AUTOMATIC DETECTION AND DENOISING OF SIGNALS IN LARGE GEOPHYSICAL DATASETS

by

Gabriel O. Trisca

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

August 2015

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Gabriel O. Trisca

Thesis Title: Automatic Detection and Denoising of Signals in Large Geophysical Datasets

Date of Final Oral Examination: 29 June 2015

The following individuals read and discussed the thesis submitted by student Gabriel O. Trisca, and they evaluated his presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Amit Jain, Ph.D.	Chair, Supervisory Committee
Hans-Peter Marshall, Ph.D.	Member, Supervisory Committee
Timothy Andersen, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Amit Jain, Ph.D., Chair of the Supervisory Committee. The thesis was approved for the Graduate College by John R. Pelton, Ph.D., Dean of the Graduate College.

ACKNOWLEDGMENTS

This research was partially supported by NASA EPSCoR grant #NNX10AN30A

Abstract

To fully understand the complex interactions of various phenomena in the natural world, scientific disciplines such as geology and seismology increasingly rely upon analyzing large amounts of observations. However, data collection is growing at a faster rate than what is currently possible to analyze through traditional approaches. These datasets, supplied by the increasing use of sensors and remote sensing, require specialized computer programs to effectively analyze complex and expansive volumes of data.

Elaborating on existing geophysical data processing approaches for infrasound data collected from an avalanche-prone area, this thesis proposes new techniques for processing large geophysical datasets. These improved techniques take advantage of Graphical Processing Units (GPU) to accelerate floating-point, computationally expensive operations. Additionally, they allow for dividing the workload among nodes in a High Performance Computing cluster, yielding a performance speedup of 3.5 times for every additional node added. Finally, a machine learning approach was used to classify events found in the processed data, which demonstrates the potential of automatic real-time avalanche detection.

Applications with characteristics similar to infrasound processing are common throughout the earth-sciences, and this work exemplifies the potential of these techniques to an array of science fields. New algorithms for efficient data processing like those presented in this work are fundamental to analyzing large geophysical datasets, as well as to improving the accuracy of computer models across many disciplines.

Contents

List of Tables

List of Figures

LIST OF ABBREVIATIONS

CPU – Central Processing Unit

GPU – Graphics Processing Unit

RAM – Random Access Memory

VRAM – Video Random Access Memory

MPI – Message Passing Interface

DAQ – Data Acquisition Card

ADC – Analog to Digital Converter

SNR – Signal-to-Noise Ratio

ANN – Artificial Neural Network

RNN, RANN – Recurrent Artificial Neural Network

Chapter 1

INTRODUCTION

1.1 Data in Geophysics

Scientific disciplines such as geology and seismology require large amounts of observations to understand the interaction between different phenomena. By collecting additional data, the existing geophysical models can be fine-tuned to produce more accurate predictions.

More complex models allow for precise predictions of the future and a better understanding of the past, and with this information, researchers can continue to explore the correlation between variables in geophysics.

The data used in geophysical research is extracted from measurements that usually have a low noise-to-signal ratio [55], meaning that the information is hard to distinguish from the noise. Additionally, samples often require human interpretation and processing before the scientific questions can be answered [38].

1.2 Modelling Natural Phenomena

Models are mathematical descriptions of natural phenomena that allow scientists to systematically analyze and interpret physical observations. In order to create new models, scientists start by identifying and understanding the primary physics that

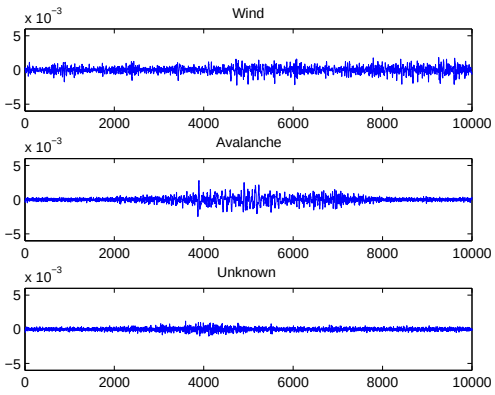


Figure 1.1: Sample signals created by different phenomena

influence the natural phenomena. The scientists will then analyze the interaction between these the physical constants and variables and introduce new constants and variables to produce a model called *theoretical model* that can be used to characterize and predict future occurrences of the same phenomena. In modeling natural phenomena, scientists will make assumptions (for reasons like reducing complexity) and omit variables in the model to create a simpler or more understandable model.

An example of a situation where oversimplifications occur is in snow-pack modeling, where a physical model is developed to predict the characteristics of snow for diverse geographical areas. The accuracy of models that try to predict snow water equivalent and snow depth vary depending on the input variable —the assumptions and omissions can make the model produce inaccurate predictions in different geographical areas, and there are relatively few observations to constrain the model[45]. For example, one of the physics-based snow-pack models known as "SNOWPACK" takes into account variables like wind speed, humidity, and air temperature. Although SNOWPACK is generally accurate in predicting snow-pack density, it does so by assuming all snow crystals are spherical, which is a problem when the features in the

snow-pack change the thermal conductivity in complex ways [45].

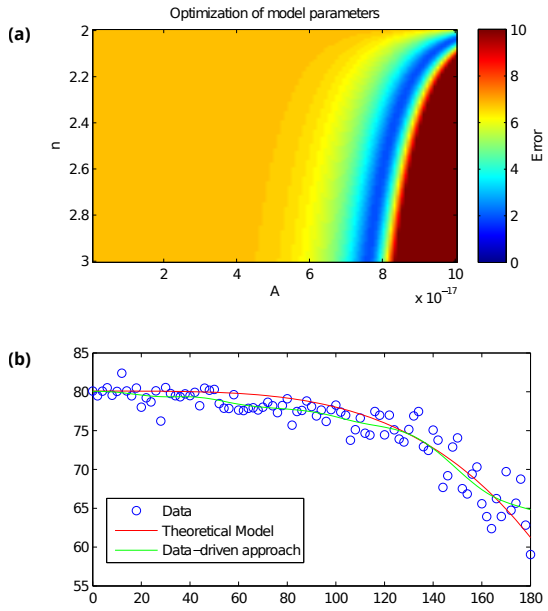


Figure 1.2: (a) Optimization of a function with parameters (n, A). (b) Observations as circles, a theoretical model $y = f(x)$ for the phenomenon and a numerical, data-driven prediction for y

In contrast with the process used to develop theoretical models, there is an alternative technique that is based on statistics and patterns found in past observations of the phenomenon being studied. These kinds of models are called *empirical models*. Just as with physics-based models, empirical models make the assumption that the natural process can be successfully characterized with the available historical observations and that in the future, the statistics will be similar. These empirical models are more flexible because they are not constrained by physics, but are not useful in predicting outcomes that have never been observed before.

One of the most common examples of empirical models is a regression model, in which the relationship among variables is represented as a function with multiple

parameters that are optimized to reduce the error with the observed data. See Figure 1.2.

Finally, in the study of geophysical processes where dozens of variables interact, theoretical models are in many cases hard to develop. Alternatively, empirical models can be used to help to understand and identify some of the controlling factors behind phenomena and their influence based on historical observations.

Chapter 2

AVALANCHES

Avalanches are destructive events that pose a threat to infrastructure like roads and towns and put lives at risk. Since the mid-20th century, efforts have been made to predict when and where an avalanche may occur based on weather and snow conditions, which is a very difficult task. The next best step is to be able to detect when they occur using warning systems. Measures can then be taken to protect the structures and people that are in the path of the avalanche [56], [10]. One of the ways to reduce the risk of people being caught in an avalanche is by closing roads when an avalanche is detected or by pre-emptively closing roadways during periods of time when avalanches are likely to occur.

When studying avalanches, seismic, acoustic and electromagnetic sensors are used to collect data on different processes that affect and trigger them. These measurements are noisy due to pressure changes in the air and ground vibrations coming from other sources outside of the avalanche and noise in the instruments used to record the signals [40].

Because of the noise present in the measurements and the enormous number of variables that are involved in the creation of avalanches (snow-pack layer strength, temperature, wind, snow grain structure, load, etc.), developing a physics-based, theoretical model is an intractable task due to a lack of necessary observations [46].

The best alternative is to use an empirical model that can incorporate some of the variables and be iteratively optimized to identify when avalanches are happening based on historical observations.

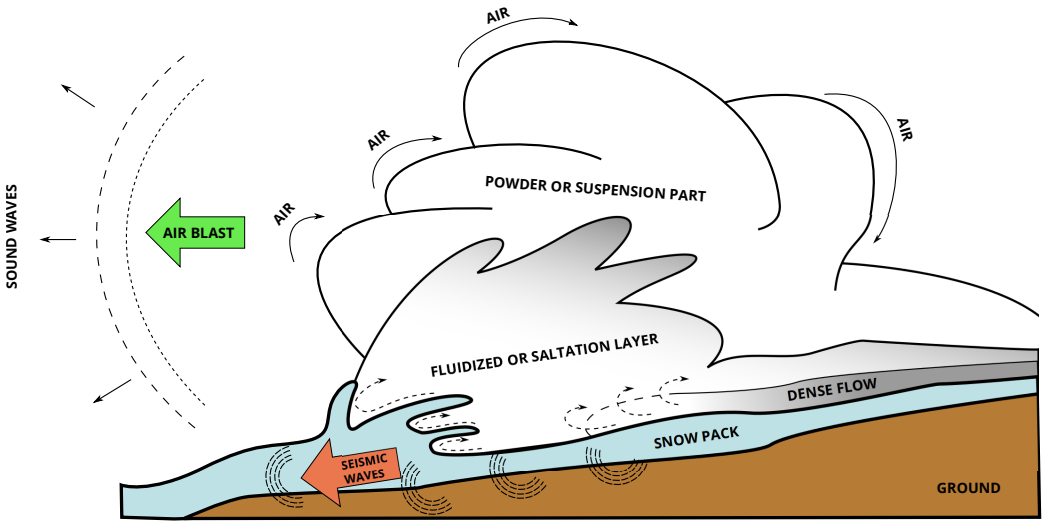


Figure 2.1: Cross section of a mixed avalanche, indicating the different parts (modified after Kogelnig et al., 2013) [36]

2.1 Avalanche Detection

The first step in studying avalanches is to identify when one has taken place. When visual confirmation is not possible (due to inaccessible roads, avalanches happening at night or in remote areas, etc.), the signals captured by sensors in the vicinity of the avalanche, after being processed, can either confirm or deny the occurrence of an avalanche.

There are multiple sources of sound that can be present in areas where avalanches occur. Airplanes and cars in the vicinity of an avalanche zone will produce sound signals that can be similar to the signals created by avalanches. Vehicles emit sound

at a specific range of frequencies and produce vibrations on the ground just as distant explosions or earthquakes produce seismic signals [10].

In order to determine if an avalanche has occurred at any given time, the signals from sensors need to be filtered to remove unwanted noise that may mask the presence of a signal. The signal is reviewed by an expert who can determine the most likely source for the signal. Based on the signal's properties, it is possible to estimate if any data recorded originated from an avalanche or from another type of source, like low-flying aircraft or strong winds [51].

Retrieving signals simultaneously from multiple sensors and processing them is called array processing and is a common technique used in geophysical applications [44]. Scientists can learn more about the characteristics of the event by studying these signals. It is possible to identify the direction of the source of the sound (pressure changes in the air) or the frequency components of the signal by performing a Fast Fourier Transform (FFT).

2.1.1 Seismic and Radar-based Methods

The three most commonly used sensors in avalanche detection are seismic-based, radar-based, and sound-based.

Avalanche detection using seismic methods consists in placing devices that measure ground movement called geophones close to locations where avalanches are known to take place [40]. Geophones can be placed directly in the slide path of an avalanche [51] or at a distance [10]. The most effective technique based on the results of Herwijnen and Sweizer (2011) is to instrument the avalanche slide paths. However, these techniques will only provide results for a limited number of avalanche paths and its prohibitively expensive to instrument a large geographical area.

Seismic approaches rely on detecting changes in the signal and associating those with avalanche events. The most common method to find these changes is by using the signal's amplitude. Processing data from an instrument that records F samples per second (see Equation 2.1), the average amplitude is calculated by averaging the absolute ground oscillation velocity v. If the value of A exceeds a previously defined threshold, a warning is triggered [6].

$$A = \frac{\sum_{i=1}^{F} |v_i|}{F} \tag{2.1}$$

Radars can be also used to detect avalanches by means of measuring the total travel time of a signal emitted by the radar until it is reflected back from the source [38]. The radar is aimed at a fixed point in the avalanche path —and if an avalanche takes place, the radar-emitted signal is reflected off the front of the avalanche. The moving avalanche reduces the travel time and provides an easily identifiable change that indicates that an obstacle is in the path of the radar's beam [40]. The biggest limitations of using radars is their limited detection range and limited angle coverage. A common radar installation can monitor an area of less than 0.3 Km^2 and only a single avalanche path [36].

2.1.2 Infrasound-based Methods

Sound is created by vibrations in the air and these vibrations can be recorded using microphones [47]. The human ear can perceive sounds that have a frequency approximately between 20 Hz to 20 kHz therefore the term *infrasound* refers to vibrations in the air that have a frequency below 20 Hz and are inaudible to humans.

Based on experimental data and theoretical models, it has been determined that

avalanches produce sound that has a characteristic frequency signature, with most of the signal power being concentrated in frequencies below 20 Hz [50], [55], [35], [42]. Most infrasound approaches rely on arrays of microphones that can be deployed in different configurations varying in relation to the avalanche path location, the expected amount of snow that accumulates in the study area, and the amount of noise and extraneous sources of sound in the area, among others [47], [50].

Recorded infrasound signals can be analyzed by experts with tools similar to those used in seismic approaches to determine whether an avalanche has happened or another event has taken place. The confidence in the correct classification of an event varies based on the amount of noise in the signal, the meteorological conditions, the skill of the analyst, and whether there is visual confirmation of an avalanche in the hours or days following the detection [36].

2.2 Avalanches in Idaho

Highway 21 is a two-lane highway in Idaho connecting the cities of Boise and Stanley [4], has a maximum elevation of 7057 ft above sea level and receives approximately 300 inches of snowfall during the winter. Approaching Banner Creek Summit on Highway 21 and about 30 miles from the city of Stanley [3] is a 12 mile-long stretch of road with high-slope mountain faces nicknamed "avalanche alley" that sees more than 100 avalanches occur annually, forcing the road to be closed to the public due to the risk of large amounts of snow being deposited on the road by avalanches.

The Idaho Department of Transportation (ITD) maintains an avalanche forecasting program that assesses the risk of keeping the road open to traffic based on weather and the snow conditions. The number and size of avalanches that took place in

the 2012-2013 cycle have been considered the largest in the 15 years the forecasting program has been operating [28]. In the 2013-2014 cycle the IDT reported at least 120 avalanches, of which roughly 50% deposited snow on the road.

Understanding avalanche dynamics and having timely warning systems can prove invaluable for the public traveling through this region of Idaho and by extension inhabitants of other avalanche-prone locations throughout the globe.

2.3 Big Data Processing

Big data is a relatively new field in computer science that focuses on data processing at a scale where traditional algorithms are inadequate. Large datasets are becoming more common as companies see 30% to 50% year-to-year growth of stored data in addition to the declining cost of digital storage [49]. These trends are enabling a new kind of data analysis known as *data mining*, defined as trying to find patterns and information in large amounts of data.

A computer, as defined by Von Neumann, is able to work on one task at a time, sequentially executing instructions stored in memory until the program finishes [53]. The performance of programs on this computer architecture depends on how fast a single instruction can be executed. For decades, processor clock speeds had exponentially increased. This trend continued until the early 2000s, when heat dissipation issues and current leakages—among others—limited the maximum clock speeds that could be achieved [48]. In response to CPU's clock speeds stalling, all the major chipmakers made a transition from faster processors to incorporating more processing *cores* on the same CPU. These cores or units share access to main memory, where the data resides. They also have independent caches and registers, allowing

each core to operate independently. These changes in hardware design are grouped under a new type of computer architecture called "shared memory parallel machines" (SMP), where more than one program can run at the same time and access the same memory [32].

Traditional algorithms—where one CPU core did all the processing—became obsolete with the surge in stored data. New algorithms and techniques had to be developed. The changes in CPU architecture had opened the door to running multiple programs at the same time which revolutionized the high performance software paradigm. The paradigm effectively changed from minimizing the number of instructions required to complete a task, to where the performance is driven by how much concurrency can be achieved [48].

Just as businesses store more data because of declining costs, in the field of earth sciences a new trend has emerged where researchers can stora —and are accustomed to storing experiment results and observations in real time from virtually any location on earth. The low cost of storage systems combined with the advent of cheaper sensors that can record and transmit high-precision data at higher bitrates leads to ever growing volumes of data being collected. These vast amounts of data are data-mined to create simulations, new models, etc., something that is possible in large measure to new techniques for data processing [33].

The first step to process large datasets on SMP machines is to divide the dataset into smaller subsets and process them concurrently. This is known as parallel execution or *parallelization* [30].

2.4 Parallelization

Parallelization can be done at multiple scales, from a single computer to a group of interconnected computers. We start with a simple program P that performs a computation f on input data $X = (x_1\ x_2\ ...\ x_n)$ to produce X', where f is comprised of multiple instructions. If the function $f(i)$ doesn't require any other data than i, it is considered *embarrassingly parallel*.

On an SMP machine that consists of p processing units or processors, we can divide the input X into p parts, and compute f on each part concurrently. These parallel computations are called tasks [21].

There are a number of rules of varying complexity and applicability that define whether a program can be parallelized or not [21]; however, generally a program can be parallelized if the result of the computation is independent of the order in which the steps take place and these steps do not have any dependencies on previous computations.

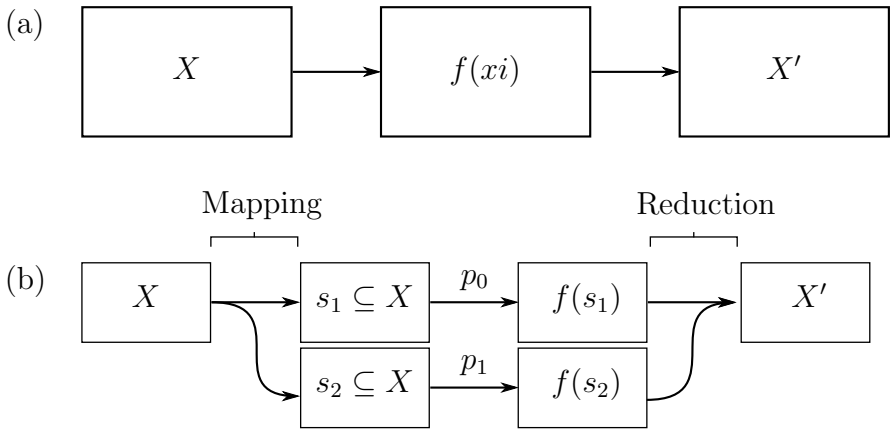


Figure 2.2: (a) A program that performs a computation f on data X. (b) The data is split (mapped), processed in parallel on processors p, and then recombined (reduced).

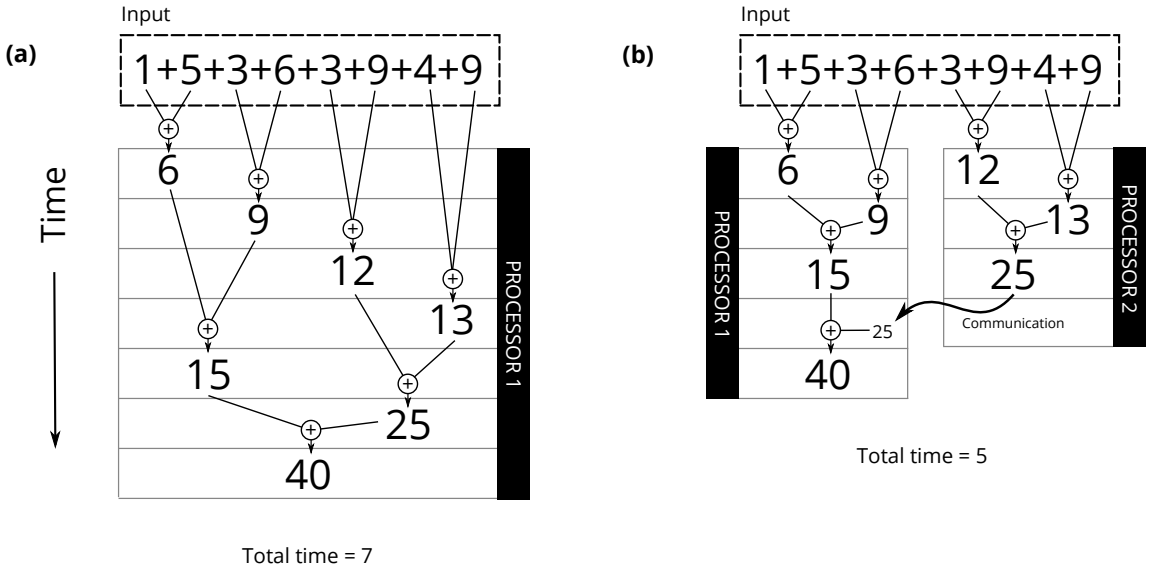


Figure 2.3: (a) Optimal steps to sum an array of 8 numbers on a single processor. (b) Simplified diagram of summation on two processors.

We have discussed *embarrassingly parallel* problems where the data is divided into pieces and an identical computation is performed on each of them, but in reality, many programs are more complex. It is possible that one instruction in P requires the result of a computation that has not yet taken place. In this situation, P is described as having data dependencies [9].

To solve data dependencies, processors wait for the dependencies to be satisfied and then continue the computation. It is evident that on an SMP machine, the higher the degree of data dependency, the closer the performance will be to that of a machine executing instructions sequentially [20].

The different types of parallel architectures were first proposed by Michael Flynn in 1966 falling in one of four main categories known as: SISD (Single Instruction Single Data), SIMD (Single Instruction Multiple Data), MISD (Multiple Instruction Single Data), and MIMD (Multiple Instruction Multiple Data). These traditional categories capture the two distinctive characteristics of a parallel system: how the

instructions are fetched and how the data is accessed.

In *single instruction* machines, the instructions are fetched sequentially, one at the time, and every processing unit in the parallel computer executes it. In SISD machines, cores operate on one data point from memory per every instruction; whereas in SIMD machines, cores can operate on different data points.

If cores can execute different instructions on the same parallel machine, they belong to the *multiple instruction* category. MISD machines can execute different programs on every core, but operate on a single data point. MIMD machines can run different programs on every processing unit while accessing different data points.

Modern parallel computers can keep track of which instructions of a *program* (a collection of instructions) are being executed on its parallel processors [17]. For practical purposes, processors are more easily understood as being able to run programs on their cores, therefore modern computers are "Multiple Instruction" by default.

With these considerations in mind, Darema introduced two subcategories to the MIMD parallel architecture: Single Program Multiple Data (SPMD) and Multiple Program Multiple Data (MPMD). In the case of embarrassingly parallel programs, an SMP machine using the SPMD architecture is able to execute on each processor p the function f on different input data. GPUs are an example of this architecture (see Section 2.4.1).

On an MPMD machine, each processor can have a different program that can execute in parallel on different input data. This is particularly important when the program is complex and there are computations that are dependent on each other or when control structures like branches and loops make the program behave in different ways depending on the input data. Multicore CPUs and SMP machines are examples of the MIMD architecture.

2.4.1 GPU Programming

Graphics Processing Units (GPUs) are specialized processors that are used in conjunction with CPUs to generate images that are presented to the user in a computer screen. They have been traditionally associated with video games.

To render images on the screen, video games have to perform millions of computations on three-dimensional geometries, distance calculations between points in space and image post-processing steps to generate a two-dimensional output. In order to create smooth animations, these steps are repeated multiple times per second—from 24 frames per second (fps) to more than 100fps.

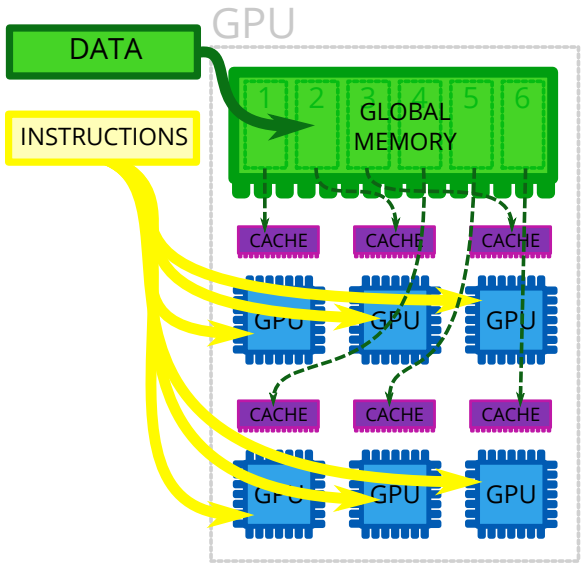


Figure 2.4: Simplified diagram of a GPU. It loads a program on each core and executes it in parallel on data that is stored in the GPU main memory, also known as VRAM.

GPUs are designed to perform simple operations in parallel, on large amounts of data, in contrast with multicore CPUs, which can execute a wide variety of programs but have only a handful of cores. GPUs have hundreds of cores but can only execute a limited set of instructions. This additionally limits the complexity of programs

in GPUs. The same collection of instructions is executed by every core, making GPUs not fit for general computing in personal computers, but provide performance advantages for repetitive operations.

An example of repetitive operations is a common post-processing visual effect used in video games called "bloom." It imitates the glow that can be naturally perceived around extremely bright objects. In order to create the effect, sources of light in a 2D image are identified, blurred, and then stacked on top of each other. The final step is to overlay them on the original 2D image [24].

For the bloom effect, every single point in the 2D input image (pixel) can be blurred independently, only requiring information about some of it neighbors. In the same fashion, to stack or overlay images, the value of every destination pixel can be computed knowing only the values of pixels from the other images in the set.

The bloom effect is embarrassingly parallel: to compute the output, only a small fraction of the input is needed and there is virtually zero dependency between the steps except for the final "stacking" step. This exemplifies how GPUs, due to their architecture, are extremely efficient for operations that require high parallelism.

Using GPUs for purposes other than graphics processing is an area of research that started more than 10 years ago [1], but for a long time there were no easy ways to send data to the GPU for processing. The only way to interact with the GPU was to represent the data as bitmaps, lists of vertices, and other graphics-related structures, and perform graphics operations on them: matrix addition becomes texture addition, etc. This changed with the introduction of programming interfaces that made the GPU cores accessible for general processing. NVIDIA, one of the major GPU manufacturers, released CUDA, a parallel computing framework that simplified the way software developers write code that is executed on the GPU [34].

Embarrassingly parallel data processing tasks like convolutions of signals [27], real time signal processing [41], image analysis [8], [31], and others, are routinely being computed using GPUs.

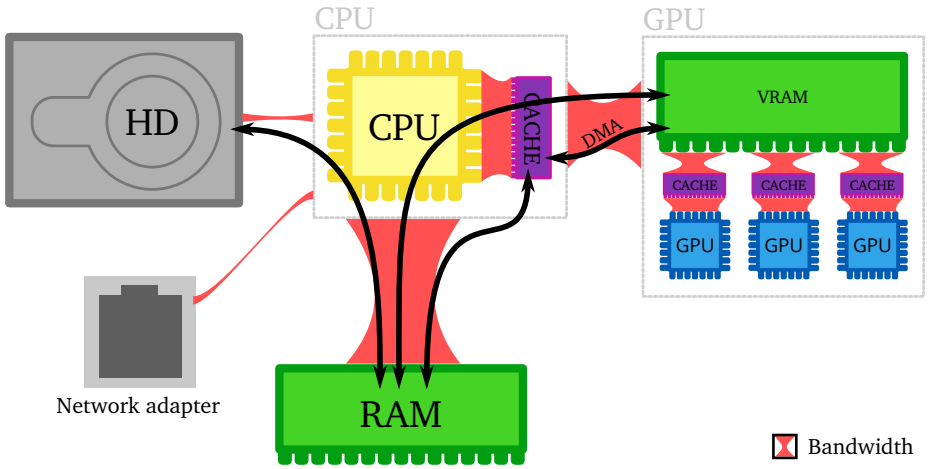


Figure 2.5: Simplified diagram of bandwidths between I/O devices and the paths that data follows to get from the hard drive to the GPU

All the approaches described in relation to parallelization, either using CPU cores or GPU cores to process information, are only viable if the amount of input data is not greater than the physical available memory of the system. CPU and GPU operations require any data to be processed to be moved to RAM or VRAM. If the total amount of input data is many orders of magnitude greater than the memory on which processing cores operate, the overall processing speed is going to be greatly reduced [25] (see Figure 2.5).

Since CUDA's launch in 2007, other CPU and GPU manufacturers have developed specifications and frameworks for high performance applications. Apple began developing a framework called the Open Computing Language (OpenCL) with the aim of standardizing the way programs are written to take advantage of the processing power of GPUs that otherwise were available only for graphical applications. After

defining the first implementation, Apple ceded control of the specification and the framework to the Khronos Group, a non-profit industry group conformed by Intel, AMD, and NVIDIA, among others.

OpenCL defines a C-like language that when compiled can be executed on *processing elements* (or processing cores) in parallel. For running the programs, OpenCL defines an API that is implemented by hardware manufacturers as a library that provides a compiler targetting a specific compute device. Programs written in OpenCL are called *kernels* and they are compiled at run time (they are generally distributed as source code), thus allowing different hardware to execute them.

2.4.2 Distributed Systems

Distributed systems are defined as a collection of computing resources that are linked together by a communication network [26]. The objective of these systems is to make multiple computers, also called *clusters*, behave as if they were a single computer. When the data that needs to be processed doesn't fit in the memory of one computer, it can be split up into multiple parts, called *tasks*, that can be distributed to the computers that conform the system.

In order for these tasks to execute in parallel, a *coordinator* decides which task will go to each computer and keeps track of which tasks have been completed. It is desirable to have a coordinator that *balances the load* of the system; a system where all the computing resources are utilized at all times is much more efficient than one where some resources are underutilized.

Load balancing is a fundamental factor in fulfilling the expectations placed on the distributed system [26]. Requests for data to be processed are called *jobs*, and they have constraints in the form of metadata that define their priority, ownership, etc.

Systems—through coordinators—are expected to execute them in a way that satisfies those constraints.

There are a multitude of paradigms and implementations for load balancing, scheduling, communication protocols, etc. For real-world applications, software developers generally use frameworks that facilitate writing distributed parallel programs by abstracting some of these complexities.

One of such frameworks is OpenMPI, an open-source library that implements the MPI-2 specification [23]. MPI is based on the message passing model. Programs running on different machines can send messages to each other and coordinate the execution of a job. The framework abstracts all the complexities related to network communication and synchronization across computers in the cluster, and does not force the programmer to adopt a data model or protocol for data distribution. It is up to the developer to decide how the data will be sent to the machines, along with defining procedures for failure recovery, tracking of task completion, etc. In synthesis, MPI, and therefore OpenMPI, provide the tools to write software that runs on clusters, but does not provide the logic and software infrastructure required to fully utilize clusters: it doesn't provide a coordinator.

Another framework that reduces the complexity of writing distributed code is Hadoop, originally developed by Doug Cutting and Mike Cafarella. It was subsequently adopted by Yahoo and eventually became a top-level open source project at Apache [54]. It provides an easy-to-use parallel programming model based on MapReduce.[1] Hadoop facilitates communication between computers in a cluster and also provides a redundant distributed filesystem that is tolerant to failures [12].

[1]MapReduce is a parallel programming model where computations are represented by two functions: *Map* and *Reduce* [18]. See Figure 2.2.

Both in OpenMPI and Hadoop, just as with general paralellization techniques, the difference in performance between a sequential approach and a parallel approach is driven by how much concurrency can be achieved.

2.4.3 Heterogeneous Systems

Clusters were traditionally built using expensive computers and high-reliability network connections. With the introduction of failure-tolerant technologies and the cost of key technologies (for example Gigabit ethernet and high-performance GPUs), clusters conformed of consumer-grade, commodity computers are more and more available. Moreover, companies that have large numbers of computers that are idle overnight or have long periods of inactivity during business hours can convert them to be part of a cluster, sending data processing jobs over the network in what is called a "desktop cluster" [22].

These emerging distributed systems are not uniform. The local resources at every node are different, and implementing techniques used in traditional distributed systems would require armies of information technology administrators making sure that the distributed software runs efficiently. The alternative is to have software and practices that can secure the data as it is moved to the nodes, and software running on the computers that can decide how to utilize the local resources efficiently, without human intervention [22].

This is the challenge with heterogeneous distributed systems: different computers with different capabilities have to operate together as a single compute unit. Additionally, to fully utilize cluster resources, all available CPUs and GPUs need to be used for processing. Generally, parallel programs that operate in cluster environments are executed on the CPU, and there are no simple ways to automatically take advantage

of GPU resources available in the system, leaving the task of writing efficient code that utilizes all computational resources up to the software developer [25].

Chapter 3

METHODS

3.1 Study Site and Infrastructure

Boise State University maintains an experimental remote sensing station equipped with a low power infrasound array that can record data continuously for months at a time. This station is located on "Avalanche Alley," roughly 120 meters south-east of Highway 21, at the 100.5 mile marker. Its exact position is 4414'15.8"N 11512'44.2"W.

On the opposite side of the highway there are three frequent avalanche paths (areas known to have regular avalanche activity), designated as 100.4, 100.48, and 100.62. There are in total more than five avalanche paths, but they don't produce avalanches as often.

The system consists of five microphones positioned in a star pattern and connected to a data logger via copper cables running inside plastic conduit to protect them from the elements (see Figure 3.1). Systems with multiple distributed sensors are called *arrays*.

Because of the remoteness of the study site and the large amounts of data that are generated every day, there are no practical ways of transmitting this data out. For the proof of concept study site, the data files where manually retrieved. These visits are also used to verify that all the components are functioning appropriately and to perform maintenance on those that are not.

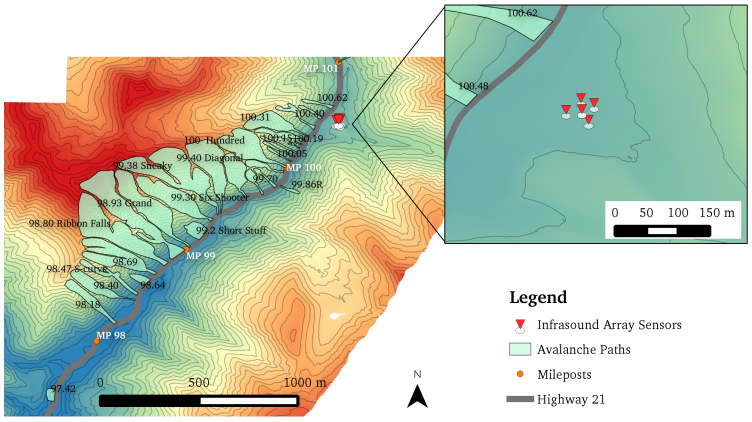


Figure 3.1: Avalanche paths along Highway 21 and study site location

The power for all the instruments and sensors is provided by two 12 volt batteries that are charged during daytime by an 80 Watt solar panel (See Figure 3.2). The batteries are housed inside a plastic box along with a solar battery charger. The microphones are connected to an adjacent box that contains a computer, a data acquisition instrument (also known as data acquisition card or DAQ), power adapters for the computer, and satellite transmission equipment that allows sending small amounts of data at very low speeds.

3.2 Data Acquisition and Formats

The raw electrical signal coming from the sensors has to be converted to a digital signal that can be stored in a computer. Microphones can measure pressure changes in the air and convert them to voltages, which in turn need to be converted to digital

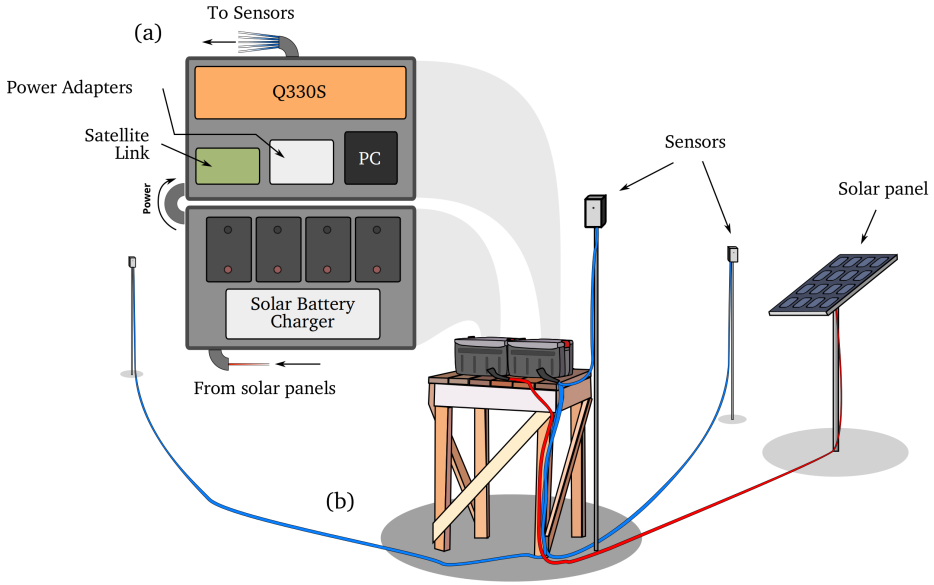


Figure 3.2: (a) Layout of the components inside boxes. (b) Illustration of part of the study site

values. Specialized devices called Analog to Digial Converters (ADCs) are used for this purpose. ADCs convert a measured voltage sample to a digital number of certain precision. DAQs are comprised of multiple ADCs called analog inputs.

During the winter of 2013 and through the end of the spring in 2014 this research station recorded infrasound signals from 5 microphones (from now on referred to as sensors) using a QUANTERRA Q330S+ DAQ.[1] The measurements from the sensors are continuously recorded on-board—in what is known as time-series data—and after a set interval are transferred to the computer to be written to files. Each one of these sensors has a name assigned to it, which is later used to distinguish them.

The physical analog inputs to the ADC and also the individual digital sensor recordings by each analog input are known as *channels*. Each ADC channel can convert sensor outputs to 24-bit precision integers many times in a second, and the

[1]The product details can be found at http://www.q330.com/Products/Q330SPlus-RevD.pdf

number of samples recorded in one second is called the *sample rate*, measured in Hertz. For this project, the 5 channels were sampled at 100Hz and the total amount of data collected was 25GB.

The data is organized and stored using the MiniSEED standard, a subset of the SEED standard (Standard for the Exchange of Earthquake Data). SEED and by extension MiniSEED are maintained by the International Federation of Digital Seismograph Networks, and documentation on the data format is available online.[2]

In order to process these signals, the first step is to extract them from the MiniSEED files. Software libraries are available for most programming languages: libmseed[3] for C/C++ and Python via `ctypes` and SeisFile[4].

The DAQ can store sensor readings for long periods of time in big files, but for this project the data is copied to an external computer, in other words offloaded, at configurable intervals. When offloading, the data is transferred to the computer and stored as many relatively small files. The alternative is to offload big files from the DAQ with the increased risk of data loss. Each offloaded file contains metadata that indicates, among other things, the start time and end time of the data contained in that file.

3.3 Event Detection

An event is defined as a moment in time when something of importance happened. In our application, and because we are continuously recording, the definition of

[2]The full specification can be accessed at http://www.fdsn.org/seed_manual/SEEDManual_V2.4.pdf

[3]The source code of libmseed can be accessed at:
https://seiscode.iris.washington.edu/projects/libmseed/repository

[4]Distribution versions and the source code for SeisFile can be downloaded for Java from: http://www.seis.sc.edu/seisFile.html

something of importance is any moment in time when a signal caused by an external *coherent* source arrives at the array. Coherent sources are those whose waves originate at a spatially constrained point and whose signal is similar on all sensors in the arrays; some examples are cars, explosions, and avalanches. In contrast, the arrival direction of incoherent sources cannot be reliably estimated, or the waves they produce are uncorrelated and the signals are not coherent on all sensors in the array. The most common example is wind: although its speed and direction can be estimated and there is some correlation in the signal, the pressure changes at the sensors are expected to vary (i.e., due to turbulence) more than for coherent, slow-moving sources. [13].

One of the most difficult challenges in digital signal processing is removing the noise that is always present in real-world measurements. The noise can hide events in the signal or can make non-events look like events.

To summarize, in order to detect events, we are interested in coherent signals that are always obscured to some degree by noise originating from inside the sensors and external noise coming from mostly uncorrelated signals like wind. When an event occurs and the amplitude of the signal is similar to the amplitude of the noise (low signal-to-noise ratio or low SNR), additional techniques need to be employed to detect these events.

The following sections discuss techniques to process signals recorded in an array and theory behind noise reduction(Section 3.3.1), an additional strategy to determine similarity between signals in sensor arrays (Section 3.3.2), approaches to detecting when an event has taken place (Section 3.3), how to estimate the arrival direction of an event (Section 3.3.4), and strategies to make these computations more efficient by performing them in parallel in the GPU (Section 3.3.5).

As a final point, these techniques are only useful in determining that an event

has taken place. The steps to identify what kind of event occurred are discussed in Section 3.4.

3.3.1 Array Processing

In the context of array processing (the analysis of data being recorded simultaneously by different sensors), *correlation* is an important concept. When two signals are correlated, it means that they are similar and also that they are likely recordings of the same event: the waves generated by an event will be captured by multiple sensors. An extension of this concept is called cross-correlation, in which recorded signals are compared with each other, as a function of the lag, or the delay, of one relative to the other. See Figure 3.3.

We expect signals recorded by an array to be different depending on the relative position of the source and each sensor. For sensors that are closer to the source, the wave-front will arrive and be recorded earlier than in the further-away sensors. The time difference in the arrival of signal to a pair of sensors is called a "time lag." The amount of time that the signals need to be shifted in every sensor to eliminate the lags are called "time shifts."

$$(f \star g)[s] \ = \sum_i f[i] \ g[i+s] \tag{3.1}$$

Cross-correlation is defined in Equation 3.1, where f and g are discrete functions (in computer science, arrays), and the cross-correlation for time shift s is computed by multiplying every distinct value at the time i of the first function by the value at time $i + s$ from the second function. Figure 3.5 shows three signals recorded by the infrasound array before and after applying time shifts.

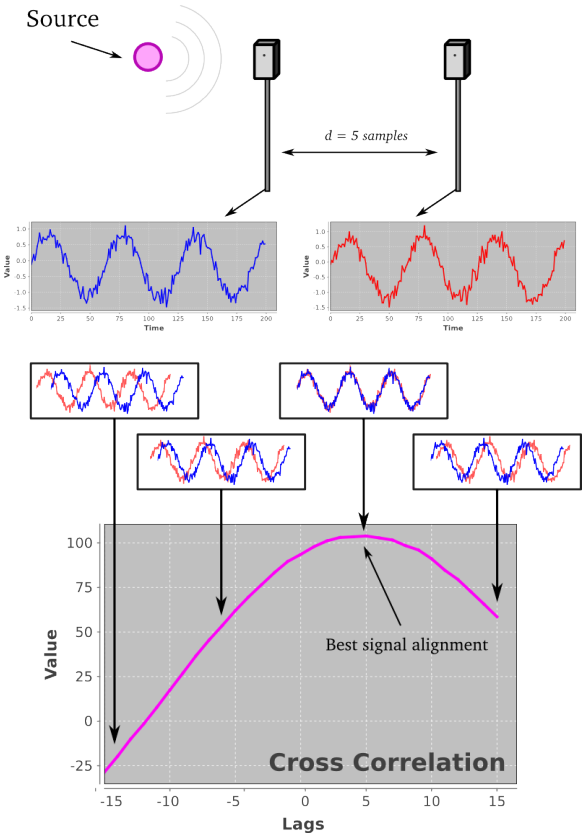


Figure 3.3: The difference in distance between sensors and the source of the event makes the recorded signals have a delay relative to each other. By computing the cross-correlation, they can be aligned

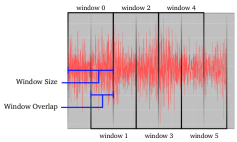


Figure 3.4: Processing windows over a signal

For practical and physical purposes, cross-correlation is not applied to the the complete discrete function but to small sections called *processing windows*. Sources of sound that are moving will create different time lags in the sensors, which can be more precisely identified by using smaller processing windows and overlapping them (see Figure 3.4).

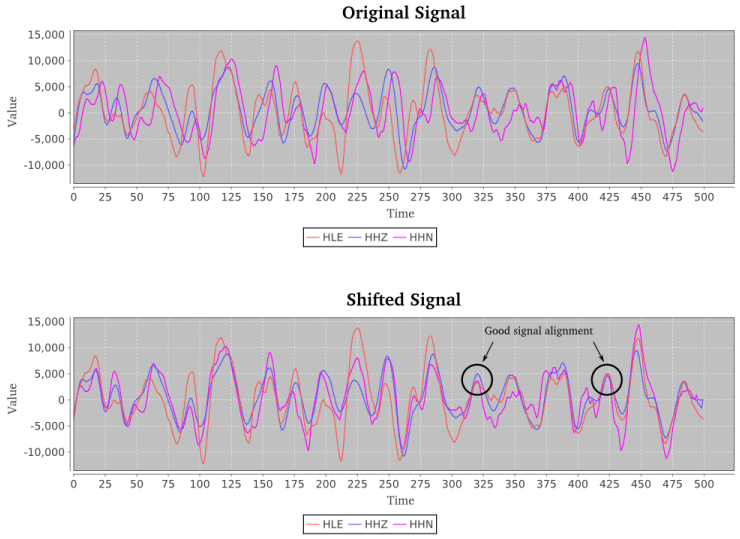


Figure 3.5: By identifying the time lags and shifting, signals can be aligned.

Further, when a signal arrives at the array, all the sensors will record the signal plus an uncertain amount of noise. The main assumption is that any noise present *will not be correlated*, will be stationary[5], and will follow a Gaussian distribution with

[5]This requirement excludes the sources of noise that change coherently over time, in which case it cannot be noise—it must be a signal.

a mean of 0 [52]. To put it another way, the sensors will be recording the sum of a coherent signal and a random noise signal that fluctuates in value around 0 [11].

Another assumption is that the speed of the wave traveling through the medium—in this case air, is constant. We assume the wave moves at a constant speed so there are no sudden accelerations that would make the wave arrive at a sensor farther from the source before it arrives at a sensor closer to the source. This is reasonable as the speed of sound in air is relatively constant.

The relationship of noise and a coherent signal for an array of sensors located equidistantly to a source is shown in Equation 3.2. The function $x_i(t)$ is the output of sensor i at time t, $s(t)$ is the coherent signal at time t, and $N_i(t)$ is the noise recorded by sensor i at time t; finally, n is the total number of sensors in the array.

With these assumptions in mind, and for a source that is equidistant to all sensors in the array, we can expect that any coherent source will be recorded by all sensors and that by summing the signals and normalizing, the uncorrelated noise can be removed (see Equation 3.3).

$$\begin{aligned} x_i(t) &= s(t) + N_i(t) \\ s(t) &= x_i(t) - N_i(t) \end{aligned} \tag{3.2}$$

$$\begin{aligned} &= \frac{\sum_{i=0}^{n} x(i,t) - N_i(t)}{n} \\ &= \frac{\sum_{i=0}^{n} x(i,t)}{n} - \cancelto{0}{\frac{\sum_{i=0}^{n} N_i(t)}{n}} \end{aligned} \tag{3.3}$$

At any rate, we must first calculate the time shifts, or in other words, *align* the signals so that events occur at the same time in all signals. When the signals are

aligned, we can analyze their correlation.

3.3.2 Fisher Statistic

Another correlation method is called the Fisher Statistic in the time domain. It is defined as the power of the beam over the residual power [28]. The Fisher statistic is computed using Equation 3.4, where M is the number of sensors in the array, N is the size of the processing window, and l_j represents a time shift to be applied to each sensor j. $x_j(n)$ is a discrete function representing the output of sensor j at time n.

The important aspect of this statistic is how it performs when there is a correlated signal in the data. The numerator will increase when the time shifts selected align the signals, as exemplified in Figure 3.3. The denominator is similar, but every value in x_j is subtracted from the average power of all channels (see last term in the denominator's summation). When there is high correlation in the signals, the denominator decreases because the average will be closer to the magnitude of the signals, lowering the values and increasing the Fisher statistic. The first term in the equation normalizes the statistic for the number of sensors in the array.

$$F = \left(\frac{M-1}{M}\right) \times \frac{\sum_{n=1}^{N}\left[\sum_{j=1}^{M} x_j(n+l_j)\right]^2}{\sum_{n=1}^{N}\left[\sum_{j=1}^{M}\left[x_j(n+l_j) - \frac{1}{M}\sum_{m=1}^{M} x_m(n+l_m)\right]^2\right]} \tag{3.4}$$

In order to efficiently calculate the F statistic, we represent the observations of every channel x as a column and individual observations over time as rows. This creates a matrix of dimensions $M \times N$, allowing us to represent the problem as a series of operations that happen on each row of the matrix (notice the outer summations

are done over columns).

To calculate the l_j terms in Equation 3.4, we test source locations for events and calculate the time lags that those locations would create. With these time lags, the signal can be shifted. Additionally, we know that certain time lags are impossible because of the assumption that the wave's speed doesn't change as it moves through the array: it cannot reach a further sensor *before* a closer one or move slower than the speed of sound.

Therefore, to test all the possible time shifts, we place virtual sources in a three-dimensional grid around the array and simulate the arrival times of the wave at the different sensors. Sound moves at approximately 320 meters per second, and the arrival times can be calculated based on the distance between the source and each sensor. Further, we know the sampling rate of the DAQ, or in other words, how many samples are being recorded per second, which enables us to convert distances and times of arrivals to numbers of samples, effectively converting time lags to *sample lags* that are represented as indices. The full algorithm for this calculation is described in Algorithm 1.

We can calculate all the potential positions where a sound source could be located and calculate all the possible time shifts that would need to be applied to the signals to align them. The steps outlined in Algorithm 1 are applied for every grid node in Figure 3.6. The grid is spaced based on the speed of sound and the sample rate: sound moves 320 m/s, which means that for a sampling rate of 100 samples per second, a sound wave travels 3.2 meters in one sample. If the grid nodes are placed at 3.2 meters in every direction x, y, and z, we arrive at a compact and relatively small search grid, with bounds set to the minimum and maximum value of every component (x, y, z) for each sensor position, plus two samples-distance in every direction. Additionally,

Algorithm 1 Compute time shifts for arbitrary source. *src* is the position (x, y, z) of an arbitrary source, in meters, relative to the position of the reference sensor *ref*, *total* is the total number of sensors, *pos* is the location of every sensor represented as an array of objects that contain the three position components (x, y, z), *ref* is the position of the reference sensor with components (x, y, z), and *rate* is the sampling rate of the DAQ.

```
1: function CalculateShift(src, total, pos, ref, rate)
2:     for i = 0 to total do                    ▷ Make locations relative to reference
3:         pos[i].x −= ref.x
4:         pos[i].y −= ref.y
5:         pos[i].z −= ref.z
6:     end for
7:
8:     shifts = []
9:     deltas ← (0, 0, 0)
10:    for i = 0 to total do ▷ Compute distance between the source and every node
11:        deltas.x ← pos[i].x − src.x
12:        deltas.y ← pos[i].y − src.y
13:        deltas.z ← pos[i].z − src.z
14:        distance ← √(deltas.x^2 + deltas.y^2 + deltas.z^2)
15:        shifts[i] ← (distance/SPEED_OF_SOUND)/rate ▷ Convert meters to time and finally to
   samples
16:    end for
17:
18:    for i = 0 to total do          ▷ Make the reference sensor have a shift of 0
19:        shifts[i] −= shifts[ref]
20:    end for
21:
22:    return shifts
23: end function
```

only unique time shifts are retained to avoid computing duplicate source locations.

The reason for removing duplicate shifts is rooted in how signals arrive at the array. Increasingly distant source locations with the same relative distance to the array sensors will yield identical time shifts: it is unknown how far the source is, the signal is only detected when it reaches the array, not at the moment it is generated.

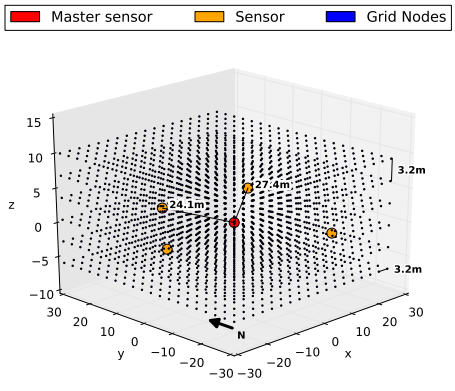


Figure 3.6: Grid used to compute all possible time shifts for signals with varying source locations.

With this information, we can assert, for example, that a signal source located at coordinates $\Delta x = 12.8, \Delta y = 16$, and $\Delta z = 3.2$ relative to the the central array sensor would require the signals from every sensor in the array to be shifted $(0, -1, -4, 6, 7)$ samples to align them. Notice the first shift is zero, meaning the first sensor was used as the reference or *master* sensor. Which sensor is selected to be the reference sensor is irrelevant, since the shifts to be applied can be both positive or negative.

Having calculated all possible time shifts, we can compute the Fisher statistic for arbitrary signals with arbitrary time shifts.

3.3.3 Signal Filtering and Detection

We discussed how signals that are coherent are a tell-tale sign of events and how calculating the Fisher statistic can provide a measure of how coherent these signals are. We have also discussed that the Fisher statistic must be evaluated for varying time shifts, and that these time shifts are calculated from all possible signal source locations. The maximum Fisher statistic value will be the one for which the chosen time shifts better align the signals, and its magnitude will be greater when the signals are highly correlated.

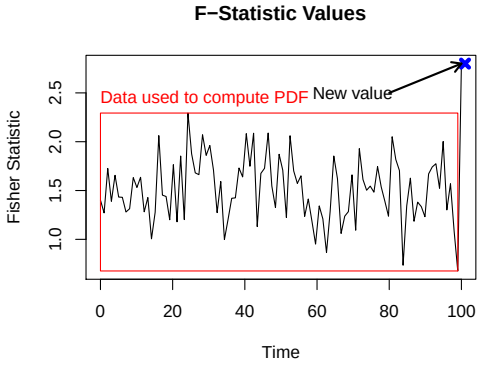
A simple approach to detect events is to calculate the Fisher statistic for a sliding window of samples and compare it with a threshold value. Any time this value is exceeded, it is assumed that an event has taken place.

Algorithm 2 Naive event detection

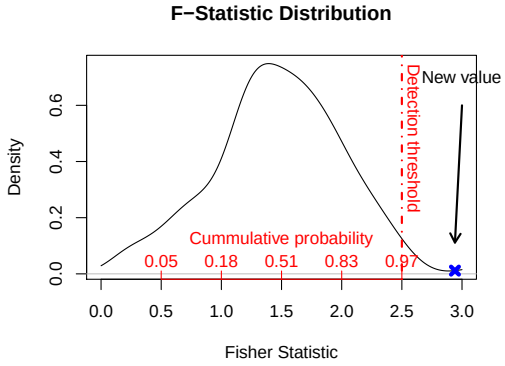
```
1: max_fisher = 0
2: for all possible_time_shifts do
3:     fisher ← computeFisherStatistic(signal, time_shift)
4:     if fisher ¿ max_fisher then
5:         max_fisher ← fisher
6:     end if
7: end for
8: if max_fisher ≥ threshold then
9:     ...                                          ▷ Event detected
10: end if
```

One of the difficulties and drawbacks of this approach is having to choose a threshold. Changing weather conditions, noise, and other factors make it impossible to choose a single threshold value that can accurately distinguish events from non-events.

We use an adaptive statistical approach after Arrowsmith et al. [7]. This technique compares the latest Fisher statistic to the a number of stored Fisher statistics called the "back window." This is repeatedly performed on processing windows.



(a) Sample Fisher statistic values over time



(b) Non-parametric distribution and testing for a new Fisher statistic value

A non-parametric probability distribution (specifically a kernel smoothed density function) is created by using the values from all the windows except the last. The last Fisher statistic is compared against this distribution: if the probability is greater than a threshold, an event has been detected (see Figure 3.7a).

The difference in this method is that the threshold represents the likelihood of a value belonging to a distribution, and this distribution is constantly updated. If the previous Fisher statistics are changing over time, the distribution will change and the probability of a future point being above the threshold remains the same.

It was experimentally established that finding the probability of an F-value being above a probability threshold was a robust strategy to detect events, but in many cases, short-duration wind bursts could be spatially correlated and yield false positive detections.

Since the power of frequencies present in wind signals are concentrated above 20 Hz, we filter the signals using a Butterworth band-pass filter[6] (a type of filter that removes unwanted frequencies from a signal [14]) by retaining frequencies in bands of

[6]Java implementation by Christian d'Heureuse, accessible at http://www.source-code.biz/dsp/java/

2 Hertz in width, from 2 Hz to 20 Hz. The first filtered signal only retains frequencies between 2 Hz and 4 Hz, the second, frequencies between 4 Hz to 6 Hz and so on with the last being 18-20 Hz. Finally, the original signal is filtered to exclude frequencies below 2 Hz and above 20 Hz.

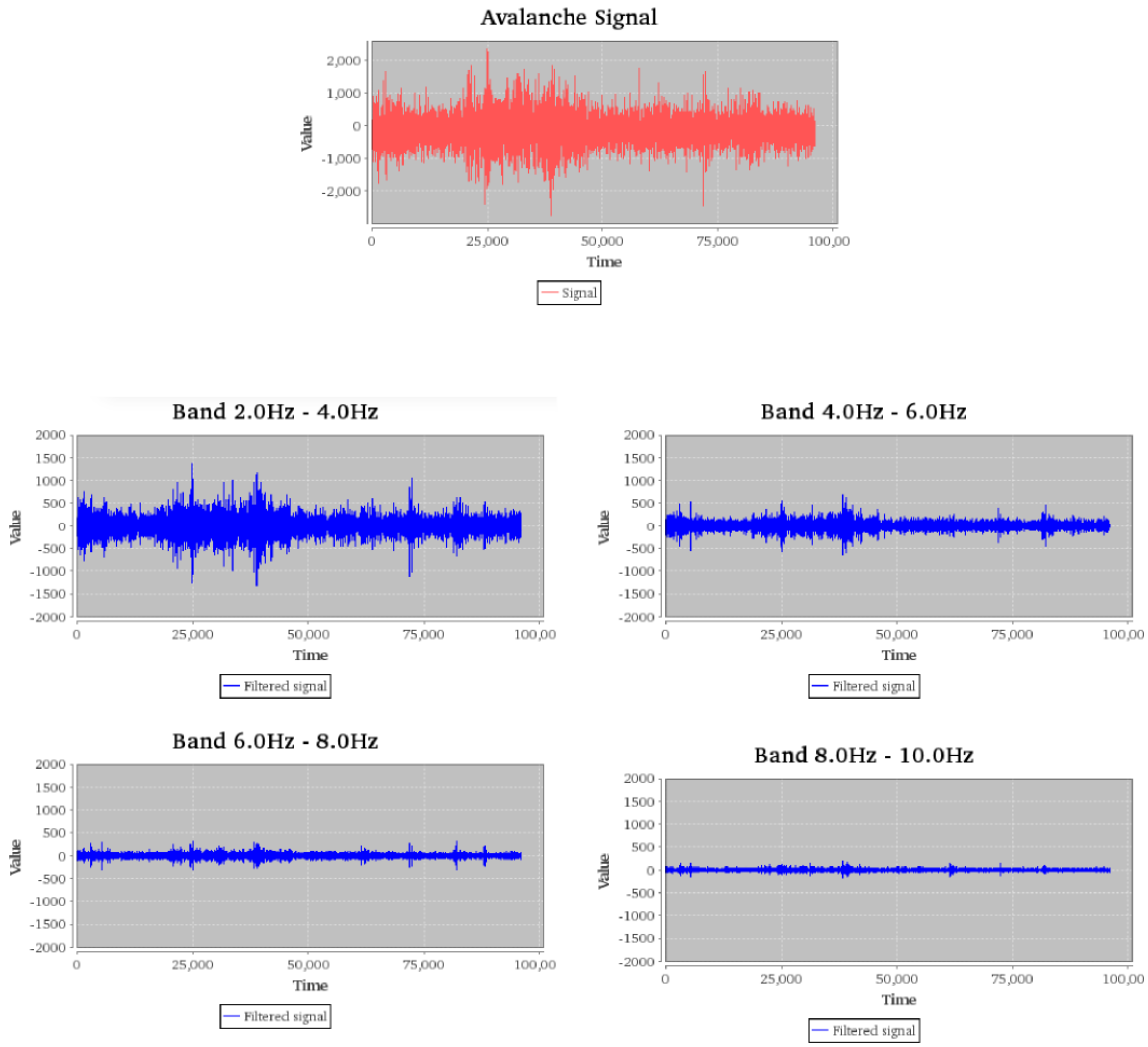


Figure 3.8: Result after filtering the signal with a band-pass Butterworth filter.

Algorithm 3 Event detection for a single signal (one band) divided into processing windows. Constant `BACK_WINDOW_SIZE` is the number of samples used to create the non-parametric distribution.

```
1: for all window in processing_windows do
2:     for all possible_time_shifts do          ▷ For every pre-calculated time shift
3:         fisher_values ← computeFisherStatistic(window, time_shift)
4:     end for
5:     fisher_for_window[window] ← Max(fisher_values)
6:     shifts_for_window[window] ← FindBestShifts(fisher_values)
7: end for
8:                ▷ Distribution created using a number of previous processing windows
9: distribution ← NonParametricDistribution(fisher_for_window[1:BACK_WINDOW_SIZE])
10:
11: for window in processing_windows starting at BACK_WINDOW_SIZE do
12:     probabilities[window] ← CalculateDensity(distribution,fisher_for_window[window])
13:     UpdateDistribution(distribution,fisher_for_window[window])
14: end for
```

Subsequently, and following the same steps as with a single signal, these 10 filtered signals are divided into processing windows. The Fisher statistic is then calculated for each one of the windows and a non-parametric probability density function is created using some of these values. The values used to create the distribution are called the *back window*). Finally, the density of the value of the windows not in the back window are calculated.

This algorithm is further described in Algorithm 3.

The probabilities from the 10 bands are then multiplied together and this value is used for detection: when the value is below a threshold, an event has been detected

(see Figure 3.9).

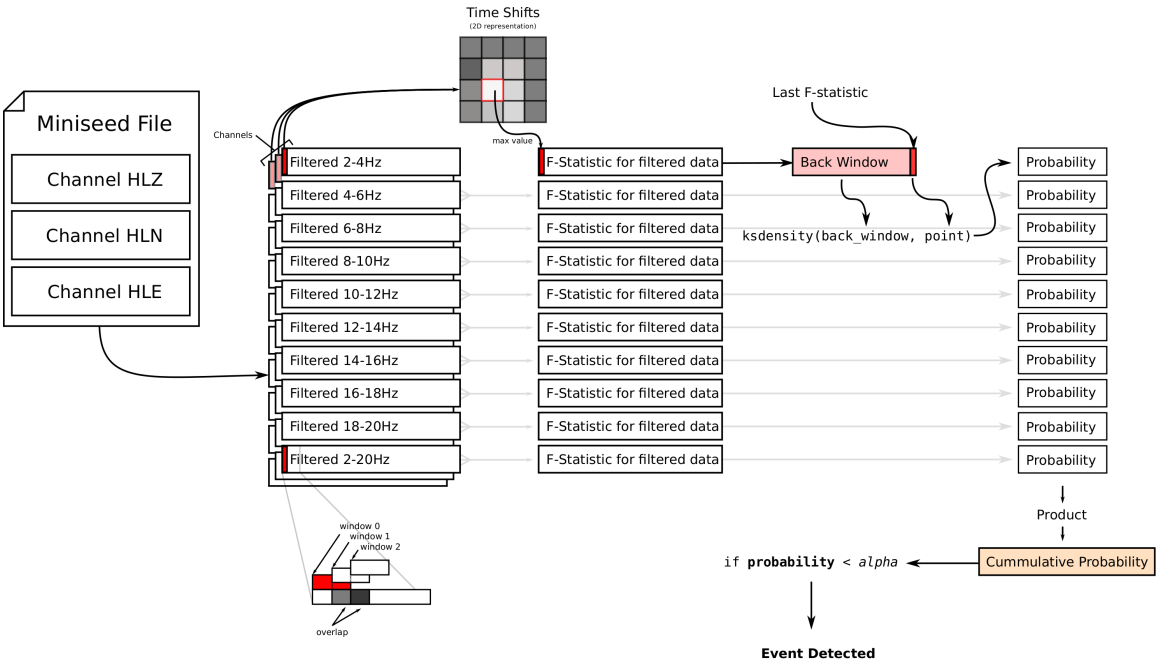


Figure 3.9: Data flow for the event detection algorithm

3.3.4 Backazimuth Estimation

Previously, we discussed how in the process of calculating the Fisher statistic, the maximum Fisher statistic will be one for which the time shift values align the signal, and how these can be used to determine the direction of the signal relative to the array.

Because the possible time shifts were calculated for sources located at specific spatial locations, it is trivial to calculate the incidence and backazimuth angles (see Figure 4.8).

The objective of estimating the incidence angle (degrees above the horizontal plane) and the backazimuth (degrees from north) is to aid in the classification of events.

$$\begin{aligned} \text{incidence angle} &= \measuredangle\text{ABC} = sin^{-1}\left(\overline{AC}/\overline{AB}\right) \\ \text{backazimuth angle} &= \measuredangle\text{CBD} = sin^{-1}\left(\overline{CD}/\overline{CB}\right) \end{aligned} \tag{3.5}$$

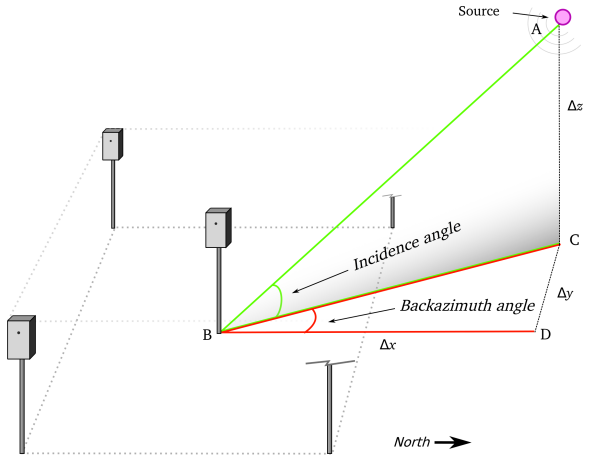


Figure 3.10: Diagram of angles involved in source location estimation.

3.3.5 GPU Implementation

In Section 3.3, we explained how the Fisher statistic and a non-parametric distribution are used to detect events and how the time shifts selected in the process are used to calculate the source of the event.

At this point, we can visualize the Fisher statistic equation as a matrix, where columns are individual sensor outputs and every row is one observation in time. In order to compute the statistic, we must *a*) divide the data into windows, *b*) shift the channels by a certain number of samples as defined by time shifts, and finally *c*) calculate the F-statistic on the shifted data for the current window by summing every row, then calculating the average of said row and accumulating these values as we progress down the matrix.

Since these steps are not sensitive to the order of the operations, this is an embarrassingly parallel problem.

Algorithm 4 Compute the F statistic for a matrix `data` where columns are channels and rows observations.

```
1: for every window in windows do
2:     for every time_shift in time_shifts do
3:         matrix ← ShiftDataForWindow(data, window, time_shift)
4:         for every row in matrix do
5:             row_average ← average(row)
6:             row[:] -= row_average        ▷ subtract average from all elements in row
7:             row_sum ← sum(row)
8:             row_square ← row_sum^2
9:             normalized ← (N−1)/N × row_square
10:        end for
11:        fisher[time_shift] ← normalized / row_sum
12:    end for
13:    best_time_shift[window] ← FindMaxTimeShift(fisher)        ▷ Find the maximum Fisher value and get its associated time_shift for the current window
14: end for
```

As mentioned in Section 2.4.1, OpenCL is an open standard that allows programs written in a C-like language to run on compute devices. APARAPI[7] is an open source API and library developed by AMD and currently open source that allows Java bytecode to be converted to compatible OpenCL for parallel execution.

It places constraints on what code that can be converted, currently supporting only native Java data types and static memory access. These limitations do not affect our application, where all the computations are done over two dimensional arrays and the size of the output is known beforehand.

[7]Available from AMD at: http://developer.amd.com/tools-and-sdks/opencl-zone/aparapi/

3.4 Event Classification

In the following sections, we discuss the techniques used to *classify* events after they have been detected. Classification can be achieved by analyzing statistical indicators of the data, like the length of the event or its most likely source (described in Section 3.3.1). Another statistic that can be analyzed is the set of frequencies that are present in the signal.

3.4.1 Fast Fourier Transform

Fourier analysis is used to convert waves in the time domain to frequencies. The discrete Fourier transform (DFT) can be used to estimate the signal amplitudes as a function of frequency.

$$X_k \stackrel{\text{def}}{=} \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N}, \quad k \in \mathbb{Z} \tag{3.6}$$

Equation 3.6 shows the real discrete Fourier transform. There are N real values in x, and the function is usually computed for $N/2$ different k. X_k is a complex number that represents the amplitude and phase of a sinusoidal component of the original function x. Each X_k is related to a frequency present in the signal, for example, if X_5 has a smaller value than other X_k, we conclude that the frequency equivalent to k cycles per N is a greater contributor to the content of the signal.

In reality, the DFT can be represented as a multiplication between a vector of length N with an $N \times k$ matrix, with k being calculated up to N or $N/2$ depending on the application. Furthermore, the matrix is symmetric, which allows for optimizations that reduce the cost from $O(N^2)$ to $O(NlogN)$. The Fast Fourier Transform (FFT) is an example of one of these optimizations [16].

For this application, we compute the FFT of the signal and summarize it by using *power bands*. Bands are ranges of frequencies that are characteristic of a certain type of event as shown in Table 3.1.

Frequency band	Common sources
2-8 Hz	Avalanches, Earthquakes, Atmospheric phenomena
8-16 Hz	Avalanches, Explosions
16-20 Hz	Avalanches, Vehicles, Wind
20-50 Hz	Vehicles, Wind, Planes, etc.

Table 3.1: Frequency bands and common signal sources that contain high power in a given band.

A power band is defined as the total power content of the frequencies that fall between its lower limit and upper limit. We use five power bands, the first band being the sum of all the frequencies from 1 Hz to 5 Hz, the second is a sum from 5 Hz to 10 Hz. Other bands are defined from 10-15 Hz, 15-20 Hz, and 20-50 Hz.

As was previously stated in Section 2.1.2, avalanche signals have a characteristic frequency signature, therefore calculating the power along bands can serve as a good indicator to differentiate and classify events.

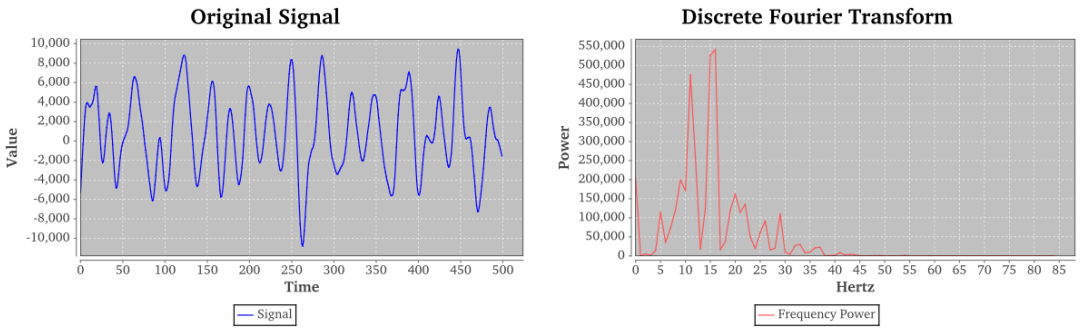


Figure 3.11: A signal is converted to its frequency components

3.5 Machine Learning Techniques

3.5.1 Artificial Neural Networks

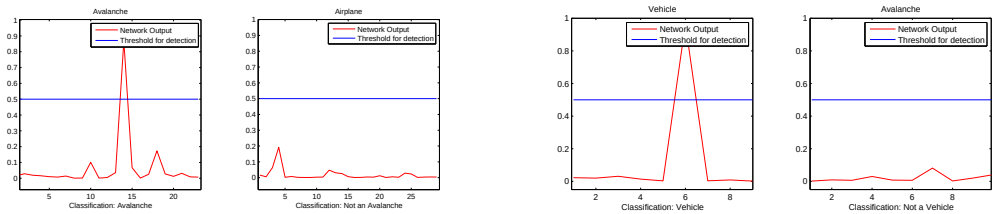
Artificial neural networks (ANNs) are based on biological brains, and as such are conformed by units (neurons) that are connected to each other. These connections can have a varying *strength*, which is the basic factor that modifies the behavior and output of the network [39]. In order to use artificial neural networks for classification, these connections need to be modified to achieve a desired output. The process of tweaking the strength of connections is called training [37].

The most common type of neural network is the feedforward network first described by McCulloch and Pitts[39]. The neurons are organized in layers with the connections between units in layers made strictly in a forward direction. The first layer is denominated the *input layer*, where the variables or inputs are supplied to the neurons, and the last layer is called the *output layer*, where the result of the neural network for the given inputs is retrieved. The layers of neurons in between the input layer and output layer are called "hidden layers." Every neuron computes its output based on its inputs multiplied by the strength of the connection with a previous neuron. These connection strengths are called weights, and are what determines how much the output of the previous neuron affects the current neuron.

Recurrent neural networks (RNNs) are a type neural network similar to feedforward networks that can maintain an internal state by giving neurons the ability to store its previous output and can allow for complex temporal decision-making and classification in continuous data [19]. Results obtained using RNNs are discussed in Section 4.

3.5.2 Training

Just as with biological neural networks, ANNs (and by extension RNNs) are trained by providing an input and an "ideal output." The input—also called sample—can be an image, a sound snippet, or any other data that can be represented as numbers. Along with the sample, there is a label (ideal output) that describes what the network should output. The output of the network is usually a number that is greater for a positive input and smaller for a negative output. When all the available samples in the training dataset have been supplied to the network for training, it is said that the network has been trained for an *epoch*.



(a) **Avalanche Classification Network** (b) **Vehicle Classification Network** out-Network output for avalanche and airplane put for a vehicle and an avalanche

The database of events used to train the neural network was compiled by Scott Havens, a recent PhD in Geophysics now at USCA NWRC. The events were classified by manually looking for patterns in the signals and analyzing characteristics like length and origin of the signal (calculated using a similar method to the one used in this project). These characteristics in the signal help to determine what kind of event took place. The most likely type of event is labeled using a descriptive word like "Avalanche," "Vehicle," etc., or—for signals which are not clearly identified—the label "unknown." The events stored in the database are considered to be correctly labeled, and in some cases they are correct beyond any reasonable doubt. In other cases, it is possible that the label applied to the event is not correct.

Training a network is a computationally expensive process [37]. As long as there are enough samples (labeled inputs), the network will continue to get more accurate in its classification through training. In the opposite scenario, where there are only a few samples available for training, there is a high probability that the network will be accurate in classifying all the samples in the training set but will be very inaccurate when classifying samples that belong to the same class but are substantially different (the network *over-learns* or has been over-trained).

The artificial neural network's architecture determines—to a certain extent—its capacity to recall (or remember) learned patterns. For this project, different types and architectures of neural networks were used (see Section 4), and the training was done using backpropagation as described by Hecht-Nielsen [29] and simulated annealing, a technique that can reduce the training time by modifying the rate of learning [15].

For this project, Encog[8], an open source neural network and machine learning framework was used to define, train, and subsequently evaluate the neural networks.

3.6 Distributed Processing

In Section 2.4.2, we discussed the requirements and difficulties of developing software that can operate in distributed systems, and the necessity of using frameworks to accelerate the development of solutions that are easily scalable and resilient to errors.

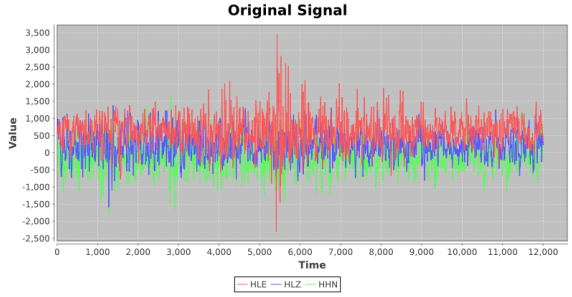
Apache Hadoop was used in this project for its fault tolerance and its scalability. Its most characteristic feature is that every operation is defined exclusively as having a `<key, value>` pair as its input (values can be of different types), and a `<key,value>` pair as its output.

[8]Developed by Jeff Heaton and available at http://www.heatonresearch.com/encog

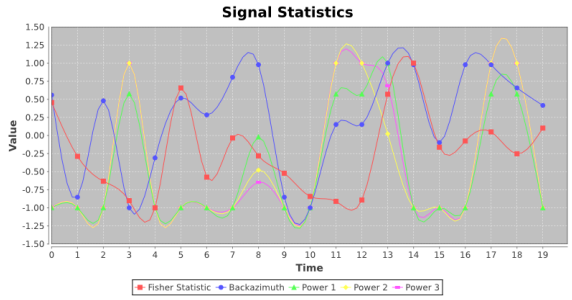
Hadoop uses the map-reduce design pattern, a pattern similar to divide-and-conquer where the data is first split into multiple parts, and then these parts are *reduced* or combined to produce the final output. An example of this design pattern is counting words in text documents. The individual documents are analyzed line by line in the map stage. The mapper receives the line number of the document as its key and the content of the line as its value and produces key-value pair every word in the line as the key, and the number one as the value. In the reduce step, all the records that have the same key are combined and the total number of occurrences can be calculated.

The first step to the detect and classify events in Hadoop is to load the data onto a distributed filesystem so that every node has access to every file. After, files are generated that contain information about the processing, the interval of data that is going to be processed, etc. These files (called InputSplits) are read by Hadoop and a new or various mappers depending on the size of the file are instantiated to process them.

The mapper then streams each of the files from the distributed file system and performs all the event detection steps (as outlined in Section3.3). If an event has been detected at any point in the data file, it is classified using the machine learning techniques described in Section 3.4. These events are then gathered and compiled in a detailed report. The details of this process are explained in detail in Section 4.3.



(a) Original signal recorded on 2014-03-06 at 19:09:33



(b) Normalized statistics derived from the signal

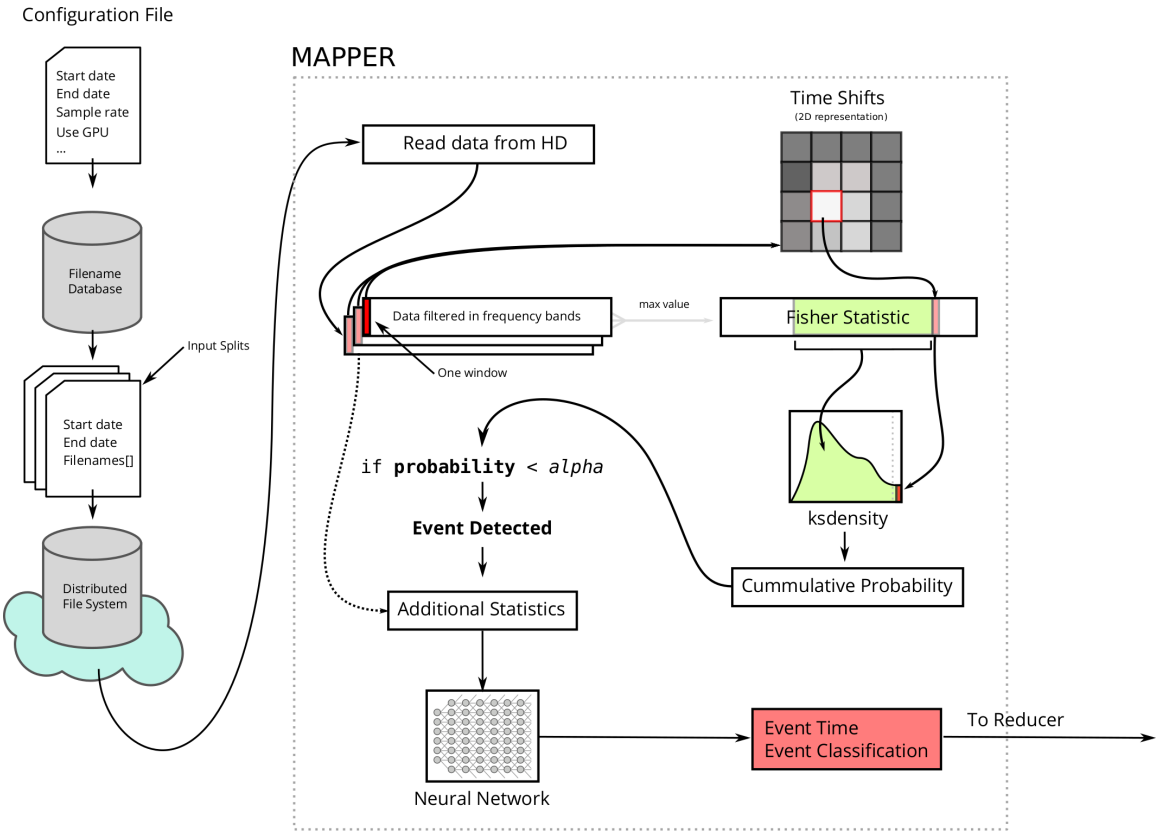


Figure 3.14: Diagram of steps to prepare input files and execution on every mapper instance.

Chapter 4

RESULTS

The results of this thesis are presented in three parts. The first part is focused on the results of performing event detection by applying the techniques described in Section 3.3.

The second part is focused on the training of neural networks and the evaluation of event classification, and the third part describes the results of running the project in a high performance distributed system.

Additionally, the operational parameters used throughout this work are listed in Appendix A.

4.1 Event Detection

To estimate the probability of a new Fisher statistic belonging to a given distribution, we calculate a kernel-smoothed probability function of the past Fisher statistic values, but these probabilities are affected by the selected kernel and its characteristics. For our application, we use a Gaussian kernel with standard deviation of 5 (see `KernelWidth` Appendix A).

When the density of a new Fisher statistic value is calculated from the non-parametric distribution, we expect the probability density function value and the probability of observing that value or greater to be low for signals that are part an

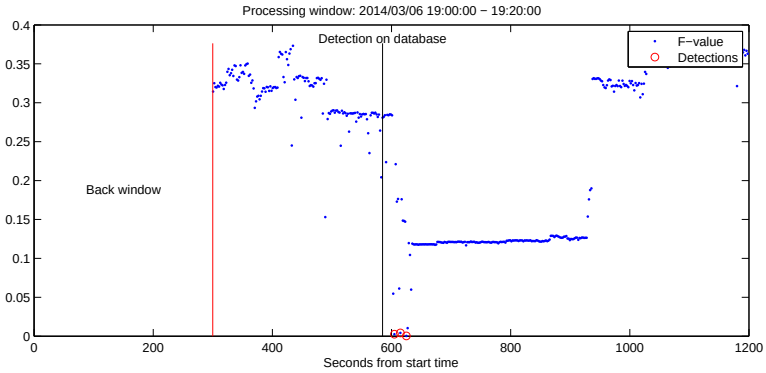


Figure 4.1: Validation of the detection technique. This event had previously been identified and stored in a database of known events

event. These densities are calculated for every signal filtered by a frequency band; therefore, before the detection is performed, there are 10 individual densities.

To detect events, the densities from each band are multiplied together and the result is compared with the threshold. For this reason, by multiplying very small numbers, the compound density is an even smaller number. The threshold for event detection (see `Alpha` in Appendix A) is set to a very small number that is dependent on the number the factors in this multiplication: in essence, the number of filtered bands on which the Fisher statistic is computed.

Changes to the value of the threshold control how *rare* a signal in time has to be in order to be considered an event. To prevent an overly strict detection procedure, the threshold is set to a value that will guarantee that there are more false positive detections than the opposite. It is expected that a large portion of these false positive events are accidental correlations of non-coherent sources and that they would be discarded in the classification stage.

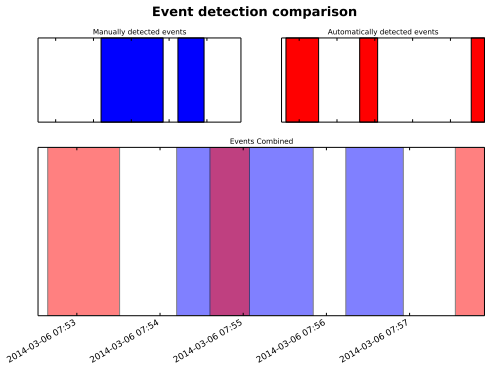


Figure 4.2: Automatic event detection compared to manual event detection.

4.1.1 GPU Parallelization

As described in Section 3.3.5, APARAPI was used to convert a Java program to OpenCL code that is then executed in the GPU. The program computes the Fisher statistic for processing windows (see `WindowSize` and `WindowOverlap` in Appendix A) that contain an input data matrix consisting of N rows and M columns, and a pre-calculated time shift matrix of dimensions $834 \times M$.

Figure 4.3 shows the results of GPU benchmarking by plotting the mean execution time of five observations and the 95% confidence interval in a lighter color. It is evident that as the input matrices grow in size, the execution time increases linearly for single thread CPUs. On the contrary, the GPU executes in sub-linear time.

Although OpenCL is a standard framework, vendors provide their own implementation in the form of a userspace library (called ICD, installable client driver) that communicates with the graphics driver. This, along with different vendors supporting different versions of OpenCL, requires the developer to modify the OpenCL program to work correctly on multiple devices, even in the case where this code is auto-generated as with APARAPI.

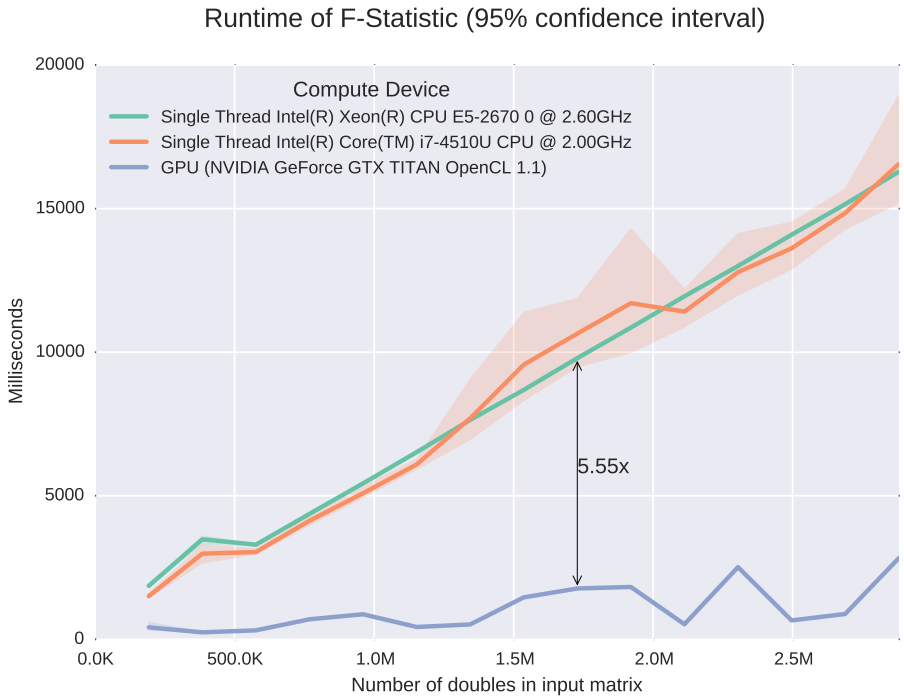


Figure 4.3: Evaluation of floating point processing performance on multiple devices.

Two versions of the Fisher Statistic calculation for GPU execution were developed: one targeting Intel devices and another targeting NVIDIA devices. The main differences between them are related to strategies for array indexing and memory management. There are known issues with local memory arrays and long-executing tasks in the NVIDIA platform.[1]

APARAPI provides an "explicit memory management" mode that allows for fine grained control over which data structures are copied to GPU memory. This mode can be used to guarantee that copy operations, which are very expensive, are made only when necessary.

Finally, in order to prevent crashes in the Java Virtual Machine when attempting to free resources, a patch had to be applied to an APARAPI JNI library file located

[1]See https://code.google.com/p/aparapi/issues/detail?id=145

in the APARAPI distribution folder.[2]

4.2 Event Classification

4.2.1 Training Dataset

To generate datasets for neural networks training, large amounts of data had to be loaded to memory. It was evident that there were inefficiencies in reading data for time intervals that were bigger than the number of samples stored in one single data file. When this happened, it was necessary to read additional files to acquire the remaining samples. Identifying which files contain the data for a given time range could, in the worst of the cases, lead to reading the metadata of all the data files.

As a consequence, an indexing system based on an Interval Tree was implemented. Interval trees, as the name indicates, store intervals instead of single values. In order to create the tree, a pivot interval is chosen and subsequently every interval whose start is greater than the pivot's midpoint, it is added to the right child list. If the end of the interval is smaller than the pivot's midpoint it is added to the left child list. If the interval is fully contained by the pivot, it is added as a "middle" child of the pivot (see Figure 4.4).

Search operations are performed similar to binary search, which in a balanced tree discards half the intervals. Similarly, like in all tree structures, it is important to pick a good pivot that evenly divides the input and balances the tree. Finding the optimal pivot takes $O(n \log n)$ time, and building the tree is also $O(n \log n)$.

Interval trees are used for search and to measure performance more accurately, an additional variable m that represents the total number of results is introduced. In

[2]The full path to the file is `aparapi/com.amd.aparapi.jni/src/cpp/runKernel/JNIContext.cpp` and the patch can be found in Apendix C

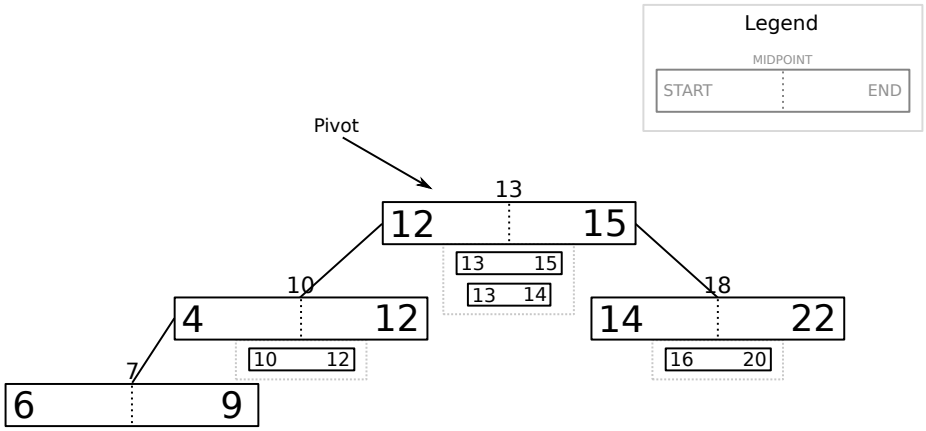


Figure 4.4: Resulting Interval Tree of intervals [4,12], [10,12], [11,12], [12,15], [13,14], [13,15], [14,22] and [16,20].

other words, if the search is expected to return all of the inputs, the algorithm would take $O(m)$; otherwise, the performance is $\Theta(\log n + m)$.

The worst case scenario is when the pivot interval is erroneously chosen and it contains all other intervals. In this scenario, the search is $O(n)$. For our application, we know that there are very few overlapping intervals of data, therefore this is not a concern.

This indexing system can be used to efficiently retrieve filenames that contain data for any given interval of time. This functionality is useful for extracting the samples that belong to an event from the database of events by knowing its start and end times.

Knowing which files to read, the input data for ANN training can be generated.

4.2.2 Artificial Neural Network Training

For this project, the inputs to the ANN were the statistics used in the event detection phase, along with frequency information. The details of the generation of the frequency information is discussed in Section 3.4.1.

The network is trained using the following variables:

Variable Name	**Description**
F	The Fisher Statistic
power1	Power in the frequency band between 1 Hz and 5 Hz
power2	Power in the frequency band between 5 Hz and 10 Hz
power3	Power in the frequency band between 10 Hz and 15 Hz
backazimuth	Angle relative to north for the calculated source of the signal

Table 4.1: Variables used for Artificial Neural Network training and evaluation

These variables are normalized before they are input to the network. Outlier values beyond the 90^{th} percentile are discarded and the remainder are normalized to the range $[-1, 1]$. See Equation 4.1.

$$\text{normalize(data,point)} = \left(\frac{\text{point} - \textsc{Min}(\text{data})}{\textsc{Max}(\text{data}) - \textsc{Min}(\text{data})} \times 2 \right) - 1 \tag{4.1}$$

The training data—that is, pairs of inputs and ideal outputs— were generated using an extensive event database developed over three avalanche periods using different infrasound arrays placed along Highway 21. This dataset contains information about different types events, such as vehicles, airplanes, and avalanches. Table 4.2 contains a summary of the events in the database.

Every event was divided into multiple samples, some of which were subsequently used to train the neural network. To avoid over-training, 30% of these samples with at least 50% of them belonging to avalanche events were retained from the training set. In total, the training set contained 1308 individual samples generated from 113 labeled events.

Networks with the same architecture were trained with one of two ideal outputs: *a*) a function outputs 1.0 for any sample that belonged to an event labeled as an

Algorithm 5 Hybrid training strategy useful for training large artificial neural networks.

```
function BackpropagationTrain(network,epochs)
    ...                                   ▷ Train network using backpropagation
end function

function SimulatedAnnealingTrain(network,epochs)
    ...                                   ▷ Train network using simulated annealing
end function
prev_error ← ∞
while training do
    error ← evaluate(network)
    if prev_error - error < 0.00001 then
        SimulatedAnnealingTrain(network,5)
    end if
    prev_error ← error
    BackpropagationTrain(network,5)
end while
```

avalanche and 0 otherwise and *b*) the ideal function was set to 0 for all non-avalanche events. For avalanches, the output was a normalized Gaussian function (with a maximum of 1.0) centered at the middle of the length of the event. The standard deviation was half of the length of the event.

The event start times in the manually classified database were not set to the exact time when the event started, but rather a few seconds before the event began. Penalizing the neural network for miss-classifying those samples increased the training time, a problem solved by using the Gaussian approach.

Training the neural networks was performed using a hybrid strategy. The network is trained using backpropagation (a gradient descent method optimization technique for ANNs) until the percentage of improvement in the network output from one epoch compared to the previous falls below 0.001%. Then, the network is further trained using Simulated Annealing for 5 epochs. Afterward, the network is trained again using

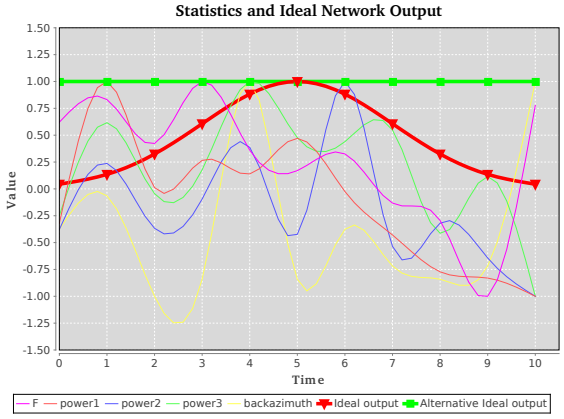
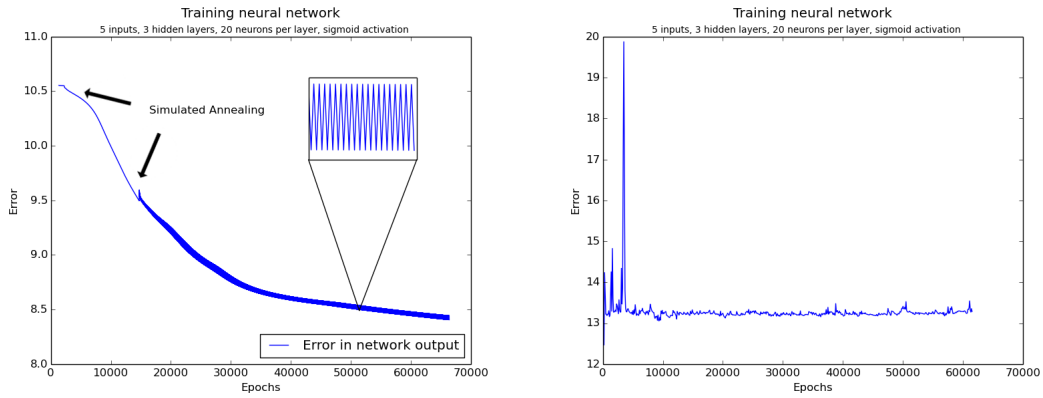


Figure 4.5: Signal statistics and Artificial Neural Network ideal output. This is the result of converting an event to multiple training samples.

Event classification	**Number of events**
Vehicle	7
Unknown	21
Plane	25
Avalanche	24
Explosive	9
MHAFB	14
Rotary	6
Earthquake	2
Helicopter	5
Total	113

Table 4.2: Summary of events classifications in the event database



(a) Neural Network training with a hybrid strategy

(b) Neural Network training using only backpropagation, stuck in a local minimum

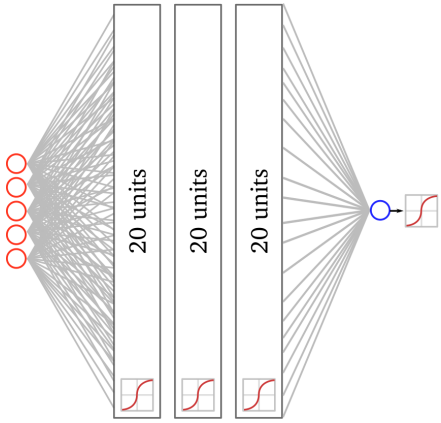
Figure 4.6: Differences in training and error using different strategies

backpropagation for 5 epochs (see Algorithm 5). The results of using this technique compared to a training strategy using only backpropagation are presented in Figure 4.6.

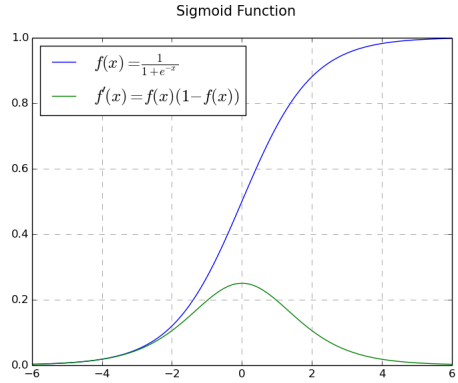
The architecture of the neural network consisted of five inputs (listed in Table 4.1), three layers of 20 units each, and one output. The activation function for the hidden units as well as the output neuron was a sigmoid function (a squashing function). The sigmoid function is one of the most commonly used because its derivative is easily computable, which facilitates training (see Figure 4.7).

A satisfactory classification accuracy was achieved after approximately 70,000 epochs of training.

Recurrent artificial neural network (RNNs) were trained alongside the feed-forward ANNs but results of training were never satisfactory. While recurrent ANNs have been demonstrated to be good at modeling repetitive events because of their stored internal state, no two avalanches are exactly the same. Additionally, the length of events is always different, meaning that any cyclical features would be obscured,



(a) ANN architecture used for event classification



(b) Sigmoid activation function

Figure 4.7: Artificial Neural Network used for classification and activation function

making training more difficult. The final reason why the training was not successful is related to the way the samples are arranged: every training example represented a sample from an event, which could be of any class. Therefore, one avalanche event could be preceded by a vehicle event.

In conclusion, the recurrent neural network is not able to train for a repetitive pattern that might not be always present and the events being given to the network back to back obscures any pattern that might have been present.

4.2.3 Performance Metrics

The trained classification neural network outputs a value closer to 1.0 when the input sample belongs to an event labeled as an avalanche. Since events are either avalanches or not avalanches, a threshold of 0.5 (see `NeuralNetworkAvalancheTreshold` in Appendix A) was used to produce a boolean output of true or false, true being all values greater than 0.5 and false otherwise. The network is evaluated on the retained events and figures are provided with the performance over the training set.

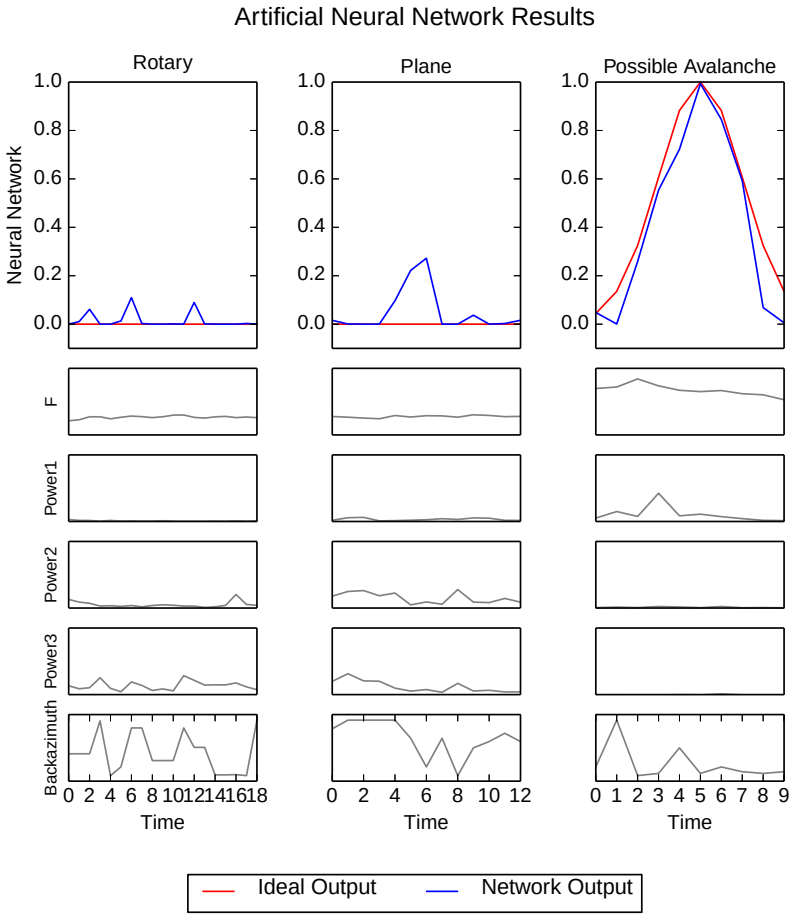


Figure 4.8: Neural Network output for sample events

Neural network performance is usually evaluated using a "confusion matrix," in which the results of the classification are categorized as *actual positives* and *actual negatives* meaning the true class for a given sample, and *predicted negative* and *predicted positive* being the output of the network for a given example. The cells in the first row of the table are labeled a and b, and the remaining two in the second row c and d.

	Actual Positive	**Actual Negative**
Predicted Positive	21	0
Predicted Negative	3	89

Table 4.3: Confusion Matrix

With this information, we can compute useful statistics about the performance. See Table 4.4.

Metric Name	**Formula**	**Value**
Correct Classification Rate	$\frac{(a+d)}{N}$	0.97
Positive Predictive Power	$\frac{a}{(a+b)}$	1
Negative Predictive Power	$\frac{d}{(c+d)}$	0.96
False Negative Rate	$\frac{c}{(a+c)}$	0.14
False Positive Rate	$\frac{b}{(b+d)}$	0

Table 4.4: Performance Metrics for Artificial Neural Network

In Figure 4.9, every training sample is plotted along with the the ideal function and the network output.

The network was also used to classify every event in two 24-hour windows for which some events had been manually classified. Unlike the training and evaluation, events were extracted directly from the dataset using the algorithms described in Section 3.3. These events were then classified using the feed-forward neural network. While the network was trained to output values closer to 1 when the event was an avalanche, there were many false positives in the form of spikes for only one of the samples that conform an event.

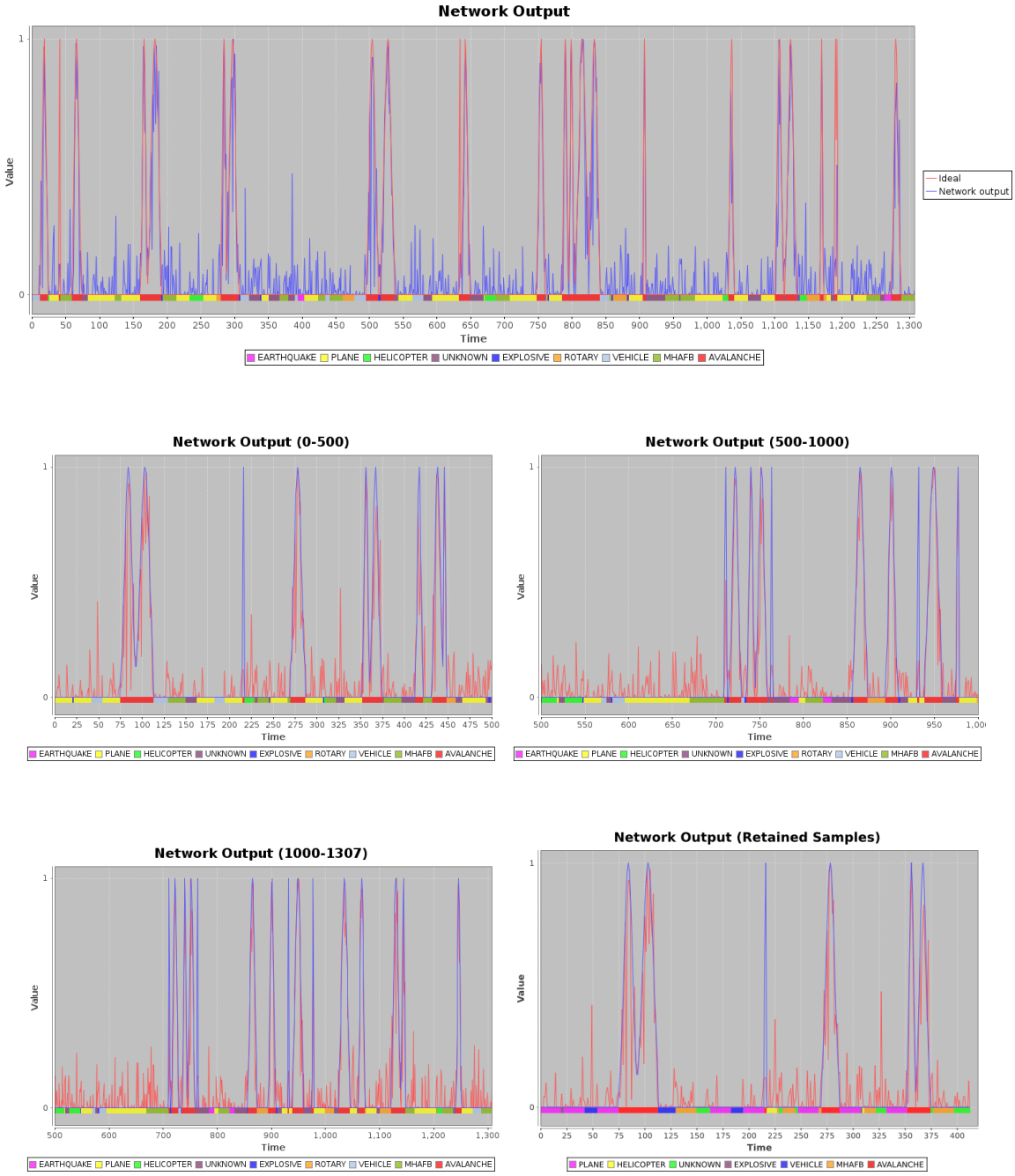


Figure 4.9: Classification results for all training samples. Colored bars represent the class of each sample

To filter out false positives, the network output was only considered positive if it remained above the parameter `NeuralNetworkAvalancheTreshold` for at least 50% of the event samples. In other words, if the ANN output remained above the threshold value for more than half the number of samples for the event, it would be considered a positive classification. On the other hand, if the network output was above the threshold less than half the time, the event would not be classified as an avalanche.

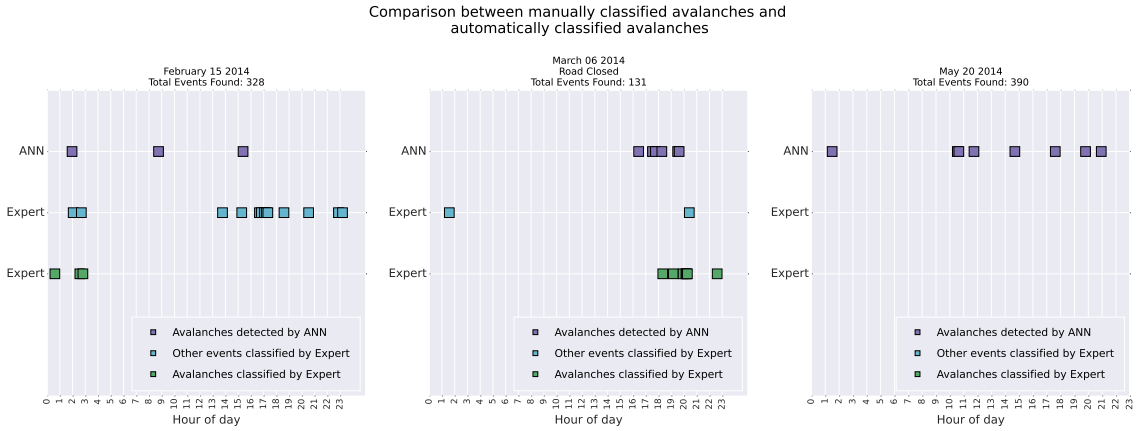


Figure 4.10: Comparison of event classification

Figure 4.10 shows the results of performing event detection and classification on full days, two of which had been manually analyzed and labeled by an expert. The neural network classifies events as avalanches that coincide with manual picks, and some false positives can also be observed.

The third day, May 20$^{\text{th}}$, shows the ANN identifying avalanches where there were none. One potential reason for this misclassification is related to vehicle flow: on both February 15$^{\text{th}}$ and May 20$^{\text{th}}$, the road was open and vehicles were producing signals. On March 6$^{\text{th}}$, the highway was closed, and accordingly, there were a smaller number of events detected that day.

One of the severely under-represented event classes in the ANN training database are vehicles. There are thousands of cars that travel through Highway 21 every winter, and

those signals are surely recorded in the infrasound dataset. Yet, in the training data, there are only 7. It is not surprising then that the ANN produces false positives in days without avalanches for times where vehicles are traveling. Having a more complete dataset of labeled events would immediately lead to better classification accuracy.

An alternative to mitigate this problem is discussed in Section 4.7.2.

4.3 Distributed Processing

In the steps required to detect and classify events, some of the techniques can be controlled by parameters. These parameters are configurable and are listed in Appendix A with a name, description, scope (what part of the processing steps utilizes it), and default value with unit (the value for the parameter used in this project).

Boise State University owns a high-performance cluster named Kestrel [5]. It is composed of 32 nodes that can be utilized to compute tasks. These tasks are submitted to a *master node* that allocates resources on the nodes and and then executes the task on each of them. The process of allocating and distributing tasks is managed using the PBSPro[3] software package.

CPU	2 Intel Xeon E5-2600 series processors 16 cores/32 threads
GPU	2 Tesla K20 (nodes 1-22)
RAM	2 TB
Local Storage	400GB
Storage	64 TB Panasas Parallel File Storage
Networking	Mellanox ConnectX-3FDR Infiniband interconnect

Table 4.5: Kestrel node configuration

[3]Information about this software along with guides and manuals can be obtained by visiting `http://www.pbsworks.com/SupportGT.aspx?d=PBS-Professional,-Documentation`

In order to use Hadoop in a cluster setting like Kestrel, the distributed file system used by Hadoop needs to be configured at runtime, along with the resource manager. In Hadoop 2.6, HDFS is used for data storage, and the default resource manager is Yarn [2].

When submitting a new task using PBSPro, one of the nodes will run the NameNode (centerpiece of an HDFS file system) and YARN, while all the other allocated nodes will start the DataNode (local HDFS replica) along with TaskTrackers and JobTrackers (the components responsible of keeping track of progress).

Scripts were developed based on the work of Ravi Preesha Geetha [43] to automate the starting and stopping of NameNodes and DataNodes and new scripts were developed to manage YARN. These scripts and the accompanying configuration files are provided in Appendix B.

Finally, in order to execute code in the GPU using Hadoop, the APARAPI libraries have to be compiled targeting the same JVM where Hadoop jobs are being executed and the native APARAPI libraries need to be within Java's `java.library.path`.

4.3.1 Hadoop Algorithm

In this project, we are generating very small files, called input files, that contain in a single line the information about a time interval and which files should be opened to retrieve data for it. For example, to process a day-worth of data, 24 input files are created containing information about the start and end of the interval (one hour per file) and a list of files where the data for that interval is located.

The input files are loaded onto HDFS, where they are processed by a Hadoop. The length of the interval for which a new input file will be generated can be configured at runtime by modifying the property `ChunkProcessingTime` (See Appendix A).

The full algorithm that generates the input files is described in Algorithm 6.

Algorithm 6 Algorithm for generating input files for Hadoop.

```
function GenerateInputFiles(configuration)
    start,chunk_start,chunk_end←configuration.get("start date")
    end←configuration.get("end date")
    chunk_size←configuration.get("chunk size")
    while chunk_end ¡ end do
        chunk_end ← chunk_start + chunk_size
        filenames←getFilenamesForInterval(chunk_start,chunk_end))
        writeToHDFS(chunk_start, chunk_end, filenames)
        chunk_start ← chunk_start + chunk_size
    end while
end function
```

4.3.2 Optimizing Task Size

Hadoop will spawn a new mapper or many mappers to handle every input file (InputSplit). the input splits are small files that contain information about the interval of time that is being processed, and the filenames that contain the data for said interval. We call each of these intervals a "chunk" of data that needs to be processed.

The amount of data per one mapper affects its running time, but running tests for different chunk sizes didn't show great differences between small chunks and big chunks, as demonstrated by Figure 4.11.

The ultimate conclusion from this observation is that there is not a significant penalty from moving data from the hard drive to memory and from memory back and forth from the GPU. The most likely reason is due to the computationally-heavy operations that take many orders of magnitude more time than moving data between storage forms.

4.3.3 Benchmark

The program was executed in a variety of ways to measure its performance. In Figure 4.12, the running time (a measure of performance) is compared between a single node in Kestrel running Java code sequentially on the CPU, one node using the GPU along the CPU and

10 nodes using both GPU and CPU.

The results show a 3.72 speedup between the CPU-only implementation and the version optimized for running on the GPU. This speedup is less dramatic that what could be expected on the basis of GPU parallelism (see Figure 4.3), but can be explained by analyzing the total amount of time a task runs in the GPU vs CPU.

In Figure 4.13, the different stages of the event detection and their running time mechanism are examined. As expected, there is a similar speedup in the stage where processing can be offloaded to the GPU, but the other stages remain the same. As the data being processed grows, more time is spent on CPU-only tasks, decreasing the speedup.

In conclusion, we demonstrate a ∼4x speedup over single threaded event detection and classification that can be linearly improved for every node added. The benchmark in Figure 4.12 shows a ∼37x speedup by using 10 nodes and taking advantage of the heterogeneous compute capabilities in those nodes.

4.4 Conclusion

To conclude, we provide a summary of the findings of this project and potential areas of future research in the field of massive-scale data processing.

4.5 Summary

We started by presenting a background on the challenges and limitations of modeling naturally occurring phenomena. There are solutions to these problems, but at the cost of making assumptions and oversimplifications.

In Section 2, we described what avalanches are, the risk they pose to infrastructure, and why it is important for the state of Idaho to have effective systems to detect when avalanches are occurring.

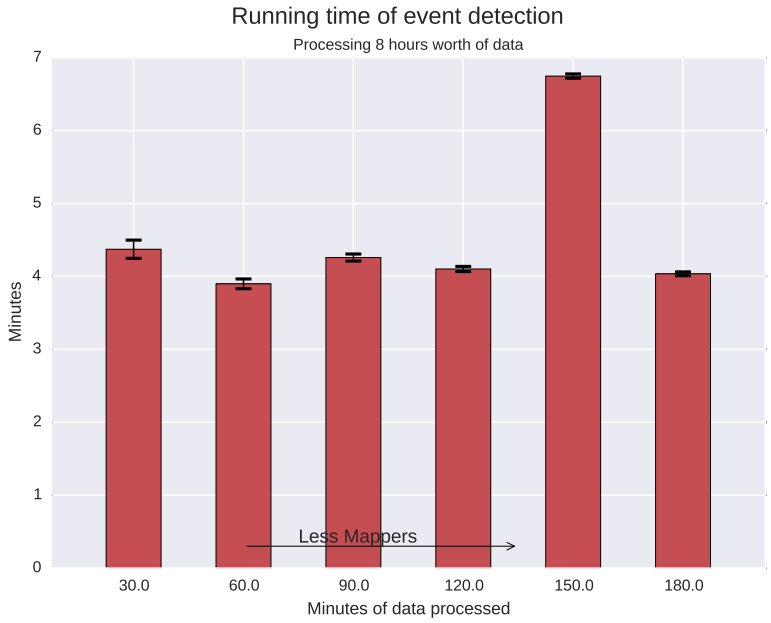


Figure 4.11: Running time of different chunk sizes on 1 node, processing 8 hours of data.

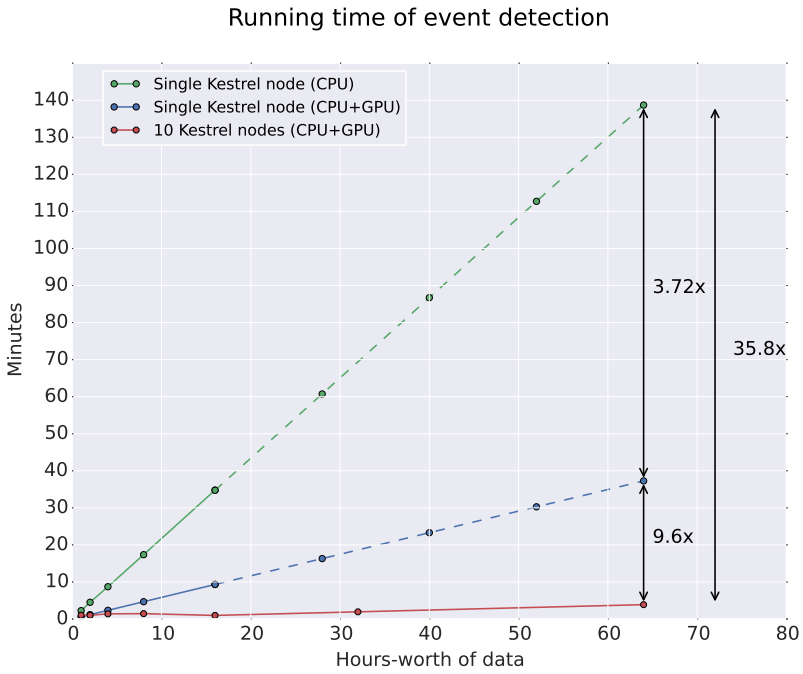


Figure 4.12: Event detection performed using different compute capabilities. Dashed lines represent expected running times.

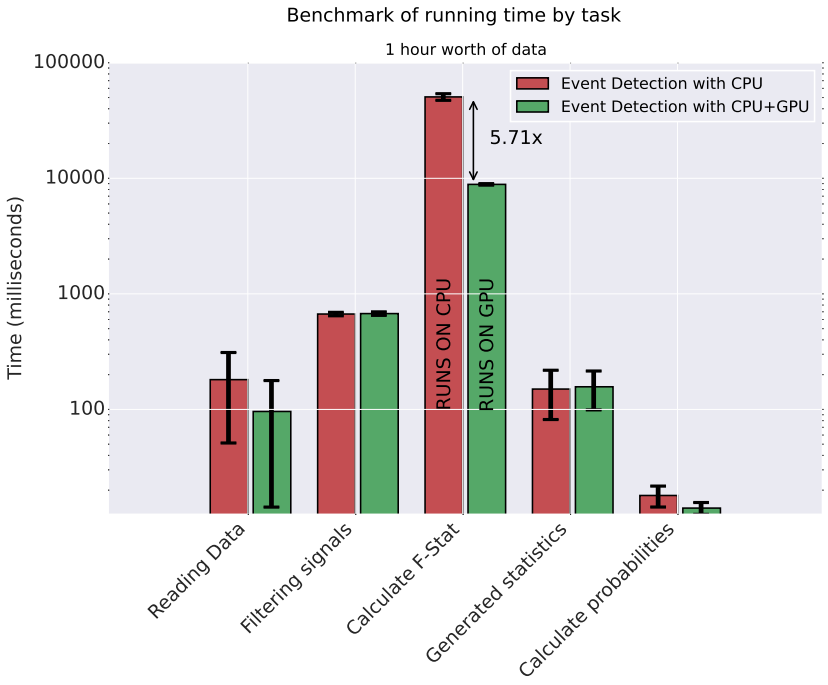


Figure 4.13: Running time of different stages of processing. Y axis is represented in a logarithmic scale

Boise State University has a study site located in an avalanche-prone area, where data has been collected for a number of years, yet, due to the large volume of this data, it has not been studied thoroughly. Scenarios like this are increasingly more common, due to the rise of "big data."

When trying to process large amounts of data, one of the most powerful and useful techniques is parallelization. This is especially true if the problem can be simplified to a series of smaller problems that can be solved in parallel. We covered two ways to parallelize: one that is done in the same machine by utilizing existing resources like CPU cores or GPU units, and distributed processing, where computers are connected to each other through a network.

In Section 3, we described the algorithms that are used to process signals recorded by an array of sensors. We discussed different approaches to reducing noise, finding correlation in signals, and detecting when something of importance has happened. After detecting that an event has occurred, we proceeded to describe techniques for classifying these events, in other words, to label them.

Finally, in Section 4, the results of this work are presented, along with differences specific to this work in the implementation of standard techniques.

4.6 Results and Implications

This work shows that processing massive-scale geophysical datasets is feasible and that the results can be used to improve existing models, especially observation-based models that would benefit from as much processed data as possible. Unless the vast amounts of unexplored data in historical datasets is analyzed, these models cannot be thoroughly validated or improved.

In this work, we propose a system for processing large amounts of data by taking advantage of multiple sources of parallelism. Signal processing techniques for event detection

were optimized for performance and a novel machine learning approach is applied to event detection, something that has not been historically used for avalanche detection.

Although the speedup for event detection is very attractive in terms of the analysis that can be performed on the data—whether it contains avalanches or not—event classification is not robust enough yet for real-world applications without additional training on a larger dataset of labeled events.

In conclusion, earth sciences can benefit greatly from high-performance data processing. This work enables scientists to manipulate and understand large datasets many orders of magnitude faster than what was possible before. Additionally, the algorithms and techniques used are not limited to event detection and classification, but can be easily modified to solve other geophysical problems that are computationally expensive.

4.7 Future Work

4.7.1 Fisher Statistic Computation

In the process of detecting events, a list of values of the Fisher statistics is stored in order to compute the probability distribution of the back window. This list is relatively expensive to create and is likely to contain events as well.

Currently, every Map operation in Hadoop recreates this list from the original data every single time, even if some other Mapper has already calculated it. Unfortunately, to achieve true parallelism, no task should require data generated by any other task, which means that there is no easy solution to this problem.

In the future, an alternative approach could be to retain the back window data and generate a database of Fisher statistics for all signals that can be re-analyzed using different techniques.

4.7.2 Artificial Neural Network Ensembles

Event classification on full days, as opposed to manually labeled intervals of time, was not as accurate as expected. In training artificial neural networks, there are multiple factors that affect the accuracy, but the most important is the training data. With a limited dataset of events, the training is limited and the networks fail to generalize and produce correct outputs for unseen events.

ANN Ensembles are groups of ANNs that are trained to classify different classes. For example, one network is trained to detect avalanches while another network is trained to detect vehicles. When an event needs to be classified, both networks produce an output, which is then weighted, and a final classification is produced. ANN ensembles would provide a more robust classification of events by specializing ANNs to detect one kind of event, which would help to reduce the number of false positive avalanche detections.

4.7.3 Optimization to Detection Algorithm

The non-parametric approach described in this work was evaluated against manually identified events. Yet these events were selected from a very small subset of all the available data. While the technique is statistically sound, the probability threshold selected could very well be inadequate for other types of events or for events that occur over long periods of time.

Ultimately, all approaches have to make a compromise between detecting too many false events or missing important events that should have been detected. For this reason, having a database of Fisher statistics would help develop even more robust detection algorithms in the future.

4.7.4 GPU Optimizations

Developing software for GPUs requires an understanding of how the memory architecture is defined in OpenCL. There are multiple optimizations that can be applied to OpenCL Kernels to take advantage of a specific compute device. For example, local memory can be used as a scratch pad that is orders of magnitude faster than global GPU memory, yet it requires careful management.

In future implementations of this software, GPU optimizations could lead to even better performance for embarrassingly parallel floating point operations.

Bibliography

[1] General-purpose computation on graphics hardware. http://gpgpu.org/about. Accessed: 2014-11-24.

[2] Hadoop - Apache Hadoop 2.6.0. `https://hadoop.apache.org/docs/r2.6.0/`. Accessed: 2015-05-14.

[3] Idaho Transportation Department - Highway Info. `http://lb.511.idaho.gov/idlb/mountainpasses/mountainpass.jsf;jsessionid=1548FECCC0E6331FDE868372D2454481?id=7&view=state&text=m&textOnly=false`. Accessed: 2014-11-14.

[4] Idaho Transportation Department - Milepoint Log- State Highway System. `http://itd.idaho.gov/highways/milepointlog/logs/stateHW/SH_21_MPLog.pdf`. Accessed: 2014-11-17.

[5] The Kestrel CPU/GPU cluster — Boise State University Wiki. `http://wiki.boisestate.edu/wiki/the-kestral-cpugpu-cluster/`. Accessed: 2015-05-14.

[6] M. Arattano. On the Use of Seismic Detectors as Monitoring and Warning Systems for Debris Flows. *Natural Hazards*, 2–3(20):197–213, 1999.

[7] Stephen J Arrowsmith, Rod Whitaker, Steven R Taylor, Relu Burlacu, Brian Stump, Michael Hedlin, George Randall, Chris Hayward, and Doug ReVelle. Regional monitoring of infrasound events using multiple arrays: application to utah and washington state. *Geophysical Journal International*, 175(1):291–300, 2008.

[8] Ian C Atkinson, Geng Liu, Nady Obeid, Keith R Thulborn, and Wen-mei Hwu. Rapid computation of sodium bioscales using GPU-accelerated image reconstruction. *International Journal of Imaging Systems and Technology*, 23(1):29–35, 2013.

[9] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David A Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, 1993.

[10] Bessason Bjarni, Eirksson Gsli, Thrarisson Oinn, Thrarisson Andrs, and Einarsson Sigurur. Automatic detection of avalanches and debris flows by seismic methods. *Journal of Glaciology*, 53(182), 2007.

[11] R. Blandford. An automatic event detector at the tonto forest seismic observatory. *GEOPHYSICS*, 39(5):633–643, 1974.

[12] Dhruba Borthakur. Hdfs architecture guide. *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*, 2008.

[13] J. Roger Bowman, G. Eli Baker, and Manochehr Bahavar. Ambient infrasound noise. *Geophysical Research Letters*, 32(9):n/a–n/a, 2005.

[14] Stephen Butterworth. On the theory of filter amplifiers. *Wireless Engineer*, 7(6):536–541, 1930.

[15] Luonan Chen and Kazuyuki Aihara. Chaotic simulated annealing by a neural network model with transient chaos. *Neural networks*, 8(6):915–930, 1995.

[16] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[17] F. Darema, D.A. George, V.A. Norton, and G.F. Pfister. A single-program-multiple-data computational model for EPEX/FORTRAN. *Parallel Computing*, 7(1):11–24, 1988.

[18] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[19] S. E. Fahlman. The recurrent cascade-correlation architecture. *Advances in Neural Information Processing Systems*, (3):190–196, 1991.

[20] Michael Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972.

[21] Ian Foster. *Designing and building parallel programs*. Addison Wesley Publishing Company, 1995.

[22] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a new computing infrastructure*. Elsevier, 2003.

[23] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[24] James Greg and John O'Rorke. Real-time glow. `http://www.gamasutra.com/view/feature/2107/realtime_glow.php`, 2004. Accessed: 2014-11-31.

[25] Max Grossman, Mauricio Breternitz, and Vivek Sarkar. HadoopCL: MapReduce on distributed heterogeneous platforms through seamless integration of Hadoop and OpenCL. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 1918–1927, Washington, DC, USA, 2013. IEEE Computer Society.

[26] Daniel Grosu, Anthony T Chronopoulos, and Ming-Ying Leung. Cooperative load balancing in distributed systems. *Concurrency and Computation: Practice and Experience*, 20(16):1953–1976, 2008.

[27] Chris Harris, Karen Haines, and Lister Staveley-Smith. GPU accelerated radio astronomy signal convolution. *Experimental Astronomy*, 22(1-2):129–141, 2008.

[28] Scott Havens, Hans-Peter Marshall, Jeffrey B. Johnson, and Bill Nicholson. Calculating the velocity of a fast-moving snow avalanche using an infrasound array. *Geophysical Research Letters*, 41(17):6191–6198, 2014.

[29] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural Networks, 1989. IJCNN., International Joint Conference on*, pages 593–605. IEEE, 1989.

[30] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, December 1986.

[31] Robert Jacques, Russell Taylor, John Wong, and Todd McNutt. Towards real-time radiation therapy: GPU accelerated superposition/convolution. *Computer methods and programs in biomedicine*, 98(3):285–292, 2010.

[32] Ruoming Jin, Ge Yang, and G. Agrawal. Shared memory parallelization of data mining algorithms: techniques, programming interface, and performance. *Knowledge and Data Engineering, IEEE Transactions on*, 17(1):71–89, Jan 2005.

[33] William J. Kaufmann and Larry L. Smarr. *Supercomputing and the Transformation of Science*. W. H. Freeman & Co., New York, NY, USA, 1992.

[34] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2 edition, 2010.

[35] A. Kogelnig, E. Suriñach, I. Vilajosana, J. Hübl, B. Sovilla, M. Hiller, and F. Dufour. On the complementariness of infrasound and seismic sensors for monitoring snow avalanches. *Natural Hazards and Earth System Science*, 11(8):2355–2370, 2011.

[36] Arnold Kogelnig, Giacomo Ulivieri, Emanule Marchetti, and Samuel Wyssen. Infrasound detection of avalanches, a new approach on managing avalanche risks. In *International Snow Science Workshop Grenoble Chamonix Mont-Blanc*, 2013.

[37] Bart Kosko. *Neural Networks and Fuzzy Systems: A Dynamical Systems Approach to Machine Intelligence/Book and Disk*, volume 1. Prentice hall, 1992.

[38] Hans-Peter Marshall and Gary Koh. FMCW radars for snow research. *Cold Regions Science and Technology*, 52(2):118–131, April 2008.

[39] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[40] Lorenz Meier and Daniel Lussi. Remote detection of snow avalanches in Switzerland using infrasound, doppler radars and geophones. In *Proceedings of International Snow Science Workshop*, 2010.

[41] Kenneth Moreland and Edward Angel. The FFT on a GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119. Eurographics Association, 2003.

[42] K. Naugolnykh and A. Bedard. A model of the avalanche infrasonic radiation. In *Geoscience and Remote Sensing Symposium, 2002. IGARSS '02. 2002 IEEE International*, volume 2, pages 871–872 vol.2, June 2002.

[43] Ravi Preesha Geetha. Accelerated radar signal processing in large geophysical datasets, 2015.

[44] Sebastian Rost and Christine Thomas. Array seismology: Methods and applications. *Reviews of Geophysics*, 40(3):2–1–2–27, 2002. 1008.

[45] Kazuyuki Saito, Satoru Yamaguchi, Hiroki Iwata, Yoshinobu Harazono, Kenji Kosugi, Michael Lehning, and Martha Shulski. Climatic physical snowpack properties for large-scale modeling examined by observations and a physical model. *Polar Science*, 6(1):79–95, 2012. Special Issue: The Second International Symposium on the Arctic Research (ISAR - 2).

[46] Jürg Schweizer, J. Bruce Jamieson, and Martin Schneebeli. Snow avalanche formation. *Reviews of Geophysics*, 41(4):n/a–n/a, 2003.

[47] Ernest D. Scott, Christopher T. Hayward, Robert F. Kubichek, Jerry C. Hamann, John W. Pierre, Bob Comey, and Tim Mendenhall. Single and multiple sensor identification of avalanche-generated infrasound. *Cold Regions Science and Technology*, 47(1–2):159–170, 2007. A Selection of papers presented at the International Snow Science Workshop, Jackson Hole, Wyoming, September 19-24, 2004.

[48] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), March 2005.

[49] Paul P. Tallon. Understanding the dynamics of information management costs. *Commun. ACM*, 53(5):121–125, May 2010.

[50] G. Ulivieri, E. Marchetti, M. Ripepe, I. Chiambretti, G. De Rosa, and V. Segor. Monitoring snow avalanches in Northwestern Italian Alps using an infrasound array. *Cold Regions Science and Technology*, 69(2–3):177–183, 2011. International Snow Science Workshop 2010 Lake Tahoe.

[51] Alec Van Herwijnen and Jrg Schweizer. Seismic sensor array for monitoring an avalanche start zone: design, deployment and preliminary results. *Journal of Glaciology*, 52(202), 2011.

[52] Harry L. Van Trees. *Detection, estimation, and modulation theory. Part IV. , Optimum array processing.* Wiley-Interscience, New York, 2002.

[53] John von Neumann. First draft of a report on the edvac. *IEEE Ann. Hist. Comput.*, 15(4):27–75, October 1993.

[54] Tom White. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.

[55] Jamie Yount, Adam Naisbitt, and Ernie Scott. Operational highway avalanche forecasting using the infrasonic avalanche detection system. In *Proceedings of Whistler International Snow Science Workshop*, 2008.

[56] A. Zischg, S. Fuchs, M. Keiler, and J. Stötter. Temporal variability of damage potential on roads as a conceptual contribution towards a short-term avalanche risk simulation. *Natural Hazards and Earth System Science*, 5(2):235–242, 2005.

Appendix A

OPERATIONAL PARAMETERS

Name	Description	Scope	Default Value
`SamplingRate`	The sampling rate of the signals retrieved from the DAQ	Event Detection	100 Hertz
`MasterChannel`	Index of the channel used as a reference of correlation	Event Detection	0
`ChannelsToUse`	Names of the channels used for all operations	Event Detection	HHN, HLZ, HLN, HLE, HHZ
`Channels`	The positions of each sensor	Event Detection	GPS coordinates, see Figure 3.6
`FilterFreqLow`	Frequency for the first frequency	Event Detection	2.0 Hertz
`FilterFreqIncrement`	Whidth of each frequency band	Event Detection	2.0 Hertz
`FilterFreqHigh`	Upper frequency for the last frequency band	Event Detection	20.0 Hertz
`WindowSize`	The size of each processing window	Event Detection, Event Classification	6 Seconds
`WindowOverlap`	How much each processing window will overlap	Event Detection, Event Classification	3 Seconds
`WindowBack`	Number of processing windows used to create the *back window*	Event Detection, Event Classification	900 Seconds
`KernelWidth`	Standard deviation used for ksdensity	Event Detection, Event Classification	5.0
`Alpha`	Threshold value for event detection (compound probability)	Event Detection, Event Classification	1^{-9}
`TimeThreshold`	Amount of time used that determines when event is created as opposed to merged with a previous event	Event Detection	27 seconds
`MaxEventLength`	The absolute maximum time an event can last	Event Detection	60 seconds
`PowerBands`	A list of power bands calculated for the data	Event Detection, Event Classification	1-5Hz, 5-10Hz, 10-15Hz, 15-10Hz, 20-50Hz
`NeuralNetworkAvalancheTreshold`	Threshold at which the network output is considered positive	Event Classification	0.5

`IdealChannelGaussian`	Whether or not generate training samples with a Gaussian function as the ideal output versus the value 1.0	Event Classification	True
`ChunkProcessingTime`	How many seconds will be loaded generated for every input file. As a consequence, this property also controls the number of mappers that will be utilized for processing.	Distributed Processing	1800
`FileLookupTable`	Path to the filename lookup table that resolves times to filenames	Distributed Processing	Enviroment-dependent
`FilenameReplacements`	Any replacements that need to be made to the filenames returned by *FilenameDatabasePath*	Distributed Processing	Comma-separated find and replace
`ClassificationNeuralNetPath`	Path to the neural network used for classification	Distributed Processing	Enviroment-dependent
`OutputPath`	HDFS path where the events are going to be stored	Distributed Processing	./EventDetectionClassification
`ProcessingStartDate`	Start date for distributed processing	Distributed Processing	Date
`ProcessingEndDate`	End date for distributed processing	Distributed Processing	Date

Appendix B

SCRIPTS FOR HADOOP CONFIGURATION ON KESTREL

B.1 Configuration Files

B.1.1 yarn-site.xml

```
      <?xml version="1.0"?>
<!--
  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

      http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License. See accompanying LICENSE file.
-->
<configuration>
       <property>
              <name>yarn.resourcemanager.hostname</name>
<value>node-04</value>
       </property>
       <property>
              <name>yarn.resourcemanager.address</name>
<value>node-04:58010</value>
       </property>
       <property>
              <name>yarn.resourcemanager.scheduler.address</name>
<value>node-04:58020</value>
       </property>
       <property>
              <name>yarn.resourcemanager.resource-tracker.address</name>
<value>node-04:58030</value>
       </property>
       <property>
              <name>yarn.resourcemanager.admin.address</name>
<value>node-04:58040</value>
       </property>
       <property>
              <name>yarn.resourcemanager.webapp.address</name>
<value>node-04:58050</value>
       </property>
       <property>
              <name>yarn.nodemanager.webapp.address</name>
<value>0.0.0.0:58060</value>
       </property>
       <property>
              <name>yarn.nodemanager.localizer.address</name>
<value>0.0.0.0:58070</value>
       </property>
       <property>
              <name>yarn.nodemanager.vmem-check-enabled</name>
              <value>false</value>
```

```
        </property>
        <property>
                <name>yarn.nodemanager.aux-services</name>
                <value>mapreduce_shuffle</value>
                <description>shuffle service that needs to be set for Map Reduce to run </description>
        </property>

</configuration>
```

B.1.2 mapred-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!--
  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License. See accompanying LICENSE file.
-->

<!-- Put site-specific property overrides in this file. -->

<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
    <property>
        <name>mapreduce.map.memory.mb</name>
        <value>2048</value>
    </property>
    <property>
        <name>mapreduce.reduce.memory.mb</name>
        <value>2048</value>
    </property>
        <property>
                <name>mapreduce.jobhistory.address</name>
                <value>0.0.0.0:58060</value>
        </property>
        <property>
                <name>mapreduce.jobhistory.webapp.address</name>
                <value>0.0.0.0:58070</value>
        </property>

</configuration>
```

B.1.3 mapred-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!--
  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License. See accompanying LICENSE file.
-->

<!-- Put site-specific property overrides in this file. -->

<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
```

```
    <property>
        <name>mapreduce.map.memory.mb</name>
        <value>2048</value>
    </property>
    <property>
        <name>mapreduce.reduce.memory.mb</name>
        <value>2048</value>
    </property>
        <property>
                <name>mapreduce.jobhistory.address</name>
                <value>0.0.0.0:58060</value>
        </property>
        <property>
                <name>mapreduce.jobhistory.webapp.address</name>
                <value>0.0.0.0:58070</value>
        </property>

</configuration>
```

B.1.4 core-site.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!--
  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License. See accompanying LICENSE file.
-->

<!-- Put site-specific property overrides in this file. -->

<configuration>
  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://0.0.0.0:9999</value>
  </property>
</configuration>
```

B.2 Scripts

B.2.1 cluster-pickports.sh

```
#!/bin/sh

# This script modifies ports in config files to allow multiple instances of hadoop to
# run on the same cluster using common namenode but separate datanodes.
# author: Marissa Hollingsworth and Amit Jain
# modified for Hadoop >=2.6 by Gabriel Trisca

HADOOP_HOME=$HOME/hadoop-install

if test ! -d "${HADOOP_HOME}"
then
    echo
    echo "Error: missing hadoop install folder: ${HADOOP_HOME}"
    echo "Install hadoop in install folder before running this script!"
    echo
    exit 1
fi

if [ $# -ne 1 ]
then
    echo "Usage: $0 <base port>"
        exit 1
fi

baseport=$1
```

```
if [ $baseport -gt 62000 -o $baseport -lt 10000 ]
then
       echo "$0: bad base port range: choose between 10000 and 60000"
       exit 1
fi
if [ $baseport -eq 50000 ]
then
       echo "$0: forbidden base port range: 50000 is standard port so avoid it!"
       exit 1
fi

cd ${HADOOP_HOME}

MASTER=`head -1 etc/hadoop/masters`

/bin/cp etc/hadoop/core-site.xml etc/hadoop/core-site.xml.backup
/bin/cp etc/hadoop/hdfs-site.xml etc/hadoop/hdfs-site.xml.backup
/bin/cp etc/hadoop/yarn-site.xml etc/hadoop/yarn-site.xml.backup
/bin/cp etc/hadoop/mapred-site.xml etc/hadoop/mapred-site.xml.backup

#
# Configure paths for HDFS
#

param="dfs.namenode.name.dir"
newvalue="<value>file:$TMPDIR/hadoop-$(whoami)/hdfs/name</value>"
sed -i "/$param/ {
n
c\
$newvalue
}" etc/hadoop/hdfs-site.xml

param="dfs.datanode.data.dir"
newvalue="<value>file:$TMPDIR/hadoop-$(whoami)/hdfs/data</value>"
sed -i "/$param/ {
n
c\
$newvalue
}" etc/hadoop/hdfs-site.xml

#
# Configure uri to NameNode
#

# 58000
port=$baseport
param="fs.defaultFS"
newvalue="<value>hdfs://$MASTER:$port</value>"
sed -i "/$param/ {
n
c\
$newvalue
}" etc/hadoop/core-site.xml

#
# Configure Resource Manager (YARN)
#

# 58010
param="yarn.resourcemanager.hostname"
newvalue="<value>$MASTER</value>"
sed -i "/$param/ {
n
c\
$newvalue
}" etc/hadoop/yarn-site.xml

# 58010
port=$[baseport+10]
param="yarn.resourcemanager.address"
newvalue="<value>$MASTER:$port</value>"
sed -i "/$param/ {
n
c\
$newvalue
}" etc/hadoop/yarn-site.xml

# 58020
port=$[baseport+20]
param="yarn.resourcemanager.scheduler.address"
newvalue="<value>$MASTER:$port</value>"
sed -i "/$param/ {
n
c\
```

```
$newvalue
}" etc/hadoop/yarn-site.xml

# 58030
port=$[baseport+30]
param="yarn.resourcemanager.resource-tracker.address"
newvalue="<value>$MASTER:$port</value>"
sed -i "/$param/ {
n
c\
$newvalue
}" etc/hadoop/yarn-site.xml

# 58040
port=$[baseport+40]
param="yarn.resourcemanager.admin.address"
newvalue="<value>$MASTER:$port</value>"
sed -i "/$param/ {
n
c\
$newvalue
}" etc/hadoop/yarn-site.xml

# 58050
port=$[baseport+50]
param="yarn.resourcemanager.webapp.address"
newvalue="<value>$MASTER:$port</value>"
sed -i "/$param/ {
n
c\
$newvalue
}" etc/hadoop/yarn-site.xml

# 58060
port=$[baseport+60]
param="yarn.nodemanager.webapp.address"
newvalue="<value>0.0.0.0:$port</value>"
sed -i "/$param/ {
n
c\
$newvalue
}" etc/hadoop/yarn-site.xml

# 58070
port=$[baseport+70]
param="yarn.nodemanager.localizer.address"
newvalue="<value>0.0.0.0:$port</value>"
sed -i "/$param/ {
n
c\
$newvalue
}" etc/hadoop/yarn-site.xml

#
# Configure MapRed settings (JVM's on every node)
#

# 58060
port=$[baseport+60]
param="mapreduce.jobhistory.address"
newvalue="<value>0.0.0.0:$port</value>"
sed -i "/$param/ {
n
c\
$newvalue
}" etc/hadoop/mapred-site.xml

# 58070
port=$[baseport+70]
param="mapreduce.jobhistory.webapp.address"
newvalue="<value>0.0.0.0:$port</value>"
sed -i "/$param/ {
n
c\
$newvalue
}" etc/hadoop/mapred-site.xml


echo
echo "Updated files hdfs-site.xml, core-site.xml, yarn-site.xml, mapred-site.xml in hadoop conf folder"
echo
```

B.2.2 create-hadoop-cluster.sh

```
#!/bin/sh
# author: Amit Jain
# modified for Hadoop >=2.6 by Gabriel Trisca

export JAVA_HOME="/cm/shared/apps/java/gcc/64/jdk1.8.0_31"
export HADOOP_HOME=$HOME/hadoop-install
export SLAVES_FILE=${HADOOP_HOME}/etc/hadoop/slaves

if test ! -d "${HADOOP_HOME}"
then
    echo
    echo "Error: missing hadoop install folder: ${HADOOP_HOME}"
    echo "Install hadoop in install folder before running this script!"
    echo
    exit 1
fi

cd ${HADOOP_HOME}

/bin/rm -fr logs pids
mkdir {logs,pids}

pbsdsh -v "/bin/rm -fr $TMPDIR/hadoop-‘whoami‘"
pbsdsh -v "mkdir $TMPDIR/hadoop-‘whoami‘"
pbsdsh -v "chmod 700 $TMPDIR/hadoop-‘whoami‘"

cd etc/hadoop/
for node in $(cat $SLAVES_FILE)
do
    mkdir $HADOOP_HOME/pids/$node
done

mkdir $HADOOP_HOME/pids/‘hostname‘
cd $HADOOP_HOME

java -version

MASTER=‘head -1 etc/hadoop/masters‘
HOST=‘hostname‘

echo "[MASTER][$HOST] Formatting the DFS filesystem"
# Formating has to be executed on NameNode
ssh $MASTER $HADOOP_HOME/bin/hdfs namenode -format

echo "[MASTER][$HOST] Starting the DFS on all nodes"
sbin/start-dfs.sh
sleep 30

echo "[MASTER][$HOST] Launching YARN on $MASTER"
# ResourceManager has to be stopped from the same host
ssh $MASTER $HADOOP_HOME/sbin/start-yarn.sh
```

B.2.3 create-hadoop-cluster.sh

```
#!/bin/sh
# author: Amit Jain
# modified for Hadoop >=2.6 by Gabriel Trisca

export JAVA_HOME="/etc/alternatives/jre_openjdk"
export HADOOP_HOME=$HOME/hadoop-install
export SLAVES_FILE=${HADOOP_HOME}/etc/hadoop/slaves

cd $HADOOP_HOME

MASTER=‘head -1 etc/hadoop/masters‘

sbin/stop-dfs.sh
# ResourceManager has to be stopped from the same host
ssh $MASTER $HADOOP_HOME/sbin/stop-yarn.sh
#/bin/rm -fr logs pids
echo
echo "Removing hadoop filesystem directories"
pbsdsh -v "/bin/rm -r $TMPDIR/hadoop-‘whoami‘"
```

B.2.4 removeDuplicateHosts.py

```
#!/usr/bin/python

import sys

source=open(sys.argv[1],'r')
destination=open(sys.argv[2],'w')

unique = {}

for line in source:
       unique[line.strip()] = 1

for key,vaue in unique.iteritems():
       destination.write(key)
       destination.write('\n')

source.close()
destination.close();
```

B.2.5 runHadoopOnKestrel.sh

```
#!/bin/bash

### Set the job name
#PBS -N EventDetection

### Run in the queue named "batch"
#PBS -q MRI

### Use the bourne shell
#PBS -S /bin/sh
#PBS -V

### To send email when the job is completed:
#PBS -m ae
#PBS -M gabrieltrisca@boisestate.edu

#PBS -j oe
#PBS -o localhost:/home/gabrieltrisca/logs/out.log

### Specify the number of cpus for your job.
#PBS -l select=10:ncpus=1:ngpus=1

#PBS -l place=scatter

###PBS -l mem=1gb
#PBS -l min_walltime=00:08:00
#PBS -l max_walltime=00:25:00

HADOOP_HOME="${HOME}/hadoop-install/"

module load pdsh/2.9
module load pbspro
module load java/gcc/64/1.8.0_31

cd $PBS_O_WORKDIR
echo Working directory is $PBS_O_WORKDIR

cp $PBS_NODEFILE $HOME/logs/

# Calculate the number of processors allocated to this run.
NPROCS=`wc -l < $PBS_NODEFILE`

# Calculate the number of nodes allocated.
NNODES=`uniq $PBS_NODEFILE | wc -l`

### Display the job echo Running on host `hostname`
echo Time is `date`
echo Directory is `pwd`
echo Using ${NPROCS} processors across ${NNODES} nodes
echo "Temporary directory is $TMPDIR"
echo "======================================="
echo
for line in $(cat $PBS_NODEFILE) ;do
       ssh-copy-id -i $HOME/.ssh/id_rsa.pub $line
       ssh -x $line rm -rf $TMPDIR/hadoop-`whoami`
done
```

```
mv ${HADOOP_HOME}/etc/hadoop/masters ${HADOOP_HOME}/etc/hadoop/masters.orig
mv ${HADOOP_HOME}/etc/hadoop/slaves ${HADOOP_HOME}/etc/hadoop/slaves.orig
rm ${HADOOP_HOME}/etc/hadoop/nodes


echo "*** Trying to load Java on all nodes"
pbsdsh -v "module load java/gcc/64/1.8.0_31"

cat $PBS_NODEFILE > ${HADOOP_HOME}/etc/hadoop/nodes
sed 's/.cm.cluster//' ${HADOOP_HOME}/etc/hadoop/nodes > ${HADOOP_HOME}/etc/hadoop/pbsnodes

$HOME/classifier/resources/kestrel/removeDuplicateHosts.py ${HADOOP_HOME}/etc/hadoop/pbsnodes ${HADOOP_HOME}/etc/hadoop/slaves

MASTER=`head -1 ${HADOOP_HOME}/etc/hadoop/slaves`
echo $MASTER > ${HADOOP_HOME}/etc/hadoop/masters

#echo $PBS_O_HOST ${HADOOP_HOME}/etc/hadoop/masters

${HADOOP_HOME}/local-scripts/cluster-pickports.sh 58000

echo "*** Creating hadoop cluster"
pbsdsh -n1 -v "${HADOOP_HOME}/local-scripts/create-hadoop-cluster.sh"
echo "*** Sleeping for 60s..."
sleep 30

echo "*** Executing hadoop job"
echo Starting time: `date`
$@
echo Ending time: `date`

echo "*** Copying output data to local filesystem"
${HADOOP_HOME}/bin/hdfs dfs -ls ./EventDetectionClassification/* > $HOME/logs/folder_contents.txt
${HADOOP_HOME}/bin/hdfs dfs -get ./EventDetectionClassification/* $HOME/logs/
${HADOOP_HOME}/bin/hdfs dfs -get /input_filenames/* $HOME/logs/

#echo "*** Deleting the cluster"
pbsdsh -n1 -v "${HADOOP_HOME}/local-scripts/cluster-remove.sh"
```

B.2.6 runOnKestrelHadoop.sh

```
#!/bin/bash

#export JAVA_HOME="/etc/alternatives/jre_openjdk"
export JAVA_HOME="/cm/shared/apps/java/gcc/64/jdk1.8.0_31"
export HADOOP_HOME="$HOME/hadoop-install/"

$HADOOP_HOME/bin/hadoop jar $HOME/classifier/target/classifier-0.0.1-SNAPSHOT-jar-with-dependencies.jar edu.boisestate.cgiss.distributed
      .DistributedEventClassificationKestrel $@
```

Appendix C

PATCH TO PREVENT ERRORS ON DISPOSING KERNELS IN APARAPI

```
Index: com.amd.aparapi.jni/src/cpp/runKernel/JNIContext.cpp
===================================================================
--- com.amd.aparapi.jni/src/cpp/runKernel/JNIContext.cpp  (revision 1700)
+++ com.amd.aparapi.jni/src/cpp/runKernel/JNIContext.cpp  (working copy)
@@ -19,9 +19,8 @@
    deviceId = OpenCLDevice::getDeviceId(jenv, openCLDeviceObject);
    cl_device_type returnedDeviceType;
    clGetDeviceInfo(deviceId, CL_DEVICE_TYPE, sizeof(returnedDeviceType), &returnedDeviceType, NULL);
-   //fprintf(stderr, "device[%d] CL_DEVICE_TYPE = %x\n", deviceId, returnedDeviceType);
+   //fprintf(stderr, "device[%ld] CL_DEVICE_TYPE = %x\n", deviceId, returnedDeviceType);

-
    cl_context_properties cps[3] = { CL_CONTEXT_PLATFORM, (cl_context_properties)platformId, 0 };
    cl_context_properties* cprops = (NULL == platformId) ? NULL : cps;
    context = clCreateContextFromType( cprops, returnedDeviceType, NULL, NULL, &status);
@@ -32,7 +31,7 @@
 }

 void JNIContext::dispose(JNIEnv *jenv, Config* config) {
-   //fprintf(stdout, "dispose()\n");
+   //fprintf(stderr, "dispose()\n");
    cl_int status = CL_SUCCESS;
    jenv->DeleteGlobalRef(kernelObject);
    jenv->DeleteGlobalRef(kernelClass);
@@ -67,21 +66,41 @@
       for (int i=0; i< argc; i++){
         KernelArg *arg = args[i];
         if (!arg->isPrimitive()){
-           if (arg->arrayBuffer != NULL){
-              if (arg->arrayBuffer->mem != 0){
-                 if (config->isTrackingOpenCLResources()){
-                    memList.remove((cl_mem)arg->arrayBuffer->mem, __LINE__, __FILE__);
+           if (arg->isArray()) {
+              if (arg->arrayBuffer != NULL){
+                 if (arg->arrayBuffer->mem != 0){
+                    if (config->isTrackingOpenCLResources()){
+                       memList.remove((cl_mem)arg->arrayBuffer->mem, __LINE__, __FILE__);
+                    }
+                    status = clReleaseMemObject((cl_mem)arg->arrayBuffer->mem);
+                    //fprintf(stdout, "dispose arg %d %0lx\n", i, arg->arrayBuffer->mem);
+                    CLException::checkCLError(status, "clReleaseMemObject()");
+                    arg->arrayBuffer->mem = (cl_mem)0;
                  }
-                 status = clReleaseMemObject((cl_mem)arg->arrayBuffer->mem);
-                 //fprintf(stdout, "dispose arg %d %0lx\n", i, arg->arrayBuffer->mem);
-                 CLException::checkCLError(status, "clReleaseMemObject()");
-                 arg->arrayBuffer->mem = (cl_mem)0;
+                 if (arg->arrayBuffer->javaArray != NULL) {
+                    jenv->DeleteWeakGlobalRef((jweak) arg->arrayBuffer->javaArray);
+                 }
+                 delete arg->arrayBuffer;
+                 arg->arrayBuffer = NULL;
+              }
+              if (arg->arrayBuffer->javaArray != NULL) {
+                 jenv->DeleteWeakGlobalRef((jweak) arg->arrayBuffer->javaArray);
+              }
+           } else if (arg->isAparapiBuffer()) {
+              if (arg->aparapiBuffer != NULL){
+                 if (arg->aparapiBuffer->mem != 0){
```

```
+                  if (config->isTrackingOpenCLResources()){
+                     memList.remove((cl_mem)arg->aparapiBuffer->mem, __LINE__, __FILE__);
+                  }
+                  status = clReleaseMemObject((cl_mem)arg->aparapiBuffer->mem);
+                  //fprintf(stdout, "dispose arg %d %0lx\n", i, arg->aparapiBuffer->mem);
+                  CLException::checkCLError(status, "clReleaseMemObject()");
+                  arg->aparapiBuffer->mem = (cl_mem)0;
+               }
+               if (arg->aparapiBuffer->javaObject != NULL) {
+                  jenv->DeleteWeakGlobalRef((jweak) arg->aparapiBuffer->javaObject);
+               }
+               delete arg->aparapiBuffer;
+               arg->aparapiBuffer = NULL;
             }
-            delete arg->arrayBuffer;
-            arg->arrayBuffer = NULL;
+
          }
         }
         if (arg->name != NULL){
@@ -91,7 +110,7 @@
           jenv->DeleteGlobalRef((jobject) arg->javaArg);
         }
         delete arg; arg=args[i]=NULL;
-     }
+     } // for
      delete[] args; args=NULL;

      // do we need to call clReleaseEvent on any of these that are still retained....
```