as they involve every machine waiting to send and receive data across the network. One way to solve this would be to have machines calculate partial estimates using only the data they have and then merge partial estimates in a smaller Reduction operation. This would limit the amount of data transmitted over the network and allow machines to operate independently for longer time periods. Another possible solution would be to have approximation policies where M samples out of N samples are enough to offer a complete Estimate for a given target rather than waiting for the complete N samples.

Pre-Computation

As much as is possible computationally expensive results should be precomputed and broadcast to the network. Centrally computing shared data and transmitting cached copies avoids costly re-computations.
Bibliography


