Apache Storm-based SDP Pipeline

SKA Time Domain Team[a]

[a]*R. J. Lyon, L. Levin-Preston, B. W. Stappers*

Summary

We have developed a prototype pulsar search pipeline, built entirely from off-the-shelf software components. The prototype, developed within the Apache Storm Framework, has been designed to maximise computational efficiency, and minimise cost where possible. It encapsulates all of the off-line processing steps typically undertaken during a real-world pulsar search, including data pre-processing, machine learning-based candidate filtering, sifting, and known source matching. However it is very different from existing software search pipelines. It is able to operate on streams of data arriving sequentially over time, a feature which most modern pipelines lack. To achieve this, numerous aspects of standard pulsar search have had to be modified for parallel and distributed computation. This had not been attempted before.

We have evaluated the performance of the prototype with respect to SKA design requirements. This was done via simulations that mimicked a future SKA search for pulsars. During the simulations, the prototype was able to consistently process over 1,000 candidates per second. This was achieved using only modest commodity computational resources (a single laptop). Based upon these results, it is plausible for the prototype (and similar software) to meet the design requirements for SKA pulsar search, however at very low cost compared to bespoke tools. Use of the prototype also demonstrates that it is possible to develop an end-to-end science data processor (SDP) for the SKA, as such a system had not been demonstrated before. It also illustrates our capacity to do so.

1. Introduction

State-of-the-art pulsar search pipelines are generally comprised of two key components. The first is a signal processing system. This converts the voltages induced in the receiving element of a radio telescope, into digital signal detections for analysis. For the remainder of this document, we refer to this first system as the central signal processor (CSP). The second is a filtering system that reduces the large number of detections found via the CSP, facilitating scientific study. We refer to this second system as the science data processor (SDP). This memo is focused exclusively on developing a prototype SDP, capable of supporting Square Kilometre Array (SKA) pulsar searches.

2. SDP Preliminaries

Modern pulsar search pipelines are comprised of custom software tools, written by research groups within the radio astronomy community (e.g. Lorimer, 2016; Keith, 2016; Ransom, 2016). These are tailored to achieve specific science goals, and are often written with a particular observing set-up, search target (e.g. millisecond pulsar vs. longer period pulsars), or observing instrument in mind (Lyon et. al., 2016). When combined these tools form a pipeline. For the purposes of this memo such pipelines are considered to be SDPs. Given the heterogeneity of search tools, there are effectively multiple pipelines in active use at any one time, spanning search efforts around the globe (e.g. compare, Barr, 2014; Bhattacharyya et. al., 2016).

2.1. Problem Definition

The goal for a SDP is to reduce an input set of candidate detections C, to a subset of promising detections C'. These are the detections possessing the most scientific utility. The input set contains n elements, and n varies according the exact processing configuration used. For example if processing data per observation, then $n \approx 1000$ is reasonable. Whilst if processing data for an entire survey, $n > 10^6$ is not unusual. In either case the SDP should return a set C', such that $|C'| \leq |C|$, though in reality we desire $|C'| \ll |C|$.

Each element in C is most easily described as a candidate tuple. A tuple is simply a list of m elements. An individual candidate tuple is defined as $c_i = \{c_i^1, \ldots, c_i^m\}$. Here each tuple is uniquely identifiable in C via the index i. For all $c_i \in C$, it holds that $|c_i| > 0$ (in others words $m > 0$). For simplicity we assume that all $c_i^j \in \mathbb{R}$, and that there is no implicit ordering in C. In practice the numerical components of a tuple c_i^j, represent candidate characteristics. These would include the signal-to-noise ratio (S/N), dispersion measure (DM), period, pulse width, beam number, coordinates, integrated pulse profile etc. The SDP must use these characteristics to make accurate filtering decisions, producing as pure an output set as possible.

We note that the set C is not a multi-set, thus it does not contain any exact duplicates. Whilst there are no exact duplicates, there are however non-exact duplicates. Non-exact duplicates include pulsars detected by different telescope beams,

or at different S/N, DM, acceleration, and even period values (harmonic detections). Only the 'strongest' detection need be retained, with sub-optimal detections removed to minimise $|C'|$. The strongest detection is usually the candidate with the highest S/N. If the data in C is available off-line, the strongest detection amongst a collection of duplicates is easy to find, via exhaustive comparisons. Most detections in C however will be attributable to noise or interference. Thus finding and removing these results in the greatest reduction in $|C'|$.

2.2. Existing SDPs

Existing pulsar search pipelines execute filtering tasks sequentially, on a candidate set C. The set is usually processed off-line (Sclocco et. al., 2015), either upon the completion of a pulsar survey, or at the end of an individual observational session if a faster pace of discovery is desired (see for example, Barr et. al., 2013; Ng, 2012; Coenen et. al., 2014; Bhattacharyya et. al., 2016). At present it is feasible to permanently store all the candidates in C. This allows them to be processed off-line more than once, with no real-time constraints. It is common for new pulsars to be found in C, even after it has been searched multiple times (e.g. Keith et. al., 2009; Mickaliger et. al., 2012). This happens when improved search algorithms or search parameters are applied, helping to isolate otherwise hidden detections.

A small number of recently developed pipelines process batches of data in real-time (e.g. Thompson et. al., 2011; Ait-Allal, 2012; Barr, 2014; van Heerden et. al., 2014; Law et. al., 2015; Petroff et. al., 2015; Karastergiou et. al., 2015; Naidu et. al., 2015). However these are generally designed to identify transient events (see McLaughlin et. al., 2006; McLaughlin, 2009) and fast radio bursts (FRBs, see Lorimer et. al., 2007; Keane et. al., 2012; Thornton et. al., 2013; Keane et. al., 2016) rather than pulsars. The transition to real-time processing has occurred due to changing science requirements (rapid follow-up), and in response to increasing survey data output volumes (Lyon et. al., 2016). The latter makes it difficult to store data permanently for off-line analysis, given the associated cost of storage media (Sclocco et. al., 2015; Lyon et. al., 2016). Current trends in survey data volumes, suggest that in the SKA era, pulsar search will necessarily become a real-time process (Lyon, 2016; Lyon et. al., 2016). Transitioning off-line SDP code to the real-time paradigm at SKA scales, is challenging.

2.3. SKA SDP Challenge

It is sensible to assume that a SDP built for the SKA must be capable of real-time operation. To be viable, real-time SDP operation must be completed within strict time and resource constraints as dictated by the SKA design (see, Dewdney et. al., 2013, 2015; Nijboer et. al., 2015; Tan et. al., 2015). How then to transition existing off-line tools to the real-time paradigm? This could be achieved via algorithmic optimisations that reduce the runtime, memory, and computational complexity of SDP components. Efforts are already under-way to optimise numerous aspects of the CSP system in this manner (see for example, Dimoudi & Armour, 2015; Sclocco et. al., 2015). SDP operations must also be able to operate functionally within an SKA search pipeline. However not all processing steps can be directly translated to a real-time environment. This is generally true of tasks that require data to be aggregated prior to processing, an operation that is extremely difficult to achieve at SKA scales. In such cases existing methods must be redesigned, or new techniques developed, to maintain the effectiveness of SDP for SKA pulsar searches (Broekema et. al., 2015). We therefore embarked on a prototyping effort to find these problems, and attempt to solve them. We now introduce our design desiderata for the prototype pipeline.

3. Design Desiderata & Assumptions

The design of a science data processor is driven by multiple considerations (for the design drivers see Dewdney et. al., 2015). For the purposes of this prototyping effort, we focused principally on utilising off-the-shelf software components in order to reduce cost. This is our primary design driver. However our choice of software components was necessarily driven by the need to reduce computational overheads, whilst maximising scalability.

3.1. Hardware

Modern SDP software is most often executed on general purpose computing resources, and not graphics processing units (GPUs, Magro et. al., 2011; Couturier, 2013; Dimoudi & Armour, 2015)[1] or field programmable gate arrays (FPGAs, Kilts, 2007; Dubey, 2008; Woods et. al., 2008; Vanderbauwhede & Benkrid, 2013; Wang & Sinnen, 2015). We therefore only consider SDP execution upon CPUs. We believe this restriction to be reasonable. The latest SKA design documents (Dewdney et. al., 2013, 2015; Broekema et. al., 2015) indicate that SDP activities will execute upon generic computing resources (compute islands, see Broekema et. al., 2015) over which tasks can be distributed in parallel. Where possible we have endeavoured to produce a modular prototype that is hardware agnostic, to enable porting to hardware specific resources if required, at a later date.

3.2. Processing Scenario

We consider a 'worst case' processing scenario, based upon SKA-Mid described by Broekema et. al. (2015). Here the input data rate to the SDP is capped at 5.2 Tbps. We assume this is high enough to prevent the storage of data for off-line analysis. Similarly it is assumed to be impossible to cache significant quantities of this data, making batch processing impractical. We further suppose that the SDP is able to utilise only limited computational resources due to cost. At present the SDP is expected to require 24 PFLOPs of compute capacity (Broekema et. al., 2015), which we treat here as an upper limit. It is highly desirable for the prototype to be as far from this limit as possible. Finally we assume traditional disk access is unavailable,

[1] There are some exceptions, e.g. machine learning based candidate filters (Zhu et al., 2014).

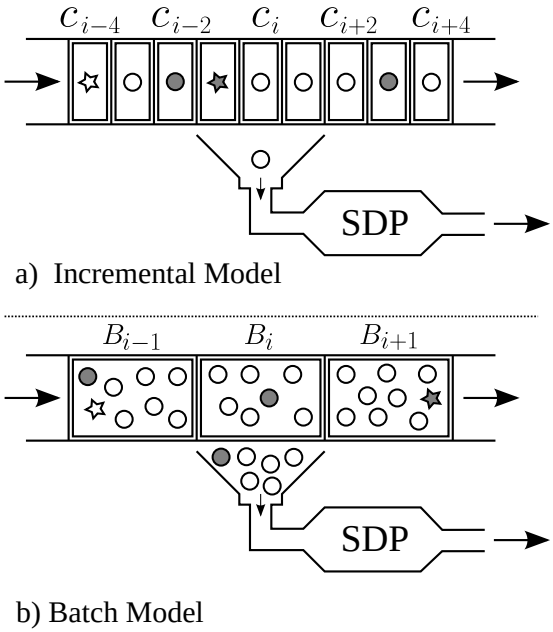

Figure 1: Incremental and batch models of learning over data streams.

since read/write operations are slow and obstruct real-time operation. Without the capacity to store data for analysis, we are faced with a data stream processing scenario.

4. On-line Data Stream Problem Definition

There are two distinct models for data stream processing. The incremental model applies when individual data items arrive at discrete time steps. Thus an item c_i arriving a time i, is always processed after an item arriving at time $i-1$, and before $i+1$. Whilst the batch model applies when groups of items arrive together at discrete time steps, such that batch B_i arrives after B_{i-1} and before B_{i+1}. This is summarised in Figure 1. Both these models temporally order streaming data, and this has implications for how the data can be processed (especially with respect to sifting, see Section 7.4).

To accommodate our processing restrictions, an incremental streaming model is adopted. Here the CSP is viewed as a data stream producer. In this scenario the size of the input set C is unbounded (unknown n). This means that SDP is receiving a theoretically limitless supply of individual candidates $c_i \in C$ incrementally, according to some interval t_{int}. These arrive in a temporally ordered fashion, and **only one** will arrive at a time. We can thus assume a discrete time model (candidates arrive at discrete time steps). We extend the notation used previously, such that c_i is now the candidate arriving at time step i, and c_i always arrives before c_{i+1}. The set of candidates C can now be easily viewed as a stream containing candidates $C = \{c_1, c_2, \ldots, c_n, \ldots\}$.

To maintain real-time operation in this scenario, each arriving c_i must be processed before the next one arrives (or quickly enough to prevent queuing). This is necessary given our assumption that cost restrictions prevent us from caching data. Thus filtering decisions must be made on an individual candidate basis, with very limited knowledge. For example, suppose c_i arrives for processing having observed c_{i-10} to c_{i-1}. Here a filtering decision can only be made using knowledge of the candidates already seen and c_i. If c_i is most likely noise or interference, then the decision is conceptually simple. However if c_i is a pulsar detection, do we retain it? Is it the strongest detection we are likely to see? If it is weak, should we throw it away, assuming a stronger detection will come along? Do we keep every weak detection, and greatly risk increasing the size of C'? There are no simple answers to such questions. To proceed, we must assume that arriving candidates will be ordered according to c_i^0 (an ordering variable). This ordering must be strict, so that $\forall c_i \in C,\ c_i^0 \leq c_{i+1}^0$. This definition specifies an ascending order, though it would make no difference if it were descending. By using an ordering we can have confidence that similar items will be close together in the stream (arrive close temporally). This simplifies our data processing, and will be discussed more in Section 7.4. However now we look for frameworks that can accommodate this processing model.

5. Candidate Frameworks for Prototyping

There are many frameworks that could be used to create an SDP prototype based upon our requirements. These are briefly reviewed, before the chosen framework is introduced.

5.1. Apache Hadoop

This is a framework for managing 'big' data processing (White, 2012; Jankowski et. al., 2015; Apache Software Foundation, 2014). It utilises a cluster resource manager/job scheduler called Yet Another Resource Negotiator (YARN). This a Java based system built around a batch processing component called Hadoop MapReduce. It is designed for the distributed processing of distributed data, using the Hadoop Distributed File System (HDFS) format. As it is designed for batch processing, it is not suitable for prototyping.

5.2. Apache Spark

Spark is a batch processing framework like Hadoop (Apache Software Foundation, 2016), though it is often preferred over Hadoop when processing can be accomplished in memory on commodity hardware. This is because Spark can achieve significant processing speed-ups over hadoop, using a data structure known as the resilient distributed dataset (RDD). This is a read-only multi-set of data items distributed over a cluster of machines. It allows for a distributed form of shared memory with iterative access. This is useful for developing specialised data analysis tools, particularly those that employ machine learning (ML) algorithms (for an ML introduction see Duda et. al., 2000; Bishop, 2006; Russell & Norvig, 2009) for data analysis. Spark can now process data outside of a batch model, using the Spark Streaming component of the framework. This is described as a micro-batch streaming tool by practitioners (Jankowski et. al.,

2015). This is because it has to convert an incoming stream of data into a batch, in order to operate (due to it being based on Spark). Given the computational overhead involved in micro-batching data, this framework is not preferred for prototyping.

5.3. Apache Samza

Samza is a stream processing system built to run on the YARN system (Apache Software Foundation, 2015d). It carries an at-least-once delivery promise for stream data items, and maintains state management on a computational node. It is designed to be scalable and fault-tolerant, whilst maintaining a parallel processing model. Samza is currently at an early stage in its development, and is yet to include all planned functionality. It is not preferred for prototyping given its smaller user base, and limited user support infrastructure.

5.4. S4

S4 is a distributed stream processing system initially developed internally at Yahoo (Apache Software Foundation, 2015c). It is similar to Apache Samza in design and purpose, though with subtle implementation specific differences. It is capable of real-time incremental stream processing, load-balancing and managing fault tolerance. However its supporting documentation is lacking. The configuration of the framework is also known to be complex.

5.5. Apache Storm

Apache Storm is a distributed software framework that processes streaming data in real-time (Apache Software Foundation, 2015a; Jain & Nalya, 2014; Jankowski et. al., 2015). It is designed to be scalable on modern computational infrastructures, and is claimed to be robust to error due to its fail-fast methodology. The system guarantees that each data item passed into the framework, will be processed at least once, irrespective of errors. The framework takes care of fault tolerance, message passing and load balancing automatically, allowing software engineers to concentrate their efforts on task-specific code. Storm is used by organisations that manage and extract knowledge from high velocity and volume streams (Twitter, Spotify, Weibo) (Apache Software Foundation, 2015b). Storm has recently been superseded at Twitter by the Heron framework (Kulkarni, 2015), however this is not yet publicly available.

5.6. Choosing a Framework

We have chosen to use Apache Strom as the framework for our SDP prototype. Storm shares many commonalities with other frameworks such as Apache Samza and S4, and is thus representative. However it was chosen over the alternatives as an exemplar, primarily due to its application in the real-world to SDP scale problems. For example at Twitter, where it was used to process 500 million events per day (almost 6,000 per second) (Jones, 2013; Toshniwal, 2014; Weiler, 2015). Storm also has a large user base and a software support infrastructure, providing sources of help in the event of problems.

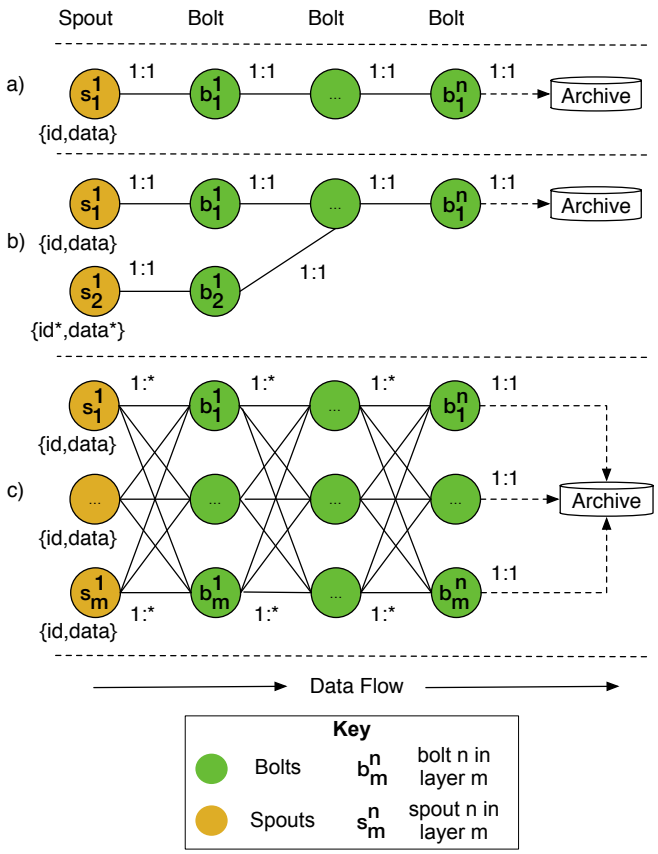

Figure 2: Here three example topologies are shown. Topology a) consists of a single spout which emits data tuples to a series of bolts. Here the tuples move from one bolt to the next for processing in a linear fashion. The spouts and bolts have a one-to-one relationship. The final bolt in the topology stores the outcome of the processing to a data archive. Topology b) has two input spouts, s_1^1 and s_1^2. The tuples from spout s_1^1 pass through the bolts linearly as before. Whilst spout s_1^2 sends tuples to the bolt b_1^2. This in turn emits the tuples to be merged at a bolt in layer 1. Thus the bolt in the centre of layer 1 processes two sets of incoming streams. Finally topology c) has multiple input spouts and bolts. These are connected via one-to-many relations. Depending on how the topology is configured, individual tuples moving through topology c) could be emitted from the spouts to all connected nodes, or the just one bolt chosen at random.

6. Storm Framework

Apache Storm is a Java based framework. It operates under an entirely incremental data stream model. The system is underpinned by the notion of a directed-acyclic graph (DAG). The graph describes the processing steps to be completed, and the flow of data through the system. Within Storm this graph is known as a *topology*. In general Storm topologies only allow data to flow in one direction, making recursive/reciprocal processing steps impractical to implement. The topology itself is comprised of nodes. Nodes can represent data output sources known as *spouts*, and processing tasks applied to the data which are called *bolts*. Data flows from spouts to the bolts via edges in the topological graph, in the form of n-tuples. These are finite ordered lists of items, which could be numbers, text strings, or objects.

Three example topologies are shown in Figure 2. Topology a) consists of a single spout, which emits tuples to a series of bolts. Here the tuples move from one bolt to the next, such that tuples are processed in a linear fashion. The spouts and bolts have a simple one-to-one relationship. The final bolt in the topology stores the outcome of processing to a data archive. Topology b) has two input spouts, s_1^1 and s_1^2. The tuples from spout s_1^1 pass through the bolts linearly as before. Whilst spout s_1^2 sends tuples to the bolt b_1^2. This in turn emits the tuples to be merged at a bolt in layer 1. Thus the bolt in the centre of layer 1 processes two incoming streams. Finally topology c) has multiple input spouts and processing bolts. These are connected via one-to-many relations. Depending on how the topology is configured, individual tuples moving through topology c) could be emitted from the spouts to all connected bolts, or to just one bolt chosen at random. The bolts themselves are heterogeneous, allowing for different combinations of processing to be performed. Thus topologies can be used to describe extremely complex processing scenarios.

The edges in the topology represent generic connections between processing units. In practice, edges are generally connections made via a wide area network (WAN), or a local area network (LAN). Data flows via the edges as tuples. Upon arriving at a bolt a tuple may be modified, and the updated tuple emitted, for the next bolt in the topology to process. The final bolt that tuples reach may archive their contents to permanent storage, or simply discard them as appropriate - there is no requirement that tuples be processed in a particular way.

Each bolt in a Storm topology should encapsulate a generic task, that can be completed without interaction with other bolts. Thus each bolt should be modular, be able to process data in parallel, and be distributable across computing resources without causing a loss in functionality. Adherence to this means that bolts can be easily duplicated. Using duplication a topology can be scaled to process increasing amounts of data, without additional development effort. It also means that when a failure occurs at a bolt, it can quickly be replaced with a new instance without a loss of functionality.

This modular design is not ideal for all processing scenarios. Particularly those where the persistence of state is required. Generally state persists within a spout/bolt only whilst it is executing. If it fails or is stopped, that state is lost. Note that bolts and spouts are usually very lightweight serializable components. This enables them to be sent quickly across a network connection, and initialised (via deserialization) on a worker node running upon an arbitrary computing resource. Storm is unsuitable for processing components that are i) programmatically large, ii) require a great deal of persisted state to function to correctly, or iii) require data to be aggregated/batched.

6.1. Tuple Flow

Tuples moving through a Storm topology do not necessarily arrive at all bolts. Storm allows the programmer to control how many bolts an individual tuple arrives at. This is achieved by

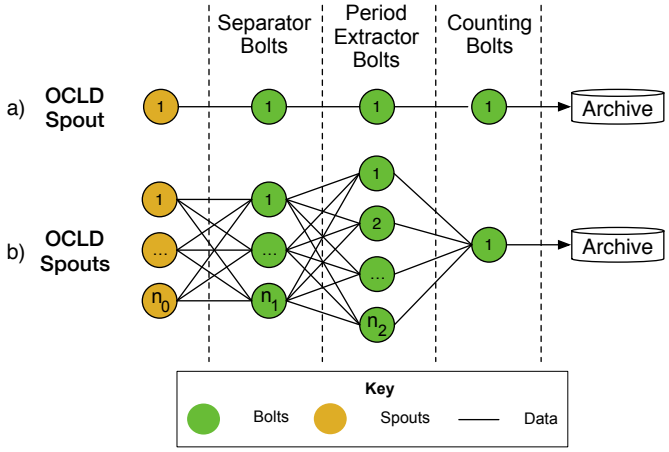

Figure 3: An example of a data processing topology. Version a) is a simple linear processing version, whilst b) achieves the same goal, however it has been scaled to achieve faster processing.

via a cardinality relation. This is a relation between spouts and bolts, and the bolts that lay before them. This relation can be one-to-one, one-to-many, or one-to-all. If using a one-to-one relation, Storm can be made to move tuples forward to a single random bolt, or a value specific bolt based on a tuple attribute test[2]. Upon sending a tuple, an acknowledgement message can be requested, that is used to enforce fault tolerance in the storm network. If a spout or bolt sending a tuple receives no acknowledgement message, it will resend the tuple. This enables Storm to maintain the property that all data is processed *at least* once.

6.2. Heterogeneous Bolts

It is important to note that different types of bolts are typically used to construct a useful topology. Suppose a data source is producing OCLDs. Imagine we are then tasked with maintaining a count of candidates, which have pulse periods coinciding with some known period range (i.e. a range know to be polluted with RFI). How could we solve this problem? A single spout could first pass the data to a bolt that extracts individual candidates. This in turn could emit candidates to a 'period extractor' bolt. This would extract the pulse period, and send this value to a 'counting' bolt that updates the period counts. These hypothetical bolts contain different program logic, and require specific input tuples to function. This overall functionality can be implemented as simple linear processing chain topology as shown in Figure 3 a), or as a deeper topology that can be scaled to accommodate more than one input source shown in Figure 3 b).

6.3. Deployment

Storm topologies are deployed upon computational clusters which are comprised of two distinct types of node. There is a master node that runs a daemon called Nimbus, and one or more worker nodes which each run a daemon called a Supervisor. The master node (and Nimbus) is responsible for executing tasks, and restarting spouts/bolts after failure. The worker

[2]For example if $c_i^1 \leq 10$, send to bolt b_1^1, else send to bolt b_1^2.

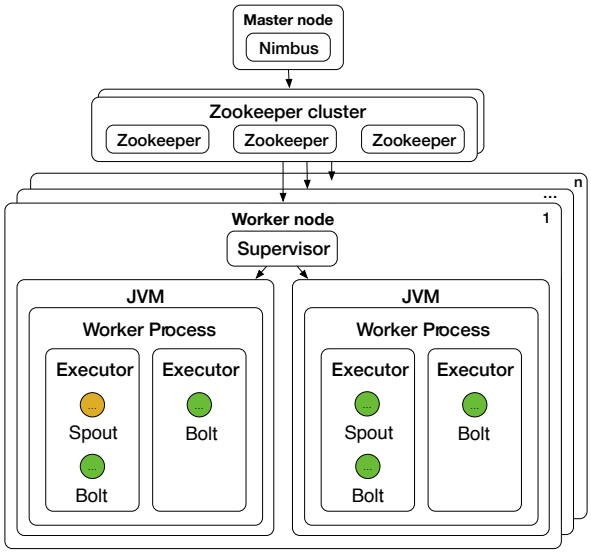

Figure 4: Components of a Storm cluster. The cluster has a single master node that controls topology execution. This master connects to a cluster of Apache Zookeeper instances (one or more), which are used to track the state of the worker nodes which actually perform the processing. Each worker node, of which there could many, executes a single supervisor process and many worker processes. The supervisor connects to the master node to receive execution instructions, while the worker processes execute the code contained within spouts and bolts via 'executors'. Worker processes execute within a Java Virtual Machine (JVM), and this JVM can spawn multiple threads allowing multiple worker processes to execute concurrently on a worker node. Topologies deployed to a Storm cluster can be scaled by adding more worker nodes as required.

nodes execute the spouts and bolts which do the real processing work. Note that spouts and bolts execute within a thread spawned by the Java Virtual Machine (JVM) of a worker node (general computational node), as directed by the supervisor.

Communication between the master and worker nodes is managed by a separate tool. Storm relies on Apache Zookeeper for coordinating the communication between Nimbus and the Supervisors (Jankowski et. al., 2015). Zookeeper maintains any state required by the master and the supervisors, thus if either fail, Zookeeper can restore that state once restarted. A Storm cluster therefore consists of three components - Nimbus, one or more Supervisors, and a Zookeeper.

7. SDP Prototype Design

The complete prototype design is shown in Figure 5. The pipeline has a single 'layer' of spouts, which emit individual pulsar candidates at a controllable rate. Candidates emitted from the spouts are propagated forwards, to a single random bolt at each subsequent layer in the topology. The first layer contains bolts which initially process the candidate data. These modify it so that it is transmitted forward in a tuple format amenable to further processing.

The second layer attempts to filter out duplicate detections of the same candidate with slightly different observational parameters. This is known more generally as sifting. To achieve this, the sifting bolts employ a new distributed multi-beam sift algorithm. This is described in more detail in Section 7.4. Following sifting candidate tuples are sent to bolts that perform pre-processing (i.e. data normalisation). This is required prior to machine learning feature extraction. Once complete, the next layer of bolts extracts the machine learning heuristic features from the candidates. These features describe the signal detection in various ways, and are chosen to facilitate high separability between pulsar and non-pulsar candidates. The features are appended to the tuple, and sent on for further processing.

Following feature extraction, tuples are passed to machine learning bolts. These bolts contain the intelligent machine learning algorithm described by Lyon et. al. (2016). This predicts with high accuracy the true class origin of each candidate. The predictions are appended to the candidate tuple, and passed on to secondary sifting bolts. These remove duplicate candidates in lieu of the additional information obtained during machine learning classification. Candidate tuples making it through this second sift, are passed to known source matching bolts. These attempt to match promising candidates to known pulsar sources in the ATNF pulsar catalogue. Candidates not matched to known sources (likely new pulsars) are sent to alert bolts, which generate dummy alerts for follow-up action.

The topology has also been design to accommodate SDP auditing. Auditing nodes are connected to the bolts via the dashed lines as shown in Figure 5. The auditing nodes are not used during normal execution. Rather they are used to audit the performance of individual bolts, or the entire topology as required, during testing. The individual components of significance are now described in more detail.

7.1. SDP Input

The input data received by the science data processor will consist of Optimal Candidate List and Data (OCLD) objects. OCLD batches will be delivered to the SDP per receiver beam (1 batch per beam), upon completion of a single observation. The SDP can expect OCLD data from 1,500 beams per observation, with an upper limit of 1000 candidates per beam (1 list and 1000 candidates per beam). A single observation will therefore return 1.5 million candidates in total for analysis. With a single SKA observation expected to take 600 seconds to complete, we enforce the following constraint. Candidates collected during an observation must be processed *before the next batch of candidates arrives*. Thus SDP data processing must process the 1.5 million candidates collected, within a 600 second time frame.

It is impractical to generate real OCLD inputs given the resources it would require (and the data volumes involved). Instead we opt to simulate the delivery of OCLDs, via the generation of candidate data on the fly.

7.1.1. Prototype Candidate Generation

The prototype uses custom Storm spouts to generate pulsar candidates that drive our survey simulations. These spouts

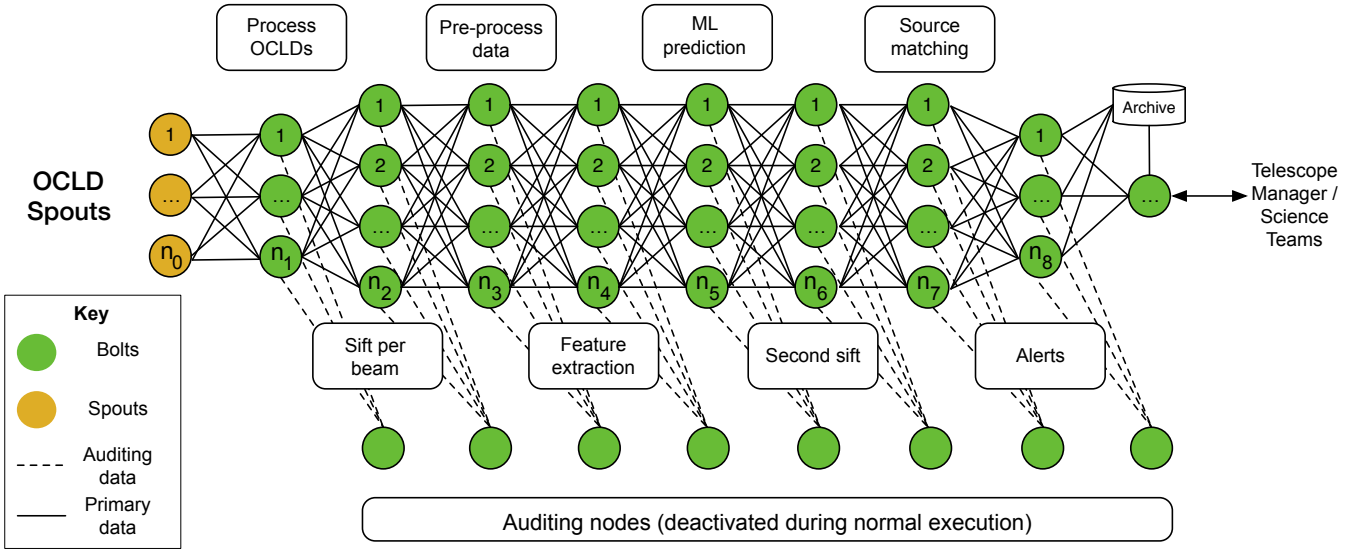

Figure 5: The SDP prototype Storm topology. Data is supplied to the topology via OCLD spouts, which generate candidate data as tuples. The tuples are propagated through the topology, such that a tuple only ever moves 'forwards' to a single randomly chosen bolt in the topology ahead of it. Thus no tuple is duplicated, and is only ever at one bolt in the topology at any given time. There are nine distinct types of bolts in the topology, described in the sections that follow.

play the role of the CSP, delivering data for analysis at a very high throughput. The candidate generation spouts have been designed to emit two types of pulsar candidates.

Type 1 candidates are completely contrived randomly generated examples. Whilst these candidates are correctly formed, they do not describe real world candidates well, and resemble candidates created from white noise. Type 2 candidates are representations of real candidates, sampled from data (see Lyon, 2015) obtained during the High Time Resolution Universe Survey (South) (HTRU, Keith et. al., 2010). Type 2 candidates contain data useful for analysis, and provide a genuine test of our prototype's discriminative capabilities. Note that as candidates are generated by input spouts, their true origin is known in advance. This enables the filtering decisions made by the prototype to be evaluated accurately.

Both forms of candidate consist of a data cube describing the detection in time, phase, and frequency; plus the variables RA, DEC, SNR, period, DM, beam number, and acceleration. These candidates are therefore similar to the candidate objects inside the OCLD, delivered by the CSP in the real world.

The spouts used to generate candidates are highly customizable. They can generate a fixed number of candidates, or as many as possible within a fixed period of time. To be realistic, the spouts must produce candidate distributions similar to that observed in the real-world. After all, only a small fraction of candidates entering the SDP will be of scientific interest. For the prototype we define a ratio to describe this reality. It is directly used to control the generation of candidates at the spouts. Suppose each candidate c_i is associated with a label defining its true origin. This can be modelled numerically via a binary variable y, where $y_i \in Y = \{-1, 1\}$. Here $y_i = -1$ equates to non-pulsar and $y_i = 1$ to pulsar. The set of candidates which describe real pulsar detections $P \subset C$ contains only those candidates for which $y_i = 1$. Similarly $\neg P \subset C$ contains only those candidates for which $y_i = -1$. Thus the ratio,

$$c_{\text{ratio}} = \frac{|P|}{|\neg P|}, \tag{1}$$

describes the imbalance between pulsar and non-pulsar candidates in the generated data. The ratio exhibited by real pulsar surveys varies from 1:7,500 (Keith et. al., 2010; Thornton, 2013) to 1:33,000 (Cordes et. al., 2006; Lazarus, 2012; ALFA Pulsar Consortium, 2015). For the prototype, we assume a ratio of 1:10,000 (0.0001). Empirical evidence suggests this figure to be representative (Lyon et. al., 2016), whilst also being challenging to deal with.

7.2. Prototype Outputs

Only a small fraction of candidates entering the prototype will be of scientific utility. Thus very few candidates making it through the pipeline will be flagged for i) archival storage, and/or ii) alert generation. As no archive or alert system exists, archival and alert generation will not be simulated. Instead we can analyse the filtering performance of the prototype using the auditing capabilities built into the design.

7.3. Data rate and Volume

Candidate data files are expected to be $\approx$ 2.2MB in size. This comprises a 262,144 sample data cube (128 bins, 64 channels, 32 sub-ints) using 8 bytes per sample, plus 56 bytes for the 6 additional variables (S/N, DM, RA, DEC, period, acceleration). The total amount of data passed through the prototype per observation (1,500 beams) is thus $\approx$ 3.3 TB. The self imposed 600 second time constraint implies the data must be processed at a rate of 5.5 GB/s, to maintain the 'real-time' property. All assumptions made thus far are summarised in Table 7.3.

Variable	Value
t_{obs}	600 seconds
n_{beam}	1,500 (SKA-Mid)
c_{beam}	1,000
c_{obs}	1,500,000
c_{size}	2.2 MB
c_{ratio}	0.0001
d_{rate}	5.5 GB/s
d_{volume}	3.3 TB

Table 1: Summary of assumptions made when designing the SDP prototype. Here t_{obs} is the observation time in seconds, n_{beam} the number of beams used per observation, c_{beam} the total number of candidates anticipated per beam, c_{obs} the total number of candidates anticipated per observation, c_{size} the size of an individual candidate, c_{ratio} the ratio of pulsar to non-pulsar candidates in SDP input data, d_{rate} the anticipated data rate per second, and finally d_{volume} which is the expected total data volume.

7.4. Sifting

Sifting is a batch processing procedure that aims to filter out duplicate detections of the same pulsar, occurring with slightly different observational parameters (DM, period, acceleration etc). Sifting is usually been done on a per beam basis[3]. Here all candidates obtained within a beam are compared to one another, potential matches flagged, and definite duplicates removed. Candidates surviving sifting are then passed on to the next stage of processing.

7.4.1. Traditional Sift

Standard sift is effectively an off-line matching problem. It determines whether or not two arbitrary candidates are equivalent, say c_i and c_k where $i \neq k$, using a similarity measure. We call this measure s for convenience. The similarity measure is usually associated with some decision threshold value t, applied over one or more variables that make up a candidate tuple. If the similarity value returned by this measure is above some threshold, it is typically considered a match. Otherwise tuples are considered disjoint.

The accuracy of the similarity measure can be quantified, if the ground truth is known for which tuples are matches. In such cases the performance of s can be quantified using a simple metric p, such as accuracy of the output matching,

$$\text{Matching Accuracy} = \frac{\text{Total matched correctly}}{|C'|}. \tag{2}$$

In reality we would want to use more complex measures than this, to capture the nuances of the problem. Using basic probability theory, we can describe the goal for a sifting function f more clearly. Suppose $P(c_i)$ is the prior probability of the tuple c_i having a match in C. Now suppose $P(\text{match}|c_i)$, is the probability of a match, conditioned on c_i (what is the chance of a match, given I've observed c_i). An optimal matching algorithm

[3]Candidates observed within the same beam of a radio telescope, which is pointed at a specific location on the sky.

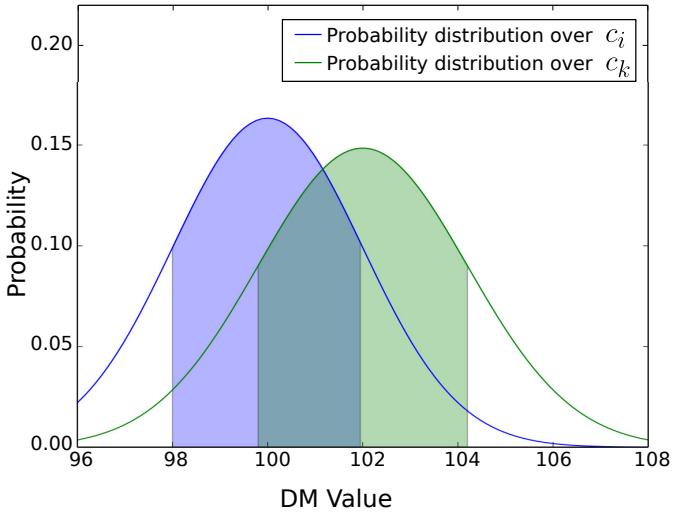

Figure 6: Here we see two separate probability distributions over DM value. The distribution for c_i is centred on its DM value of 100. It shows the probability of an arbitrary DM value being a match to its actual value of 100. Likewise for c_k, which has a DM value of 102. Here c_i and c_k have very similar DMs and are possibly a good match.

will do the following,

$$f(c_i) = \underset{\text{label}\in\{match,\neg match\}}{\text{argmax}} P(\text{label}|c_i). \tag{3}$$

This says $f(c_i)$ chooses the label that maximises the probability $P(\text{label}|c_i)$ (picks the most likely label). A function that does this well achieves high accuracy. However matching is not usually done in an explicitly probabilistic way for practical reasons. Generally heuristics over key variables (e.g. DM, period, acceleration etc) are used to constrain the matching via thresholds. For instance if the DM and period values of c_i and c_k, are within less than t of one another, they could be considered potential duplicates. Any threshold choice is effectively a probabilistic proxy, chosen to maximise $P(\text{label}|c_i)$.

For example, suppose we use a simple similarity measure s_1. Imagine that it considers c_i and c_k to be matches, if their DM values are within $t/2$ of one another (assuming t could be used in this way!). For this test, there are two underlying probability distributions. There is a distribution describing the probability of an arbitrary DM being considered the same as the DM of c_i, and the same goes for c_k. This is shown in Figure 6 which gives a contrived example.

These distributions could be used to estimate the probability of a value randomly falling in a predefined range. You could compute the joint probability $P(\text{DM in 68\% interval for } c_i) \times P(\text{DM in 68\% interval for } c_k)$, and use that to estimate the similarity. You then choose the most likely outcome based on the probability. In theory these sorts of estimates are the best we can do, in the absence of other useful information.

7.4.2. Traditional Sift Computational Complexity

The current 'best' sift approach does a brute force comparison of each c_i, to every c_k in C. This has a memory complexity of $O(n)$ given that all n candidates must be stored in memory. Run time is typically $O(n^2)$, quadratic complexity. We note that minor adjustments to the brute force approach can yield better runtime performance. Combinatorics tells us that a decreasing number of comparisons only need be done for each c_i. The total number of possible permutations for a set of length n, where k items are compared at a time, is given by,

$$\frac{n!}{(n-k)! \cdot k!}. \tag{4}$$

Technically speaking this approach is dominated by $O(n!)$ runtime complexity for all k, but in practice, for $k = 2$, it is dominated by $O(\frac{1}{2}(n-1)n)$ (rearranged original formula assuming $k = 2$). This yields a practical improvement in complexity.

7.4.3. Prototype Sift

For the purposes of the prototype, we have designed a new multi-beam sifting approach. The algorithm is able to sift one candidate at a time, as opposed to candidate batches. Its success is predicated on their being some way to sort candidates entering SDP. If candidates can be sorted by their pulse periods for example, then candidates within similar period ranges can be processed at the same bolts in the prototype topology. If similar candidates **always** arrive at the same bolts, we can use a simple frequency counting approach to check for duplicates. Such frequency counting allows us to check very quickly, if a period and DM combination has been observed before.

Using this approach, we employ simple logic. If a period has been observed in the same beam, it is likely a duplicate. Similarly if a candidate period has been seen before in another beam, with a similar DM (and other variables), it too is likely a duplicate. This way possible duplicates can be flagged or removed at this stage of the processing.

The general approach is summarised in Algorithm 1. This describes the code at a single bolt. It assumes Storm has partitioned the stream data, such that candidates arriving at the same node have similar periods (measured in μs). On lines 2-6 the basic variables required by the bolt are initialised. Here F is simply an array that maintains the frequency count of observed periods. The variable n maintains a count of the candidates observed. Note that as F is an array, it can be easily used for counting, providing the array indexes $0, \ldots, n$ can be mapped to individual periods or period ranges. For example, suppose an arbitrary bolt is counting observed periods between 1-2 seconds using 9 bins. Take the first bin which includes periods from 1.0 to 1.1 seconds. The periods counted in this range should go into $F[0]$. A value of 1.05 seconds has to mapped to the array index 0, so that $F[0]$ can be incremented. We use a scaling and a rounding operation to achieve this. This operation finds the correct bin index to increment for arbitrary period values (double values). The variables $floor$ and $celing$ help to do this. The scaling of the data is done on line 8, with the integer rounding operation done on line 9. As our prototype assumes μs periods, we must ensure that p_b is large enough to cover the desired period range to microsecond resolution (e.g. 1 second requires 10^6 bins).

Algorithm 1 Multi-beam Sift

Require: An input stream $C = \{..., (c_i), ...\}$, such that each c_i is a candidate, and c_i^j its j-th feature. Requires a similarity function s which can be user defined, the number of period bins p_b to use, the smallest period value expected at the bolt *min*, and the largest period value expected at the bolt *max*.

1: **procedure** MULTI-BEAM SIFT(C, s, p_b, min, max)
2: **if** F = null **then**
3: $F \leftarrow array[p_b]$ ▷ Init. counting array
4: $n \leftarrow 0$ ▷ Init. candidate count
5: $floor \leftarrow 0.0$
6: $ceil \leftarrow p_b$
7: $n \leftarrow n + 1$ ▷ increment observed count
8: $p \leftarrow ((ceil - floor) * (c_i^1 - min)/(max - min)) + floor$
9: $index \leftarrow (int)p$ ▷ Cast to int value
10: **if** $F[index] > 0$ **then**
11: F[index]++;
12: **return** s(true); ▷ Run similarity check, period seen
13: **else**
14: F[index]++;
15: **return** s(true); ▷ Run similarity check, period not seen

The if-else statement on lines 10-15 contains the only real logic in the algorithm. It checks if the period counter has been updated before for the current candidate's period. If it has, then we pass this information to the similarity function for checking. In reality the actual code is more complex. The algorithm itself can be greatly improved for added accuracy. For example, by maintaining frequency counts of multiple variables, and then estimating the probability of a match across all of them.

The weakness of this approach is that it is an incremental. Consequently the first candidates arriving at a sift bolt will not be flagged as duplicates. This is because no prior periods will have been observed. Furthermore the approach will treat duplicates with different S/Ns the same. However we would like to retain only the detection with the highest S/N, and discard the rest. The model described thus far can't retain the highest S/N candidate only. This is because we cannot know in advance if, or when, a higher S/N version of a candidate will enter the topology.

To mitigate these issues, we first assume that all candidates are ordered by S/N upon entering the topology on a per beam basis (c_i^0 is the ordering variable). Second, we apply a secondary sift post machine learning classification. This uses the predicted labels to assist with the sifting. During this secondary sift some aggregation of tuples can occur, allowing for more refined sifting comparisons to take place. Together these steps should ensure that most candidates are sifted correctly. However the approach is an approximation method - it is not as exact as running a batch sifting algorithm.

Feature	Description	Definition
$Prof_{\mu}$	Mean of the integrated profile P.	$\frac{1}{n}\sum_{\mathrm{i}=1}^{\mathrm{n}} p_{\mathrm{i}}$
$Prof_{\sigma}$	Standard deviation of the integrated profile P.	$\sqrt{\frac{\sum_{\mathrm{i}=1}^{\mathrm{n}}(\mathrm{p_i}-\bar{\mathrm{P}})^2}{n-1}}$
$Prof_{k}$	Excess kurtosis of the integrated profile P.	$\frac{\frac{1}{n}(\sum_{\mathrm{i}=1}^{n}(p_{\mathrm{i}}-\bar{P})^4)}{(\frac{1}{n}(\sum_{\mathrm{i}=1}^{\mathrm{n}}(p_{\mathrm{i}}-\bar{P})^2))^2}-3$
$Prof_{s}$	Skewness of the integrated profile P.	$\frac{\frac{1}{n}\sum_{\mathrm{i}=1}^{\mathrm{n}}(p_{\mathrm{i}}-\bar{P})^3}{(\sqrt{\frac{1}{n}\sum_{\mathrm{i}=1}^{\mathrm{n}}(p_{\mathrm{i}}-\bar{P})^2})^3}$
DM_{μ}	Mean of the DM-SNR curve D.	$\frac{1}{n}\sum_{\mathrm{i}=1}^{\mathrm{n}} d_{\mathrm{i}}$
DM_{σ}	Standard deviation of the DM-SNR curve D.	$\sqrt{\frac{\sum_{\mathrm{i}=1}^{n}(d_{\mathrm{i}}-\bar{D})^2}{n-1}}$
DM_{k}	Excess kurtosis of the DM-SNR curve D.	$\frac{\frac{1}{n}(\sum_{\mathrm{i}=1}^{n}(d_{\mathrm{i}}-\bar{D})^4)}{(\frac{1}{n}(\sum_{\mathrm{i}=1}^{\mathrm{n}}(d_{\mathrm{i}}-\bar{D})^2))^2}-3$
DM_{s}	Skewness of the DM-SNR curve D.	$\frac{\frac{1}{n}\sum_{\mathrm{i}=1}^{\mathrm{n}}(d_{\mathrm{i}}-\bar{D})^3}{(\sqrt{\frac{1}{n}\sum_{\mathrm{i}=1}^{\mathrm{n}}(d_{\mathrm{i}}-\bar{D})^2})^3}$

Table 2: The eight features derived from the integrated pulse profile $P = \{p_1, \ldots, p_{\mathrm{n}}\}$, and the DM-SNR curve $D = \{d_1, \ldots, d_{\mathrm{n}}\}$. For both P and D, all p_{i} and $d_{\mathrm{i}} \in \mathbb{N}$ for $i = 1, ..., n$.

7.5. Feature Extraction

Machine learning classification algorithms learn from data. The data must describe the concept being learned via examples, such that each example is described using numerical or textual descriptors known as features. Using features ML classifiers can learn to accurately separate data, provided the data is representative, and of sufficient descriptive power.

The prototype extracts 8 features from each candidate passing through the topology. These are described in full in Table 2. The first four are simple statistics obtained from the integrated pulse profile (folded profile) shown in plot (A) in Figure 7. The remaining four similarly obtained from the DM-SNR curve shown in plot (E) in Figure 7. These features are described in more detail elsewhere (Lyon, 2016; Lyon et. al., 2016).

7.5.1. Feature Generation Cost

The computational cost of generating the features is extremely low. The first four moments, mean, standard deviation, skewness and kurtosis, require minimal computational power to generate. The computational cost of generating each feature can be directly computed from the formulae in Table 2. This is achieved by counting the number of floating-point operations required to perform each arithmetic operation. If n represents the number of bins in the integrated profile and DM curve, then the cost is as follows:

- **Mean cost:** 2(n-1) + 4 floating point operations.
- **Standard deviation cost:** 3(n-1) + 9 floating point operations.
- **Skew cost:** 3(n-1) + 7 floating point operations.
- **Kurtosis cost:** 3(n-1) + 7 floating point operations.

This assumes the following costs: addition, subtraction and multiplication all require 1 FLOP, whilst division and square root calculations require 4. The Standard deviation calculation above, assumes the mean has already been calculated first. Likewise, the skew and kurtosis calculations assume the mean and standard deviation values have already been computed, and are simply plugged in. In a typical scenario, where we have 128 bins across the integrated profile and DM curve, the total cost per candidate is a mere 2,848 FLOP. If there are 1.5 million candidates returned per typical observation, calculating heuristics requires only 4.272 gigaflops[4] of computational capacity in total. A single modern laptop is capable of generating these heuristics in a few seconds. The computational runtime complexity of generating these features is $O(n)$, for a partially optimised algorithm that iterates over each item in the integrated profile and DM curve.

7.6. Machine Learning Classification

The prototype uses a tree-based data stream classifier, the GH-VFDT (Lyon et. al., 2014, 2016; Lyon, 2016). This classifier has been designed specifically for pulsar candidate selection over SKA scale data streams. It is an on-line algorithm, capable of learning incrementally over time. It is therefore able to adapt to changing data distributions, and incorporate new information as it becomes available. The GH-VFDT is extremely resource efficient, and has been designed to reduce resource use where possible.

The algorithm uses tree learning (see Duda et. al., 2000; Bishop, 2006; Russell & Norvig, 2009) to classify candidates. This involves the partitioning of input 'training data', using feature split point tests (see Figure 8) that aim to maximise the separation between pulsar and non-pulsar candidates. In practice this means choosing the variable that acts as the best class separator, and then finding a numerical threshold 'test point' for it that maximises class separability. This process repeats recursively, producing a tree-like structure as shown in Figure 8.

The memory complexity of the algorithm is $O(lf \cdot 2c)$ (sublinear in n), where l describes the number of nodes in the tree, f the number of candidate features used ($f = 8$ for the prototype, see Section 7.5), and finally c number of classes (here $c = 2$, pulsar & non-pulsar). The runtime complexity of the algorithm is difficult to quantify, as it is often data dependent. However it is of the order of $O(n)$.

[4] A gigaflop is a billion floating-point operations.

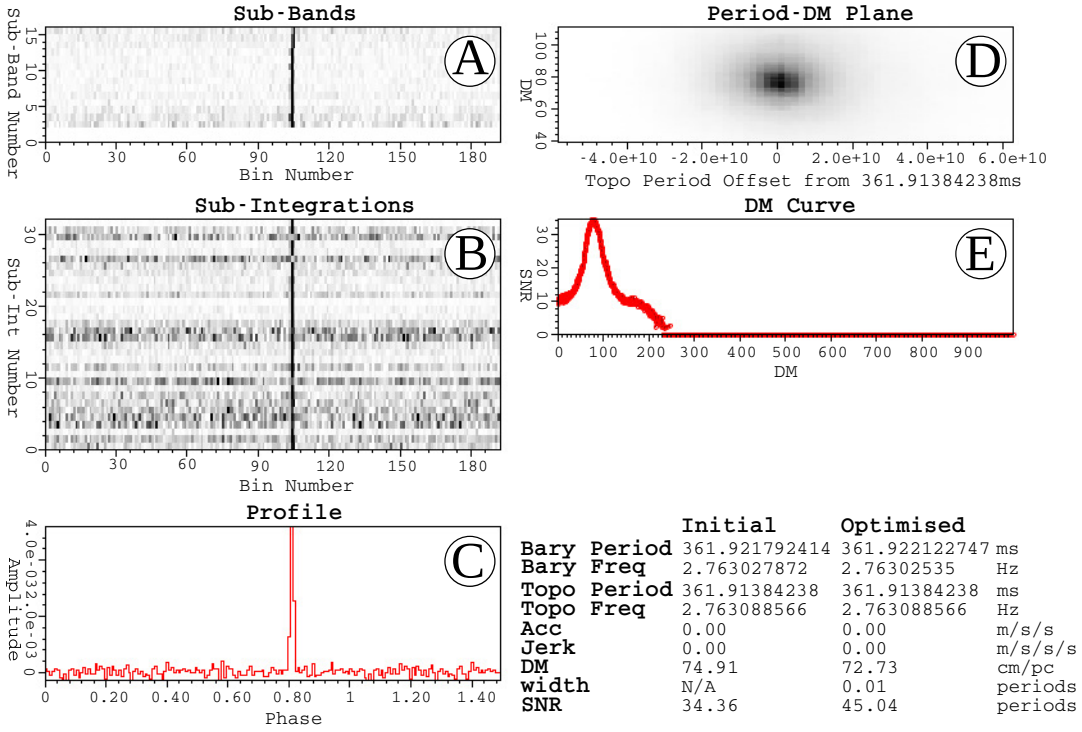

Figure 7: An annotated example candidate summarising the detection of PSR J1706-6118. The candidate was obtained during processing of High Time Resolution Universe Survey data by Thornton (2013).

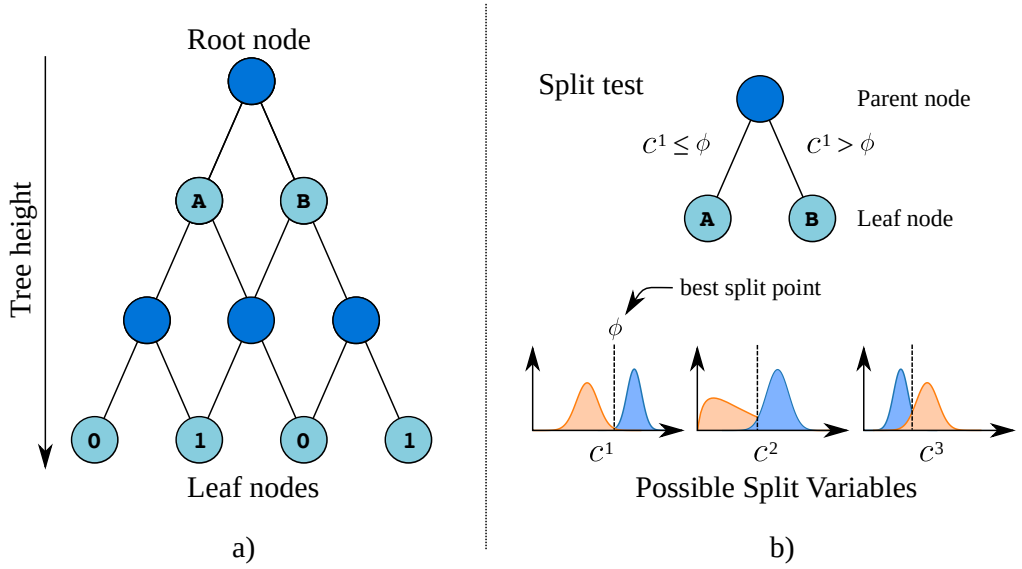

Figure 8: An overview of how a decision tree partitions the data space using binary split point 'tests' at each node. The best feature variable at each node is first determined, then an optimal numerical split point threshold chosen. Candidates with feature values below the threshold are passed down the left hand branch of the tree, and possibly subjected to further split tests. Similarly for candidates with feature values above the threshold, except these are passed down the right hand branch. Eventually candidates reach the leaf nodes, where they are assigned class labels.

7.7. Post Classification Sift

Post-classification sifting aims to remove duplicate candidate detections, in lieu of assigned machine learning classifications. If two candidates reaching such a node had been assigned the same classification label (pulsar or non-pulsar), and have very similar period and DM values (other variables can be considered), then they are likely duplicates. In this case, only the strongest detection need be forwarded on through the topology. For now the functionality of the bolt in the topology is limited, but it has been included to enable improvement in the future. The plan is to implement a sliding window over the tuples reaching the node. Thus enables a primitive form of caching. Only after the window moves do the tuples exiting the window get forwarded. Even then, this will only happen if they are not duplicates of any of the examples in the sliding window.

7.8. Known Source Matching

Known source matching involves determining which detections correspond to known pulsar sources. It requires a set K of known sources, usually obtained from the ATNF pulsar catalogue. Each individual source in K is defined as $k_i = \{k_i^1, \ldots, k_i^m\}$, with each tuple uniquely identifiable in via the index i. As before for candidates, all $k_i^j \in \mathbb{R}$, with the meaning of each k_i^j the same as for candidates.

Known source matching is similar to sifting. The goal is to compare each candidate surviving machine learning filtering, to the known sources in K. A brute force comparison approach compares each candidate c_i to those in K. This corresponds to a runtime complexity of $O(n \cdot m)$. As new pulsars (and other possible radio sources) continue to be found over time, m is gradually increasing. The brute force method is thus computationally expensive, given an increasing m and an unbounded n. Though it does not practically reach quadratic complexity, as $m \ll n$. Nonetheless as there are ≈ 2400 known sources, a brute force algorithm will undertake many comparisons for each c_i. The memory complexity of known source matching is $O(m)$, as each known source is typically stored in memory to facilitate fast comparison operations.

7.8.1. Improved Matching

We have designed a fast known source matching algorithm. It utilises a divide and conquer programming approach. This attempts to recursively divide the matching space, greatly reducing the number of comparisons required. It relies on an ordering applied over the set K. In particular it requires a total ordering of elements in K under $\leq$. For all k_i, k_{i+1}, k_m, where $m > i + 1$,

$$\text{if } k_i \leq k_{i+1} \text{ and } k_{i+1} \leq k_i \text{ then } k_i = k_{i+1}, \tag{5}$$

$$\text{if } k_i \leq k_{i+1} \text{ and } k_{i+1} \leq k_m \text{ then } k_i \leq k_m, \tag{6}$$

$$k_i \leq k_i, \tag{7}$$

$$k_i \leq k_{i+1}. \tag{8}$$

Here Equation 5 defines the antisymmetry property, Equation 6 defines the transitivity property, Equation 7 defines the reflexive property, and Equation 8 defines the totality property.

To apply the desired ordering, we must obtain a single numerical value per known source, that satisfies Equations 5-8. This has been achieved via a simple mathematical manipulation of the on sky RA and DEC coordinates of a known source. Consider first the RA component of a known source's coordinates. The RA varies between 0 hours ($00^h\ 00^m\ 00^s$), through to 24 hours ($23^h\ 59^m\ 59.\dot{9}^s$). This can be expressed as a single numerical value in seconds,

$$\text{RA value} \quad = \quad (h \times 3600) + (m \times 60) + s. \tag{9}$$

According to this formula, an RA of $00^h\ 00^m\ 00^s = 0$, whilst an RA of $23^h\ 59^m\ 59.\dot{9}^s = 86,400$. Implicitly, similar RA values will obtain similar numerical values. For the DEC component of a known source's coordinates we perform the same computation out of convenience. The DEC varies between $-90°$ ($-89°$ $59'$ $59.\dot{9}''$) and $90°$ ($+89°$ $59'$ $59.\dot{9}''$).

$$\text{DEC value} \quad = \quad (d^\circ \times 3600) + (m' \times 60) + s''. \tag{10}$$

The two values are then summed, producing a single numerical value. The simple nature of this computation can lead to some issues. For example, consider a coordinate with RA $03^h\ 00^m$ 00^s and DEC $00°\ 00'\ 00''$. This gives a sum of $10,800$. Whilst a coordinate with RA $00^h\ 00^m\ 00^s$, and DEC $03°\ 00'\ 00''$, also gives a sum of $10,800$. Clearly these are not close together on the sky, but obtain the same values. This does not matter. The sort variable is only used to guide the matching process to reduce the search space. It need not be exact for our purposes.

Whilst the sort variable has zero astronomical meaning, it is useful for speeding up the matching process. By computing the sort variable for an arbitrary c_i, we can compare its on sky location against the known sources in K. The candidate c_i need only be compared to those elements in K which are nearby, i.e. with similar sort values.

The comparison process divides up the search space recursively, until a suitable search position in an ordered K is found. This is achieved as follows: if the sort variable of c_i is less than the sort variable of the centre source in K, search only the first half of K. Else, search only the second half of K. This process is repeated until the closest single match is found. Once the closest match is found, the algorithm compares c_i to it. It then compares c_i to a small number of sources to the left, and right, of the closest match. This process is summarised in Figure 9.

Precisely how many comparisons are done to the left and right of the best match is up to the user. However for the prototype, an angular separation $\theta = 1.5°$ is used to determine this. Comparisons are not undertaken if the angular distance between c_i and k_i exceeds θ.

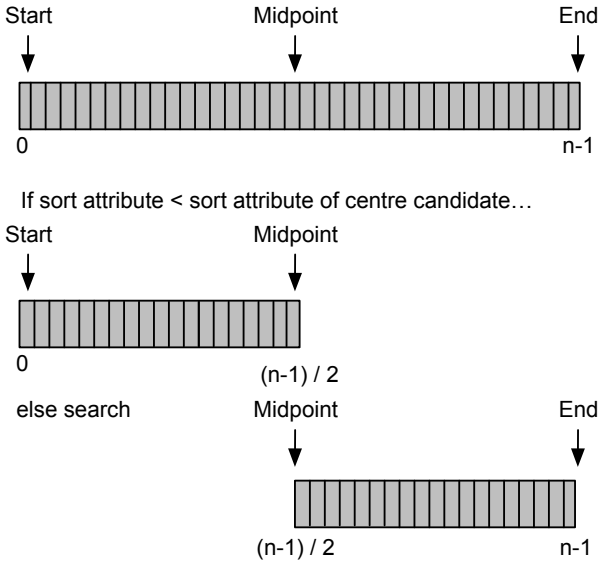

Figure 9: The divide and conquer approach to known source matching. It continually dives the search space into smaller chunks. When the chunks become too small to sub-divide, a search index is obtained. The algorithm will begin matching a candidate against known sources from that search index. Comparisons extend to the left and right of the search index, but only up until some level of user specified angular separation θ.

The search algorithm is presented in Algorithm 2, with the matching procedure used shown in Algorithm 3. The matching procedure compares the period and DM of a promising c_i, to some potential match k_i. The known source k_i is considered a possible match for c_i, only if their period and DM values are similar to within a user specified error margin. It is defined by the variable $e_{margin} \in [0, 1]$. For example, an $e_{margin} = 0.1$ corresponds to a 10% error margin. Here we would consider a known source to be possible match, only if its period and DM are within 10% of the candidate's (±5% of the candidate's values).

The computational complexity of the overall approach is $O(n \cdot \tau)$, where τ is a proxy for the number of comparisons made between known sources and candidates based on θ. The modified runtime is practically speaking linear in n (as τ is usually small). Table 3 illustrates how the number of comparisons increases as τ grows. Note the final row in the Table corresponds to the worst case, when the algorithm achieves the same performance as the brute force method. However as θ is chosen to limit comparisons to only nearby objects, τ never gets this large thus worst case performance is unlikely (unless choosing to match against the whole sky). To illustrate the difference between the two approaches, we matched 1000 candidates to 2000 known sources, using both approaches (using hardware described in Section 8.1). The brute force method took 5 minutes to complete. Whilst the optimized approach took only 10 seconds, achieving the same results.

Algorithm 2 Divide & Conquer Matching

Require: A set of known sources $K = \{k_i, \ldots, k_m\}$, the numerical *sort* variable of the candidate to compare c_i. Finally i_{start} is the index to search from in K, and i_{end} is the index to search to.

procedure DIVIDE($i_{start}, i_{end} K$, *sort*)
  if $i_{end} - i_{start} == 1$ **then**
    return i_{start} ▷ Only 1 source
  else if $i_{end} - i_{start} == 2$ **then**
    return $i_{start} + 1$ ▷ Only 2 sources, arbitrarily pick last
  else
    $i_{middle} \leftarrow$ int(ceil($(i_{end} + i_{start})$ / 2.0)) ▷ Middle index
    $k_i \leftarrow K[i_{middle}]$ ▷ Get middle source
    if $sort < k_i.sort$ **then**
      return Divide($i_{start}, i_{middle} K$, *sort*) ▷ Search 1st half
    else if $sort > k_i.sort$ **then**
      return Divide($i_{middle}, i_{end} K$, *sort*) ▷ Search 2nd half
    else
      return i_{middle} ▷ Found search location.

Algorithm 3 Matching Procedure

Require: A known source k_i, a candidate c_i, an angular separation used for matching θ, and an accuracy level for period and DM matching $e_{margin} \in [0, 1]$.

procedure ISMATCH($k_i, c_i, \theta, e_{margin}$)
  $c_p \leftarrow c_i$ ▷ Get period from candidate
  $c_{dm} \leftarrow c_i$ ▷ Get DM from candidate
  $k_p \leftarrow c_i$ ▷ Get period from known source
  $k_{dm} \leftarrow c_i$ ▷ Get DM from known source
  $p_{diff} \leftarrow (e_{margin} \times c_p)/2$
  $dm_{diff} \leftarrow (e_{margin} \times c_{dm})/2$
  $hms \leftarrow [1, 0.5, \ldots, 0.03125]$ ▷ Harmonics to look for
  for $h \leftarrow 0$, h++, while $h < |hms|$ **do**
    if $c_p > (k_p * hms[h]) - p_{diff}$ **then**
      if $c_p < (k_p * hms[h]) + p_{diff}$ **then**
        if $c_{dm} < k_{dm} + dm_{diff}$ **then**
          if $c_{dm} > k_{dm} - dm_{diff}$ **then**
            $sep \leftarrow calcSep(k_i, ci)$
            if $sep < \theta$ **then**
              $possibleMatch(k_i, ci)$

τ	Comparisons
1	10,000
10	100,000
50	500,000
100	1,000,000
1000	10,000,000
2000	20,000,000

Table 3: The increase in comparison operations as τ increases, when using a fixed $n = 10,000$ and $m = 2,000$. The first row describes optimal performance, when each c_i is compared to only one known source. The final row corresponds to the worst case, when the algorithm achieves the same performance as the brute force method. However as θ (angular separation) is chosen to limit comparisons to only nearby objects, τ never gets this large.

Machine	Instances	Instance Type	CPU (equivalent)	ECUs	RAM (GB)	Cores
Zookeeper	1	t2.micro	1 x 2.5 GHz Intel Xeon	variable	1	1
Nimbus	1	m4.xlarge	1 x 2.4 GHz Intel Xeon E5-2676v3	13	16	4
Workers	4	c4.2xlarge	1 x 2.9 GHz Intel Xeon E5-2666v3	31	16	8
Workers	8	m4.xlarge	1 x 2.4 GHz Intel Xeon E5-2676v3	13	16	4
TOTAL	14	-	-	241	209	69

Table 4: Summary of the cloud instances deployed to AWS

7.9. Alert Generation

The alert generation bolt within the prototype has limited functionality. It does not generate alerts, since the nature of SDP alerts, and the alert processing infrastructure, is yet to be decided. Thus the node simply acts as a processing unit that slows down computation as though alerts were being generated.

7.10. Auditing

The topology has been designed to accommodate auditing nodes. Each bolt is able to generate auditing information, so that the performance of the network can be evaluated. Auditing can be accomplished in two ways. There is auditing done on a per node basis. Here filtering accuracy and runtime performance is measured for specific type of node only. Whilst an end-to-end auditing measures filtering accuracy and runtime performance at the end of the pipeline. End-to-end auditing does not impact runtime, and so is used to measure prototype performance. Per node auditing adds a small additional computational overhead to the node being audited, however this type of auditing is only intended for use during development and debugging.

8. Simulations

Two forms of simulation were planned for this work. The first is a small-scale simulation undertaken on a single machine, useful for testing and debugging the topology design and processing code. The second is a larger scale deployment of the topology to the Amazon Cloud. The second simulation is intended to assess the scalability of the system, and determine how easy it is to deploy in practice. In both scenarios the goal was to recreate the delivery of data from CSP to SDP, during a plausible SKA search scenario.

8.1. Local Cluster

Here simulations were undertaken on single a machine running OSX 10.9 (Mavericks). The machine possessed a single 2.2 GHz Quad Core mobile Intel Core i7-2720QM Processor (8 threads), with a peak theoretical performance of 70.4 GFLOPs (Intel Corporation, 2013). It was equipped with 16 GB of DDR3 RAM, and two 512GB Crucial MX100 solid state drives. A Storm cluster (version 0.95) was deployed on this machine, and the SDP processing topology executed in local cluster mode. This enabled testing of the framework prior to a larger scale deployment.

8.2. Cloud Infrastructure

We were awarded compute time on Amzaon's cloud infrastructure, as part of the SKAO-AWS AstroCompute grant programme[5]. This time has been used to test the performance and behaviour of the Storm framework, when scaled beyond a single machine. Using the Amazon Web Services (AWS) console, we provisioned a number of Amazon Compute Cloud (EC2) instances[6]. The provisioned instances are described in Table 7.9. Note that it is difficult to estimate the overall compute capacity these resources amount to. This is because EC2 instances are deployed on shared hardware resources, subject to load balancing policies and stress from other EC2 users. Amazon describes the compute capacity of its virtual instances in terms of EC2 Compute Units (ECUs). According to Amazon's documentation, a single ECU corresponds to a 1.0-1.2 GHz 2007 Intel Xeon Processor. To map this ECU unit to a meaningful value, consider the the 'slowest' (lowest clock speed) Intel Xeon processor available in 2007, the Intel Xeon E7310. This CPU possesses 4 cores, performs 4 operations per cycle, and has a clock speed of 1.6 GHz. To estimate how many Floating-point operations per second (FLOPs) this processor is capable of, we use the formula,

$$\text{FLOPs} = \text{sockets} \cdot \frac{\text{cores}}{\text{sockets}} \cdot \text{clock} \cdot \frac{\text{operations}}{\text{cycle}}. \quad (11)$$

The Xeon E7310 (1 ECU) is capable of a theoretical throughput of approximately 25.6 GFLOPs, according to both Equation 11 and Intel's own export specifications[7]. The 241 ECUs used during cloud experimentation, correspond to an approximate computational capacity of 6.2 TFLOPs. This is an approximate estimate only and is subject to error.

8.2.1. Storm Cluster Initial Configuration

Each new instance provisioned via AWS, consisted of a clean operating system image running Ubuntu Server 14.04 LTS. Prior to setting up the components of the Storm system upon the instances, Java version 1.8.0_101 was firstly installed on each of them manually, via the terminal[8]. The respective 'hosts' file of

[5] See `https://aws.amazon.com/blogs/aws/new-astrocompute-in-the-cloud-grants-program/` .

[6] These are simply virtual machines.

[7] See `http://www.intel.com/content/dam/support/us/en/documents/processors/xeon/sb/xeon_7300.pdf`.

[8] To access the provisioned machines via SSH, a security certificate must be obtained from AWS. This certificate key is generated when setting up AWS user accounts. Once the key has been stored in a safe location on a local machine, the command *ssh -i <certificate> ubuntu@<instance fully qualified domain name>* can be used to access the virtual machine.

each instance (*/etc/hosts*) was then modified, so that each machine could access any other via pre-determined host names.

Listing 1: Sample hosts file.

```
54.129.69.6  ec2-54.eu-west-1.aws.com  zkserver1
54.154.68.2  ec2-53.eu-west-1.aws.com  nimbus1
54.194.93.1  ec2-52.eu-west-1.aws.com  worker1
```

8.2.2. Zookeeper Install

Following host file modification, Apache Zookeeper was installed on a single EC2 instance. This instance possessed modest computational capabilities as described in Table 4. To install Zookeeper the following command was used.

Listing 2: Zookeeper installation command.

```
sudo apt-get install zookeeperd
```

A Zookeeper configuration must then be created at */etc/zookeeper/conf/zoo.cfg*, and saved with the following contents:

Listing 3: The Zookeeper configuration file.

```
tickTime=2000
dataDir=/var/lib/zookeeper
clientPort=2181
autopurge.purgeInterval=6
autopurge.snapRetainCount=3
```

This defines the communication port used by Zookeeper, and ensures that log files are purged regularly. After the configuration file has been correctly modified, the Zookeeper daemon can be started with the following command.

Listing 4: The Zookeeper start command.

```
sudo start zookeeper
```

8.2.3. Nimbus, UI & Supervisor Install

With Zookeeper correctly configured, it remains to setup the Nimbus and supervisor nodes that comprise the Storm cluster. Firstly, a new Storm user account must be created on each of the Nimbus and worker nodes. This enables the Storm cluster to execute under a user account with the correct permissions. This prevents user account problems from halting execution. To add the new user, the commands provided in Listing 5 were executed at the terminal sequentially, on all instances (except the Zookeeper instance).

Listing 5: Storm user setup command.

```
sudo adduser --system storm
sudo groupadd storm
sudo usermod -a -G storm storm
sudo chown -R storm:storm /home/storm
sudo chmod 775 /home/storm
```

After user account creation, the Storm system is ready to be installed. Storm version 0.95 was downloaded to each node[9] using the commands shown in Listing 6.

[9]Note that the version of Storm used to develop the topology, and the version of Storm running on the cluster, **must** be the same.

Listing 6: Storm download command.

```
cd /home/storm
sudo wget http://www.apache.org/dyn/closer.lua/storm
    /apache-storm-0.9.5/apache-storm-0.9.5.tar.gz
sudo tar -xvzf apache-storm-0.9.5.tar.gz
```

Once downloaded and unpacked, the Storm configuration file must be updated. This tells Storm which machines host the Zookeeper and Nimbus installations respectively. To edit the configuration file, enter the commands shown in Listing 7.

Listing 7: Storm configuration file edit command used.

```
sudo rm /home/storm/apache-storm-0.9.5/conf/storm.
    yaml
sudo nano /home/storm/apache-storm-0.9.5/conf/storm.
    yaml
```

Once the file has been opened, enter the contents of Listing 8 into it. The configuration file firstly describe the names of the Zookeeper and Nimbus servers, which should also be listed in the hosts file modified earlier. The Line containing the text,

```
nimbus.thrift.max_buffer_size: 4048576
```

describes the maximise buffer size in bytes, used by Nimbus and Storm to transmit messages. If a single tuple, bolt, or the topology itself is larger than the buffer size, then data cannot be transmitted. in such cases the Storm deployment will encounter communication errors. The buffer size used here is a shade over 4MB, which is much larger than the default value. A higher value is used to accommodate bolts which maintain lists of known sources, which have an implicitly larger memory footprint on disk, leading to greater network transfer overheads. The section beginning *supervisor.slots.ports* describes which ports to use on each node for computational work. Here 16 ports are listed. A supervisor instance using this configuration file, will listen for work on a maximise of 16 ports. This corresponds to 16 slots per worker node for task execution.

Listing 8: Storm configuration file contents.

```
########### These MUST be filled in!
storm.zookeeper.servers:
 - zkserver1
nimbus.host: nimbus1

nimbus.thrift.max_buffer_size: 4048576
supervisor.slots.ports:
 - 6700
 - 6701
 - 6702
 - 6703
 - 6704
 - 6705
 - 6706
 - 6707
 - 6708
 - 6709
 - 6710
 - 6711
 - 6712
 - 6713
 - 6714
 - 6715
```

After editing the Storm configuration file, a start-up script file is created on each instance. There are three types of script, and

each allows Storm cluster components to be started and stopped conveniently. The script you create is dependent upon the job you wish that instance to undertake. The scripts include,

```
/etc/init/nimbus.conf
/etc/init/ui.conf
/etc/init/supervisor.conf
```

The first allows Nimbus to be started automatically, thus should be created on the Nimbus instance. The contents of the configuration file are shown in Listing 9.

Listing 9: Nimbus start file.

```
start on runlevel [2345]
stop on runlevel [^2345]
console log
chdir /home/storm
setuid storm
setgid storm
respawn
# respawn up to 20 times within 5 seconds.
respawn limit 20 5
exec /home/storm/apache-storm-0.9.5/bin/storm nimbus
```

The next start-up file allows the Storm user interface console (Strom UI) to be started. The Storm UI is a web-based interface accessible via the internet. The UI is usually hosted on the same machine as Nimbus for convenience, and should be created there. A screen shot of the working UI is shown in Figure 10. The interface can be accessed when the UI is started via http://<host>:8080/index.html.

Listing 10: Storm UI start file.

```
start on runlevel [2345]
stop on runlevel [^2345]
console log
chdir /home/storm
setuid storm
setgid storm
respawn
# respawn up to 20 times within 5 seconds.
respawn limit 20 5
exec /home/storm/apache-storm-0.9.5/bin/storm ui
```

The final start-up file is used to begin executing supervisors, which perform the bulk of the computational work within the Strom cluster. Such a file should therefore be created on every instance which will run a Storm supervisor.

Listing 11: Storm supervisor start file.

```
start on runlevel [2345]
stop on runlevel [^2345]
console log
chdir /home/storm
setuid storm
setgid storm
respawn
# respawn up to 20 times within 5 seconds.
respawn limit 20 5
exec /home/storm/apache-storm-0.9.5/bin/storm
    supervisor
```

The components of a Storm cluster can be started and stopped via the commands,

```
sudo start <component>
sudo stop <component>
```

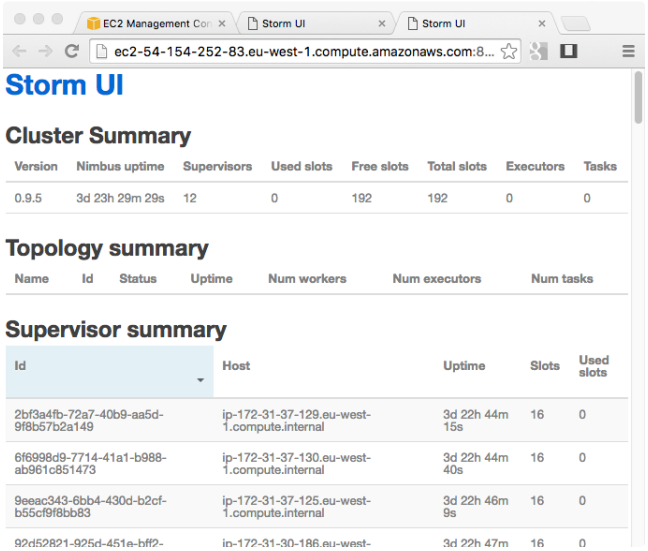

Figure 10: A screen shot of the Storm UI.

where <component> is replaced by “nimbus”, “ui”, or “supervisor” as appropriate. Note that these Storm installation steps were undertaken on 13 nodes in total as described in Table 7.9.

9. Evaluation

9.1. Performance

Performance of the topology has been measured using auditing nodes. These maintain a number of statistics which are updated after each tuple is processed. The statistics maintained are listed in Table 4. They enable runtime performance and filtering accuracy to be computed at each individual bolt in the topology. Thus we can determine how long each filtering process (feature extraction, sifting etc) took to complete per candidate. Overall end-to-end performance, is determined using the time stamp attached to each tuple upon entering the topology, and the time stamp attached at the end of the topology. This allows an estimation of the time taken for an individual tuple to move from input spouts to alert bolts. Aggregate results are then obtained by averaging over all tuples reaching the end of the topology.

Note that performance results are linked to the precise topology configuration used. The configuration describes the number of spouts and bolts at each layer in the topology. We adopt a colon delimited notation to describe the configuration in terms of layers, e.g. $1^{1...*} : 2^{1...1} : 1$. This corresponds to one input spout in layer 1, two processing bolts in layer 2, and a lone bolt in layer 3. The superscript defines the cardinality relation between the current layer and the next. For this example, there is a one-to-many relation between the spout and the bolts in the second layer, and a one-to-one relation between the bolts in the second layer, and the bolt in the third.

9.2. Filtering Accuracy

As the spouts in our topology generate candidate data based upon pre-existing examples, the true class label of all candi-

Metric	Description
c_{rec}	The total number of candidates received
c_{pro}	The total number of candidates processed
c_{forw}	The total number of candidates forwarded
t_{tot}	The total tuple processing time in milliseconds
t_{avg}	The average tuple processing time in milliseconds
obs_{real}	Positive pulsar candidates observed
obs_{false}	Non-pulsar candidates observed
f_{pos}	Positive (real) pulsar candidates observed
f_{false}	Non-pulsar candidates forwarded

Table 5: The variables maintained and monitored by auditing nodes in the topology.

		Filtering Prediction	
		-	+
Actual	-	True Negative (TN)	False Positive (FP)
	+	False Negative (FN)	True Positive (TP)

Table 6: Confusion matrix describing the outcomes of binary classification.

dates entering the topology is known *a priori*. It is therefore possible to evaluate a filtering decision over any candidate, at any bolt or spout. There are four possible outcomes for a filtering decision, assuming pulsars are considered to be positive (+), and non-pulsars negative (-). These outcomes are summarised in Table 6. A **true** negative/positive, is a negative/positive candidate **correctly** filtered. Whilst a **false** negative/positive, is a negative/positive candidate **incorrectly** filtered. It is desirable to have as few false negative/positive outcomes as possible. The outcomes listed in Table 5 are not typically evaluated independently. Rather they are evaluated using standard metrics, such as those outlined in Table 7.

These metrics are computed at the auditing nodes, allowing filtering accuracy to be continuously monitored. These metrics are actually equivalent to those used to evaluate machine learning classification accuracy. They have therefore been well studied in the machine learning domain. Other metrics have been included in the prototype (e.g. Kohen's kappa, and the Mathews' Correlation Coefficient), however for brevity we do not discuss these here. The subset described in Table 7 are sufficient to establish an impression of filtering accuracy.

10. Results

10.1. Local Cluster

The results of local cluster experimentation, are presented in Tables 7 and 8. The prototype pipeline is able to process 1.5 million pulsar candidates in well under 600 seconds, as shown in Table 7. It is able to process candidates at a very fast rate, exceeding 1,000 per second (upto 6,000 per second in the largest tests). Thus the prototype is capable of running at the desired SKA scale, on present day commodity hardware. It is worth highlighting that this was accomplished on a single laptop, possessing only 70.4 GFLOPs (Intel Corporation, 2013) of computational capacity (theoretical max[10]).

Statistic	**Description**	**Definition**
Accuracy	Measure of overall filtering accuracy.	$\frac{(TP+TN)}{(TP+FP+FN+TN)}$
False positive rate (FPR)	Fraction of non-pulsar instances incorrectly filtered.	$\frac{FP}{(FP+TN)}$
G-Mean	Imbalanced data metric describing the ratio between positive and negative accuracy.	$\sqrt{\frac{TP}{TP+FN} \times \frac{TN}{TN+FP}}$
Precision	Fraction of forwarded instances correctly filtered.	$\frac{TP}{(TP+FP)}$
Recall	Fraction of forwarded instances that belong to real pulsars.	$\frac{TP}{(TP+FN)}$
F-Score	Measure of accuracy that considers both precision and recall.	$2 \times \frac{precision \times recall}{precision+recall}$
Specificity	Fraction of non-pulsar candidates correctly filtered.	$\frac{TN}{(FP+TN)}$

Table 7: Standard evaluation metrics for filtering performance. True Positives (TP) and True Negatives (TN) are those candidates correctly filtered. False Positives (FP) and False Negatives (FN) are those incorrectly filtered. All metrics produce values in the range [0, 1].

The prototype's filtering accuracy is reasonable. It consistently achieves 84% accuracy when using a dumb sifting approach (a 50:50 sift, that propagates 50% of tuples chosen at random). This compares favourably with the accuracy of some recent machine learning methods used for candidate filtering (for more details see Lyon et. al., 2016; Lyon, 2016). The accuracy of the prototype increases when using the distributed sift algorithm. This successfully removes most uninteresting candidates and duplicate detections. A 99% accuracy level is achieved using distributed sift, with a good level of pulsar recall (81%). This recall rate is not really high enough for use with the SKA, as if applied, it would miss almost 20% of pulsars. However this prototype is but a first attempt at building a single SDP pipeline for pulsar search. There is room for improvement, and some modifications have already been mentioned in earlier sections.

Note that for the random sift approach, the average processing time per tuple, decreases as more data is processed. This is likely due to Storm increasing the scalability of the topology automatically, in response to increased processing demand. For distributed sift, which uses a slightly different topology configuration[11], the runtime per tuple increases as more data is pro-

[10] In practice this theoretical max cannot be achieved. After taking into account operating system and Storm overheads, the true computational capacity available for processing is less than 70.4 GFLOPs.

[11] There are 11 nodes in the sifting layer, to accommodate the period distribution splits required by distributed sift.

Configuration	c_{obs}	Acc.	Recall	F1	t_{avg}(ms)	t_{tot} (s)	Notes
$1^{1...*}:4^{1...*}:4^{1...*}:4^{1...*}:4^{1...*}:4^{1...*}:4^{1...1}:1$	100,000	.842	.882	. 357	1078	100	Random sift
$1^{1...*}:4^{1...*}:4^{1...*}:4^{1...*}:4^{1...*}:4^{1...*}:4^{1...1}:1$	200,000	.842	.883	. 358	601	150	Random sift
$1^{1...*}:4^{1...*}:4^{1...*}:4^{1...*}:4^{1...*}:4^{1...*}:4^{1...1}:1$	500,000	.842	.883	. 359	713	360	Random sift
$1^{1...*}:4^{1...*}:4^{1...*}:4^{1...*}:4^{1...*}:4^{1...*}:4^{1...1}:1$	1,000,000	.842	.883	. 358	261	360	Random sift
$1^{1...*}:4^{1...*}:4^{1...*}:4^{1...*}:4^{1...*}:4^{1...*}:4^{1...1}:1$	1,500,000	.842	.883	. 358	124	510	Random sift
$1^{1...*}:4^{1...*}:11^{1...*}:4^{1...*}:4^{1...*}:4^{1...*}:4^{1...1}:1$	100,000	.999	.811	.771	44	38	Distributed sift
$1^{1...*}:4^{1...*}:11^{1...*}:4^{1...*}:4^{1...*}:4^{1...*}:4^{1...1}:1$	200,000	.999	.811	.771	23	60	Distributed sift
$1^{1...*}:4^{1...*}:11^{1...*}:4^{1...*}:4^{1...*}:4^{1...*}:4^{1...1}:1$	500,000	.999	.811	.771	59	80	Distributed sift
$1^{1...*}:4^{1...*}:11^{1...*}:4^{1...*}:4^{1...*}:4^{1...*}:4^{1...1}:1$	1,000,000	.999	.810	.770	58	120	Distributed sift
$1^{1...*}:4^{1...*}:11^{1...*}:4^{1...*}:4^{1...*}:4^{1...*}:4^{1...1}:1$	1,500,000	.999	.811	. 771	100	225	Distributed sift

Table 8: Performance and filtering accuracy results for the prototype pipeline run on a local cluster. Here c_{obs} is the total number of candidates entering the topology, t_{avg}(ms) the time taken on average to process a tuple, and t_{tot} (s) the total time for the pipeline to process all candidates. Results listed for pipelines using random sift (randomly make sift decision with 50:50 split), and full distributed sift. For all tests a candidate ratio of $c_{\text{ratio}} = 0.05$ was used. See Section 9.1 for details on how to interpret the configuration used.

Bolt	Acc.	Recall	F1 Score	t_{avg}(ms)
Preprocessing	-	-	-	1
Distributed Sift	.991	.599*	.371	2
Feature Extraction	-	-	-	2
ML Classification	.844	.880	.358	2
Known Source matching	-	-	-	7
Alert generation	-	-	-	1

Table 9: The runtime and filtering performance of individual bolts. Accuracy values are only provided for those bolts filtering the data. Known source matching has no filtering accuracy results, as the data used during candidate generation has not been matched to known sources. Here c_{obs} is the total number of candidates entering the bolt, and t_{avg}(ms) the time taken on average to process a tuple. Experiments run using $c_{\text{obs}} = 50,000$, and $c_{\text{ratio}} = 0.05$ (produces many duplicates for sifting). * This sift returned 133 out of 2480 real pulsars injected, of which there are only 222 unique examples. Of the 50,000 candidates entering the system during such test, 180 made it to alert generation, of which 115 are definite pulsars.

cessed. However the average remains much lower than for random sift.

The individual bolts in the topology appear to be runtime efficient. Each tuple required on average, only a few milliseconds of processing time at each bolt, as shown in Table 8. Given the average tuple processing times shown in Table 7, there appears to be a disparity. If tuples are only processed for a few milliseconds at the bolts, why is their total processing time so high by comparison? The difference is likely due to the tuple transmission time from bolt to bolt. This indicates that transmission time plays a significant role in the total runtime of a Storm based SDP prototype.

10.2. Cloud Infrastructure

Cloud infrastructure testing results are presented in Tables 10 and 11. Table 10 shows topology performance according to the *total time* it takes to process a tuple on average. There a two sets of results in Table 10. These describe the performance when using random 50:50 sift, and when using distributed sift. In both cases the total number of ECUs and bolts in the topology are indicated.

The results in Table 10 show an increase in processing time, when initially beginning to scale the topology (when using 1-4 workers). This is followed by a decrease in processing time when more workers are added, which increases the computational power used to execute the topology. This result is observed for both the random sift, and distributed sift topologies.

The initial increase in processing time observed when beginning to scale the topology, is explainable by the results shown in Table 11. These describe the performance of individual bolts, as they are scaled within a topology. When a topology contains only a few instances of each type of bolt, these bolts generally reach their maximum computational capacity quickly. This causes an overall increase in processing time due to resource contention, as observed in Table 10. As more bolts are added to the topology, counter-intuitively, the contention is not necessarily reduced. Whilst some bolts will begin to cope with the load when more instances are added, others will not. This happens when resource contention is moved to another location in the topology, increasing processing times. This situation arises when bolts undertake different quantities of computational work. In the case of the pulsar search topology, known source matching is the most computationally expensive[12]. Adding more bolts to the pulsar search topology therefore initially increases the demand upon known source matching bolts, which they cannot meet. It isn't until at least 8 worker nodes are available, with enough known source matching bolts, that the con-

[12]The results here indicate that it requires more computational power than all other pulsar search tasks.

tention disappears and processing time performance improves. This is an important observation, as the computational demands of known source matching are surprising.

Table 11 also shows which bolts are the points of resource contention within the pulsar search topology. This is indicated by the capacity value, which is given by,

$$\frac{(\text{tuples executed} \times \text{average execute latency})}{\text{measurement time}}, \qquad (12)$$

thus a value of 1.0 corresponds to a bolt at full capacity. Values greater than 1.0 indicate a bolt over capacity, and less than 1.0 under capacity. When only two bolts of each type are included in the topology, all bolts are at, or near capacity. As the topology is scaled, most bolts begin to become under utilised. The exceptions are the ML classification bolts, and the known source matching bolts. However for all bolts, as more instances are added to the topology, execution time latency reduces. In some cases the improvement in performance scales linearly with the number of worker nodes used. However, this is not always the case. There are fluctuations in the bolt results which make it difficult to discern a genuine trend. The overall results shown in Table 10 indicate an improvement in performance which scales sub-linearly with the total number of worker nodes. This general impression agrees with empirical experience, and is based on averaged results. It is therefore likely the most accurate indication of true system scalability. In any case the AWS infrastructure used was able to support the pulsar search topology, even when executed at SKA scales.

10.2.1. Implementation Issues

Deploying a Storm topology to a remote cluster reveals implementation issues otherwise hidden in local mode. These are not serious, and do not represent bugs in the Apache Storm system. They only arise as our pulsar search processing topology possesses characteristics which stretch Storm in ways its designers could not anticipate. This is perhaps best illustrated through the main issue encountered - communication errors. Storm topologies, which are comprised of bolts and spouts, are supposed to be lightweight units. A topology should require little disk space to store, making it possible to transmit them rapidly across both LAN and WAN connections. The Java Archive (JAR) file which describes a topology, should thus be as small as possible. However for our pulsar search topology, the JAR file is relatively large[13]. This is due to the bolts in the topology requiring access to external data sources (e.g. known source lists, ML training sets), and further data used to generate new candidates at the spouts. These resources increase the size of the JAR, making it difficult to deploy without altering the Storm installation (the *Nimbus max_buffer_size* variable must be increased).

Similarly the topology components should be lightweight, and require as little memory as possible. This is because these components are Java objects, which are initialised before being serialised for transmission to the supervisor nodes. If post-initialisation these objects are large, Storm can struggle to transmit them in two ways. First, if the objects are too large for the Storm transmission buffer, then they will not be sent. Second, if sending succeeds, but the messages are not processed quickly enough on the receiver, then such objects can begin to queue. The more items added to the queue, the more memory required to maintain them. If the queue becomes full, Storm will try to find additional memory. When more memory cannot be found, bolt or spout failures are encountered[14].

The same argument applies to tuples moving through the topology. Storm tuples are expected to be only simple strings, or a small number of strings. For the pulsar search topology, the tuples are relatively large objects representing pulsar candidates. Thus if tuples are not processed fast enough forcing them to queue, memory use can increase rapidly. Again if the queue continues to grow, memory will run out and tuples will be lost.

10.2.2. Recommendations

Packaging a Storm topology into a JAR file can be difficult, if the topology has external dependencies. It is therefore recommended that each JAR be completely self contained, and not use external libraries. This incurs a development cost, but this is acceptable compared to the hours that can be spent debugging problems within Storm. It is also important that topologies be developed and tested concurrently on both local and remote hardware resources. Running unit tests in local mode only, is insufficient to iron out all possible problems (especially communication issues). Moreover debugging a remote Storm topology is difficult, and the Storm UI often reports unhelpful error messages. These are almost always attributable to communication problems, usually caused by contention within the topology causing failures due to increasing queue sizes. It is recommended in such situations to increase the *Nimbus max_buffer_size* value, and minimise tuple/bolt memory use.

The optimal topology configuration to use for arbitrary hardware resources is unclear. Multiple trial and error runs are required to find a configuration that is stable[15]. It is similarly unclear how many worker slots to assign to each node. In the absence of additional information, 1 slot per CPU thread is a sensible recommendation. Finally it is recommended that any similar topology developed within Storm, should minimise tuple, bolt, and spout memory footprints as far as possible. This will minimise the occurrence of communication errors, and make any new topology devised far more stable.

[13] Note that the Maven build system should be used to build the topology JAR. Do not include Storm sources in the JAR. When the JAR is executed by Storm, the required Storm JARs are automatically added to the class path.

[14] Java garbage collector (GC) errors commonly occur when this happens.

[15] Does not form queues at any processing node.

Configuration	Bolts	e_{avg} (ms)	Workers	ECUs	Notes
$2^{1...*} : 2^{1...*} : 11^{1...*} : 2^{1...*} : 2^{1...*} : 2^{1...*} : 2^{1...1} : 2$	23	15.693	1	20	Random sift
$2^{1...*} : 4^{1...*} : 11^{1...*} : 4^{1...*} : 4^{1...*} : 4^{1...*} : 4^{1...1} : 4$	35	17.044	2	40	Random sift
$2^{1...*} : 8^{1...*} : 11^{1...*} : 8^{1...*} : 8^{1...*} : 8^{1...*} : 8^{1...1} : 8$	59	23.47	4	80	Random sift
$2^{1...*} : 16^{1...*} : 11^{1...*} : 16^{1...*} : 16^{1...*} : 16^{1...*} : 16^{1...1} : 16$	107	9.308	8	160	Random sift
$2^{1...*} : 24^{1...*} : 11^{1...*} : 24^{1...*} : 24^{1...*} : 24^{1...*} : 24^{1...1} : 24$	155	6.255	12	240	Random sift
$2^{1...*} : 2^{1...*} : 11^{1...*} : 2^{1...*} : 2^{1...*} : 2^{1...*} : 2^{1...1} : 2$	23	16.503	1	20	Distributed sift
$2^{1...*} : 4^{1...*} : 11^{1...*} : 4^{1...*} : 4^{1...*} : 4^{1...*} : 4^{1...1} : 4$	35	21.622	2	40	Distributed sift
$2^{1...*} : 8^{1...*} : 11^{1...*} : 8^{1...*} : 8^{1...*} : 8^{1...*} : 8^{1...1} : 8$	59	7.801	4	80	Distributed sift
$2^{1...*} : 16^{1...*} : 11^{1...*} : 16^{1...*} : 16^{1...*} : 16^{1...*} : 16^{1...1} : 16$	107	6.891	8	160	Distributed sift
$2^{1...*} : 24^{1...*} : 11^{1...*} : 24^{1...*} : 24^{1...*} : 24^{1...*} : 24^{1...1} : 24$	155	2.045	12	240	Distributed sift

Table 10: Performance and filtering accuracy results for the prototype pipeline run on a remote AWS cluster. Here e_{avg} (ms) is the total time taken on average to process a tuple within the bolts (not counting communication overheads). Results listed for pipelines using random sift (randomly make sift decision with 50:50 split), and full distributed sift. For all tests a candidate ratio of $c_{\text{ratio}} = 0.1$ was used. See Section 9.1 for details on how to interpret the configuration used.

Bolt	Instances	Capacity	Execute Latency (ms)	Process latency (ms)
Preprocessing	2	1.984	1.471	1.955
Preprocessing	4	0.048	0.340	0.238
Preprocessing	8	0.012	0.167	0.698
Preprocessing	16	0.008	0.169	0.111
Preprocessing	24	0.008	0.189	0.369
Sift	11	0.039	0.204	0.094
Feature Extraction	2	1.036	1.570	1.488
Feature Extraction	4	0.006	0.176	0.942
Feature Extraction	8	0.006	0.141	0.103
Feature Extraction	16	0.029	0.620	0.205
Feature Extraction	24	0.126	0.595	0.128
ML Classification	2	1.730	2.695	1.669
ML Classification	4	0.002	0.061	6.781
ML Classification	8	1.445	8.884	0.576
ML Classification	16	0.003	0.120	0.694
ML Classification	24	0.074	0.429	0.213
Known Source matching	2	0.967	1.846	1.844
Known Source matching	4	0.298	21.167	6.636
Known Source matching	8	0.700	14.232	16.097
Known Source matching	16	0.271	8.148	5.065
Known Source matching	24	0.277	4.829	4.961
Alert generation	2	0.014	0.024	0.022
Alert generation	4	0.002	0.143	0.167
Alert generation	8	0.003	0.067	4.686
Alert generation	16	0.001	0.047	0.050
Alert generation	24	0.002	0.047	0.360

Table 11: The runtime performance of individual bolts using random sift (randomly make sift decision with 50:50 split). Here process latency is the time taken to 'ack' a tuple after it is received. Note that 'acking' a tuple involves sending an acknowledgement to the transmitter of a tuple, so it knows it has been received. Execute latency is the time taken for the bolt to complete processing a tuple. Experiments run using $c_{\text{ratio}} = 0.1$.

11. Conclusions

An Apache Storm based prototype SDP pipeline has been developed. The prototype follows a streaming methodology, and processes pulsar candidates arriving from the CSP incrementally, one at a time. It employs a combination of resource efficient algorithms, and optimised filtering methods, to enable large numbers of pulsar candidates to be processed using limited computational resources.

The performance of the prototype has been assessed on a single commodity laptop. During testing it was able to process 1.5 million pulsar candidates[16], in under 600 seconds. It is therefore able to process data fast enough to meet SKA design requirements (exceeds a processing rate of 6,000 candidates per second).

The filtering accuracy of the system is good, reaching up to 99% accuracy. However it can be improved. The prototype design compromised filtering accuracy to achieve high computational efficiency. This has led to a pulsar recall rate of between 81-88%, which is not high enough for SKA use. The prioritisation of computational efficiency over accuracy was intentional, and the drop in recall expected. It was done to ensure the system could meet SKA requirements. There is room however to reverse this trade-off somewhat, without significantly compromising runtime performance. It would not be unreasonable to double (or quadrupole!) the resource use of the system, to achieve much higher filtering accuracy. This is probably advisable given the resources that will be made available for SKA science data processing.

The prototype has also been deployed to a cloud-based software infrastructure. When deployed the system was able to process pulsar candidate data at SKA scales, using only modest computational resources (the equivalent of 241 ECUs). The performance of the cloud-based prototype was good. Its performance scaled sub-linearly with the number of worker nodes used to execute the topology, however in practice this scaling did lead to a tangible improvement in runtime performance.

Finally the prototype has demonstrated the usefulness of off-the-shelf software components, for rapidly building useful science data processing pipelines. The Storm system is relatively easy to use, and has the potential to assist us in the future. It has both strengths (computational efficiency) and weaknesses (strict stream processing model), yet proved adaptable enough for our SDP prototype. We do not advocate that Storm be used to build a real-world SDP. However the utility of open source tools should be considered. By using Storm, a single developer was able to build an entire prototype SDP pipeline, at very low cost, within a small number of weeks (6-8 weeks). This work would have taken much longer to complete, if the functionality provided by Storm had to be programmed from scratch.

[16]The quantity delivered per SKA observation

12. Future Work

Future work should analyse the performance of the individual algorithms installed at the bolts in the topology. The distributed and optimised algorithms developed here in particular, have not been studied in great detail. This is because they were developed during the course of this prototyping effort. They can be optimised to give improved runtime performance, and more crucially, filtering accuracy. They are certainly deserving of more detailed analysis. We are currently working on a data generator that will assist us in such an investigation.

13. Acknowledgements

This work utilised data obtained by the High Time Resolution Universe Collaboration using the Parkes Observatory, funded by the Commonwealth of Australia and managed by the CSIRO. Cloud testing supported by the SKAO-AWS AstroCompute grant programme.

References

Ait-Allal D. et. al., 2012, "Blind detection of giant pulses: GPU implementation", Comptes Rendus Physique, 13(1):80-87

ALFA Pulsar Consortium, 2015, "ALFA Pulsar Studies", on-line resource, `http://www.naic.edu/alfa/pulsar/`, accessed 06/09/2015

Apache Software Foundation, 2015a, "Apache Storm", on-line, `http://storm.apache.org`, accessed 22/02/2016

Apache Software Foundation, 2015b, "Companies Using Apache Storm", on-line, `http://storm.apache.org/Powered-By.html`, accessed 22/02/2016

Apache Software Foundation, 2015c, "S4 Distributed Stream Computing Platform", on-line, `http://incubator.apache.org/s4/`, accessed 22/02/2016

Apache Software Foundation, 2015d, "What is Samza?", on-line, `http://samza.apache.org`, accessed 22/02/2016

Apache Software Foundation, 2014, "Welcome to Apache Hadoop", on-line, `http://hadoop.apache.org`, accessed 22/02/2016

Apache Software Foundation, 2016, "Spark: Lightning-fast cluster computing", on-line, `http://spark.apache.org`, accessed 22/02/2016

Barr E. D., 2014, "Survey for Pulsars and Extra-galactic Radio Bursts", on-line, `http://www3.mpifr-bonn.mpg.de/div/jhs/Program_files/EwanBarrCrete2014.pdf`, accessed 06/09/2015

Barr E. D. et. al., 2013, "The Northern High Time Resolution Universe pulsar survey I. Setup and initial discoveries", MNRAS, 435(3):2234–2245

Bhattacharyya B., et. al., 2016, "The GMRT high resolution southern sky survey for pulsars and transients. i. survey description and initial discoveries", ApJ, 817(130)

Bishop C. M., 2006, "Pattern Recognition and Machine Learning", Springer

Broekema P. C., van Nieuwpoort R. V., and Bal H. E., 2015, "The square kilometre array science data processor. preliminary compute platform design", *Journal of Instrumentation*, 10(07):C07004

Carilli C. and Rawlings S., 2004, "Motivation, key science projects, standards and assumptions", New Astronomy Reviews, 48(11-12):979–984

Ng C., 2012, "Conducting the deepest all-sky pulsar survey ever: the all-sky High Time Resolution Universe survey", In *Neutron Stars and Pulsars: Challenges and Opportunities after 80 years*, volume 8 of *Proceedings of the International Astronomical Union Symposium S291*, pages 53–56

Coenen T., 2014, "The LOFAR Pilot Surveys for Pulsars and Fast Radio Transients", A & A, 570(A60)

Cordes J. M., 2006, "Arecibo Pulsar Survey Using ALFA. I. Survey Strategy and First Discoveries", ApJ, 637(1):446–455

Couturier R., 2013, "Designing scientific applications on GPUs", CRC Press

Dewdney P. E., 2015, "SKA1 system baseline design v2", Technical report, Square Kilometre Array Organization

Dewdney P. E., 2013, “SKA1 system baseline design”, Technical report, Square Kilometre Array Organization

Dimoudi S. and Armour W., 2015, “Pulsar acceleration searches on the GPU for the square kilometre array”, In *25th annual ADASS conference*, page 6

Dubey R., 2008, “Introduction to embedded system design using field programmable gate arrays”, Springer Science & Business Media

Duda R. O., Hart P. E., and Stork D. G., 2000, “Pattern Classification”, Wiley-Interscience, 2nd Edition

Eatough R. et. al., 2014, “Observing radio pulsars in the galactic centre with the square kilometre array”, *Advancing Astrophysics with the Square Kilometre Array, Proceedings of Science, PoS(AASKA14)040*

Intel Corporation, 2013, “Intel®Core i7-2700 Mobile Processor Series”, on-line, `http://www.intel.com/content/dam/support/us/en/documents/processors/corei7/sb/core_i7-2700_m.pdf`, accessed 28/04/2016

Jain A. and Nalya A., 2014, “Learning Storm”, Packt Publishing

Jankowski M., Pathirana P., and Allen S. T., 2015, “Storm Applied: Strategies for Real-time Event Processing”, Manning

Jones M. T., 2013, “Process real-time big data with twitter storm”, IBM Technical Library

Karastergiou A. et. al., 2015, “Limits on fast radio bursts at 145 MHz with Artemis, a real-time software backend”, MNRAS, 452(2):1254–1262

Keane E. F. et. al., 2014, “A cosmic census of radio pulsars with the SKA”, *Advancing Astrophysics with the Square Kilometre Array, Proceedings of Science, PoS(AASKA14)040*

Keane E. F., 2016, “The host galaxy of a fast radio burst”, Nature 530, 453–456

Keane E. F. et. al., 2012, “On the origin of a highly dispersed coherent radio burst”, MNRAS: Letters, 425(1):L71–L75

Keith M. J., 2016, “Pulsar hunter”, on-line, `http://www.pulsarastronomy.net/wiki/Software/PulsarHunter`, accessed 22/02/2016

Keith M. J. et. al., 2010, “The High Time Resolution Universe Pulsar Survey - I. System Configuration and Initial Discoveries”, MNRAS, 409(2):619–627

Keith M. J. et. al., 2010, “Discovery of 28 pulsars using new techniques for sorting pulsar candidates”, MNRAS, 395(2):837–846

Kilts S., 2007, “Advanced FPGA design: architecture, implementation, and optimization”, John Wiley & Sons

Kulkarni S. et. al., 2015, “Twitter heron: Stream processing at scale”, In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 239–250

Lam M. T. et. al., 2016, “Systematic and stochastic variations in pulsar dispersion measures”, ApJ, 821(1)

Law C. J. et. al., 2015, “A Millisecond Interferometric Search for Fast Radio Bursts with the Very Large Array, ApJ, 807(1)

Lazarus P., 2012, “The PALFA Survey: Going to great depths to find radio pulsars”, In *Neutron Stars and Pulsars: Challenges and Opportunities after 80 years*, volume 8 of *Proceedings of the International Astronomical Union Symposium S291*, pages 53–56

Lorimer D. R., 2016, “Sigproc”, on-line, `http://sigproc.sourceforge.net`,
accessed 22/02/2016

Lorimer D. R., 2007, “A Bright Millisecond Radio Burst of Extragalactic Origin”, *Science*, 318(5851):777–780

Lyon R. J., 2015, “HTRU2”, on-line, `https://dx.doi.org/10.6084/m9.figshare.3080389.v1`

Lyon R. J., 2015, “ Why Are Pulsars Hard to Find?”, PhD thesis, University of Manchester, School of Computer Science

Lyon R. J. et. al., 2013, “A Study on Classification in Imbalanced and Partially-Labelled Data Streams”, In *Simple and Effective Machine Learning for Big Data, Special Session, IEEE International Conference on Systems, Man, and Cybernetics*, pages 1506–1511

Lyon R. J. et. al., 2014, “Hellinger Distance Trees for Imbalanced Streams”, In *22nd IEEE International Conference on Pattern Recognition*, pages 1969–1974

Lyon R. J. et. al., 2016, “Fifty Years of Pulsar Candidate Selection: From simple filters to a new principled real-time classification approach”, MNRAS, 459 (1):1104-1123

Magro A. et. al., 2011, “Real-time, fast radio transient searches with GPU de-dispersion”, MNRAS, 417(4):2642–2650

McLaughlin M. A., 2009, “Rotating radio transients”, In W. Becker, editor, *Neutron Stars and Pulsars*, pages 41–66. Springer Berlin Heidelberg

McLaughlin M. A. et. al., 2007, “Transient radio bursts from rotating neutron stars”, *Nature*, 439:817–820

Mickaliger, M. B. et. al., 2012, “Discovery of Five New Pulsars in Archival Data”, ApJ, 759(2)

Naidu A. et. al., 2015, “PONDER - a real time software backend for pulsar and IPS observations at the ooty radio telescope”, *Experimental Astronomy*, 39(2):319–341

Nijboer R. et. al., 2015, “PDR.05 parametric models of SDP compute requirements”, Technical report, Square Kilometre Array Organization

Petroff E. et. al., 2015, “A real-time fast radio burst: polarization detection and multiwavelength follow-up”, MNRAS, 447(1):246–255

Petroff E. et. al., 2013, “Dispersion measure variations in a sample of 168 pulsars”, MNRAS, 435(2):1610–1617

Ransom S., 2016, “Presto”, on-line, `http://www.cv.nrao.edu/~sransom/presto/`, accessed 22/02/2016

Russell S. and Norvig P., 2009, “Artificial Intelligence: A Modern Approach”, Prentice Hall, 3rd Edition

Schilizzi R. T. et. al., 2007, “Preliminary specifications for the square kilometre array”, *SKA Memorandum*, 100

Sclocco A. et. al., 2015, “Finding pulsars in real-time”, In *2015 IEEE 11th International Conference on e-Science*, pages 98–107

Smits R., 2009, “Pulsar searches and timing with the square kilometre array”, A & A, 493(3):1161–1170

Tan G. H. et. al., 2015, “The square kilometre array baseline design v2.0”, In *1st URSI Atlantic Radio Science Conference (URSI AT-RASC)*

Taylor A. R., 2012, “The square kilometre array”, In *Neutron Stars and Pulsars: Challenges and Opportunities after 80 years*, volume 8 of *Proceedings of the International Astronomical Union*, pages 337–341, 8

Terzian Y. and Lazio J., 2006, “The square kilometre array”, In *Proceedings of SPIE 6267, Ground-based and Airborne Telescopes, Future Giant Telescopes III*, volume 6267

Thompson D. R. et. al., 2011, “Semi-supervised eigenbasis novelty detection, and application to radio transients”, In *NASA Conference on Intelligent Data Understanding*, pages 118–128

Thornton D., 2013, “The High Time Resolution Radio Sky”, PhD thesis, University of Manchester, Jodrell Bank Centre for Astrophysics School of Physics and Astronomy

Thornton D. et. al., 2013, “A Population of Fast Radio Bursts at Cosmological Distances”, *Science*, 341(6141):53–56

S. J. Tingay, 2014, “The science and technology of the square kilometre array”, *IOP Conference Series: Materials Science and Engineering*, 67(1)

Toshniwal A. et. al., 2014, “Storm@twitter”, In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’14, pages 147–156. ACM

van Heerden E. et. al., 2014, “New approaches for the real-time detection of binary pulsars with the square kilometre array”, In *General Assembly and Scientific Symposium XXXIth URSI*, pages 1–4

Vanderbauwhede W. and Benkrid K., 2013, “High-Performance Computing Using FPGAs”, Springer

Wang H. and Sinnen O., 2015, “FPGA based acceleration of FDAS module for pulsar search”, In *International Conference on Field Programmable Technology (FPT)*, pages 240–243

Weiler A., Grossniklaus M., and Scholl M. H., 2015, “Run-time and task-based performance of event detection techniques for twitter”, In *Advanced Information Systems Engineering: 27th International Conference, CAiSE 2015, Stockholm, Sweden, June 8-12, 2015, Proceedings*, pages 35–49. Springer International Publishing

White T., 2012, “Hadoop: The definitive guide”, O’Reilly Media, Inc.

Woods R. et. al., 2008, “FPGA-based Implementation of Signal Processing Systems”, Wiley Publishing

Zhu W. W. et al., 2014, ApJ, 781, 2