Karlsruhe Institute of Technology

ETP-BACHELOR-KA/2018-11

ØMQ Implementation for a Reliable Parallel Processing Framework on the High Level Trigger at the Belle II Experiment

Anselm Baur

Bachelorthesis

31.07.2018

Institute of Experimental Particle Physics (ETP)

Advisor:	Prof. Dr. Florian Bernlochner
Coadvisor:	Dr. Thomas Hauth

Editing time: 12.03.2018 – 31.07.2018

Karlsruhe Institute of Technology

ETP-BACHELOR-KA/2018-11

ØMQ Implementierung eines betriebssicheren Multiprozessing-Frameworks für den High Level Trigger am Belle II Experiment

Anselm Baur

Bachelorarbeit

31.07.2018

Institut für experimentelle Teilchenphysik (ETP)

Referent:	Prof. Dr. Florian Bernlochner
Korreferent:	Dr. Thomas Hauth

Bearbeitungszeit: 12.03.2018 – 31.07.2018

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, 31.07.2018

. . . .

(**Anselm Baur**)

Contents

1. Introduction

One of the most fundamental characteristics of the human being is the quest for knowledge and understanding of the fundamental phenomena of nature. Over the centuries, outstanding minds of mankind have created a physical construct to explain why our world works the way it does. This is a process that is far from completion. Daring and at the same time ingenious theories that have been developed describing the nature must be constantly tested and refined. For this reason, equally ingenious state-of-the-art experiments are absolutely essential. A theoretical model of modern physics that has so far passed all experiments is the Standard Model of particle physics. However, the Standard Model does not explain all pending questions. For example, to explain the baryon asymmetry after the big bang, the Standard Model of particle physics delivers no explanation. Nowadays, complicated experiments are required to find answers to open questions or to find new physics. Thus, expensive high technological and complex experiments such as Belle II are performed by scientists of many nations working hand in hand regardless of religion or ethnic origin. The technologies of these experiments are so advanced that some of them have to be invented for their construction.

In high-energy physics (HEP) huge colliders are built with giant detectors consisting of today's best technologies. However, no matter how sophisticated the hardware is if it cannot be operate properly and the data it provides cannot be processed it is downgraded to an useless piece of electrical waste. Modern hardware must also be operated with modern well-performing software.

The higher the luminosity of the colliders, the higher is the number of the particle interactions and the higher is the event data rate which has to be recorded by the detectors. In particular, the Belle II experiment is supplied with electrons and positrons by the SuperKEKB collider, which will operate on a luminosity that has never been achieved before. Since the storage of all detector data exceeds the capacity of the memory, it must be processed immediately to reduce it to a storable amount. For this reason, only for later offline analysis relevant event data is stored. In general, the single event processing takes too much time for all events to be processed sequentially. The throughput is a significant bottleneck in the readout speed of the detector. Fortunately, the problem of a large number of incoming independent events in a short time is subject to the law of Gustafson[1]. This means, increasing the amount of independent data within a time window can be compensated by increasing the processing resources. The single event processing time cannot be reduced with increasing the processing resources. Instead, the number of events, which are processed in the same single processing time - the throughput - can be increased. This is necessary for HEP experiments because of the high amount of incoming data in short periods. It would be fatal if due to full hardware buffers the existing event data would be overwritten. In the offline analysis completely wrong conclusions could be drawn from decay rates. Therefore, the data extraction is stopped when the buffers are full. Stopping the measurement because of such full buffers stops also the recording of any collision data. Therefore it is essential to readout the data of the hardware fast enough to avoid such problems. To reduce the incoming data to a storable amount, a whole server farm is filtering the event data at the Belle II experiment online. Due to this, the software framework which splits the event data to processes it in parallel mode and collects the results has to run stable and reliable.

Scientists developing software should follow modern approaches and standards of software engineering. Software must be maintainable and enhanceable. Often, bugs only become apparent during operation and must be fixed with a software update. Regularly, certain functionalities also have to be adapted to ensure, that the software is always optimally designed to meet the specific requirements. Therefore, it is essential that the implemented code is easy and clear. The core structure of the software must be built of individual easy modules which can be maintained, modified, extended or improved without causing the entire software to collapse.

To meet the aforementioned requirements and to improve the stability and reliability of the parallel framework of the Belle II analysis software framework, a message-based multiprocessing approach using the ØMQ libraries was newly implemented during this thesis. Additional features were developed to lay an easily to extend foundation that allows the monitoring and controlling of the processes. The goal is to provide a basis for a trouble-free software operation during the data acquisition of the Belle II experiment. Therefore, a tracing of dying or hanging worker processes and a self-healing method was added to the parallel framework. Further, the event data will be stored as a backup during processing, so data loss is avoided. The new implementation of the parallel framework and the new features has been tested for runtime and reliability to ensure they meet the requirements of the HLT farm of the Belle II experiment.

2. The Belle II Experiment

The Belle II experiment was designed to measure rare particle decays, to perform meson spectroscopy and to search for new physics beyond the Standard Model of particle physics. Belle II aims to collect a data set which is fifty times higher than its predecessor Belle. This chapter will give a short overview of the technical setup.

2.1 Setup of the Experiment

Belle II is located at the KEK accelerator facility in Japan. The SuperKEKB collider at KEK is as the former KEKB accelerator, a so-called B-factory, and designed to produce a large number of $\mathrm{B\overline{B}}$ meson pairs. For that, positrons and electrons are accelerated and brought to collision at the interaction point in the center of the Belle II detector.

In the low energy ring (LER) the positrons are accelerated to an energy of 4 GeV and in the high energy ring (HER) the electrons are accelerated to 7 GeV. Because of the asymmetric beam energies this leads to a Lorentz boost of $\gamma\beta = 0.28$ in direction of the accelerated electrons which is necessary to measure the decay times of B mesons. The Lorentz boost increases the lifetime of the B mesons in the observer's system according to the theory of special relativity. The collision happens at a center-of-mass energy of 10.58 GeV which corresponds to the mass of the Υ(4S)-resonance. In more than 96 % of the Υ(4S) decays, a $\mathrm{B\overline{B}}$ meson pair is produced with a mean lifetime of approximately 1.5 ps in the un-boosted system. This corresponds to a B meson propagation of approximately 130 μm.

To achieve the required statistics for the search of new physics, SuperKEKB is planned to run with a record-breaking luminosity of $8 \times 10^{35}\,\mathrm{cm^{-2}\,s^{-1}}$. This is forty times higher than the luminosity of the previous KEKB collider of the former Belle experiment. A schematic setup of SuperKEKB is shown in Figure 2.1.

The Belle II detector can detect all final-state particles created by the collisions except neutrinos. For this, the detector is equipped with various subdetectors around the interaction point:

PXD/SVD/CDC: The tasks of the pixel detector (PXD), the strip vertex detector (SVD) and the central drift chamber (CDC) are to measure the momentum and the energy loss through ionization (dE/dX) properties of charged particles in a homogeneous 1.5 T magnetic field parallel to the beam direction. Because of the Lorentz force, the charged particles move on a helical track in the magnetic field. With the high

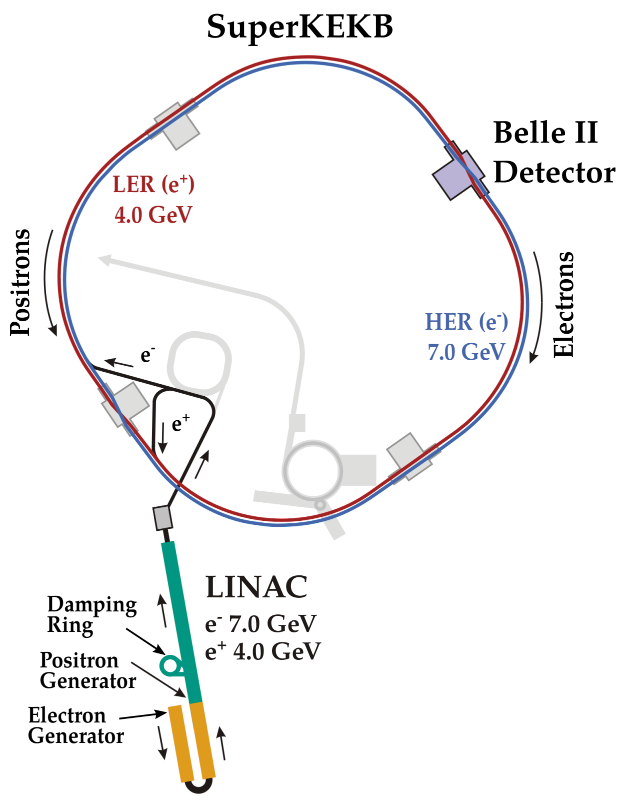

Figure 2.1: Schematic setup of the SuperKEKB collider with the Belle II detector at the KEK accelerator facility in Japan. The electrons and positrons are accelerated in the LINAC to their final energy and injected then into the storage rings. The interaction point where the two contra-rotating beams are brought to collision is in the center of the Belle II detector. Figure adapted from [2].

precision of the PXD and the SVD together with the large lever arm of the CDC, the momentum of the particles can be measured very precisely.

TOP/ARICH: To be able to distinguish between pions and kaons, two subdetectors are installed: A time-of-propagation (TOP) counter in the barrel region and an aerogel ring imaging Cherenkov detector (ARICH) in the direction of the boosted center-of-mass system in the endcap of the detector.

ECL: With the electromagnetic calorimeter (ECL) neutral photons can be detected by measuring the energy deposit of the photons with scintillating crystals. Also electrons are fully stopped in the ECL.

KLM: The K_L^0 and μ detector (KLM) is installed at the outer region of the detector.

The detector and its subdetectors are shown in Figure 2.2. A more detailed description of the whole detector and its expected performance is given in the *Belle II technical design report* [3].

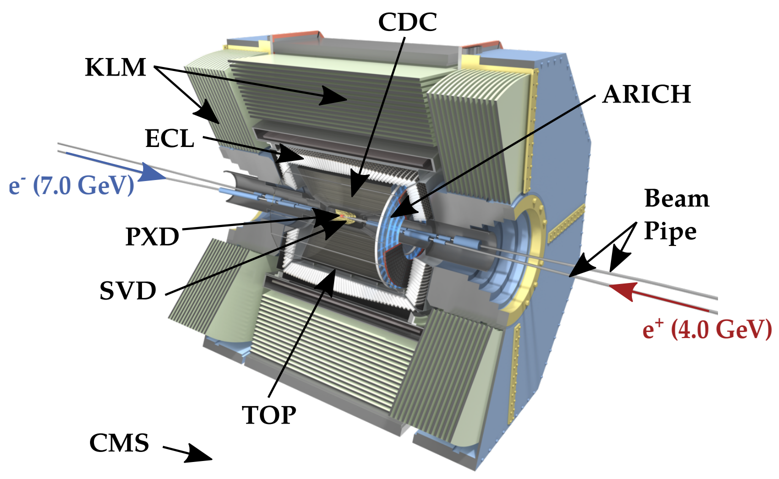

Figure 2.2: The Belle II detector at the interaction point with the subdetectors. Figure adapted from [4].

Due to the high peak luminosity a high bunch crossing rate results, which leads to a particle collision approximately every 4 ns. It is impossible to store the data of each crossing at this rate. To reduce this huge amount of data, online trigger systems have to filter out the (physical) processes of interest (e.g. $\Upsilon(4S) \rightarrow B\overline{B}$ or $e^-e^+ \rightarrow q\bar{q}$). The detector data has to pass two online trigger systems. First, the hardware trigger system (Level 1 trigger) built from independent sub-trigger systems of the subdetectors, excluding the PXD (see Section 2.2). With its final decision logic, the Level 1 trigger decides using sub-trigger data whether to keep or discard an event. The combined decision of the independent sub-trigger decisions is made in the global decision logic (GDL). The two main output requirements for the Level 1 trigger to prevent a readout buffer overflow are:

- maximum average trigger rate of 30 kHz
- minimum two-event separation of 200 ns

After the Level 1 trigger, the event data is combined with the so-called event builder and an online software trigger is applied to further reduce the rate of data. The software trigger is called high level trigger (HLT) whose decision logic is applied before the data acquisition (DAQ) system. The DAQ reads out the detector signals passed to the Level 1 trigger and the HLT. The event data is then finally saved on the storage system. The HLT is described in Section 2.2. A more detailed description of the Level-1 trigger is also given in the *Belle II technical design report* [3].

The anticipated start-up time line in which the data is recorded from the Belle II experiment is separated into three phases [5].

Phase I: Started in February 2016, the goal of the first phase was to commission the SuberKEKB accelerator and to characterize the beam environment. Phase I ended in June 2016.

Phase II: On March 2018 electron and positron beams were stored successfully in the HER and LER of the SuperKEKB accelerator complex for the first time. On 26th of April

2018, the first beam collisions were detected. The first discovered candidate for a hadronic event is shown in Figure 2.3. The first collision data was collected without the most inner silicon-based VXD tracking system. The goal of this phase is to study the sub-detectors and the machine background.

Phase III: This phase is scheduled to start end of March 2019: Collision data will be recorded with the complete Belle II detector. The main goal is to record a sizable data set of B mesons produced at the center-of-mass energy of the $\Upsilon(4S)$-resonance.

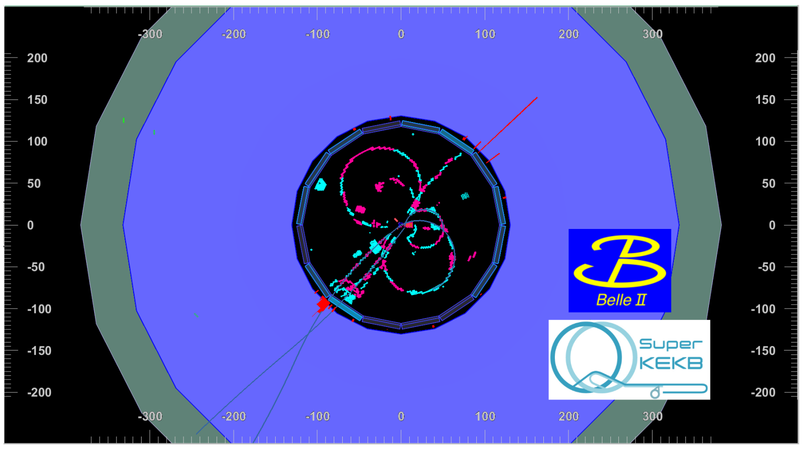

Figure 2.3: Event display of the first phase 2 collision experiments. First seen candidate for a hadronic event (event 223 in run 125 of experiment 3). [6]

2.2 The High Level Trigger

The tasks of the HLT are to reduce the incoming event data rate from approximately 3 GB/s to 1 GB/s and to determine the relevant regions of interest (ROI) to read out of the PXD sub-detector. A large server farm running with a powerful software framework is set up to provide the required computing power. The software framework of the HLT, which is the same as for offline reconstruction, is described in more detail in Section 2.3. When SuperKEKB will operate on peak luminosity, approximately 10 HLT units with approximately 400 CPUs per unit are planned to be used [7]. This corresponds to a total CPU number of $\sim$ 4000 cores, which is also the total number of events that can be processed in parallel. This would correspond approximately to a maximal processing time of 130 ms for each process to ensure the necessary throughput by an event rate of 30 kHz.

The first task is to reduce the incoming event data rate of 3 GB/s to a third of its original value. Therefore, a full reconstruction of the event data is performed, followed by an event selection based on physic-level variables. As the reconstruction is fairly slow, a two step approach is needed. In step one a fast reconstruction of the event data is performed to determine the tracks and the physical properties of the particles. A fast event reconstruction with the CDC and ECL data is executed. The number of tracks, the event vertex position, and the total deposited energy are considered for a decision. In the second step, the results of the event reconstruction are used for the physic-level event selection.

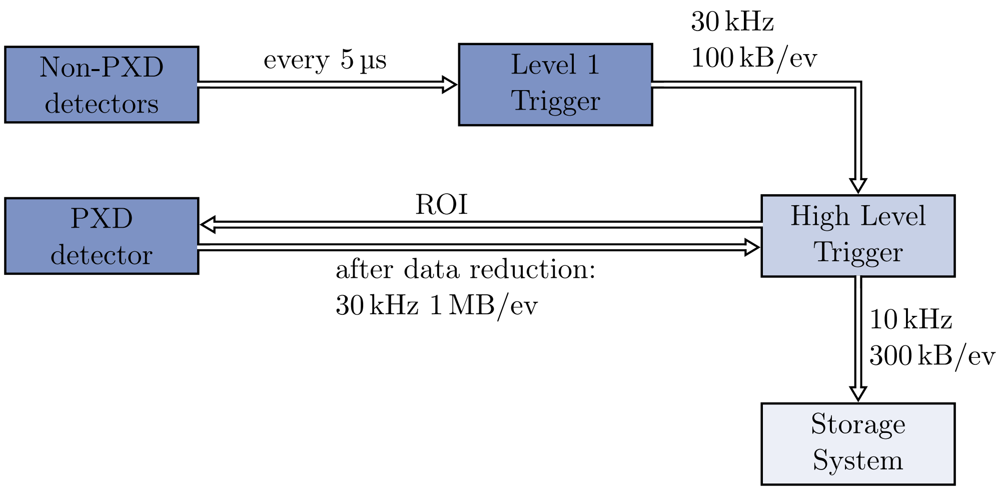

Figure 2.4: Schematic overview of the detector signals passing the two online trigger stages. If both trigger decisions are positive the non-PXD and PXD detector data in the ROIs is combined and written out to the storage for later offline reconstruction. The figure is taken and adapted from [8].

The other important task of the HLT is to reduce the huge amount of data from the PXD. Because the PXD operates with a long integration time a huge amount of single evesnts hit the pixels. Therefore the read out data must be reduced. The online HLT reconstruction is used to select regions of interest in the PXD. Only the PXD data in the ROIs is stored with the remaining subdetectors data of the events selected by the HLT in the final storage system for later offline reconstruction. Figure 2.4 shows a schematic chart of the data flow from the detector to the final data storage.

On the one hand, the processing time for a single event is one main issue. On the other hand, the event data has to be processed in parallel to obtain the required throughput to process the event data stream. Regarding the throughput it is necessary to use all available cores of the HLT. For this very reason, a stable software operation for parallel processing on the HLT is necessary. In order to improve the reliability and stability of the HLT software, which meets the requirements of modern software standards, a new implementation of the parallel processing core was developed during this thesis.

2.3 The Belle II Analysis Software Framework

The Belle II analysis software framework (basf2) is written in C++ and designed to process large data sets in a short time. Additionally, it offer a wide range of usage. It is developed for both online and offline reconstruction on several systems like the HLT, computing grids or a local workstation. The framework has different tasks, e.g. event generation, detector simulation, tracking, physics analyses, and more. This chapter gives a short overview of the basf2 architecture and the parallel processing functionalities.

2.3.1 Overview

The functionalities of basf2 are encapsulated in reusable and independent modules managed by a core framework. In particular, the modules can be loaded as required. The modules can include external libraries for special tasks. To build applications, the modules are arranged in so-called paths in Python scripts - the steering files. The core framework of basf2 is responsible for configuration, executing the module path or data exchange between the modules among other functions. This is a standard architecture for software frameworks used in high-energy physics (HEP). [9]

Complex event reconstruction or analysis applications can be built based on the modules. When the steering file is executed, the core framework calls the modules once for each event in the order of the module positions in the path. To share the event data as well as generated data between the modules, a so-called data store is used. Each module is allowed to read and write to the data store. A module path interacting with the data store is illustrated in Figure 2.5. [10]

The module path must contain at the beginning an module which loads event data into the data store. For example from the storage or from the event builder. At the end of the path must be placed a module to write out the data to the storage, for example in a ROOT file. To stream the data to another location via inter-process or inter-node communication, the data store object has a built-in function to serialize the data into a byte stream for transportation.

Each Module in basf2 is a child class of the `Module` class. It has predefined member functions like the `initialize()`, `event()`, `terminate()`, etc. that are called from the framework in specific program sequences while executing a path. For example the `initialize()` function is called on each new startup of basf2, the `event()` function is called for each event and the `terminate()` function is called when basf2 is exited.

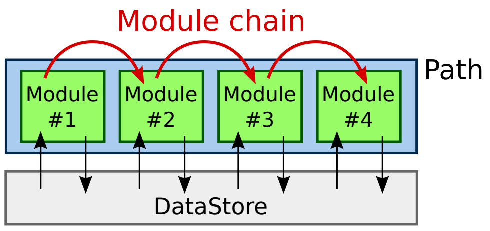

Figure 2.5: A path built of modules in a steering file with the data store for data exchange between the modules. Figrue is taken from [10, p. 2].

2.3.2 The Parallel Processing Framework

As in Section 2.2 described, basf2 needs to support parallel processing. In general a computing structure as shown in Figure 2.6 is implemented. A ventilator distributing the workload to a set of workers which can process the tasks parallel and a sink collecting the results back from the workers.

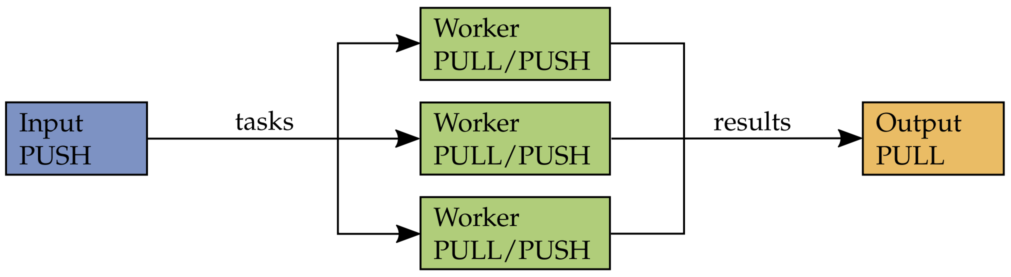

Figure 2.6: Schematic structure of task distribution by an input to a set of parallel workers. The computing results are collected by the output. Figure adapted from [11, p. 16].

Modules are parallel processing safe if they do all their I/O through the data store so that no conflicts e.g. through writing the same file occurs. If no modules were found with the parallel processing certified flag the single process sequence will be run.

For parallel processing the module path is divided into three parts. This is needed to separate the not parallel processing suitable I/O modules from the parallel processing safe modules. Accordingly, three individual paths are created for the respective process types:

- input process: loading event data (one process)
- worker process: processing event data (a specific number of parallel working processes)
- output process: storing event data (one process)

In the parallel framework, the event processor takes care of splitting up the path in parallel and serial parts. Therefore, the event processor searches parallel processing certified embedded modules in the path. If a sequence of modules is found which have the parallel processing certified flag, the path will be split by the event processor in the above described three individual paths. The parallel processing certified modules will be processed parallel in the worker processes. To transfer data between the processes, the event processor inserts before and after the worker process path so-called ring buffers. Figure 2.7 shows schematically the division of the path and the inserting of the ring buffers.

To avoid coincidental read or write access to the ring buffer, each process has to lock the ring buffer and then release it again. During this period no other process can access data on the ring buffer. This is a critical bottleneck in the parallel processing framework of basf2. Meanwhile, there are new approaches for distributing data between processes of a parallel processing application. One modern and common approach is described in Section 3.1. Another flaw of the ring buffers are that there is no event backup and process restart when processes have died while execution. But this is important for an application like the HLT. This application needs a reliable and stable operation of the software for a permanent high performance of nearly real-time reconstruction with the detector signals. For this reason, the basf2 parallel processing core has been newly implemented in this thesis and extended with some important missing features, using the ØMQ libraries as described in Chapter 3 and 4.

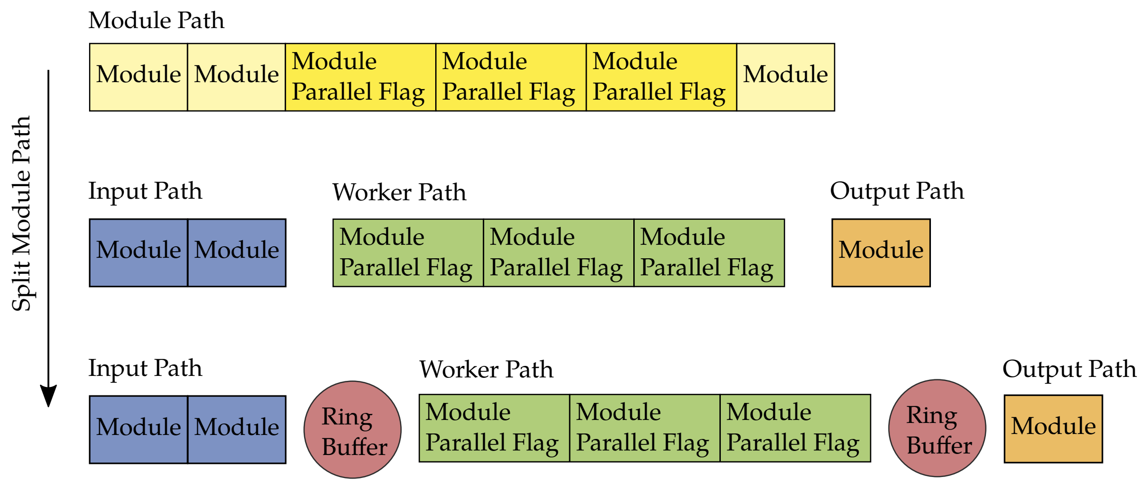

Figure 2.7: Schematic illustration of splitting the module path for parallel processing. In step 1 the modules are identified for parallel processing by reference to the parallel processing certified flag. In step 2 the module path is split into pieces suitable for parallel processing. In step 3 the ring buffers are arranged in the path before and after the parallel certified path piece.

3. New Implementation of the Parallel Processing Framework in basf2

The time-critical ring buffer locking, the lack of process monitoring and process controlling as well as possible event data loss are crucial weak spots (cf. Section 2.3.2). To provide a modern parallel framework which is easy to extend for new features, the parallel framework was rebuilt using the ØMQ library applying a message-based distribution of event data. This work builds on the first implementation by Thomas Hauth and Nils Braun [12]. This provides a basis to extend the framework with important new features. The new features are illustrated in Chapter 4. The current chapter gives a short overview of ØMQ and describes how the basic functionalities of the parallel framework were reimplemented in ØMQ.

3.1 Introduction to ØMQ

ØMQ (also ZMQ or ZeroMQ) is an asynchronous message oriented open source library based on sockets. It is designed to connect code of applications where fast working runtime is required to solve large problems with simple modules. ØMQ can be used for fast communication across in-process, inter-process or Transmission Control Protocol (TCP) between parallel working processes on the same workstation or in a network. In addition, the asynchronous I/O model of ØMQ enables to develop scalable multi-core applications. ØMQ is written in C and additional C++ bindings are available. The software is open source and has a large community which provides constant maintenance of the code. A full software manual is available as a printed edition [11] or from free online source [13]. ØMQ transfers arbitrary data in messages between sockets. The developers describe the ØMQ socket, a *super hero* derivative of a TCP socket, as follows:

"... a ØMQ socket is what you get when you take a normal TCP socket, inject it with a mix of radioactive isotopes stolen from a secret Soviet atomic research project, bombarded it with 1950-era cosmic rays, and put it into the hands of a drug-addled comic book author with a badly-disguised fetish for bulging muscles clad in spandex. Yes, ØMQ sockets are the world-saving superheroes of the networking world." [11, p. 9]

With that creative description, the developers emphasize in the ØMQ manual the easy to use and yet powerful ØMQ sockets. Sockets have long been an industry standard developed for applications. They offer an interface for network implementations. Sockets are explained in more detail in [14].

Message handling is done automatically by ØMQ with an additional background thread. As a result, messages can be received by the ØMQ socket autonomously in the background. The messages are stored in the message queue, from where they can be collected from later. Messages are easily transmitted between the sockets and ØMQ does all the work for various transport methods. ØMQ messages can consist of multiple parts, so-called message frames (Fig. 3.1). The message frames in the ØMQ message are sent sequentially via a ØMQ socket to a peer. The respective peer receives the message frames in the correct order at its corresponding ØMQ socket. ØMQ messages can have arbitrary size which is just limited by the memory of the used hardware. [11]

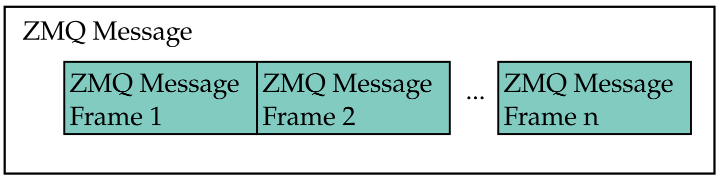

Figure 3.1: Several ØMQ message frames can be arranged to a ØMQ message which can be send by a ØMQ socket and received by a peer with the correct order of the message frames in the queue.

Almost any network topology can be realized with ØMQ. For this purpose, ØMQ provides a repertoire of different sockets with different characteristics. The socket types which are interesting for this thesis are briefly described below:

Push/Pull: The `ZMQ_PUSH` and `ZMQ_PULL` socket pair is designed for an easy 1-1 or N-1 connection. The sender opens the push socket and the receiver the pull socket. Figure 3.2 shows this connection pattern. Using this socket pair, the receiver gets no information of the identity of the sender of the received messages. It is possible to store the identity information in the message additionally but there are more comfortable ways to implement this, for example with the dealer socket.

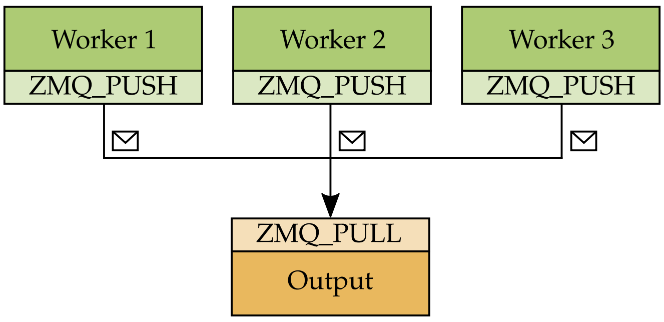

Figure 3.2: Illustration of a N-1 connection with ØMQ push and pull sockets. The graphic shows this with three worker processes and one output process. Figure adapted from [11, p. 16].

Router/Dealer: The `ZMQ_ROUTER` and `ZMQ_DEALER` socket pair is used for an 1-N messaging pattern. Because the dealer sockets have unique identities (`ZMQ_IDENTITY`), the router socket can route messages to a specific dealer. Figure 3.3 shows this connection pattern. To tell the router where to send the message a message frame with the dealer identity is added at the beginning of the ØMQ message. ØMQ Messages received from the router sent by the dealer also automatically contain a dealer identity message frame at the beginning. One characteristic of this connection type is that the router must know the identity of the peers.

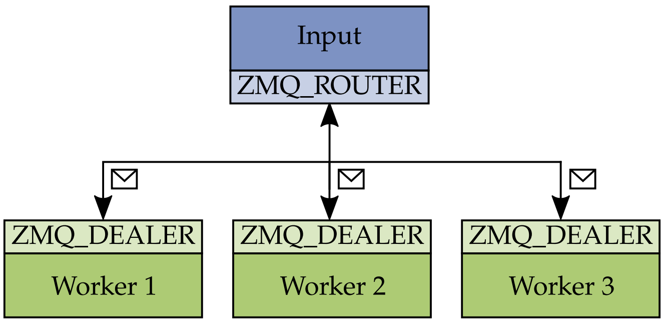

Figure 3.3: This graphic shows an 1-N connection built by ØMQ router and dealer sockets. Here an input process sends messages to a specific worker process. If a worker process sends a message to the input process, the corresponding worker identity will be transmitted as well.

Publish/Subscribe: To send many peers the same messages at the same time, the `ZMQ_PUB` and `ZMQ_SUB` sockets are designed. However, this is an one-sided data transfer. To set up a kind of broadcast, additional sockets and a proxy must be used. The proxy has a front-end and a back-end. At the front-end a `ZMQ_XSUB` and at the back-end a `ZMQ_XPUB` socket is bound to the proxy. Figure 3.4 shows this connection pattern. Every peer can connect with its `ZMQ_PUB` socket to the `ZMQ_XSUB` socket of the proxy. Additionally, every peer can connect with its `ZMQ_SUB` socket to the `ZMQ_XPUB` of the proxy. Sends a peer a message to its `ZMQ_PUB` socket, then the proxy distributes the message to all `ZMQ_SUB` sockets of the peers via its `ZMQ_XPUB` socket.

The ØMQ socket types and message patterns are described in more detail in *ØMQ - The Guide* [13].

To implement an inter-process communication in basf2 with the possibility to speak directly to asynchronous processes, ØMQ is suitable. It is possible to send the event data from a distributor via ØMQ messages to the proper worker processes. After processing, the worker processes send the event data via ØMQ messages to a collecting process. This corresponds to the structure in Figure 2.6. Finally, this is the basic structure of the new implementation of the parallel framework, which is described in the following Section 3.2.

3.2 Replacement of the Ring Buffers

In order to recreate the old core functionalities with ØMQ, the event data is sent with ØMQ messages across inter-process communication (IPC) from one process to another process as shown in Figure 3.5. In the input process, the event data is loaded into the basf2 data store

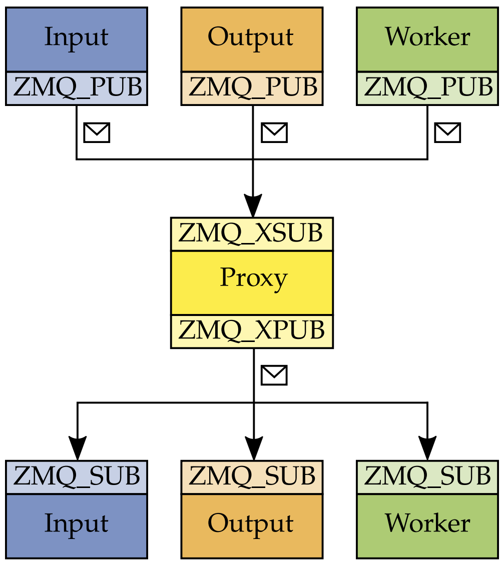

Figure 3.4: A schematic illustration of a broadcast between three different processes. Realised by ØMQ publish and subscribe sockets and a required proxy process. This figure is adapted from [11, p. 47].

and distributed from that with ØMQ messages to the parallel processing worker processes. After the event data has been processed in the worker process, it is wrapped in ØMQ messages again and eventually sent to the output process. Here the event data is written out to the storage. The exact data flow and process communication are explained in more detail below.

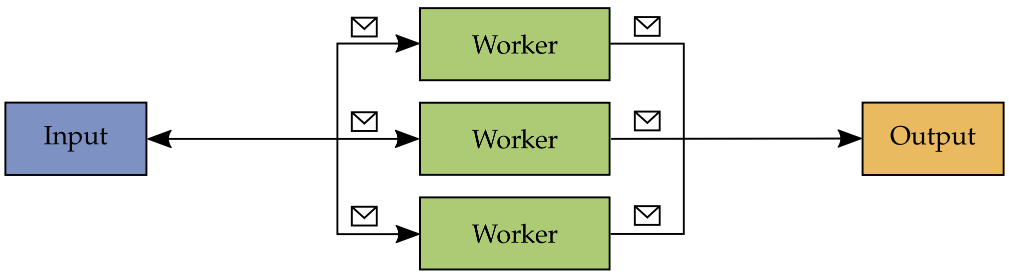

Figure 3.5: Rebuild of the old core functionality with ØMQ messages. The event data is sent by the input process to a set of worker processes. The worker processes send the event data after processing to the output process.

After starting basf2, the basf2 initial process is being created. Before loading the module path for the different processes, the module path of the steering file must be split in individual paths. For this purpose, the module path is analyzed as shown in Figure 3.6. The module path is split between the modules with and without parallel process certified flag into three individual paths (input, worker, output). At the end of the input path and at the beginning of the worker path are ØMQ modules for sending and receiving of messages inserted. The same for the end of the worker path and the beginning of the output path. These ØMQ modules are replacing the former ring buffers.

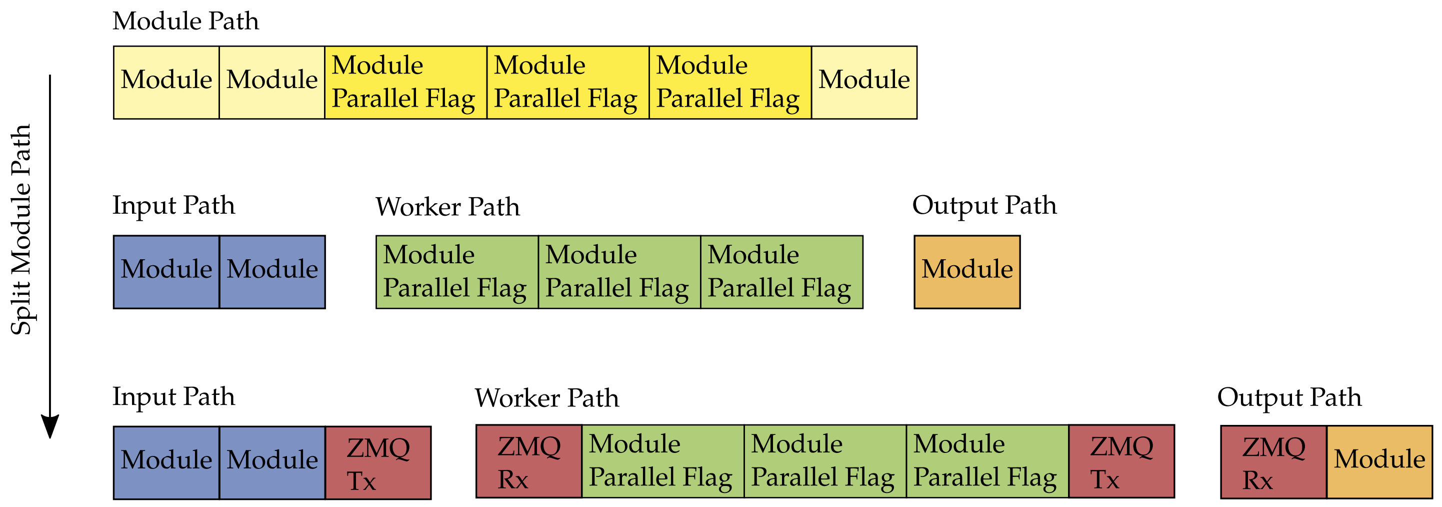

Figure 3.6: Schematic illustration of splitting the module path for message-based parallel processing. In step 1 the modules are identified for parallel processing by reference to the parallel processing certified flag. In step 2 the module path is split into pieces suitable for parallel processing. These two steps are the same as before. In step 3 the ring buffers are now replaced by ØMQ modules, which are arranged in the path before and after the parallel certified path pieces. The Tx (transceiver) modules are for sending and the Rx (receiver) modules are for receiving data between the processes.

Before processing any event data with basf2 in parallel mode, the additional processes must be created by the basf2 initial process (parent process). To create a new process, *Linux* provides the `fork()` system call. When forking a process, the memory of the parent process is copied[1]. Copying the memory during forking can be used to load libraries or data that affect all processes in the parent process before forking. After forking, all processes have this data in their own memory. Such methods can significantly reduce the initialization time and memory consumption of parallel processing software. Figure 3.7 shows which processes the basf2 initial process has to create to maintain the old core functionality.

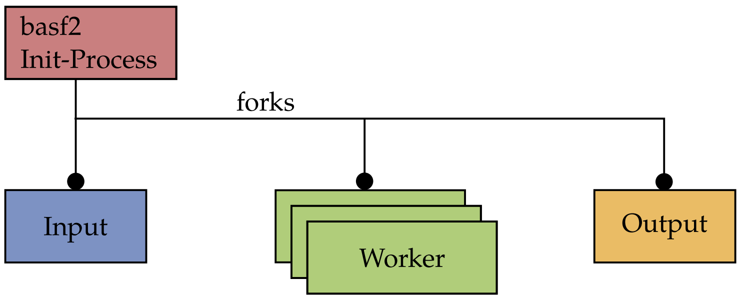

Figure 3.7: The basf2 initial process (Init-Process) is the parent process which forks all the other needed processes.

After all processes were created and the modules of the individual module paths are loaded, the actual data processing event loop can start. The detailed startup is described in Section 4.1 because it is related to a new feature. All the incoming events are sent by the input process to the worker process. The input process has a list of all the available worker processes and their free event buffer size. A number of sent `ready` messages from a worker process to the input process signals the input process how many `event` messages can be received by the worker process. Every time an event computation is finished by a worker process and the event data is sent to the output process, the worker process sends a `ready` message to the input process. All message types are described in Section 4.1.

Every process needs for the IPC at least one ØMQ socket. The input process is equipped with the `ZMQ_ROUTER` socket which is connected to all the `ZMQ_DEALER` sockets of the respective worker processes. Additionally, every worker process has a `ZMQ_PUSH` socket which is connected to the `ZMQ_PULL` socket of the output process. Before any data can be sent via ØMQ, the data must be wrapped in ØMQ messages. Therefore, the content of the data store must be serialized. This happens in the ØMQ transceiver (tx) modules. On the other side, the ØMQ receiver (rx) modules handle the unzipping of the ØMQ messages and storing of the data in the data store. The (un-)serialization is done by a data store built-in function. More details about the data store and its functionalities can be found in the dissertation of Christian Pulvermacher [9].

The ØMQ messages which are sent between the processes consist of two or three message frames. In case of two message frames, the first message frame contains the message type

[1]Actually, a copy on write method is used but this will not be discussed here, more information can be found in the book *Linux-Unix-Programmierung* [14].

(this is described in Section 4.1). The second message frame contains the actual data. In the second case, the ØMQ message contains at the beginning an additional message frame which contains the identity of the respective `ZMQ_DEALER` socket. The ØMQ messages with three message frames are used to send messages from the input process to the worker processes or when receiving in the input process messages from the worker processes.

With these few changes, the entire parallel processing could be converted to a message-based data transmission between the processes. The base for additional features is given and extending the parallel framework functionalities is quite easy.

4. New Features in the Parallel Processing Framework

In order to monitor and control processes effectively, a basic concept for process communication in basf2 was developed. With this approach, the parallel framework could be extended easily with new features. This chapter describes the process communication in basf2 and the newly implemented functionalities in the parallel framework which are particularly relevant for the HLT.

4.1 The Process Communication in basf2

In addition to the replacement of the ring buffers described in Section 3.2, a generic process communication in basf2 (PCB) was implemented. This extends the communication capability of the event data process chain which is used exclusively for communication for event data processing. Network topologically PCB is realized in form of a broadcast. The broadcast communication with ØMQ between the independent processes requires an additional proxy process. This proxy process is forked during the startup of basf2 in parallel mode together with the other processes described in Section 3.2. In order to control the forked processes via PCB, the basf2 initial process is converted to the monitoring process after forking. It plays a major role in monitoring the worker processes using PCB, this is described in more detail below.

Each ØMQ tx and rx module (Fig. 3.6) is equipped with a `ZMQ_PUB` and `ZMQ_SUB` socket, which is connected to the `ZMQ_XSUB` and `ZMQ_XPUB` socket of the proxy process (Fig. 3.4). According to this, the processes performing the event processing chain (Fig. 3.5) are able to communicate via PCB. Additionally, the init/monitoring process was extended with a `ZMQ_PUB` and `ZMQ_SUB` socket to use the PCB functionalities. Figure 4.1 illustrates the process topology and the communication paths for PCB and the event data processing chain.

The ØMQ messages sent over the broadcast are structured like the messages sent between the processes in the event data processing chain, consisting of two ØMQ message frames. The first ØMQ message frame contains the message type (type frame) and the second ØMQ message frame contains data (data frame). Each process decides for itself which broadcast message is relevant for it. To do this, the type frame of incoming broadcast messages is evaluated first. As already mentioned in Section 3.2, there are different types of messages. Especially on the broadcast messages are also used to send commands. There are message types available that correspond to a specific command. For example, a `stop` message sent to the broadcast by the monitoring process signals the input process to terminate (Sec. 4.4). In such command messages the data frame can be left empty. The corresponding recipient has stored an execution routine for the corresponding command message. Table 4.1 gives an overview of the different message types used both for the broadcast and the data processing chain. When and how the messages are used is described in the respective sections of the corresponding functionalities.

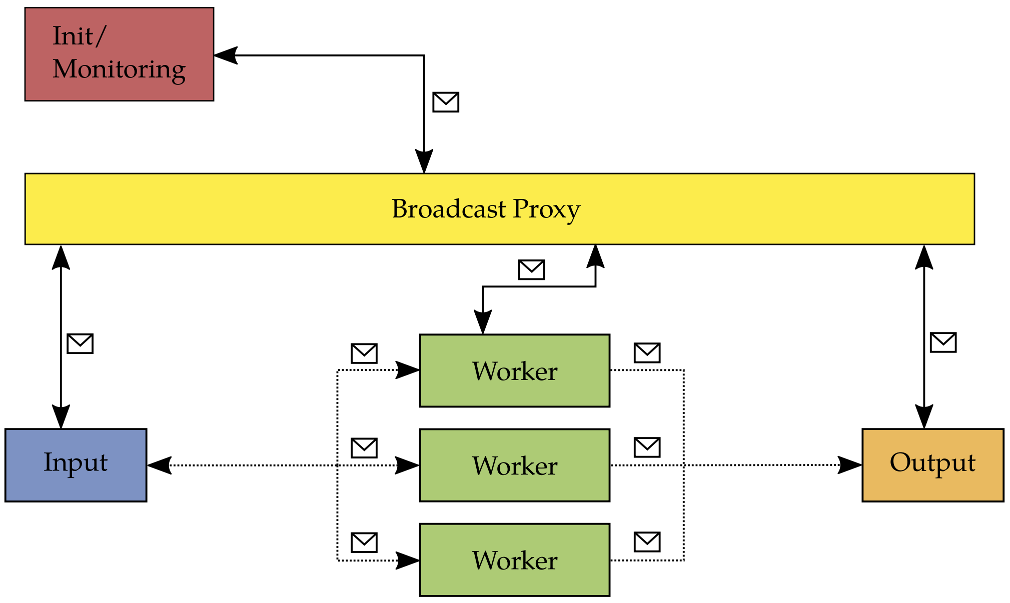

Figure 4.1: The Proxy process provides the PCB. Every process is able to send and receive broadcast messages (solid line). The broadcast communication is completely separated from the event data process chain (processes connected with dashed line). Here, the coupling of the worker processes to the broadcast is schematically illustrated on one worker.

Already during the start of the parallel framework important control tasks are performed using PCB. After forking the input and output processes, the basf2 init process waits until the input and output process report with a `hello` message on the broadcast before forking the worker processes. This step guarantees that the input process detects all worker processes and it is ensured that the worker processes fit seamlessly into the event data process chain. When the worker processes are started, they report on the broadcast with a `whello` message containing the identity of the respective `ZMQ_DEALER` socket in the data frame[1]. These messages are processed by the input process and the workers are added to a worker list. The input process replies the `whello` messages with a `hello` message to the corresponding worker via its `ZMQ_ROUTER` socket on the event processing chain. If the worker receives the `hello` message of the input process at its `ZMQ_DEALER` socket, it responds via the same socket with as many `ready` messages as it has space for event messages in its buffer. These `ready` messages are received by the input process on its `ZMQ_ROUTER` socket

[1]In detail it is the respective unique process identity (PID) of the worker process. Because the ØMQ dealer socket identity was set to the respective PID.

Table 4.1: Message types of the ØMQ messages used for communication via PCB or on the event data processing chain. The messages can be both data messages where the data frame contains data or command messages where the data frame can be left empty and the receiver reacts on the message type.

Message Type	Description
`hello`	is sent by input and output processes on startup
`whello`	is sent by a worker process on startup
`ready`	is sent by a worker process, means it has a capacity for an event
`event`	a message containing event data
`confirm`	is sent by the output process when an event has arrived
`death`	is sent by the input process when a worker process runs in timeout
`delete`	is sent by the monitoring process when it received `SIGCHLD`
`end`	is sent by the input process, indicates no more events to process
`stop`	is sent by the monitoring process in the shutdown routine
`terminate`	is sent by the input on shutdown if all events reached the output

and the worker is added to a wait list (next worker list) to receive a corresponding number of `event` messages.

Every time if the `event()` function of the ØMQ tx module of the input process (`ZMQTxInputModule`) is called by the processing loop of the parallel framework, an event is sent to the worker from the next worker list. Additionally, the respective next worker entry is removed from the list. If the worker has finished processing an event and sent it to the output process, another `ready` message is sent to the input process and the worker is added again at the end of the next worker list to receive another event if the respective entry is selected. This will be repeated until basf2 is stopped expectedly or terminated unexpectedly. The second case should not happen. If this case nevertheless occurs, the termination is uncertain. The explicit shutdown routine of basf2 is described in Section 4.4.

4.2 Event Backup

In the former ring buffer implementation event data could get lost if processes hang or terminated unexpectedly. The event data was moved from the ring buffer to buffers of the worker processes for the event processing. To avoid data loss, an event backup was developed for the event data distribution in the new ØMQ parallel framework implementation.

The `ZMQTxInputModule` was extended to store every event sent to a worker in an event backup list. Furthermore, the output process was extended to confirm the received events with a `confirm` message via the broadcast. The input process polls in every `event()` loop on the broadcast for such `confirm` messages. If the respective message is received with the event identity in the data frame, the event is removed from the event backup list. The procedure is illustrated in Figure 4.2.

If the worker process dies unexpectedly, all events that are sent to the affected worker process and have not yet been confirmed are sent by the input process to the broadcast as an `event` message. Regular processed events are received by the output process via the `ZMQ_PULL` socket, however, the `event` messages received on the broadcast are classified as event backups by the output process.

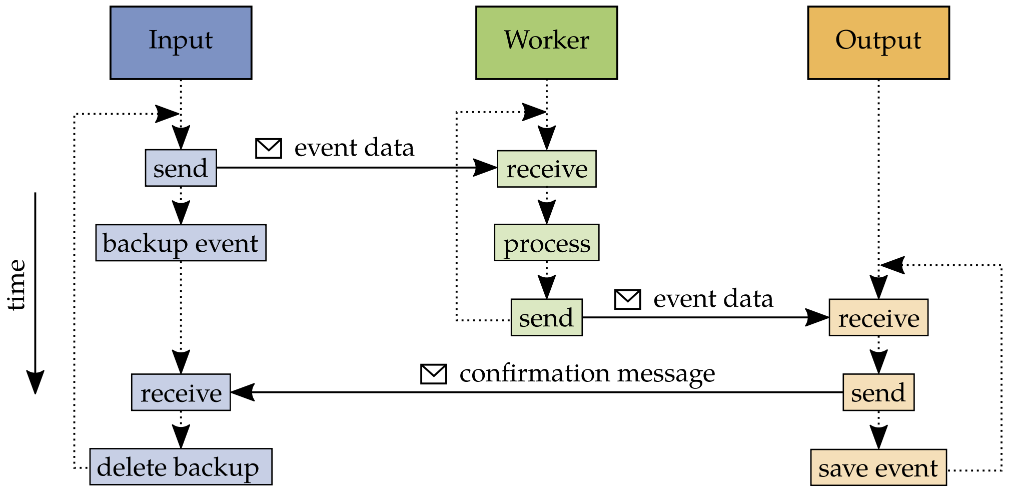

Figure 4.2: The event backup is implemented in the input process. Every by the input process sent event is saved in the event backup list. Is an event received by the output process, a confirmation message with the unique event identification in the data frame is sent to the broadcast. If the input process receives such a confirmation message, the respective event is removed from the event backup list.

4.3 Handling Died Worker Processes

In general, there are two different situations where the worker processes die and the event backup feature has to be used (Sec. 4.3.1 and 4.3.2). In this context, the signal handling in the parallel framework was redesigned. The monitoring process intercepts all signals and decides how to react. In addition, a feature has been developed that restarts dead worker processes as described in Section 4.3.3.

4.3.1 Process Timeout

That a worker process runs into a (processing) timeout means that due to some reason it takes too long to process an event. One possible reason is that the worker hangs in an endless loop and no longer reacts. The maximum time a worker is allowed to process an event is defined in a processing time variable. To monitor the worker processes in this aspect, the oldest backup entry is checked in the `ZMQTxInputModule` in every `event()` loop. If no `confirm` message was sent for the distributed events after a certain time, the corresponding worker process ran into a timeout. In this case, all events sent to this worker and stored in the event backup list are sent as `event` message to the broadcast and are removed from the event backup list. The `ZMQRxOutputModule` in the output process looks for such `event` messages on the broadcast. If the output process receives such backup `event` messages on the broadcast, they will be written out to the storage. Additionally, the input process sends a `death` message with the corresponding worker process identification in the data frame to the broadcast. This message is evaluated by the monitoring process. The monitoring process subsequently sends a `SIGKILL`[2] to the corresponding worker process. In general, single events are responsible that worker processes are running into a process timeout. This method eliminates processes that are caught in an endless loop while they are consuming valuable computing resources. The scenario described is shown in Figure 4.3. Then, a new process is started automatically as described in Section 4.3.3.

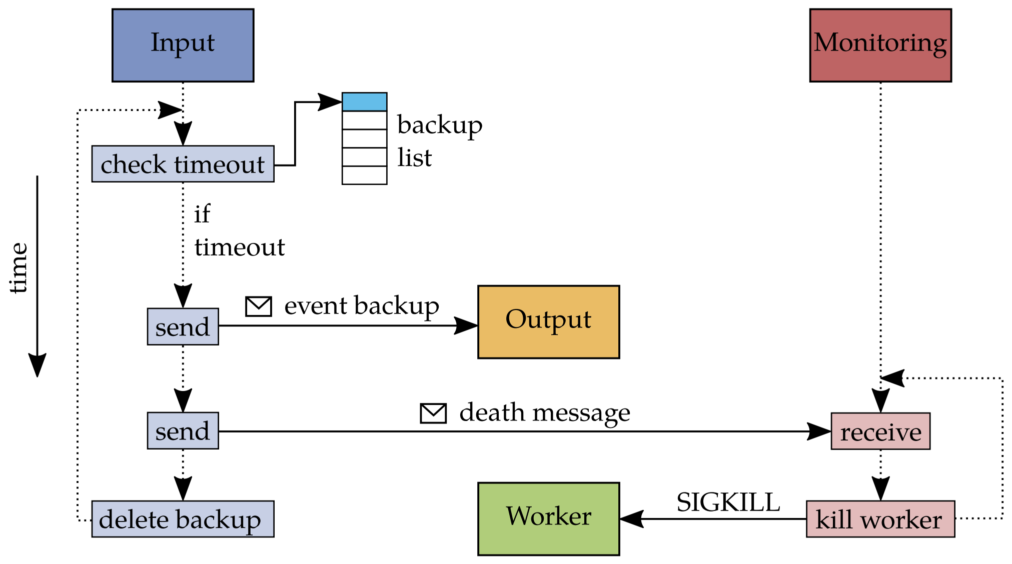

Figure 4.3: To detect hanging worker processes, the time stamp of the oldest entry in the event backup list is checked. If a worker process ran into a timeout, the backup events are sent via the broadcast to the output process and written out to the storage. Additionally, the monitoring process gets the identity of the dead process sent by a death message. Eventually, the monitoring process kills the respective worker process.

4.3.2 Process Terminates Unexpected

A process unexpectedly terminates, for example, if a critical sudden error occurs. When a process terminates, a `SIGCHLD` is received by the monitoring process. If this signal occurs, a `delete` message is sent to the broadcast. The sent `delete` message contains the corresponding worker identity in the data frame. If the input process gets such a message, it sends the respective events corresponding to the worker identity from the event backup list to the broadcast. Additionally, the respective backups are removed from the event backup list. The steps executed by the output process are the same as in Section 4.3.1. Figure 4.4 shows the procedure schematically.

[2]Signals in detail are not content of this thesis. More informations to signals can be found in the book *Linux-Unix-Programmierung* [14].

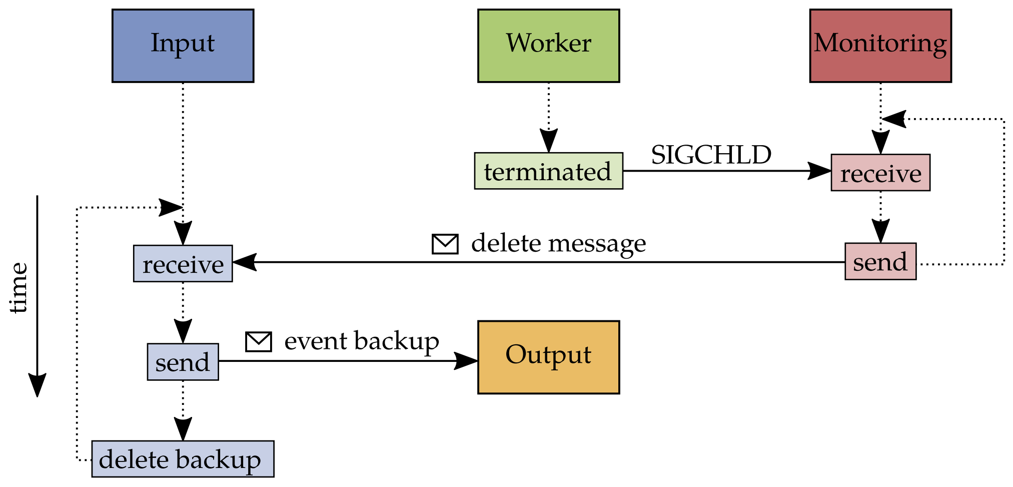

Figure 4.4: If a worker terminates unexpectedly, the monitoring process receives a SIGCHLD. According to that signal, a delete message with the corresponding worker identity is sent to the input process. The input process sends the respective event data from the corresponding worker to the output process.

4.3.3 Restart Dead Worker Processes

In order to ensure long uninterrupted operation, for example on the HLT, any dead workers must be restarted. If the monitoring process detects that a worker process has died, it ensures the complete removal of the worker process. Then, a new worker process is started instead of the old one so that the total number of worker processes is retained. For this purpose, a further `fork()` is executed in the monitoring process. From then on, the procedure is repeated as described in Sections 3.2 and 4.1.

4.4 Graceful Shutdown of basf2

It is good practice to leave a tidy workplace, so it is ready for the next user. This applies to the use of system resources as well. Open files and system resources are released by executing a so-called shutdown routine. In general, a shutdown routine should not produce pointless error messages and warnings e.g. because processes terminating in the wrong order.

There are essentially three ways why basf2 is terminated:

1. Because all event data was processed (offline reconstruction on a set of event data), basf2 terminates itself.
2. The user terminates the program (e.g. by pressing CTRL + C) sending a `SIGINT` to the program.
3. An unexpected bug in the program occurs which causes the program to crash. In this case, unexpected error messages and a program abort will inevitably occur. As a result, a shutdown routine is executed only in a limited number of cases.

When the first two scenarios occur, basically the same shutdown routine can be executed. The shutdown routine will execute the following steps:

1. If the shut down is initiated, the input process sends an `end` message to all worker processes from its worker list via the `ZMQ_ROUTER` socket.
2. A worker process will terminate if an `end` message is received by the `ZMQ_DALER` socket. This will ensure that all events are processed, stored in the buffers of the worker processes.
3. If all outstanding events have been confirmed by the output process, the event backup list is empty. Then, the input process sends a concluding `terminate` message to the broadcast. This message finally terminates the output process and the input process. The monitoring process terminates the proxy process and terminates itself last.

There is a small difference between finishing by the user and finishing after the work is done. If the user cancels basf2, the monitoring process receives a `SIGINT`, then, a `stop` message is sent to the broadcast. This message is processed by the input process and the shutdown sequence is initiated as described above. Whereas in the other case the input process itself notices that no new events arrive and then initiates the shutdown routine above.

5. Benchmark and Reliability Tests

In this chapter, the measurements of the total processing time needed to process a number of events using the new basf2 parallel framework with a variable number of worker processes will be discussed. Additionally, the ØMQ implementation is compared with the old ring buffer implementation regarding to the absolute processing time and the scaling behavior. The stability of the new ØMQ implementation was tested by adding corrupt modules to the path in the steering file to fail individual processes.

5.1 Runtime Tests

The total processing time with a single process is defined by Equation 5.1 if the startup time and the termination time is neglected. This is possible if the startup and termination time is short compared to the total processing time. T_{input} and T_{output} are tasks which can only be serially processed. T_{work} is the processing part which can also be processed in parallel mode. The total time needed to process a number of N events is $T(N)$.

$$T(N) = N \cdot (T_{\text{input}} + T_{\text{work}} + T_{\text{output}}) \tag{5.1}$$

When the workload of the parallel part is distributed to a number of processes, the total processing time ideally decreases according to Equation 5.2. A further requirement for the validity of this equation is that the single worker processes do not have to wait for the serial input and output processes[1]. In addition, the number of events N must be larger or equal to the number of processes p.

$$T(N,p) = T_{\text{input}} + \frac{N}{p} \cdot T_{\text{reco}} + T_{\text{output}} \tag{5.2}$$

To verify that the new ØMQ implementation is as fast as the old ring buffer implementation, a speed test was performed. The used hardware was equipped with 16 Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz CPUs, which is similar to the hardware of the HLT. A standard reconstruction was performed on the same sample of $\mathrm{B\overline{B}}$ events with both implementations. Therefore, special time measurement modules were added to the module

[1]The exact requirement for the processing time of a single event procession: $T_{\text{work}} > (p-1) \cdot T_{\text{input}}$. With p is the number of processes.

path in the steering file. These modules were placed at the beginning and at the end of the path to measure the time difference while executing the whole path for each event. Side effects, for example, start-up and termination influences were eliminated by ignoring the first and the last 100 events.

Figure 5.1 shows the absolute processing time of both implementations. It shows clearly, that the curve of the ØMQ implementation is overlapped by the similar absolute shape of the ring buffer implementation curve. This proves that the absolute processing time has not been declined by the ØMQ implementation. Because ØMQ is designed for such tasks a decline was not expected.

Even more interesting is the behavior of the ratio $\eta(p)$ of speedup and number of processes p defined by Equation 5.3. Because the considered ratio depends not on N, $T(N,p)$ is denoted in the following as $T(p)$.

$$\eta(p) = \frac{T(1)}{T(p)} \tag{5.3}$$

The theoretical speedup of the total processing time of tasks, which results from increasing the number of processes is given by the law of Amdahl. It is considered how much of the total processing time T can be processed in parallel mode T_p which benefits from increasing the number of worker processes. The remaining processing time T_s is the serial executed part. Equation 5.4 defines the total processing time T of a number of N events.

$$T = T_\mathrm{s} + T_\mathrm{p} \tag{5.4}$$

The parallel fraction u of the total processing time T is given by Equation 5.5.

$$u = \frac{T_p}{T} \tag{5.5}$$

The processing time of the parallel part $T_p(p)$ is decreasing with the number of processes p according to equation 5.6.

$$T_p(p) = \frac{1}{p} \cdot T_p(1) \tag{5.6}$$

Due to Equations 5.3, 5.4 and 5.6, the speedup scales according to Equation 5.7 (Amdahl's law). Considering the limiting case, the parallel processing part outweighs heavily the serial part ($u \approx 1$) and the number of events N is larger or equal than the number of processes p ($N \geq p$), Equation 5.7 simplifies to $\eta(p) = p$.

$$\eta(p) = \frac{1}{1 - u + \frac{u}{p}} \tag{5.7}$$

Figure 5.2 shows the measured ratio $\eta(p)$ of speedup and the number of processes p for both the ØMQ implementation and the ring buffer implementation. Relevant is just the scope from one to 16 processes because the used hardware was equipped with 16 CPUs.

Although Intel advertises additional speedup based on Hyper-Threading Technology[2], in this case, there was no significant speedup measured with basf2. With the Hyper-Threading Technology a single physical CPU is divided in two logical CPUs sharing the physical execution resources. Processes are executed at the same time by both logical CPUs. If the pipeline of one process is disturbed for example the process is waiting for memory access, the other process can use e.g. the execution engine in the meantime [15]. The speedup depends on the memory pattern of the executed processes. Due to the complexity of this issue, a conclusion at this point is not possible regarding the speedup using the Hyper-Threading Technology.

There is a small deviation with increasing number of processes respective to the ideal scaling in Figure 5.2. These deviations are a result of cache misses that affect both implementations. The used hardware to perform the speed tests was equipped with three different caches (L1 (32 kB L1i, 32 kB L1d), L2 (256 kB), L3 (20 480 kB)). The caches are arranged hierarchically with increasing access times. The L1 is the fastest cache decreasing to the L3. Cache misses occur if the data needed by the processor for the next processing step is not loaded into the fast L1 cache. Instead, the data is stored into higher cache levels or even into the RAM. The RAM has the longest access times of all here listed memories. This means that the processor may have to load the data from higher cache levels or from the RAM into the L1 while the process has to wait. More information about this issue can be found in [16]. In general, the speedup with ØMQ is as high as before with the ring buffers and it leaves no negative effect on the processing performance.

5.2 Further Reliability Tests

In certain circumstances, backup events of workers which ran into a processing timeout can produce specific warnings:

```
[WARNING] Event: <event nr>, no matching event backup found in backup list
```

This warning can occur if a worker process finished the event processing just before it ran into the timeout because the processing needed a lot of time. When terminating that process, the processed event is already on the way to the output process to be confirmed eventually. In the meantime, the event backups of the corresponding worker process were sent to the output process and deleted from the event backup list. When the input process receives the `confirm` message it tries to remove the event backup from the event backup list but cannot find a corresponding entry. Furthermore, the output process will write this event two times to the storage. One event will contain the backup flag and the other one not. If the stored data is later evaluated it is necessary to pay attention to this point. For example, when the regular processed event is available in the output files, the backup of this event can be discarded. If that warning occurs too often, it is recommended to increase the value for the process timeout.

During this thesis, there was, unfortunately, no better solution found to avoid this issue. The problem is that the output process has to check if there is already an arrived backup

[2]According to Intel it is possible to reach a further speedup up to 33 % due to Hyper-Threading with $2N$ processes, with N is the number of CPUs.

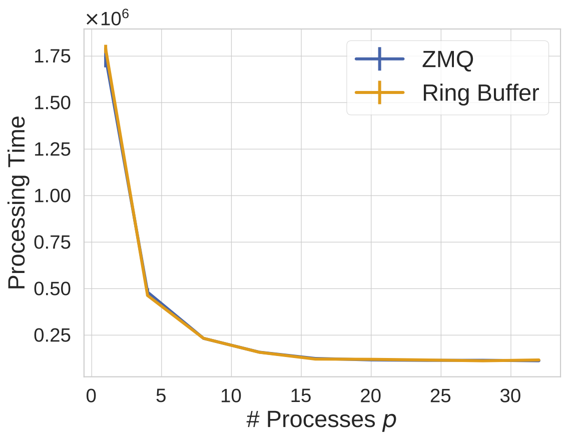

Figure 5.1: Absolute processing time by a different number of worker processes measured for both implementations. The ring buffer curve overlaps entirely the ØMQ curve.

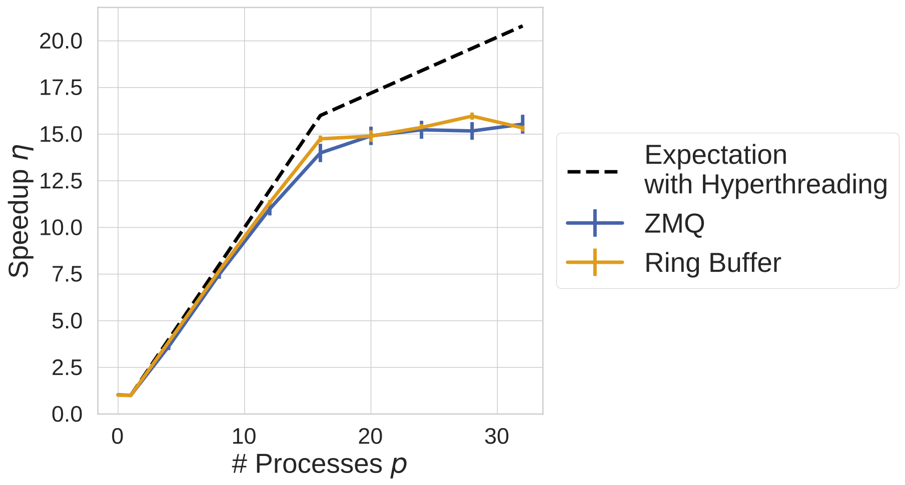

Figure 5.2: Speedup scaling with the number of processes measured for both implementations. The deviation of the measurements to the ideal curve(dashed) occur due to cash misses.

event before sending a `confirm` message for a regular arrived processed event. In particular, there is no defined point in time when the backup event should have arrived so that all backup events actually had to be synchronized continuously. This means an additional amount of workload, which is impacting on the CPU and memory resources. So the most productive way was to allow a marked redundant storage of the event data in certain circumstances.

In addition to the previous sections benchmark tests, additional reliability tests were performed to detect dead worker processes. For this purpose, a module was built into the path of a dummy reconstruction steering file, which has caused specific processes to fail. The processing of any single event terminated by raising a value error or the reconstruction of a single event was hang up by an artificial endless loop. In all cases, the corrupt processes were completely terminated and new processes started in their place. Additionally, the event data of the affected worker processes were delivered directly to the output process so that event data loss was avoided.

6. Summary and Conclusion

In this work, the core of basf2's parallel framework was upgraded with a modern approach for parallel processing. The obsolete ring buffers in the data distribution were replaced by a message-based data distribution method using IPC and the open source ØMQ libraries. This change eliminates the locking of the memory structures required to prevent processes from accessing the same memory segments simultaneously when operating with ring buffers. Thus, the data is distributed for parallel processing and collected again, which is managed by two independent processes - the input and output process. This approach lays the foundation for new control and monitoring mechanisms to ensure a stable operation of the software.

With the ØMQ implementation, it was possible to extend the parallel framework with new functionalities. An event backup, process monitoring and process controlling were implemented with the capability in case of dying or hanging worker processes to heal itself. In addition, it was taken care of a graceful shutdown of basf2 in different occurring scenarios to ensure that files are closed properly to prevent data corruption and resources are released.

The developed process communication in basf2 (PCB) allows to extend the parallel framework easily. It provides the required infrastructure during the execution of basf2 in parallel mode to communicate with the individual processes without disturbing the regular processing. Additional features such as monitoring worker processes with high memory consumption or monitoring functions that may terminate and restart processes according to other criteria are easy to implement with this design.

Since the code for implementation has partly grown historically, some improvements are still needed. On the one hand, the processes from the event data processing chain are using different messages when they report on the broadcast for the first time (`hello` and `whello` messages) this can be unified to a general `hello` message in any case. On the other hand, e.g. the shutdown routine is not yet ideal when the termination is requested by the user. Instead of processing the remaining events in the worker process buffers, it would be wiser to terminate the worker directly, as it might otherwise take some time until basf2 shuts down.

The runtime tests have shown, that for the new implementation the total processing time of the events has not increased and the speedup has scaled as well as before with the ring buffer implementation. In addition, a comprehensive testing of the reliability and stability of the parallel framework was performed during this thesis. The new ØMQ implementation of the parallel framework achieves the stability and performance required for the Belle II HLT farm.

Danksagung

Zum Abschluss möchte ich allen danken, die zur Fertigstellung dieser Arbeit beigetragen haben.

Zuallererst danke ich Prof. Dr. Florian Bernlochner für die Möglichkeit, diese Bachelorarbeit schreiben zu können.

Vielen Dank auch an Dr. Thomas Hauth für die Übernahme des Korreferats.

Bei Nils Braun möchte ich mich sehr für die allzeit gute und umfassende Betreuung und Unterstützung bedanken. Markus Prim danke ich für das Korrekturlesen dieser Arbeit.

Das offene und kreative Arbeitsumfeld in der Belle II Arbeitsgruppe am ETP hat mir sehr gefallen und den oft amüsanten Aufenthalt im Raum 9-2 habe ich sehr genossen (trotz unsäglicher sommerlicher Hitze).

Bibliography

[1] J. L. Gustafson, "Reevaluating Amdahl's Law," *Commun. ACM* **31** no. 5, (May, 1988) 532–533. http://doi.acm.org/10.1145/42411.42415.

[2] Wikipedia contributors, "KEKB (accelerator) — Wikipedia, the free encyclopedia," 2018. https://en.wikipedia.org/wiki/KEKB_(accelerator). [Online; accessed 21-April-2018].

[3] Z. Doležal, S. Uno, et al., "Belle II technical design report," *KEK Report 2010-1* (Oct. 2010) .

[4] Belle II Experiment, "The Detector," 2018. http://www.belle2.jp/. [Online; accessed 15-July-2018].

[5] N. Braun, "Hadron spectroscopy studies at Belle II," *XLVII International Symposium on Multiparticle Dynamics, Tlaxcala, Mexico* (2017) . https://indico.nucleares.unam.mx/event/1180/session/11/contribution/76/material/slides/0.pdf. [Online; accessed 07-July-2018].

[6] KEK IPNS, "A Report on the Ground at KEK: Electrons and Positrons Collide for the first time in the SuperKEKB Accelerator," 2018. https://www2.kek.jp/ipns/en/release/first-collision/. [Online; accessed 01-July-2018].

[7] R. Itoh, "Status of Belle II and physics procespects," *Sixth Workshop on Theory, Phenomenology and Experiments in Flavour Physics - FPCapri2016* (June, 2016) . https://docs.belle2.org/record/403/files/BELLE2-TALK-CONF-2016-025.pdf. [Online; accessed 26-July-2018].

[8] N. Braun, *Combinatorial Kalman Filter and Online Reconstruction for the Belle II Experiment (preliminary title)*. PhD thesis, Karlsruhe Institute of Technology (KIT), 2018. https://ekp-invenio.physik.uni-karlsruhe.de/collection/Belle2?ln=en. **Unpublished yet**.

[9] C. Pulvermacher, *Analysis Framework and Full Event Interpretation from the Belle II Experiment*. PhD thesis, Karlsruhe Institute of Technology (KIT), 2015. https://ekp-invenio.physik.uni-karlsruhe.de/collection/Belle2?ln=en.

[10] A. Moll, "The software framework of the Belle II experiment," *Journal of Physics: Conference Series* **331** no. 3, (2011) 032024.

[11] P. Hintjens, *ZeroMQ*. O'Reilly, 1 ed., 2013.

[12] T. Hauth, N. Braun, "Parallel processing with ZeroMQ," *Belle II TRG/DAQ/Software Workshop* (2017) . **Unpublished**.

[13] P. Hintjens, "ØMQ - The Guide," 2018. http://zguide.zeromq.org. [Online; accessed 01-July-2018].

[14] J. Wolf, K.-J. Wolf, *Linux-Unix-Programmierung*. Rheinwerk Computing, 4 ed., 2016.

[15] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, M. Upton, "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology Journal* **6** no. 1, (Feb., 2002) . http://www.cs.sfu.ca/~fedorova/Teaching/CMPT886/Spring2007/papers/hyper-threading.pdf. [Online; accessed 28-July-2018].

[16] J. L. Hennessy, D. A. Patterson, *Computer architecture : a quantitative approach*. Elsevier, Morgan Kaufmann, 6 ed., 2017.