technische universität
dortmund

SFB 876 Verfügbarkeit von Information
durch Analyse unter Ressourcenbeschränkung

Arbeit zur Erlangung des akademischen Grades
Bachelor of Science

Evaluating throughput performance of the `streams`-framework on Cherenkov Telescope Array data

Kai Brügge
geboren in Dortmund

2015

Lehrstuhl für Experimentelle Physik Vb
Fakultät Physik
Technische Universität Dortmund

Erstgutachter:	Prof. Dr. Dr. Rhode
Zweitgutachter:	Prof. Dr. Zweitgutachter
Abgabedatum:	31. Oktober 2015

Abstract

The Cherenkov Telescope Array (CTA) will be the largest and most sophisticated experiment in gamma-ray astronomy to date. The final array will consist of over a hundred telescopes working in unison. This creates a big challenge for data acquisition, computing and analysis. Streaming data from many telescopes into a unified analysis process is essential for gaining knowledge about monitored sources. Based on already available solutions for streamed data processing and Imaging Atmospheric Cherenkov Telescopes (IACT), the `streams`-framework and the FACT-Tools, a new solution specifically tailored to the high performance requirements of the CTA experiment is build. The ultimate goal will be the analysis of CTA raw data in near real-time. This thesis will document the efforts to build and evaluate software based on the `streams`-framework and apply it to CTA Monte Carlo data.

Kurzfassung

Das Cherenkov Telescope Array (CTA) ist das größte und komplexeste Experiment der Gammaastronomie, dass bisher entwickelt wurde. Das fertige Experiment wird über 100 Teleskope enthalten. Die Ansprüche für die Datenakquise, Rechnerinfrastruktur und Analyse sind hoch. Daten von mehreren Teleskopen müssen vereint werden um zeitnahe Informationen über die beobachtete Quelle zu erhalten. Basierend auf vorhandenen Lösungen für Datenanalyse im Strom und bekannten Analysemethoden für Imaging Atmospheric Cherenkov Telescope (IACT), dem `streams`-framework und den FACT-Tools, wird eine neue Lösung, speziell für die Laufzeitanforderungen von CTA entwickelt. Das Ziel ist die Echtzeitanalyse von CTA Rohdaten. In dieser Arbeit wird dokumentiert, wie die Software entwickelt und evaluiert wird.

Contents

1 Introduction

Earth's atmosphere is constantly being bombarded by highly energetic particles of cosmic origin. The largest fraction of these particles are protons, alpha particles and other heavier nuclei. These particles interact with atoms in the atmosphere and produce cascades of secondary particles which are hurled towards earths surface. Moving faster than the speed of light in air, the secondary charged particles produce photons in the visible and ultraviolet spectrum due to the Cherenkov effect. Ground based experiments like Imaging Atmospheric Cherenkov Telescopes (IACTs) capture and record the Cherenkov light using sensitive cameras. Cosmic gamma rays are another source of air showers. In contrast to protons or other charged particles they do not interact with electric or magnetic fields. This means that gamma rays travel in a straight line from the source to the observer. This allows to gain insights to the inner working of objects which produce high energy gamma rays like active galactic nuclei (AGN) or supernova remnants (SNRs).

The planned Cherenkov Telescope Array (CTA) will be the largest and most expensive experiment in the field of gamma ray astronomy. The final array will consist of over a hundred telescopes of various shapes and sizes distributed across two different experiment sites. Both the northern and the southern sky will be observed by the telescopes. Once operational CTA will produce a substantial amount of data. In fact its not physically possible to store the entire raw data into persistent storage using todays technologies. The volume and rate at which data is produced by the array is too large for conventional storage. Data reduction has to be performed before any information is written. This can be achieved by suppressing data from dark camera pixels, filtering triggered events which are clearly of hadronic origin, compression of raw data or any combination of these three. Reducing large amounts of data while gaining information from it in real time is a typical use case for Big-Data technology.

Over the past few years many concepts for network, hardware and software infrastructure have been created and popularized by data-driven companies like Google, Twitter, Facebook or Amazon's Cloud platform. While Big-Data management solutions like Hadoop are being used with massive success in places like the LHC at CERN [8][2][1], most of the buzzwords have not arrived in the astroparticle community just yet. Analyzing CTA data under time and resource constraints in near real time while using current Big-Data technologies fits well into the research topics of the collaborative research center (Sonderforschungsbereich 876) at the TU Dortmund. As a collaborative work between physicists, computer scientists and

software engineers we try to close the interdisciplinary gap between those fields by combining the domain knowledge of physicists and computer scientists.

One product of the SFB876 is the `streams`-framework created by C. Bockermann. The `streams`-framework was developed for simple definitions of data stream processes and data flow graphs. In this thesis the `streams`-framework and other Big-Data technologies are used to prototype a real time analysis and data reduction pipeline for CTA data. The goal is to evaluate the runtime performance and data rates that can be achieved using these technologies. Higher data rates allow for more effective noise suppression schemes and higher sensitivity for the real time analysis. The prototype builds upon the same technologies that are used by the low level computing architecture as proposed by the Cherenkov Telescope Array (CTA) data acquisition (DAQ) working group at the university of Geneva. The proposed DAQ system uses the ProtocollBuffer format to send data over the on-site network using ZeroMQ.

During this thesis the `streams`-framework is extended to read and analyze simulated CTA data. A lot of attention was put into parallelization of the process for multiple processor cores. Runtime performance and memory requirements for calculating image parameters were measured. The initial goal was to analyze raw camera data from Monte Carlos simulations with an event rate of at least 15 kHz. Extensions for the `streams`-framework were developed to receive data over network using ZeroMQ. Two different serialization formats, Protocol Buffers and MsgPack, are evaluated with the goal to receive and analyze data with at least 8 kHz when using ZeroMQ.

2 The Cherenkov Telescope Array

The Cherenkov Telescope Array will be the largest and most sophisticated experiment in gamma-ray astronomy to date. CTA aims to expand the sensitivity to gamma-rays across a wide energy spectrum by an order of magnitude compared to current Cherenkov telescope designs like MAGIC or HESS [3]. The angular resolution will be improved by a factor of 5 to 0,02° [3]. With improved sensitivity new sources of high energy gamma rays are expected to be observed. The improved angular resolution allows for measuring substructures of extended sources like supernova remnants (SNRs). The high sensitivity also opens up new possibilities for fundamental physics like dark matter search and investigation of transient phenomena and flaring sources [4]. The later imposes strong time constraints for an online analysis.

To achieve its sensitivity goals CTA will consist of multiple types of telescopes. Several prototype cameras and telescopes are currently being evaluated. The different telescope types fall into three major categories. The Large Size Telescopes (LSTs) will have a mirror diameter of 23 m and are sensitive to the lowest part of the observed energy spectrum down to 10 GeV. The Medium Size Telescopes (MSTs) have a diameter of 10 m to 12 m and a Field of View (FoV) of 6° to 8° to cover the energy range of approximately 100 GeV to 1 TeV. The Small Size Telescopes (SSTs) will observe the highest energy range above 1 TeV and have a large FoV of about 10°. The south site will consist of 4 LSTs, 25 MSTs and about 70 SSTs. The array on the north site will also contain 4 LSTs and anything between 16 and 20 MSTs. The final layout and arrangement of telescopes on site are under negotiation. The layouts will be chosen by their respective sensitivity as calculated by Monte-Carlo simulations [5]. Figure 2.1 shows an artists impression of the array.

Figure 2.1: An artists rendering of a possible CTA configuration [11].

3 Computing for CTA

Handling transfer and analysis of data from dozens of telescopes requires a reliable hardware and software stack working at the experiment site. Data from the telescopes needs to be transferred and distributed to the computing infrastructure on site. Events from the single telescopes need to be collected and packed into *array events* containing data from all triggered telescopes. The recorded data has to be stored into persistent storage and distributed to computing nodes running an online analysis of CTA data on-site. This is the task of the DAQ work package within the CTA consortium. CTA will produce tens of TiB[1] of raw data per night [13]. Though there is no way to predict what data storage technology will look like once CTA is completed, it seems unrealistic to store the raw data for every telescope to a persistent storage somewhere on site. This means that data needs to be reduced before it can be stored into offline data storage. This reduction can happen very early in the camera servers and/or later by the DAQ system on a central computing cluster. Waveforms of camera pixels that are not hit by Cherenkov photons can be reduced to a few numbers that describe the general shape of the recorded voltages. Reduction can happen through filtering and removing of background events induced by night sky background light or hadronic showers. Effective data reduction at an early stage may allow for using more advanced methodology for the analysis which in turn leads to better sensitivity for the real time analysis (RTA). A common scheme for data reduction has yet to be decided by the CTA collaboration. Handling messaging in a big network is a challenging task. The network at the CTA site will easily consist of more than a hundred, possibly heterogeneous, nodes. In a network such as this the distinction between client and server is not always clear and depends strongly on the use case. The computers at the telescope for example will have to act as both: a server for raw physics and slow data and as a client for the control system for pointing and slewing. All of these nodes send and receive messages with varying speeds and reliability. A custom software solution for message handling in a network this large will have to deal with problems of portability, extensibility, queueing and fault tolerance, just to name a few. Sections 3.2 and 3.3 present possible technologies that help to tackle some of these problems at varying abstraction levels. The solutions chosen are inspired by the DAQ architecture proposed by the CTA DAQ work group [19].

[1] Throughout this text binary prefixes are used to quantify data. The prefixes describe powers of 1024 according to the definition of the International Electrotechnical Commission (IEC). Therefore one MiB corresponds to 1024^2 bytes, one GiB corresponds to 1024^3 and so forth.

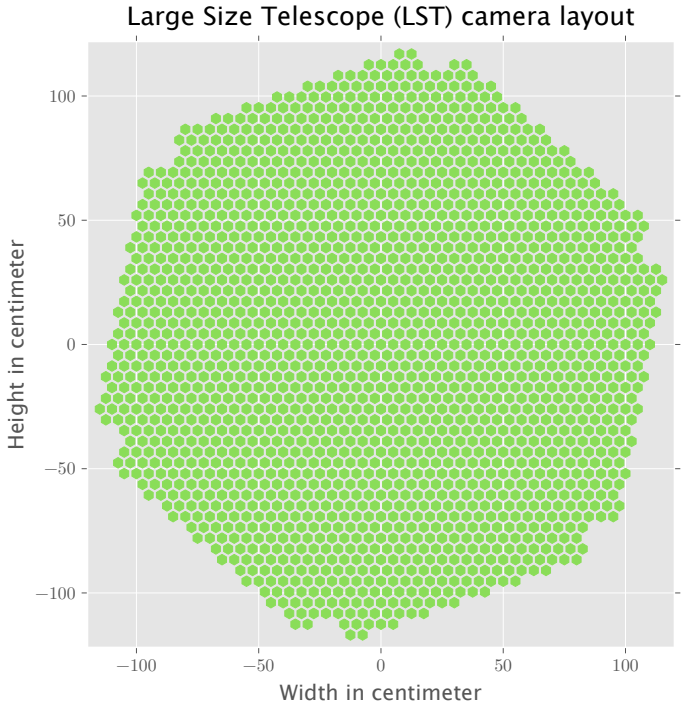
3.1 CTA data rate

Figure A.1 (see appendix) shows the flux of cosmic radiation versus energy. The higher the energy the lower the flux of particles becomes. Therefore a telescope sensitive to the lower energies will have a higher trigger rate. Using a two telescope coincidence for the array, the expected trigger rate for the southern array is 32 kHz. [13]. Due to the large collection area of 23 m (mirror diameter) the LST type telescopes will have the highest trigger rate. Estimates from Monte Carlos simulations predict rates of 15 kHz for a single LST camera [13]. Each camera is connected to a dedicated camera server by fiber optic cable. The LST prototype cameras consist of 1855 pixels comprised of Hammamatsu PMT. Once an event is triggered the camera electronics record 30 samples of the measured voltage coming from each PMT [10]. In the Monte Carlo simulation each sample is stored in 2 bytes. This produces $1855 \cdot 30 \cdot 2 = 108.7$ KiB of data per triggered event. This leads to a total rate of $15\,000\,\text{Hz} \times 108.7\ \text{KiB} = 1{,}555\,\text{GiB/s}$ which have to be transferred from the camera electronics to the camera server.[2]

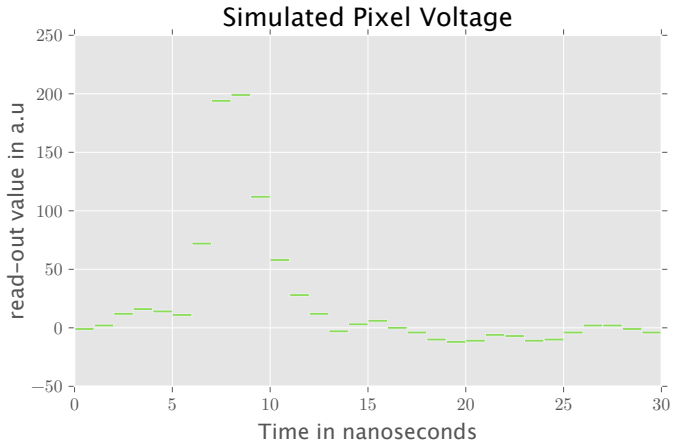
Depending on the number of pixels in the camera and the sampling rate, other telescope types will have different data rates. The overall structure of the events will be similar however.[3] From the camera servers the data will be sent to central computing nodes over an ethernet connection. The definitive event size being sent over the wire will depend on the serialization format chosen for transport as described in sections 3.3. Figure 3.1a illustrates the layout of the pixels in the LST camera. Figure 3.1b depicts the samples coming from a single pixel as simulated by the `simtelarray` program [5].

[2] According to the Technical Design Report [13] it will be 4 bytes per sample and another 5 bytes of header information per pixel. Under these assumptions the data rate will be closer to 3,236 GiB/s. However the data stored in the Monte Carlo files uses 16 bit unsigned integers per sample and provides only static header information per pixel. The later seems more reasonable judging from experiences gained from data acquisition from the FACT telescope which uses the same DRS4 sampling electronics as the LST prototype.

[3] Except for data from the ASTRI SST camera which records only a few numbers describing the overall shape of the measured pulse in each pixel.



(a) Pixel layout of the CTA LST camera. The camera consists of 1855 pixels with hexagonal light collectors.



(b) Simulated voltage samples recorded by a single pixel when hit by Cherenkov light.

3.2 ZeroMQ as an abstraction over raw TCP sockets

Once the messages have been assembled in the camera server they can be sent to computing nodes via ethernet. The protocol of choice will be Transmission Control Protocol (TCP) since message loss has to be avoided as much as possible which cannot be guaranteed using User Datagram Protocoll (UDP). ZeroMQ[4] provides a thin abstraction layer over TCP sockets. ZeroMQ is still very close to the metal and does not impose any constraints on network topologies or architectures. ZeroMQ has bindings and libraries for many popular programming languages and works on all major operating systems. It makes it easier to implement several messaging patterns through different kinds of sockets with differing functionalities. Many patterns for messaging are explained in great detail in the official user guide [17]. In the RTA software prototype developed during this thesis a simple Publish/Subscriber messaging pattern is used for communication between nodes. Figure 3.2 shows a sketch of a Publish/Subscriber setup. The publishers sent data through a TCP port without knowing its clients. The subscribers can connect to publishing nodes. This way data from several publishers can be collected or distributed among any number of subscribers. New subscribers or publishers can be added any time. This messaging pattern was chosen for all runtime or throughput tests in this thesis for sake of simplicity and flexibility.

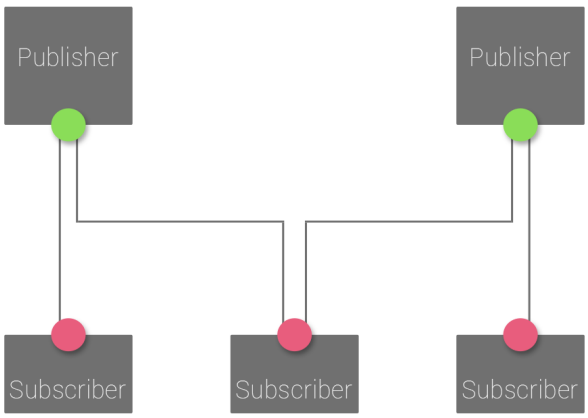


Figure 3.2: A sketch of a network using the Publish/Subscriber pattern for messaging. The subscribers connect to a specific port on the publishing server. The publishers simply send data to a specific port. They do not receive data and do not know their clients.

[4]The name is often stylized as ØMQ or ZMQ

3.3 A comparison of serialization formats

ZeroMQ sends and receives raw arrays of bytes. These bytes have to be interpreted by all possible network endpoints and used programs. Once the format of the messages has been specified, the raw bytes can be interpreted by custom readers for each program. While reading bytes is in itself not a difficult task for most programming languages, the heterogenous nature of the software used in CTA motivates the use of a language agnostic specification. In addition the compression of data, which might be necessary, is not a trivial task and will result in complex solutions that have to be implemented in all languages used. Many programming languages use the notion of objects, containers, structs, or classes to structure data. Turning these objects into raw bytes is the process of serialization. The process of deserialization inverts that operation and creates objects from bytes. The following subsections describe two popular serialization solutions which work on a number of languages and platforms.

3.3.1 Protocol Buffers

Protocol Buffers (or simply ProtoBuf) were introduced by Google in 2008. Protocol Buffers uses a schema language to structure data. The language comes with a few primitives and data types to describe structured data. These descriptions are written into `.proto` files and run through the ProtoBuf compiler. The compiler then generates code in the desired target language. The generated code then provides an interface to the data. In object oriented languages for example, the generated code provides access methods to the deserialized data and is used like any other object in the language.

From the `.proto` specification (see Listing B.1) a java class is generated that provides access to the data as demonstrated in listing 3.1. In this thesis version 3.0.0-alpha-3.1 of Protocol Buffers with the experimental Java Protocol Buffers Nano runtime is used. The Java Protocol Buffers Nano runtime in combination with the newer version had significant performance improvements over version 2. The Google Protocol Buffers project is open source software and can be accessed through a public Github repository [14]. Protocol Buffers are supported by C++, Python, Ruby and Java.

Listing 3.1: Usage of the RawCTAEvent.RawEvent class as created by the `proto` compiler version 3.0.0-alpha-3.1.

```
try {

    RawCTAEvent.RawEvent rawEvent = RawCTAEvent.RawEvent.parseFrom(raw_bytes);
    int[] rawData = rawEvent.samples;
    int telescopeId = rawEvent.telescopeId;
    int numPixel = rawEvent.numPixel;
    int regionOfInterest = rawEvent.roi;

} catch (InvalidProtocolBufferNanoException e){
    //handle error
}
```

3.3.2 MsgPack

Unlike the techniques described above, MsgPack does not need a schema language. Its messages are self-describing. This foregoes the need for another program or compiler to automatically generate code in the target programming language. This removes another layer of complexity. Usage of its Java API is quite simple as listing 3.2 demonstrates.

Listing 3.2: Writing data to a ByteArrayOutputStream using MsgPacks Java API

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
MessagePacker packer = msgpack.newPacker(out);
try {
    packer.packInt(numPixel);
    packer.packInt(telescope.telescopeId);
    packer.packInt(roi);
    packer.packArrayHeader(roi*numPixel);
    for (int pix = 0; pix < eventData.length; pix++) {
        for (int slice = 0; slice < roi; slice++) {
            packer.packShort( eventData[pix][slice]);
        }
    }
    packer.close();
} catch (IOException e) {
    //handle error
}
```

The order in which the single message parts are written has to be known by the program reading the data. Switching lines 4 and 5 in listing 3.2 without changing the reader will result in wrong results which can, in the worst case scenario, go unnoticed. Simplicity of usage might nevertheless outweigh the downside of not having a fixed scheme for messages.

4 The `streams`-framework

The data analysis chain for CTA, or really any other experiment, can be modeled by a data flow graph, especially in the context of real time analysis. The experiment has a data source which continuously emits pieces of data. This data needs to be transferred to an entity which processes the data. These can be algorithms that analyze raw data and emit new information. Once a piece of data has been processed it can be propagated to the next processing entity. At the end of this pipeline results are written to some kind of data sink, like a file or a database. The continuos flow of data from a source to a sink motivates the term *data stream*. Figure 4.1 is a drawing of a simple data flow graph. Two data sources emit data into the graph. The processors work on the data stream and pass it along to a queue where the two streams get merged into one.

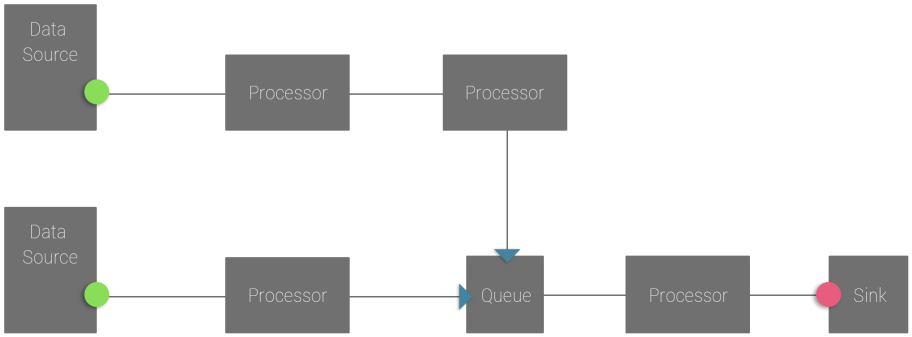


Figure 4.1: Graphical representation a data flow graph containing sources, processing nodes and sinks.

The main motivation behind the `streams`-framework is to provide an abstraction to uniquely define the topology of data flow graphs using an XML based description language. This leads to simple reproducibility of the data analysis while providing an abstraction layer for data flow independent of implementation details. This allows for robust reasoning about the data and control flow of the data analysis chain. The streams-framework is written in the Java programming language and comes with a runtime module to execute the stream as defined by the XML document. These data flow graphs can easily be mapped to topologies for the Apache Storm [20]

engine, which allows for data analysis in large scale distributed environments. The `streams`-framework comes with a set of basic processing nodes or processors for data transformations and control flow management. In addition to the already existing processors a user can also easily define an entirely new processor. In the context of the `streams`-framework data sources are called *streams*. A stream produces so called *data items* which are propagated according to the data flow graph as specified in the xml file. Each data item created by the stream is represented by a HashMap containing values of arbitrary[1] type. A node processing these data items is called a processor. A processor takes one data item at a time, performs some task on it, and returns the possibly modified data item.

In terms of CTA, the telescopes act as the source of data in the data flow graph. Once triggered, a telescope collects the time series for all pixel into one logical unit, a data item, and passes it along to some processing methods. The dedicated analysis steps can be represented as processing nodes in the graph.

4.1 Application to CTA data

Since CTA has not been build yet, simulated data stored in files are used as the data source. To apply the `streams`-framework to CTA data, the simulated data has to be read from files produced by the Monte Carlo program `simtelarray` [5]. The simulated data was then converted from the original `eventio` to the more efficient `kryo` [12] file format. A custom stream was implemented to read data from `kryo` files and act as a data source within the `streams`-framework . Simple feature extraction algorithms were implemented to extract image features from CTA raw data. Leaving out technical details, the analysis of CTA raw data can be reduced to three basic steps.

Low Level Extraction From the voltage curve, the number and arrival times of photons that hit a specific pixel has to be estimated. The maximum value of the voltage within a 15 ns window was used to estimate the number of photons for each pixel. The position of the maximum was chosen as an estimator for the arrival time.

Cleaning From the extracted arrival times and number of photons, retain only those pixels in the camera image that belong to the shower. In this step, the pixels hit by photons from the night sky background have to be discarded as much as possible. At first, pixels above a certain threshold were selected as shower

[1] All objects in the data item have to implement Java's Serializable interface which is the case for many built-in classes.

pixels. In a second pass adjacent pixels above a second, lower, threshold were selected.

Image parameter calculation Image parameters are calculated based on the pixels and extracted values in the previous steps. Among the most prominent image features are the Hillas parameters [15], which can be used for Signal-Background separation.

Listing 4.1 shows an XML representation for this process. The stream reads the data from a `kryo` file on a web server. The process reads the data items from the stream and applies the separate analysis steps to the data. Once all image parameters are found, they are written to a file containing `json` syntax [18] [21].

Listing 4.1: An XML file defining a simple data flow for analyzing data from a `kryo` file on a web server. The process gets the stream ID as an input. Inside the process several processors are called in the order in which they appear. First the data rate and memory usage are logged to the data stream. Then processors are called that estimate the number of photons and their arrival time in each pixel. Bright pixels are selected and Hillas parameter calculated. Finally, results are written to a file called results.jsonl. Each XML file needs an outer tag, which in this case is the `<application/>` element.

```
<application>

    <stream id="cta:data" class="streams.cta.io.KryoStream"
  url="http://sfb876.tu-dortmund.de/data_0.kryo?self=$eipoqcpkhs&amp;part=data"/>

    <process input="cta:data">
        <streams.DataRate every="1000" logmemory="true"/>
        <streams.cta.datacorrection.BaselineShift />
        <streams.cta.extraction.Photons />
        <streams.cta.extraction.ArrivalTime />
        <streams.cta.cleaning.TwoLevelTimeNeighbor levels="400,320" />
        <streams.cta.features.Size/>
        <streams.cta.features.COG/>
        <streams.cta.features.WidthLengthDelta/>
        <streams.cta.io.JSONWriter keys="cog,width,length,size"
  url="results.jsonl" />
    </process>

</application>
```

5 Results

The main goal of this thesis is to measure the runtime performance for different tasks and setups. Runtime is quantified by measuring the number of processed events per second. A dedicated `streams` processor measures the time it takes for streaming a fixed number of items N. After N items have been streamed, the elapsed time is logged to the stream and the clock is reset. Then the time for processing the next N items is measured. This way each execution produces M/N runtime measurements, where M is the total number of events streamed. Memory usage is also saved by logging the output of `Runtime.getRuntime().freeMemory()`. Total memory usage depends on the actions of the garbage collector and Java Virtual Machine (JVM) settings. The numbers produced however, are still useful to track memory behavior of the program over longer time periods. All tests were performed on the same machine.

5.1 Performance on a single thread

The data was originally given in the `eventio` format which was produced by the `simtelarray` Monte Carlo simulation program. The data was converted into the java specific `kryo` serialization format only retaining the sampled data for each pixels and meta data for telescope identification and event timestamps. Dropping the Monte Carlo truth information produces events which are closer to what the real events produced by the telescopes will look like in both form and size. Four measurements were performed to evaluate performance on a single thread.

1. Reading raw telescope data from Monte Carlo `eventio` files.
2. Reading raw telescope data from converted `kryo` files.
3. Reading data and extracting time series features as described in section 4.1 from converted `kryo` files.
4. Steps 2, 3 and additionally selecting the most important pixels (Cleaning) and calculating Hillas parameters as described in section 4.1.

In figure 5.1 the results of these measurements are shown. The data was streamed from a ramdisk[1] and processed in a single thread. Reading data from `kryo` files is faster by two orders of magnitude when compared to `eventio` files. Estimating time series features requires multiple iterations over the voltages in each pixels. This accounts for the large gap in runtime between measurements 2 and 3. The process of image cleaning and parametrization uses the numbers calculated in the previous steps and does not require iteration over the raw data. Hence the runtime differences between measurement 3 and 4 are smaller. Because reading from `kryo` files is clearly much faster than access to `eventio`, all succeeding tests were performed on data stored in `kryo` files.

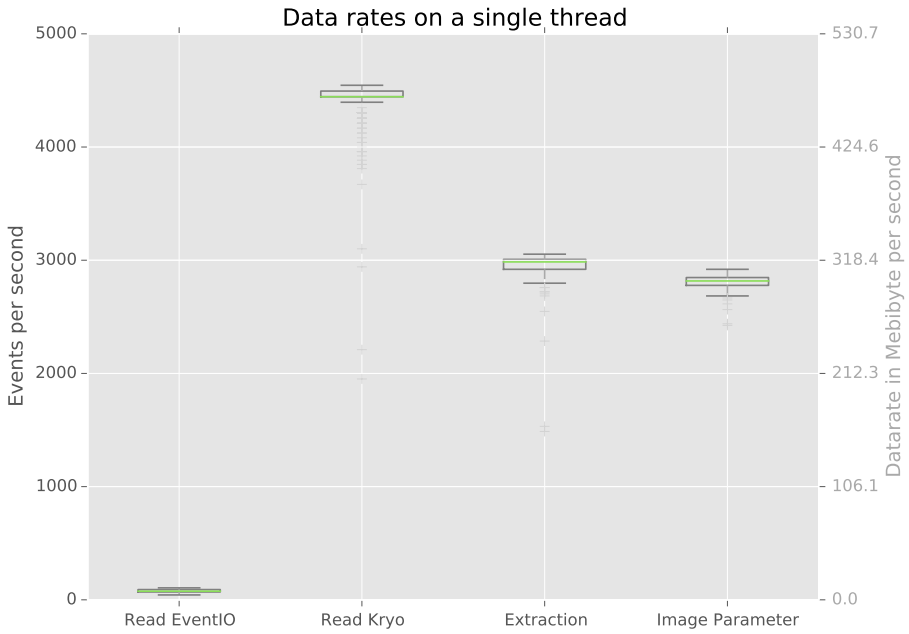


Figure 5.1: Event rates for reading and analyzing Monte Carlo data. The two boxes to the left show the event rate while reading data. The boxes on the right hand side show the event rate when extracting low level and high level features from `kryo` files. The whiskers represent the upper and lower end of $1.5 \times \mathrm{IQR}$ where IQR specifies the inter quartile range. The green line indicates the median of the observations.

In order to calculate image features the camera pixels are weighted by the number

[1] A fixed part of the computers main memory is mounted as a file system. Access is managed by the OS kernel and is slower than direct access to RAM from the user space but still much faster than a hard disk drive.

of photons that hit the pixel. Assuming the weighted pixels adhere to a two dimensional Gaussian distribution, the width and length parameters are the second order moments of that distribution. The image parameters produced by the naive extraction methods used during this thesis already provide physically reasonable results. See figure 5.2. In the CTA Monte Carlo simulation the Proton induced showers typically have a greater width than Gamma induced showers.

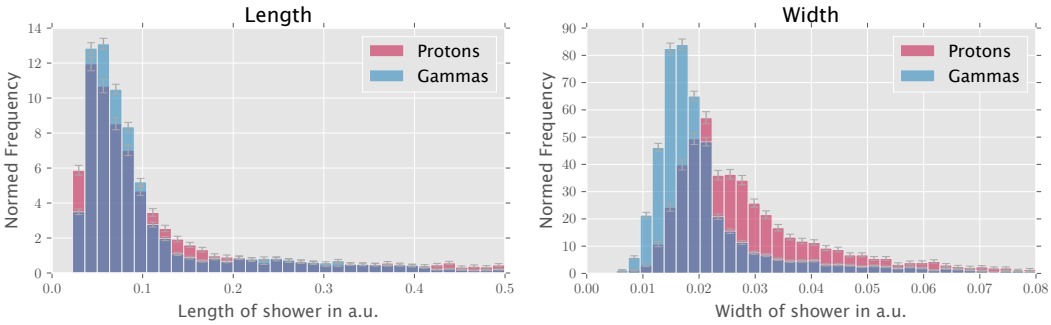


Figure 5.2: Histogram of the two image parameters width and length. Gammas are drawn in blue. Protons are drawn in red.

5.2 Multithreaded performance

To reach the target event rate of 15 kHz, the process was parallelized. The data is streamed into multiple copies of the same analysis process. A queue was used to collect the logged output of all threads into a single location. Efficiently parallelizing the stream means reducing the number of shared resources. The collecting queue is the only explicit shared resource in this stream. Other data structures needed for computation of the image parameters were designed to be immutable and can safely be copied across threads.. Figure 5.3 shows the event rate per thread while running on 8 cores in parallel. The plot shows that the workload was distributed almost equally across the used threads.

The colored boxes in figure 5.4 indicate the event rate for a single thread. The boxes are stacked to display the total event rate of the process. In order to test the stability and scalability of these results, the measurement was repeated 15 times using up to 48 threads in parallel. Figure 5.5 shows the result of theses measurements. The error bars indicate the upper and lower quartile of the sample. Event rates of over 60 kHz can be reached which, is four times higher than the original goal. This shows that the `streams`-framework can achieve the needed throughput to analyse CTA

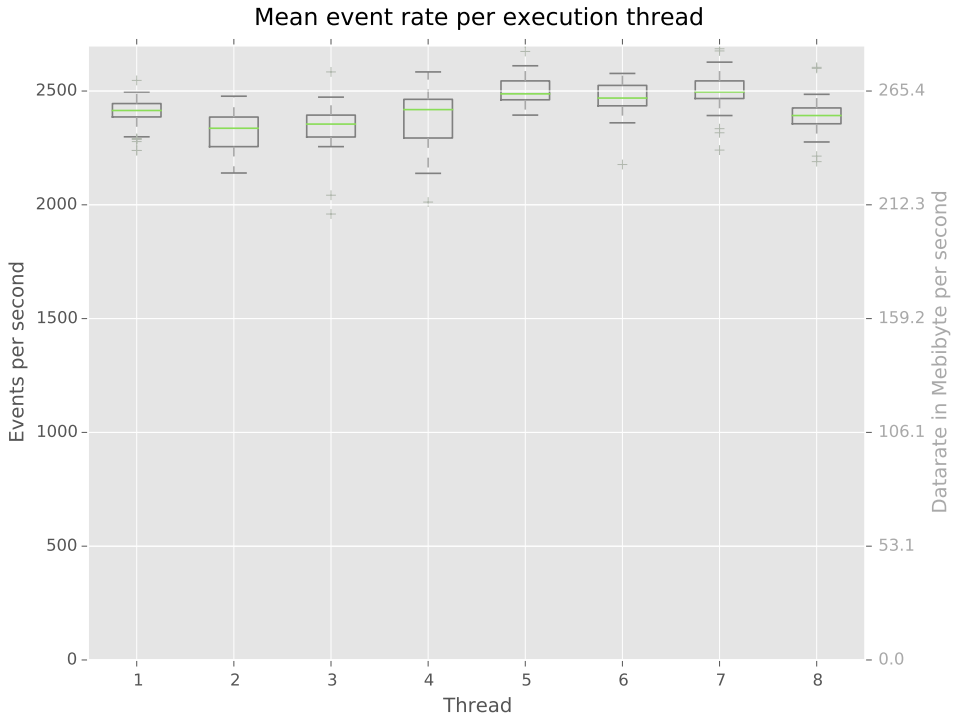


Figure 5.3: Event rate per thread when using 8 threads in parallel. The green bars show the median. The whiskers represent the upper and lower end of $1.5 \times \mathrm{IQR}$.

raw data in real time. The fluctuating error bars are due to the shared access to the machine. Especially when using many threads, users working on the same machine seem to have a strong impact on the stability of the result. Total data rate is limited by memory bandwidth and the number of physical cores. The test machine has 24 physical cores and 48 parallel threads. A measurement of memory bandwidth can be found in the Appendix A.2.

A data stream runs for an undefined amount of time. The telescopes that are part of CTA continuously produce data. An important requirement for any program working with continuos data streams is its performance and memory usage over long periods of time. In Figure 5.6 the event rate and memory usage are displayed over a period of about two hours.

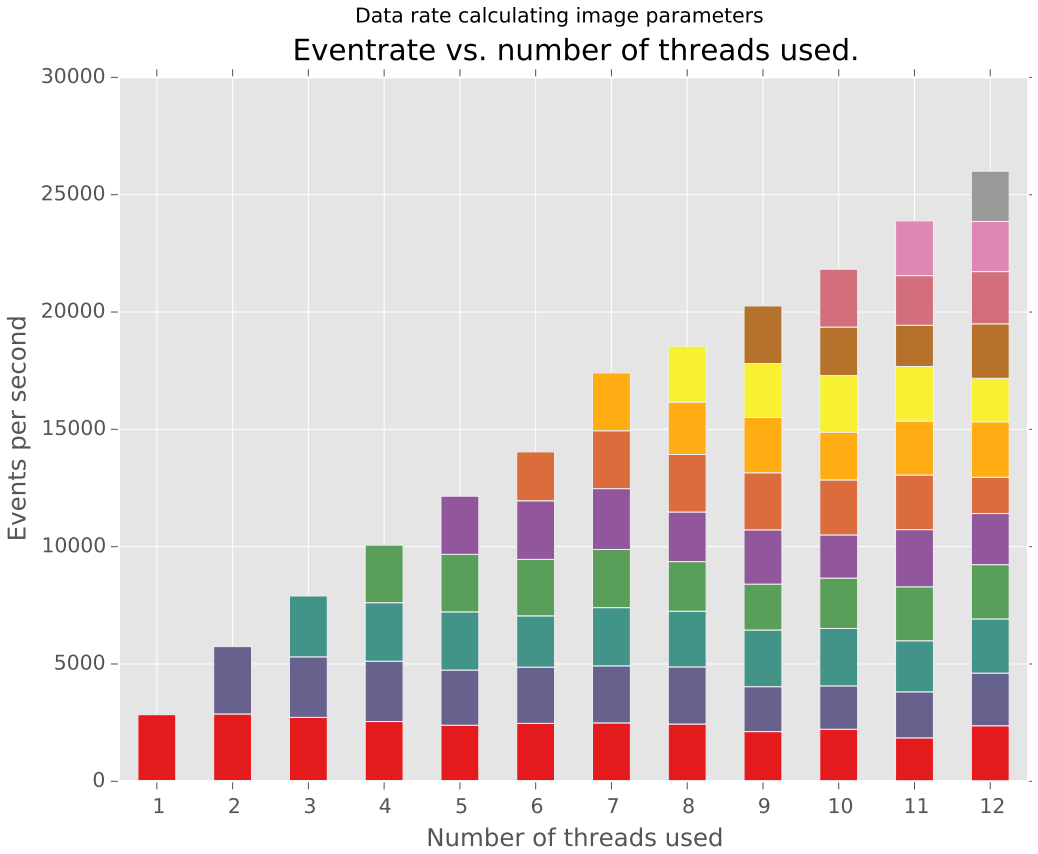


Figure 5.4: Runtime performance for analyzing files mounted in RAM. Using more CPU cores allows to reach event rates over 15 kHz

.

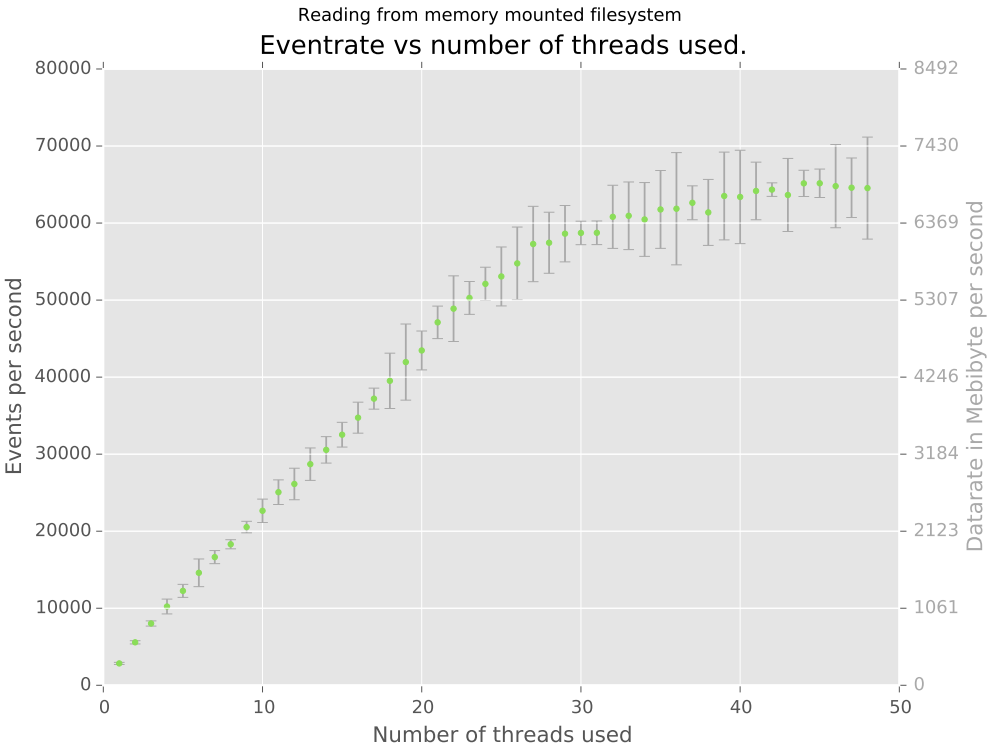


Figure 5.5: Runtime performance for analyzing files mounted in RAM. The error bars indicate the upper and lower quartile of the sample. The measurements were repeated 15 times to obtain error bars.

.

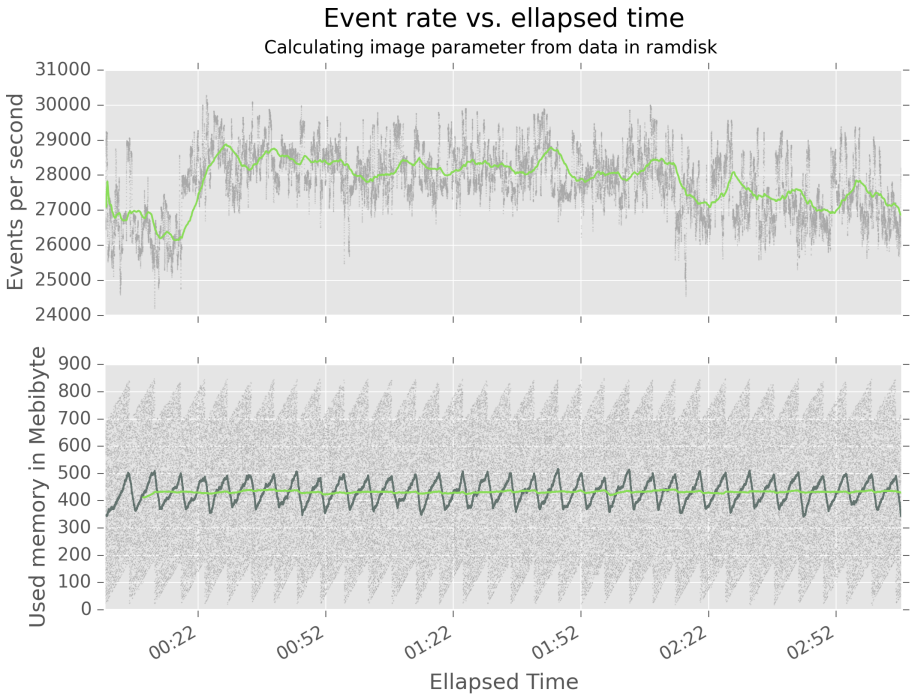


Figure 5.6: Event rate and memory usage over the course of approximately two hours. 12 threads were used during this test. The gray markers indicate sampling points which were taken after every 1000 data items. The light green line in the upper plot is a rolling mean over 2500 samples. In the lower plot the blue line is a rolling mean over 500 points and green line over 5000 points. The sawtooth pattern in the memory usage clearly shows the location of garbage collector sweeps in time.

5.3 Performance using ZeroMQ

With 1855 pixels and 30 samples each, at least $1855 \cdot 30 \cdot 2 = 111300$ byte have to be saved to memory at some point. A few more bytes are needed for timestamps, IDs and other meta data. In a 10GbE network this leads to a theoretical maximum rate of about 11200 events per second. When serialized to the wire the packet size goes down due to the compressed encoding of integers used by the serialization libraries discussed in section 3.3. The encoding efficiency for arrays of small integers is the same for both Protocol Buffers and MsgPack. They both create packages of no more than 56767 bytes which is about half of its original size. At least for Monte Carlo data this could, in theory, double the event rate send over a 10GbE network to about 22000 events per second. Protocol Buffers can be used without compression. The smallest integer data type is 32 bit. This leads to a package size of 222615 bytes. While these measurements are not suitable to infer general statements about encoding efficiency of the different libraries, they nevertheless apply to the CTA use case of serializing large arrays of small numbers. Serializing data and moving it through the operating system's networking stack introduces a non negligible amount of overhead. Data has to be wrapped into TCP packets and routed to the right destination. To measure runtime differences and to get an estimation of the overhead, Monte Carlo data was serialized and published via ZeroMQ. Another process on the same machine subscribed to the data and deserialized it. Due to lack of a properly working 10Gbit network infrastructure, all tests had to be performed on a single computer. In figure 5.7 the number of processed events per second using Protocol Buffer and MsgPack is depicted. The serialization step clearly worsens the runtime for both MsgPack and Protocol Buffer when compared to analyzing data directly from a ramdisk as in section 5.1. Using uncompressed Protocol Buffer data is not much faster than using the variable length integer encoding. The uncompressed packets are however about 4 times bigger than the compressed packets. It might be worth implementing a custom type for Protocol Buffer using only 16 bytes per sample.

Taking into account the overhead produced by the network stack and serialization process, the initial target event rate for receiving and analyzing CTA raw data over ZeroMQ was set to 8 kHz. Figure 5.8 shows the event rate achieved using 12 subscription threads over the course of about 14 hours. Event rates of over 12 kHz can be achieved and are stable over many hours. More optimizations and faster machines might allow to reach the full LST read-out rate of 15 kHz

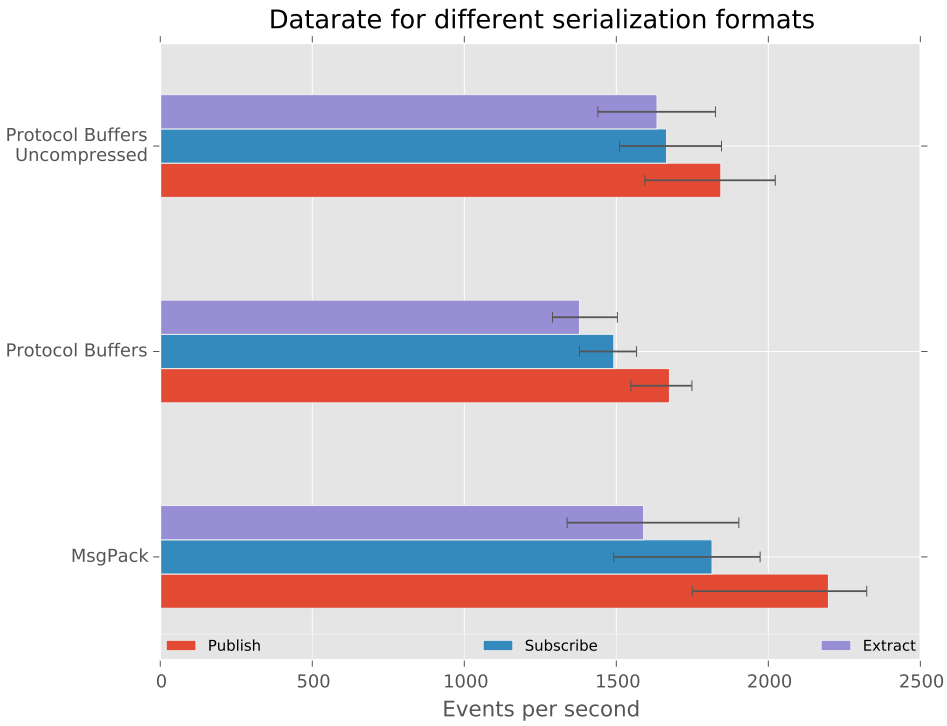


Figure 5.7: Mean event rate for publication, subscription and extraction of serialized events using ZeroMQ for transportation. Measurements were performed on a single machine in dedicated threads for publication and subscription. The error bars indicate upper and lower 0.025 quantile.

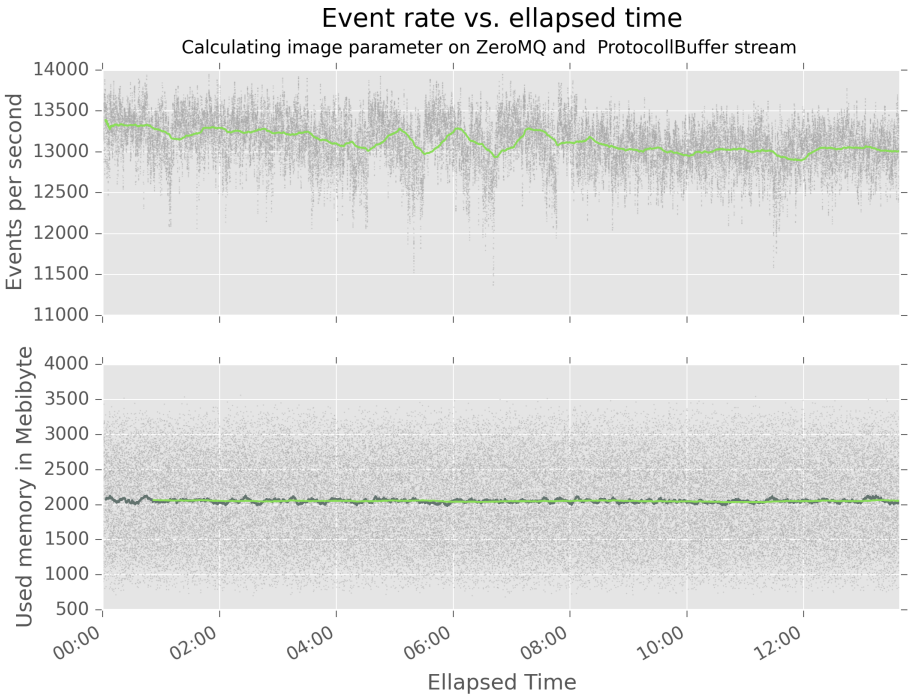


Figure 5.8: Event rate and memory usage over the course of approximately 14 hours. 12 threads were used during this test. The gray markers indicate sampling points which were taken after every 3000 data items. The light green line in the upper plot is a rolling mean over 2500 samples. In the lower plot the blue line is a rolling mean over 500 points and green line over 5000 points.

6 Conclusion and future work

The goal of this thesis was to evaluate the `streams`-framework for the CTA use-case and measure its throughput under different setups. Simple extraction and image parametrization algorithms were implemented and applied to CTA simulation data to reduce the raw voltage curves into meaningful image parameters. The performance goals set for this thesis were met and even surpassed by a large margin. Its possible to calculate image parameters from raw data with more than 60 000 events per second. The `streams`-framework was successfully extended to stream data over network using ZeroMQ with event rates of over 8 kHz. Memory usage and runtime performance were shown to be stable over long time periods. Two serialization solutions, Protocol Buffers and MsgPack were compared in terms of usage and performance. Code from the FACT-Tools [9] [7] was adapted to create a graphical user interface for visualizing CTA Monte Carlo data.

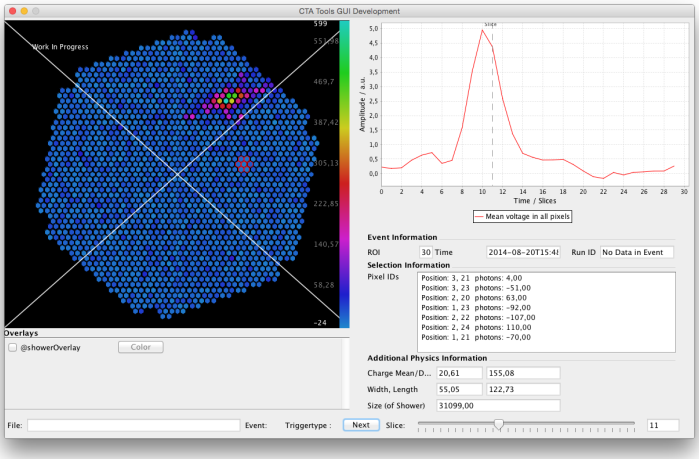


Figure 6.1: A screenshot of the graphical user interface for CTA Monte Carlo data showing pixels from a Large Size Telescope.

Once more accurate low level Monte Carlos simulations exist, more elaborate methods for feature extraction can be developed. While these will have a negative impact on overall performance, its likely that the initial goal of 15 kHz for raw data processing can be achieved even then. In the future it is planned to integrate the `streams`-framework completely into the official CTA DAQ framework.

A Images

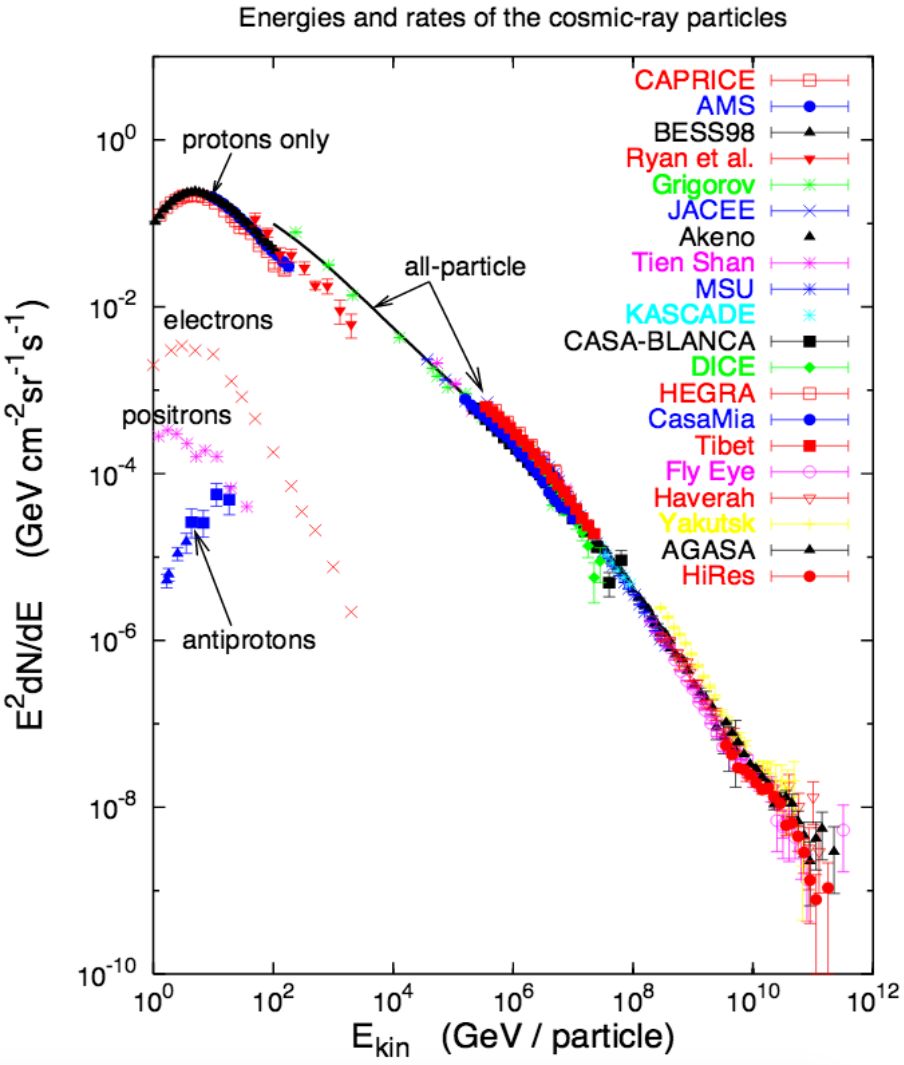


Figure A.1: Cosmic Ray flux as measured by different experiments [16].

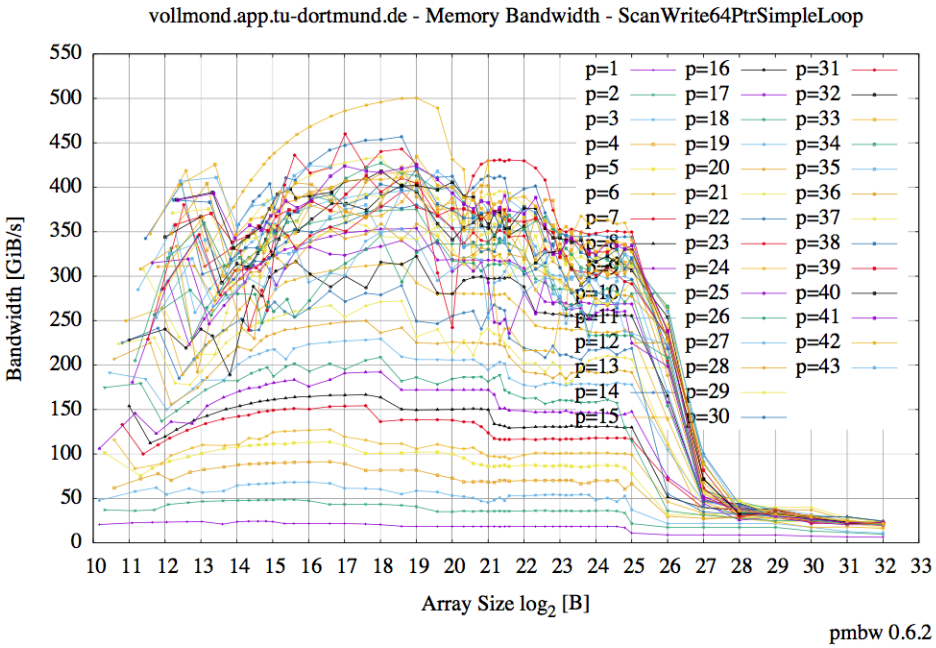


Figure A.2: Memory bandwidth measured on the test machine. Measurement was performed the by pmbw program [6].

List of Figures

B Listings

Listing B.1: The `proto` description for the data items used as CTA events.

```
syntax="proto2"
package streams.cta.io.protobuf;

option java_package = "streams.cta.io.protobuf";
option java_outer_classname = "RawCTAEvent";

message RawEvent {

    required string messageType = 1;

    required int32 telescope_id = 2;
    required int32 roi = 3;
    required int32 num_pixel = 4;
    //uses variable length encoding for integers
    repeated int32 samples = 5 [packed=true];

    //this uses fixed length for the integers
    //repeated sfixed32 samples = 5 [packed=true];

}
```

List of listings

C Abbreviations and Acronyms

DAC Digital Analog Converter

ERM Entity Relationship Model

CTA Cherenkov Telescope Array

KB Kilo Byte

EULA End User License Agreement

NACK Negative Acknowledgment

BVB Ballspielverein Borussia

ESRF European Synchrotron Radiation Facility

HTTP Hyper Text Transfer Protocoll

AGN active galactic nuclei

ADC Analog Digital Converter

RTA real time analysis

UDP User Datagram Protocoll

Ns Nano Second

GRB Gamma Ray Burst

LST Large Size Telescope

MST Medium Size Telescope

SST Small Size Telescope

JDK Java Development Kit

JVM Java Virtual Machine

FoV Field of View

SNR supernova remnant

DAQ data acquisition

SFB Sonderforschungsbereich

ZMQ Zero Message Queue

FACT First G-APD Cherenkov Telescope

SiPM Silicon Photo Multiplier

TCP Transmission Control Protocol

PMT photo multiplier tube

IACT Imaging Atmospheric Cherenkov Telescope

XML Extensible Markup Language

IQR Inter Quartile Range

Bibliography

[1] S. A. Russo. "Using the Hadoop/MapReduce approach for monitoring the CERN storage system and improving the ATLAS computing model". PhD thesis. Udine U. URL: https://inspirehep.net/record/1296389/files/327651183_CERN-THESIS-2013-067.pdf.

[2] D. Abdurachmanov et al. "Optimizing CMS build infrastructure via Apache Mesos". In: *ArXiv e-prints* (2015-07). arXiv: 1507.07429 [cs.DC].

[3] B. Acharya et al. "Introducing the CTA concept". In: *Astroparticle Physics* 43 (2013-03), pp. 3–18. ISSN: 09276505. DOI: 10.1016/j.astropartphys.2013.01.007. URL: http://www.sciencedirect.com/science/article/pii/S0927650513000169.

[4] L. Bergström. "Dark matter and imaging air Cherenkov arrays". In: *Astroparticle Physics* 43 (2013-03), pp. 44–49. ISSN: 09276505. DOI: 10.1016/j.astropartphys.2012.04.010. URL: http://www.sciencedirect.com/science/article/pii/S092765051200093X.

[5] K. Bernlöhr et al. for the CTA Consortium. "Monte Carlo design studies for the Cherenkov Telescope Array". In: *arXiv* (2012). arXiv: 1210.3503 [astro-ph.IM].

[6] T. Bingmann. *pmbw - Parallel Memory Bandwidth Benchmark / Measurement*. URL: https://panthema.net/2013/pmbw/ (visited on 10/17/2015).

[7] C. Bockermann et al. "Online Analysis of High-Volume Data Streams in Astroparticle Physics". In: *Machine Learning: ECML 2015, Industrial Track*. Springer Berlin Heidelberg, 2015.

[8] M. Braeger (CERN) and M. Devgan(Software AG Terracotta). *Unlocking Big Data at CERN*. URL: http://strataconf.com/stratany2014/public/schedule/detail/36312.

[9] K. Brügge et al. "FACT-Tools: Streamed Real-Time Data Analysis". In: *Proceedings, 34th International Cosmic Ray Conference (ICRC 2015)*. Vol. ICRC2015. 865. 2015. URL: http://pos.sissa.it/cgi-bin/reader/conf.cgi?confid=236.

[10] J. Cortina and M. Teshima for the CTA Consortium. "Status of the Cherenkov Telescope Array's Large Size Telescopes". In: *ArXiv e-prints* (2015-08). arXiv: 1508.06438 [astro-ph.IM].

[11] S. C. DESY / Milde. *Artistic computer rendering of the CTA*. URL: https://portal.cta-observatory.org/Pages/Home.aspx (visited on 09/30/2015).

[12] Esoteric Software. *kryo Github Repository*. URL: https://github.com/EsotericSoftware/kryo.

[13] M. Fuessling, P. Wegener, et al. for the CTA Consortium. *Array Control and Data Acquisition technical design report*. 2015-3.

[14] Google Inc. *ProtocollBuffers Github Repository*. URL: https://github.com/google/protobuf.

[15] A. M. Hillas. "Cerenkov light images of EAS produced by primary gamma". In: *International Cosmic Ray Conference* 3 (1985-08), pp. 445–448.

[16] A. M. Hillas. "Cosmic Rays: Recent Progress and some Current Questions". In: *ArXiv Astrophysics e-prints* (2006-07). eprint: astro-ph/0607109.

[17] P. Hintjens. *ZeroMQ: The Guide*. 2010. URL: http://zguide.zeromq.org/page:all.

[18] *The JSON Data Interchange Format*. Tech. rep. Standard ECMA-404 1st Edition / October 2013. ECMA, 2013-10. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

[19] E. Lyard et al. "Modern middleware for the data acquisition of the Cherenkov Telescope Array". In: vol. ICRC2015. 2015-08. arXiv: 1508.06473 [astro-ph.IM].

[20] A. Toshniwal et al. "Storm@Twitter". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: ACM, 2014, pp. 147–156. ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2595641. URL: http://doi.acm.org/10.1145/2588555.2595641.

[21] I. Ward. *JSON Lines*. URL: http://jsonlines.org/ (visited on 09/30/2015).

Eidesstattliche Versicherung

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit mit dem Titel “Evaluating throughput performance of the `streams`-framework on Cherenkov Telescope Array data” selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum	Unterschrift

Belehrung

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50 000 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden (§ 63 Abs. 5 Hochschulgesetz –HG–).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z. B. die Software “turnitin”) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen.

Ort, Datum	Unterschrift