

The New Event Builder of the LHCb Experiment: Network Optimisations for the Online Cluster

Alberto Perro



Relatore: Ernesto Migliore

Co-relatore: Flavio Pisani

Contro-relatore: Federica Legger

Dipartimento di Fisica
Università degli Studi di Torino
Anno Accademico 2021-2022



Summary

1	The LHCb Experiment at CERN	9
1.1	European Organization for Nuclear Research (CERN)	9
1.2	The Large Hadron Collider	10
1.3	The LHCb Experiment	12
1.3.1	The LHCb Tracking System	13
1.3.2	Particle Identification	18
1.4	LHCb Physics	23
1.4.1	Results from Run 1 and 2	23
1.4.2	Prospects for Run 3	24
2	Data Acquisition System	27
2.1	Data Flow	28
2.1.1	DAQ Boards	30
2.1.2	Event Builder	30
2.1.3	High Level Trigger 1	31
2.1.4	High Level Trigger 2	33
3	Design of the Event Builder Infrastructure	35
3.1	The LHCb Event Building Process	35
3.1.1	Detailed Overview of the Event Building Process . . .	35
3.1.2	The Event Building Network	36
3.1.3	The Event Builder Finite State Machine	39
3.2	Message Passing Interface (MPI)	40
3.2.1	Structure of an MPI Application	40
3.2.2	Data Exchange	41
3.3	InfiniBand Network Technology	45
3.4	Requirements from the EB Software	46

3.4.1	Warm-Up	47
3.4.2	Dead Processes	47
3.5	The Choice of a Custom Design	48
4	The InfiniBuilder Network Library	49
4.1	Design Choices	49
4.2	Setup	49
4.2.1	Configuration Parser	50
4.2.2	Local InfiniBand Setup	51
4.2.3	Queue Pair Detailed Description	52
4.2.4	Out-of-Band Communication	54
4.3	Basic Exchange	54
4.3.1	Memory Registration	54
4.3.2	Send and Receive	55
4.4	Barrier	60
4.5	Synchronisation	62
4.5.1	<code>sync</code> Procedures	64
4.6	Interfacing with the EB	65
4.6.1	<code>Transport_unit</code> Class	65
4.6.2	<code>Parallel_comm</code> Class	66
4.6.3	Collective Functions	66
5	Measurements and Benchmarks	69
5.1	Experimental Setup	69
5.1.1	Hardware	69
5.1.2	Software	70
5.1.3	Measurement of Throughput	70
5.2	Point-to-Point Communication	71
5.3	Event Builder	75
5.3.1	Small Batch Size	75
5.3.2	Full Size Scalability Tests	76
5.3.3	Full Size Test	77
5.4	Conclusions	81
5.4.1	Future Developments	81

A Principles of Parallel Computing	83
A.1 Distributed Shared Memory and Non Uniform Memory Access	84
A.2 Programming models	85
A.3 Message Passing	86
A.4 Threads	87
A.5 Side Note on Copies	89

Abstract

The LHCb experiment at CERN LHC collider investigates the c- and b-quarks flavour sector of the standard model of particle physics. For the Run III of LHC (2022-2025), LHCb has been upgraded to withstand proton-proton collisions of 14 TeV centre-of-mass energy up to an instantaneous luminosity of $2 \times 10^{33} \text{ cm}^{-2}\text{s}^{-1}$. The entire online data acquisition chain has been redesigned to fully exploit the higher luminosity.

The upgraded system is designed to perform full readout and online reconstruction of the full 40 MHz rate without any low-level hardware trigger, with an expected aggregated throughput of $\sim 32 \text{ Tb/s}$. This requires a state-of-the-art design in the networking and computing sides. The online computer cluster is built using 170 servers, interconnected via a 200 Gb/s InfiniBand-based network. The data from the LHCb sub-detectors are fragmented throughout all the servers and have to be assembled before the event reconstruction can take place; this process is known as Event Builder.

The first design of the Event Builder software used MPI for the parallel network communication, which presents an underperforming warmup phase and a suboptimal handling of dead processes, leading to potential dead time and consequent data loss. The solution was the design of a custom network library based on the InfiniBand Verbs API called InfiniBuilder.

InfiniBuilder has been designed as a drop-in replacement for MPI calls, which can be further customised by leveraging the hardware capabilities of InfiniBand. InfiniBuilder resolves the issues reported with MPI and is capable of achieving better performance in throughput tests. The results of the benchmarks have shown that InfiniBuilder have achieved the design goals, leading to its integration in the current version of the LHCb Event Builder software, which will enable efficient data taking during Run III.

Chapter 1

The LHCb Experiment at CERN

1.1 European Organization for Nuclear Research (CERN)

The European Organization for Nuclear Research (CERN) is an international research organization founded in 1954. The organization headquarters are located near Geneva in Switzerland on the Franco-Swiss border. At the time of writing, the member states participating in the organization are 23.

The main purpose of CERN is to investigate the fields of Nuclear and Subnuclear Physics, focusing on high-energy physics. The CERN laboratory is the largest particle physics laboratory in the world and has given the world numerous scientific achievements such as:

- 1973: the discovery of Neutral Currents in the Gargamelle bubble chamber [1]
- 1983: the discovery of W and Z bosons in UA1 and UA2 experiments [2]
- 1989: the determination of the number of light neutrino families in ALEPH, DELPHI, L3, and OPAL experiments [3]
- 1995: the first creation of anti-hydrogen with the PS210 experiment [4]

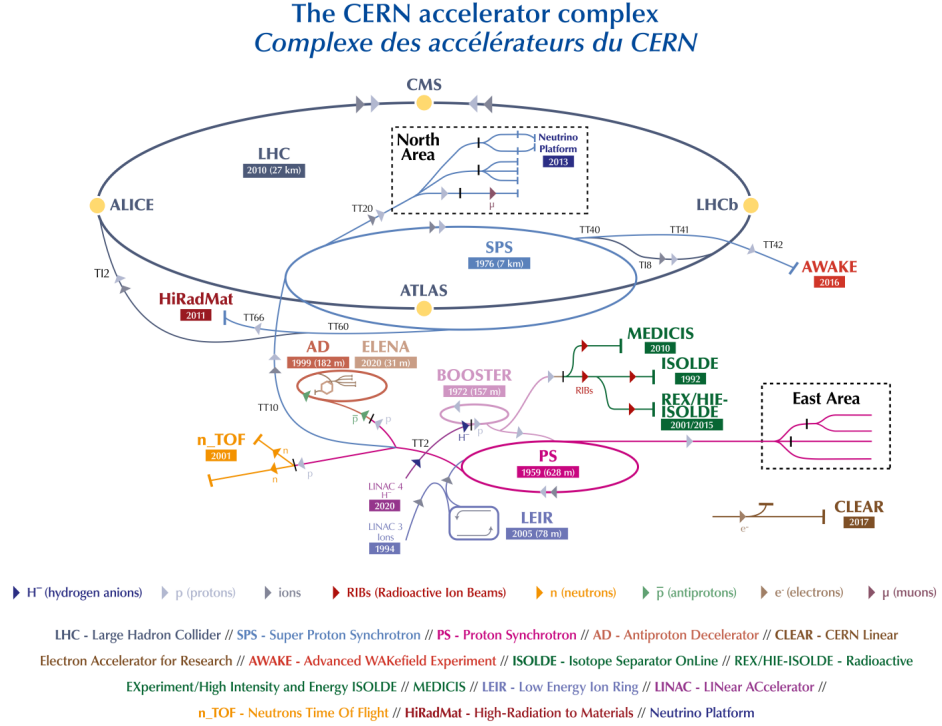


Figure 1.1: Schematic view of the CERN Accelerator Complex. Original image from CERN.

- 1999: the discovery of direct CP violation in the NA48 experiment [5]
- 2012: the discovery of the Higgs boson with ATLAS and CMS experiments [6], [7]
- 2015: the discovery of pentaquarks with the LHCb experiment [8]
- 2019: the discovery of CP violation in charm hadrons with the LHCb experiment [9]

1.2 The Large Hadron Collider

The Large Hadron Collider (LHC)[10] is a collider installed 100 m underground in a 27 km tunnel designed for hadron collisions. Proton beams are accelerated up to a design energy of 7 TeV each and then collide with a centre-of-mass energy ($\sqrt{s}$) of 14 TeV. The accelerator provides a peak

luminosity of $10^{34} \text{ cm}^{-2} \text{ s}^{-1}$ when colliding protons. The LHC is built using superconducting elements in order to achieve the specifications listed above.

The LHC is just the last part of the accelerator complex in which protons get accelerated. Hydrogen ions (H^-) are injected into the linear accelerator LINAC4. After this first acceleration step, H^- ions have an energy of 160 MeV and get their electrons removed. The resulting protons are injected in the Proton Synchrotron Booster (PSB), made up of four superimposed synchrotron rings, which brings the protons up to an energy of 1.4 GeV. The Proton Synchrotron is a 628 m circular accelerator that takes protons from the PSB and boosts them to an energy of 25 GeV. At this point, the protons have reached the relativistic limit. Protons are then injected into the Super Proton Synchrotron (SPS) and reach 450 GeV before being transferred into the LHC. Since LHC operates with protons, they need to circulate in opposite directions, thus the injection takes place via two separate tunnels. The full accelerator complex is shown in Figure 1.1.

The LHC is filled up with 2808 bunches of 10^{11} protons per beam [11]. The bunches are kept in their trajectory thanks to 1232 superconductive dipole magnets, which generate a magnetic field of up to 8.34 T. Beams pass through the four points around the ring, where the experiments have been built, at a rate of 40 MHz.

- A Toroidal LHC ApparatuS (ATLAS): it is a general purpose detector, it was designed to find the Higgs boson and super-symmetrical particles;
- Compact Muon Solenoid (CMS): it is a general purpose detector built with a different design, and aims at the same physics studies as ATLAS, ensuring a double-blind response;
- A Large Ion Collider Experiment (ALICE): it is a detector optimised for heavy-ion physics using Pb-Pb collisions. These collisions allow the exploration of quark-gluon plasma. ALICE focuses on the physics of strongly interacting matter, studying quantum chromodynamics (QCD) properties;
- Large Hadron Collider beauty (LHCb): specialised in heavy quark flavour physics, its goal is to determine CP violation parameters and to study rare decays of B hadrons.

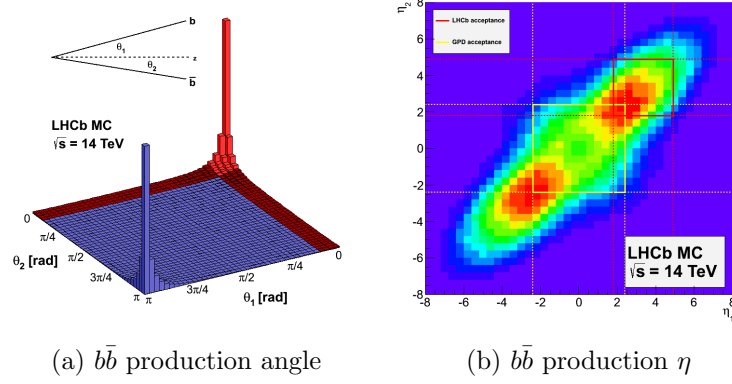


Figure 1.2: Production characteristics of $b\bar{b}$ pairs: angles with respect to the beam direction and pseudo-rapidity. The data comes from fully simulated events from pp collisions at $\sqrt{s} = 14$ TeV. LHCb acceptance region is highlighted in red. [12]

1.3 The LHCb Experiment

The LHCb experiment has been designed to study the flavour physics using heavy quarks such as b and c quarks. This is possible thanks to the production cross-section of $b\bar{b}$ [13] and $c\bar{c}$ [14] pairs in pp collision:

$$\sigma(pp \rightarrow b\bar{b}X) = (154.3 \pm 1.5 \pm 14.3) \mu b \quad \text{at } \sqrt{s} = 13 \text{ TeV} \quad (1.1)$$

$$\sigma(pp \rightarrow c\bar{c}X) = (2369 \pm 3 \pm 152 \pm 118) \mu b \quad \text{at } \sqrt{s} = 13 \text{ TeV} \quad (1.2)$$

Due to the asymmetry of the parton momentum distribution in pp collisions, b and c quarks are produced preferentially with a large boost in the beam-line direction (Figure 1.2). For this reason, the LHCb detector has been built as a forward spectrometer. This design focuses the available resources to have a more sophisticated detector while covering only the region of interest for b -hadrons. The LHCb detector is splitted on the vertical plane passing by the beam line, the two sides are identified as side A (Airport) and side C (Centre). The geometrical acceptance of LHCb is $[10,300]$ mrad in the horizontal plane and $[10,250]$ mrad in the vertical plane. The trajectories of charged particles are bent in the horizontal plane by a dipole magnet.

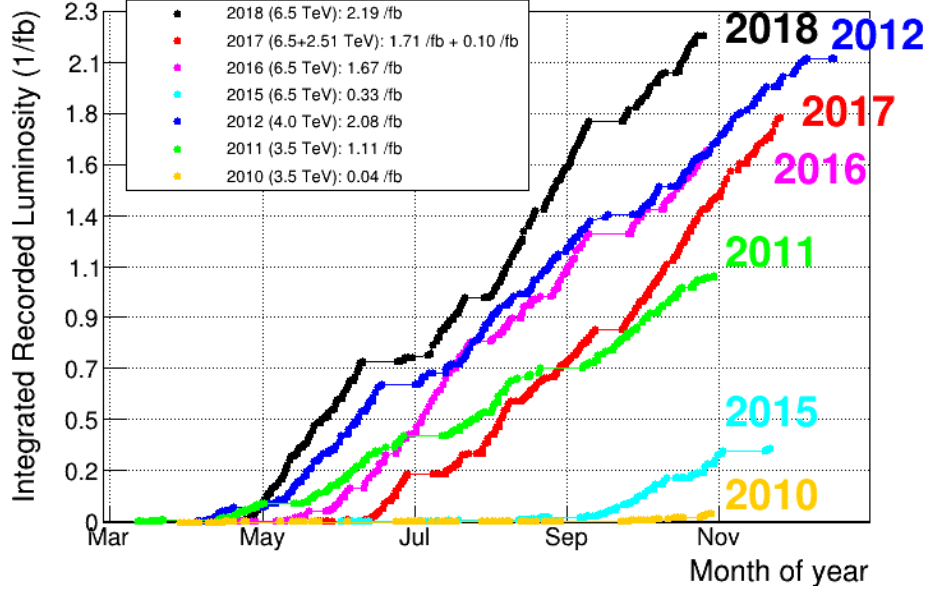


Figure 1.3: Integrated recorded luminosity at the LHCb experiment during Run 1 (2010-2012) and Run 2 (2015-2018). Courtesy of the LHCb Collaboration.

Therefore, the LHCb detector can detect particles with a pseudorapidity η between 1.8 and 4.9, where η is defined as:

$$\eta = -\ln \left[\tan \left(\frac{\theta}{2} \right) \right] = \frac{1}{2} \ln \frac{|\vec{p}| + p_L}{|\vec{p}| - p_L} \quad (1.3)$$

where θ is the angle between the particle and the beam axis and p_L is the longitudinal momentum.

During Run 1 and 2, the LHCb Experiment was able to record more than 9 fb^{-1} of integrated luminosity, as depicted in Figure 1.3.

The LHCb detector layout is described in the next sections and a complete side view is shown in Figure 1.4.

1.3.1 The LHCb Tracking System

The current LHCb tracking system is composed of three main sub-detectors: Vertex LOcator (VELO), Upstream Tracker (UT), and Scintillating Fibers (SciFi)[15].

The VELO is the sub-detector closest to the interaction point. Its main

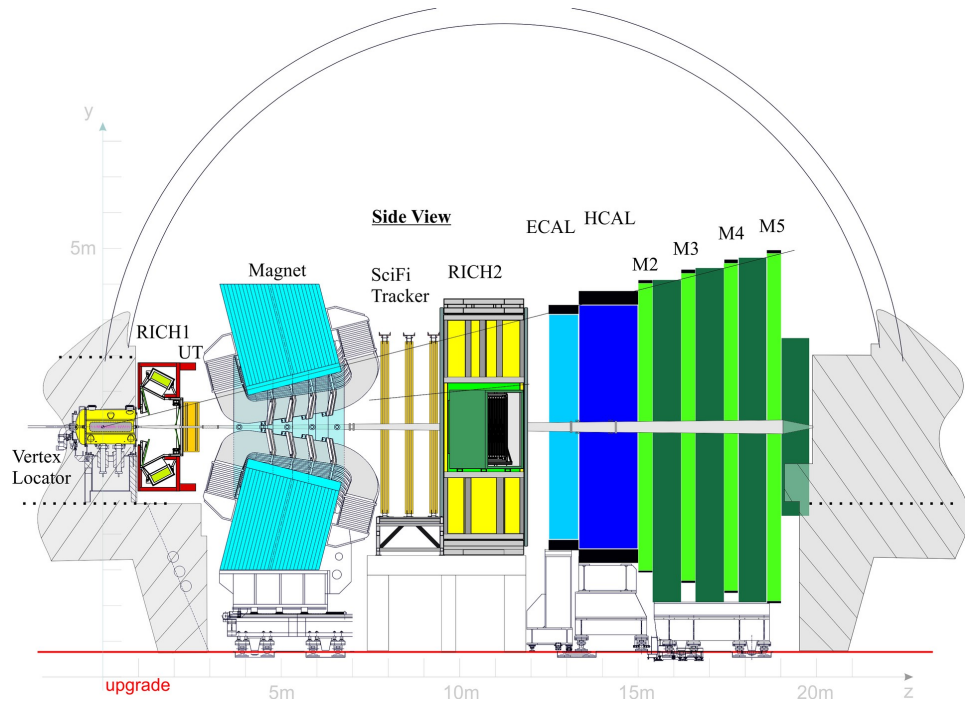


Figure 1.4: Side view of the LHCb detector layout that will be used in Run 3. From left to right all the different sub-detectors are shown: VELO, RICH1, UT, SciFi, RICH2, ECAL, HCAL, and Muon stations. Original image from the LHCb collaboration.

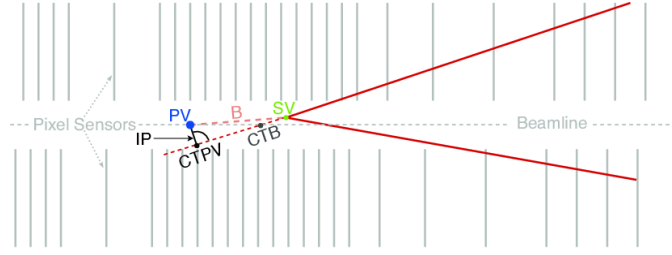


Figure 1.5: Sketch of a B meson coming from the primary vertex (PV) and decaying inside the VELO at the secondary vertex (SV) emitting two daughter particles (red lines). Closest to beam position (CTB), to PV position (CTPV), and impact parameter (IP) are shown. Original image from [16]

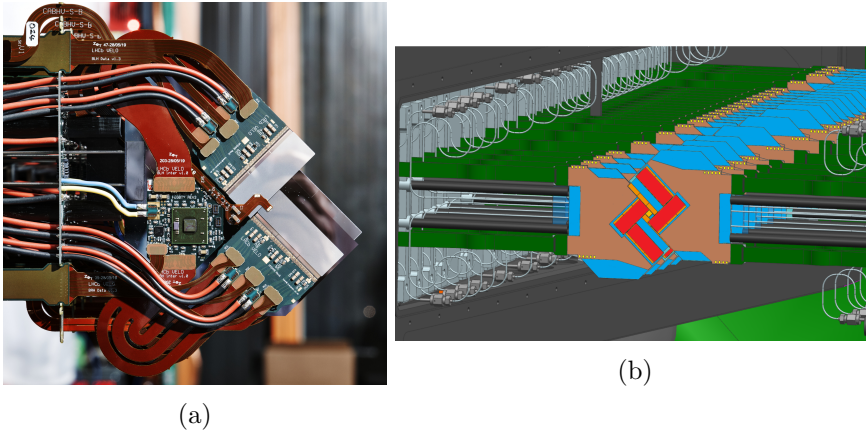


Figure 1.6: **(a)** Closeup picture of one side of the 26 stations of the VELO before the installation. These station will close onto the beam line in order to achieve high resolution in impact parameter (IP) measurements and vertex location. **(b)** An illustration of the VELO closed during stable beams [17]. Courtesy of the LHCb collaboration.

purpose is to locate the Primary Vertices (PVs) of the pp collisions, the Secondary Vertices (SVs), and to establish the Impact Parameter (IP). The IP is the distance of closest approach between a reconstructed track and the true origin of the particle, most commonly the primary pp collision vertex. At LHCb, charm and beauty hadrons travel on average 1 cm before decaying, so it is crucial to have a good IP resolution in order to reduce background and enable the correct identification of b and c flavoured hadrons. A sketch of a B meson produced in a pp collision with the reconstructed parameters highlighted is shown in Figure 1.5.

In order to achieve the specs required, the VELO is built using 52 silicon

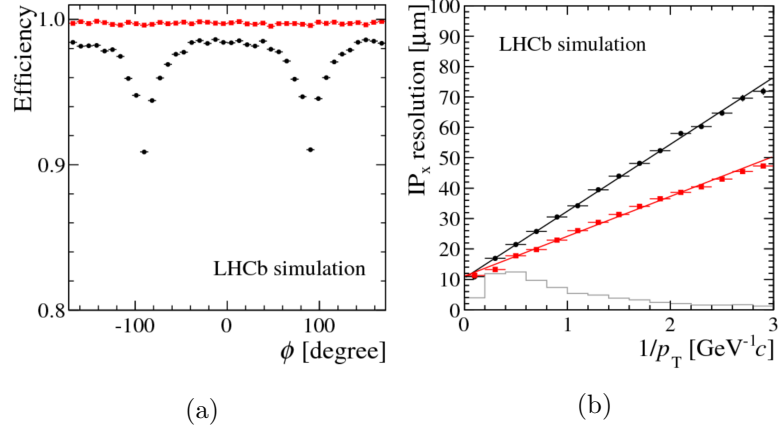


Figure 1.7: In red the upgraded VELO used for Run 3, in black the old VELO. **(a)** The plot shows tracking efficiency as a function of the azimuthal angle ϕ . **(b)** the plot shows the IP resolution as a function of $1/p_T$, while the histogram in grey shows the relative track population. Original plots from [18].

sensors placed on the two sides of the beam line and perpendicular to the beam. Each module is composed of four $200 \mu\text{m}$ thick pixel sensors with an active area of $(42.46 \times 14.08) \text{ mm}^2$, the pixels are square and have a pitch of $(55 \times 55) \mu\text{m}$. The VELO is positioned at just 5.1 mm from the LHC beams and is divided into two retractable halves that close onto the beam line only during stable collisions, in order to reduce the radiation damage to the sensor and protect the detector from eventual beam losses. The VELO modules and their mechanical construction is shown in Figure 1.6. The expected performance of the upgraded VELO based on full simulations is shown in Figure 1.7. Tracking efficiency is improved thanks to the wider coverage of the new modules, while also achieving better IP resolution.

VELO is encapsulated in an isolated vacuum vessel and separated by a thin aluminium foil from the main LHC vacuum. RF foil shields from the interference produced by the circulating beams, while keeping the effect of multiple Coulomb scattering limited thanks to its thinness and its low atomic number. A CO_2 cooling system is used to dissipate the heat produced by the detector and the electronics inside the vacuum vessel.

The Upstream Tracker (UT) is placed before the magnet and composed of four layers of silicon strips. The sub-detector is subdivided into two stations called UTa and UTb, with a distance of 315 mm from each other.

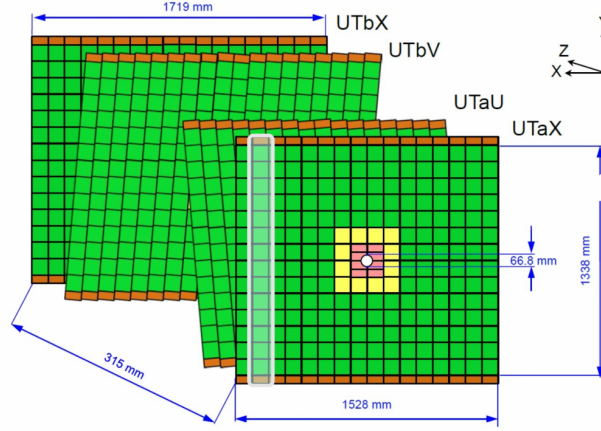


Figure 1.8: Illustration of the layout of the Upstream Tracker (UT). It can be seen the 5° offset between the modules to achieve 2D resolution. Presented in [15].

Each section contains two layers with a skew angle of 5° between them in order to provide bi-dimensional coordinates via stereoscopic projection and reduce misidentification of multiple tracks. The layout of the UT stations is shown in Figure 1.8. The silicon sensors have a thickness of $250 \mu\text{m}$ with an expected hit resolution of $50 \mu\text{m}$.

Using data from VELO and UT, it is possible to perform a fast estimate of momentum and transverse momentum of charged particles. This enables the online reconstruction framework to estimate the particle trajectory, reducing the computation and improve reconstruction quality.

The last part of the tracking system is the Scintillating Fiber tracker (SciFi). It is located between the magnet and the RICH2 and distributed in three stations (T1, T2, T3) with four layers each. A layer uses 2.5 m long multi cladding wavelength shifting scintillating fibers as the active material. In each station, the layers are tilted $\pm 5^\circ$ from each other in order to have bi-dimensional coordinates via stereoscopic projection. Each layer is composed of 12 modules, for a total of 144 modules for the whole sub-detector. The layout and placement of SciFi tracker's stations is shown in Figure 1.9.

SciFi provides a hit resolution of $100 \mu\text{m}$ and a high granularity in the active region. Due to multiple scattering in the upstream detectors, a higher resolution is not necessary.

To enable the measurement of p and p_T of charged particles, a conven-

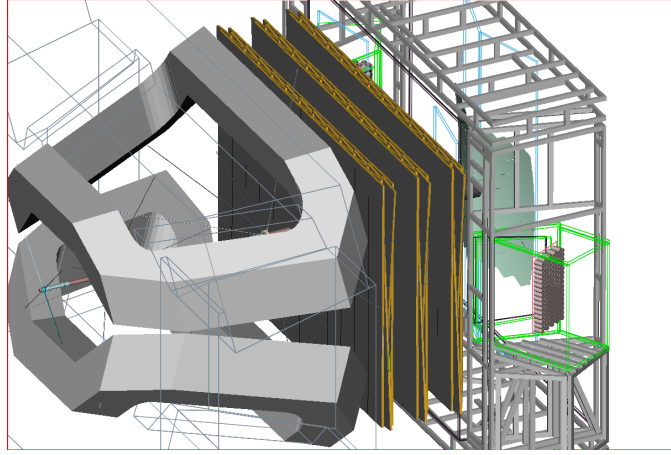


Figure 1.9: Illustration of the position of the Scintillating Fiber (SciFi) tracker between the magnet yoke and the RICH2. Presented in [15].

tional dipole is used to bend the trajectories. The magnet consists of two 25 tons identical coils symmetrically placed inside a 1450 tons yoke. The main component of the magnetic field is directed along the y axis, bending the particles' trajectories in the x - z plane. The integrated field strength is $4 \text{ T} \cdot \text{m}$ and the field profile is well known in order to reconstruct the track correctly. The magnet polarity can be inverted during data taking to compensate for the asymmetries in the detector layout, which have to be taken in consideration to evaluate CP violation parameters.

In Figure 1.10, it is shown how the tracking system classifies tracks. The relevant tracks for event reconstruction are long and downstream tracks. Long tracks have the best spatial and momentum resolution out of all tracks, thanks to the VELO data. Downstream tracks are mainly correlated with the decay products of long-lived particles and have lower resolution and efficiency, as they do not involve the VELO. The other tracks are discarded or, in the case of upstream tracks, may be used for calibration purposes [19].

1.3.2 Particle Identification

The rest of the sub-detectors are used for Particle IDentification (PID). PID is a crucial part of the detection process, since it enables the tagging of final states and reconstruction of the decays. This is done using three types of sub-detectors: Ring Imaging CHerenkov (RICH), calorimeters, and the

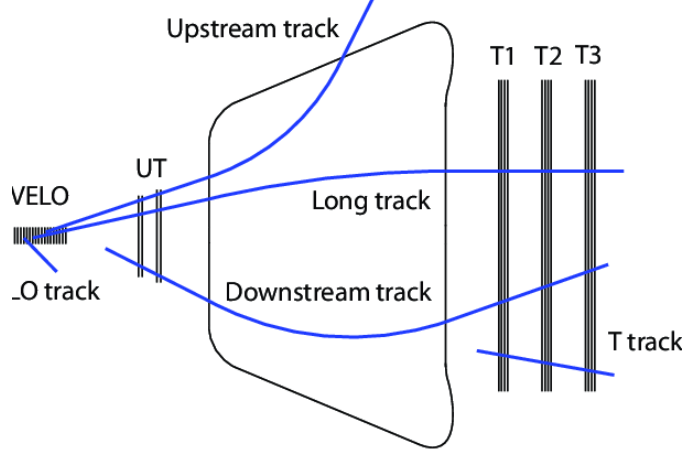


Figure 1.10: Illustration of the track reconstruction using the sub-detectors in the tracking system [16].

Muon system.

RICH detectors exploit the Cherenkov effect to measure particles' velocity. When a charged particle travels through a dielectric medium, it generates temporary electric dipoles and this leads to electromagnetic radiation to be emitted. If the speed of the particle is higher than the speed of light in that medium, constructive interference between the waves creates a wave front at a particular angle related to the speed of the particle. Assuming n the refractive index of the medium, the wave propagation angle θ in relation to the particle direction is given by:

$$\cos\theta = \frac{1}{n\beta} \quad (1.4)$$

where $\beta = \frac{v}{c}$ is the ratio between the speed of the particle and the speed of light in vacuum. As the name says, Ring Imaging Cherenkov detectors are used to reconstruct the emission ring and calculate β in order to identify and discriminate charged particles. Light is collected from the medium (called radiator) through mirrors and then detected via photon detectors.

The RICH system is composed of two stations (Figure 1.11), which use dielectrics with different n in order to measure different momentum ranges. RICH1 is positioned between VELO and UT and is designed to operate between $1 \text{ GeV}/c < p < 60 \text{ GeV}/c$ using C_4F_{10} as radiator ($n = 1.0014$).

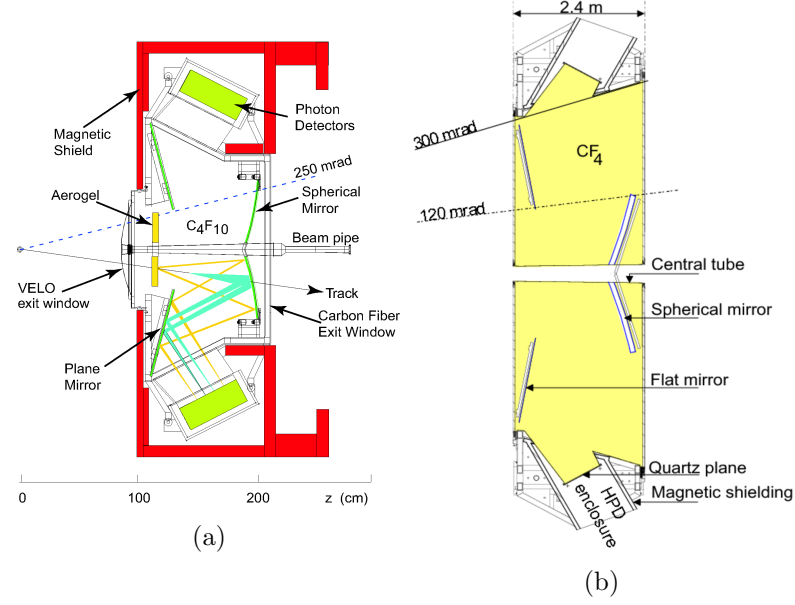


Figure 1.11: **(a)** Side view of the RICH1 detector. **(b)** Top view of the RICH2 detector. Presented in [15].

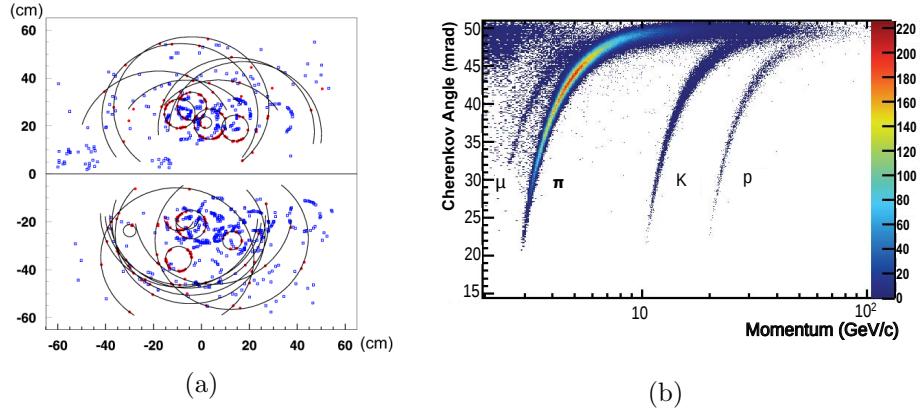


Figure 1.12: **(a)** An example of the readout from the RICH1 (Run 1 data). Photons are detected by the MaPMTs and then rings are reconstructed. **(b)** Particle identification using the reconstructed Cherenkov angle as a function of track momentum in RICH1 (Run 1 data). Original plots from [15] and [20].

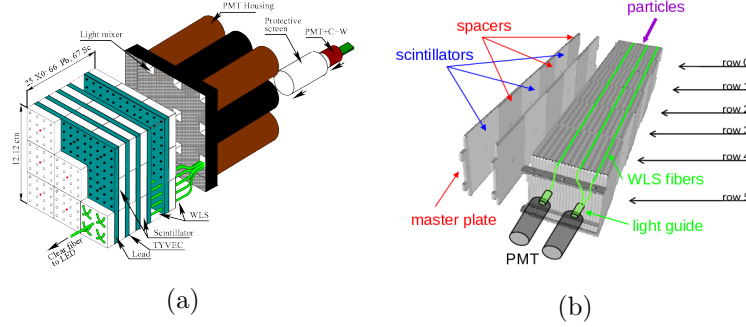


Figure 1.13: Illustrations of calorimeters modules: (a) ECAL, (b) HCAL. Courtesy of CERN.

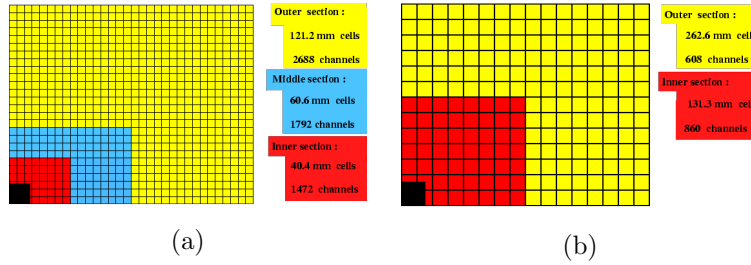


Figure 1.14: Details of the segmentation of the calorimeters: (a) ECAL, (b) HCAL. Presented in [15].

The sensors are an array of Multi-anode Photo Multiplier Tubes (MaPMTs). RICH2 is located after the SciFi and covers the momentum range $15 \text{ GeV}/c < p < 100 \text{ GeV}/c$. It uses CF_4 as the radiator ($n = 1.0005$), a 5% CO_2 is added in order to quench scintillation. An example of the RICH event is shown in Figure 1.12a, while Figure 1.12b shows how particles can be classified knowing their momentum and measuring their Cherenkov angle.

The calorimeter system is composed of two sub-detectors: ECAL and HCAL. The ECAL is an electromagnetic calorimeter, that measures the energy of electromagnetically interacting particles, such as photons and electrons. The ECAL is a sampling calorimeter segmented in modules and each module consists of alternated layers of absorber and scintillator arranged in a Shashlik design. The absorber is a slab of lead with a thickness of 2 mm, while the scintillator is 4 mm thick. All the modules are $120 \times 120 \text{ mm}^2$ and have a total of 66 layers, for a total depth of 25 radiation lengths. The light signal is then carried to the photon detectors by wavelength shifting fibers. Readout granularity depends on the location of the module inside

the detector: the closer to the beam line has a cell size of $40 \times 40 \text{ mm}^2$, the middle region $60 \times 60 \text{ mm}^2$ and the outer region $120 \times 120 \text{ mm}^2$. The energy resolution of the ECAL is $\sigma(E)/E = ((8 \div 10)/\sqrt{E} \oplus 0.9)\%$.

The HCAL has a sampling design similar to the ECAL, but using steel as absorber and scintillating tiles. The ratio between absorber and scintillator is 5.5:1 with a total depth of 5.6 interaction lengths. The readout granularity is divided in two regions, with a cell size of $131 \times 131 \text{ mm}^2$ for the inner region and $263 \times 263 \text{ mm}^2$ in the outer region. The energy resolution of the HCAL is $\sigma(E)/E = ((69 \pm 5)/\sqrt{E} \oplus (9 \pm 2))\%$. The design of the modules of the calorimeters is illustrated in Figure 1.13, the segmentation of the calorimeters is shown in Figure 1.14.

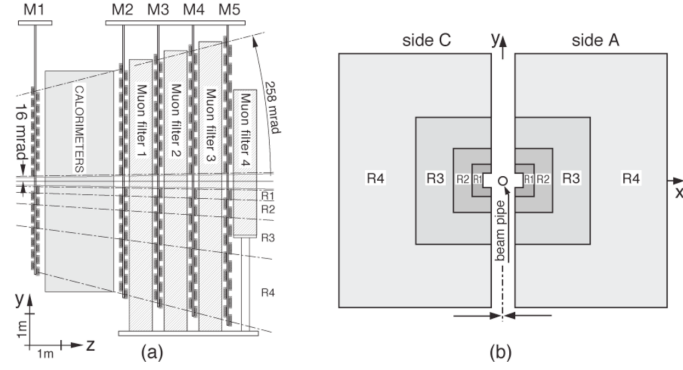


Figure 1.15: **(a)** Side view of the LHCb muon detector (M1 station is removed for Run 3), **(b)** Station layout view where the four granularity regions are indicated. Presented in [15].

The muon system is crucial to identify muons, which are present in several final states of b -hadron decay modes, e.g. $B_s^0 \rightarrow J/\psi(\mu^+\mu^-)\phi$, $B_s^0 \rightarrow J/\psi(\mu^+\mu^-)K_s^0$, $B_s^0 \rightarrow \mu^+\mu^-$. Muons are also used when their p_T is high in order to tag the spectator b -hadron associated with the signal one.

The muon system is composed of four muon stations (M2-M5), with an acceptance of $\pm 300 \text{ mrad}$ in both the horizontal and vertical plane. Since muons have a high penetration capability, the stations are placed after all the other sub-detectors. Every station is composed of an active layer and an absorber layer. The absorber is made of iron and is 80 mm thick. The active layer uses Multi-Wire Proportional Chambers (MWPCs) filled with a mixture of $\text{Ar}/\text{CO}_2/\text{CF}_4$ in a 5:4:1 ratio. Also in this design, granularity varies with the distance from the beam line: four regions (R1-R4) are defined

in order to achieve the same occupancy in all the detector.

The global muon identification efficiency is above 96% and the hadron misidentification probability is less than 1%. The placement and the segmentation of the Muon stations are shown in Figure 1.15.

1.4 LHCb Physics

1.4.1 Results from Run 1 and 2

$b \rightarrow sl^+l^-$ and tests of lepton flavour universality The SM predicts that the different charged leptons have identical electroweak interaction strengths: this property is known as *Lepton Flavour Universality* (LFU). Decays involving $b \rightarrow sl^+l^-$ are forbidden at tree level and therefore have to proceed via loop processes. However, in New Physics this does not necessarily apply. In LHCb, LFU has been tested by studying the $B^+ \rightarrow K^+l^+l^-$ decays, measuring their branching ratios and defining the R_H ratios as:

$$R_H = \frac{\int_{q_{min}^2}^{q_{max}^2} \frac{d\mathcal{B}(B \rightarrow H\mu^+\mu^-)}{dq^2} dq^2}{\int_{q_{min}^2}^{q_{max}^2} \frac{d\mathcal{B}(B \rightarrow He^+e^-)}{dq^2} dq^2} \quad (1.5)$$

where H is the hadron considered (K^+, K^{*0}) and $q_{min}^2 < q^2 < q_{max}^2$ is the dilepton mass-squared range. R_H is studied as a function of q^2 because of the $b \rightarrow s\gamma^*$ contribute, which is sensible to various hadronic resonances. The results presented in the paper [21] report that value of R_K is:

$$R_K(1.1 < q^2 < 6.0 \text{ GeV}^2 c^{-4}) = 0.846_{-0.039-0.012}^{+0.042+0.013} \quad (1.6)$$

which gives evidence for the violation of lepton universality in these decays (3.1σ discrepancy from the SM prediction). The comparison with the SM and the measurements of the same observable by the Belle and BaBar collaboration are shown in Figure 1.16.

CP violation and measurement of γ The CKM angle γ has been determined from the study of CP-violating observables in $B^\pm \rightarrow DK^\pm$ and $B^\pm \rightarrow D\pi^\pm$. As shown in Figure 1.17, the large difference between the mass peaks for B hadrons with different charges indicates CP violation of 85%, the largest ever observed.

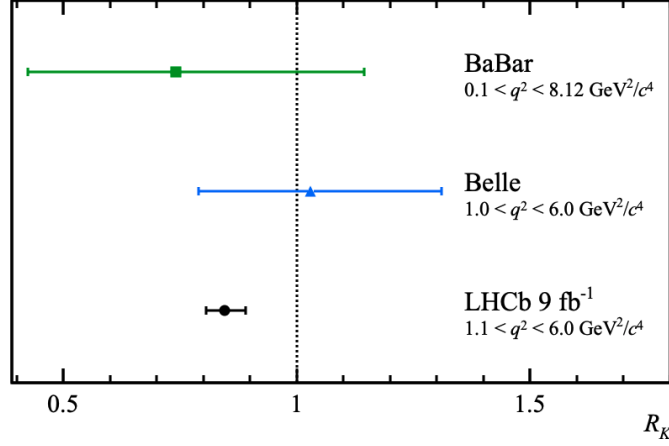


Figure 1.16: Comparison between R_K measurements which combines results from K^+ and K^{*0} channels. In addition to LHCb results, measurements from SM, Belle, and BaBar collaboration are shown. The assumption of LU SM prediction is $R_K = 1$. Presented in [21].

The result of the measurements reports a value of γ :

$$\gamma = (54.8_{-5.8-0.6+4.3}^{+6.0+0.6+6.7})^\circ \quad (1.7)$$

Exotic hadron states In the field of exotic hadrons spectroscopy, LHCb observed the first doubly charmed tetraquark $T_{cc}^+(3875)$ in the $D^0 D^0 \pi^+$ mass spectrum, consistent with a composition of $cc\bar{u}\bar{d}$ quarks [22] (See Figure 1.18). The measurements indicate a very narrow state, placed slightly below the $D^{*+} D^0$ threshold (the values are determined using the Breit-Wigner formula):

$$\begin{aligned} \delta_{BW} &= -273 \pm 61 \pm 5_{-14}^{+11} \text{ keV}/c^2 \\ \text{with } \delta_{BW} &= M_{BW} - (M_{D^{*+}} + M_{D^0}) \\ \Gamma_{BW} &= 410 \pm 165 \pm 43_{-38}^{+18} \text{ keV} \end{aligned} \quad (1.8)$$

1.4.2 Prospects for Run 3

The LHCb detector has been completely upgraded with some sub-detectors that got replaced and others which had their readout capability increased.

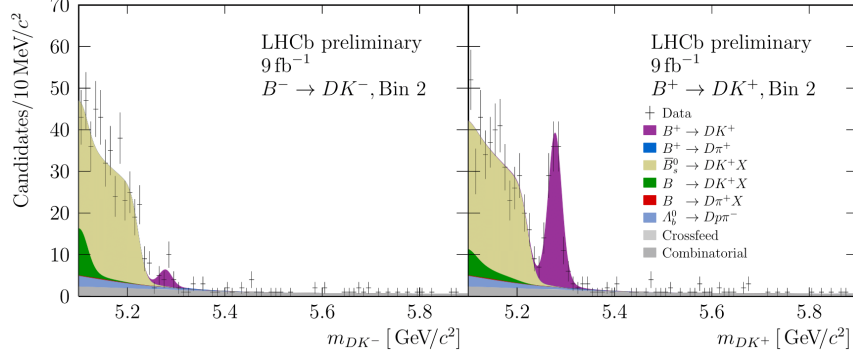


Figure 1.17: The combination plot shows the invariant mass distribution for the different charged B hadrons. The large difference between the heights of the mass peaks indicates a CP violation of 85%. Courtesy of the LHCb collaboration.

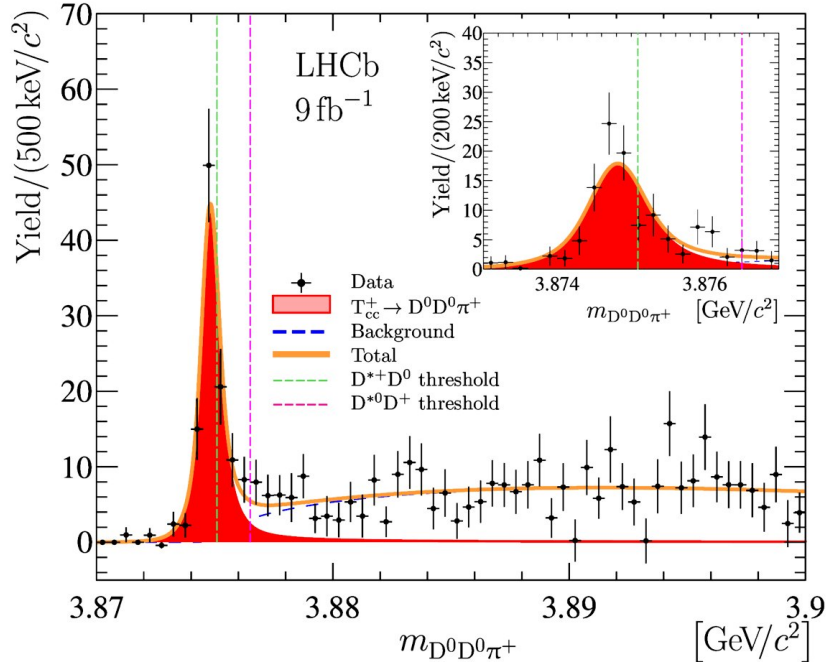


Figure 1.18: Distribution of $D^0 D^0 \pi^+$ mass where the contribution of non- D^0 background has been subtracted. Thresholds for $D^{*+} D^0$ and $D^{*0} D^+$ are indicated with vertical dashed lines. The zoomed plot shows the signal region with fine binning. Presented in [22].

This upgrade has been necessary to improve the sensitivity in flavour physics, in order to investigate possible New Physics in the Flavour Sector and make precision tests of the Standard Model (SM).

In Run 3, LHCb expects a 5 times increase in luminosity: the sample sizes collected for B and D final states will be bigger than the ones collected in the existing B -factories, being the best detector for the study of CP violation in B_s^0 decays. Precision tests of the SM could give hints of TeV-range massive particles while complementing the direct searches by ATLAS and CMS.

Continuing the LHCb legacy, CP violation measurements will be the focus of the research thanks to their strong experimental and theory constraints. Measuring branching ratios of rare decays in flavour changing neutral currents could give signs of the effects of new heavy particles.

Moreover, LHCb will be able to investigate the lepton sector and topics beyond flavour physics. Measurements in the electroweak physics, such as the effective weak mixing angle for leptons and the mass of the W-boson, will be complementary to the results from ATLAS and CMS thanks to the forward region coverage of LHCb, which makes the LHCb measurements sensitive to a different regime of the proton parton density functions. The study of Quantum Chromo-Dynamics (QCD) will also benefit from the forward coverage, extending the results reported for the central region.

Chapter 2

Data Acquisition System

To reach the physics goals of Run 3, a major upgrade of the Data Acquisition System (DAQ) is required. The design in Run 2 provided a Level 0 Trigger (L0T) implemented in hardware. Selections are made at 40 MHz using only calorimeters, muon systems, and VELO [23]. The trigger criteria are based on high deposits of transverse energy, in the order of several GeV, and momentum from muons, hadrons, electrons and photons [23]. While this provides high efficiencies on dimuon events, it typically removes half of the fully hadronic signal decays. In these hadronic decays the E_T threshold required to reduce the rate of triggered events to an acceptable level is already a substantial fraction of the B meson mass. This implies that the yield of B-hadrons is almost constant and independent of luminosity, thus the experiment cannot benefit from luminosities larger than $3 \times 10^{32} \text{ cm}^{-2} \text{ s}^{-1}$ as shown in Figure 2.1. The maximum readout rate of the L0T is limited to 1.1 MHz. For Run 3, the upgrade focuses on increasing the readout rate to 40 MHz for all sub-detectors. Moreover the hardware L0T is discarded and replaced with a real-time analysis in software.

The choice of a software trigger is more flexible and selection criteria can use all sub-detectors to make decisions. This comes at the cost of a 40 MHz readout, that with an event size of $\sim 100 \text{ kB}$ results in a total throughput of 32 Tb/s.

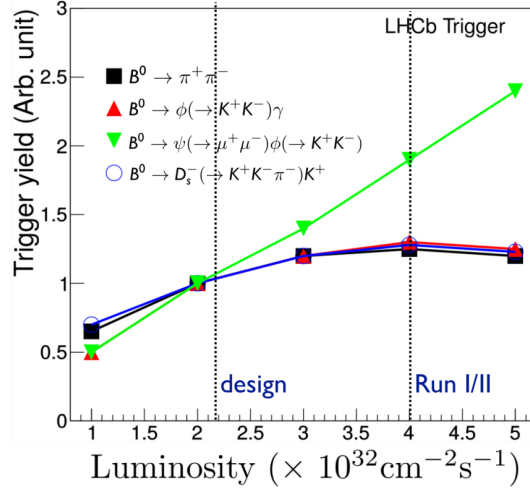


Figure 2.1: Trigger yield as a function of luminosity for the Level 0 Trigger (L0T) of Run 2. Saturation of the yield in the hadronic channels shows the limits of the old L0T. Presented in [24].

2.1 Data Flow

An overview of the new Data Acquisition System for the LHCb Upgrade is presented in this section [25]. The full Online chain is summarised in Figure 2.2. Data taken from the subdetectors is sent to the Event Building cluster via optical links. The Event Building cluster receives the data and hosts the Event Building process, which reorders and assembles full events. Full events are then processed by the High Level Trigger 1 (HLT1), which is executed on the Event Building cluster and uses the GPUs to accelerate the process. Reconstructed and selected events are then stored in a storage buffer which links the HLT1 to the High Level Trigger 2 (HLT2). HLT2 takes as input the last alignment and calibration, and the events are reconstructed with offline quality. Moreover, events are then compressed before being written in a second storage buffer. From there, events are copied to tape storage. To achieve the design requirements, almost all of the Online system has been moved to the surface, leaving just the Front-End Electronics underground. This gives more flexibility and reduce the costs, since the system is not space and power constrained.

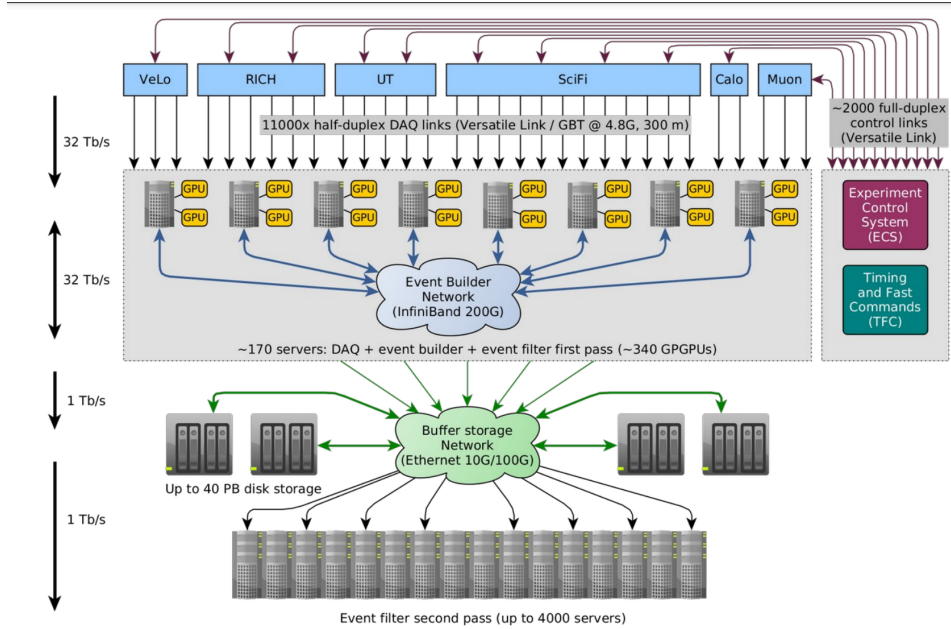


Figure 2.2: Overview of the Data Acquisition system (DAQ) for Run 3. Courtesy of Tommaso Colombo (CERN).



Figure 2.3: The PCIe40 Board: single custom-made FPGA board for data acquisition and control. Its behaviour is defined by firmware. Image courtesy of CERN.

2.1.1 DAQ Boards

The readout cards are the PCIe40 boards (Figure 2.3) [26]. These boards are equipped with FPGAs that communicate with the Front-End Electronics (FEE) via optical links using the radiation-hard gigabit transceiver (GBT) developed at CERN [27]. Moreover, they are interfaced to the software control system via the PCIe bus of the host server in which they are installed. PCIe40 are configured in three flavours depending on the application:

- **SODIN** plays the role of readout supervisor, it centrally synchronises and manages the readout of events by dispatching the LHC clock and generating clock-synchronous commands;
- **SOL40** interfaces the Timing and Fast Control (TFC) and the Experiment Control System (ECS) with the detector FEE. It distributes the timing and clock information from SODIN to the FEE via the optical link, while it is also giving the ECS the capability to both control and monitor the FEE over the same link;
- **Tell40** is responsible for the readout of the Front-End event fragments into the DAQ infrastructure: consecutive event fragments are buffered in Multiple Fragments Packets (MFPs) of suitable size and then copied to the host server RAM via Direct Memory Access (DMA).

Around 11000 GBT optical links connect the detector to the readout boards, requiring ~ 500 TELL40 in order to digest the total throughput at 40 MHz. The DAQ boards are hosted in ~ 170 servers where the event building takes place.

2.1.2 Event Builder

Since every event is fragmented across multiple servers, it is necessary to build the full event in order to proceed with the online data reconstruction. The Event Builder (EB) is responsible for this task, receiving fragments from the readout boards. The Tell40 sends fragments to the host in Multiple Fragment Packets (MFPs). The number of fragments in an MFP is set centrally by the event builder. MFPs are then exchanged via an InfiniBand network between the servers in the Event Building cluster and reordered to build the Multiple Event Packet (MEP), which concatenates the MFPs starting with the same event ID.

Sub-detector	Fragment Size (B)	# Tell40 Streams	Event Size (B)	Event Fraction
VELO	156.25	104	16250	0.13
UT	100	200	20000	0.16
SciFi	100	288	28800	0.23
RICH1	166.67	132	22000	0.18
RICH2	166.67	72	12000	0.10
CALO	156.25	104	16250	0.13
MUON	156.25	56	8750	0.07
TOTAL	1002.08	956	124050	1

Table 2.1: Sub-detector data distribution estimates based on Monte-Carlo simulations and number of links available. A TELL40 stream is one of the two interfaces present on the FPGA that connects to the sub-detector and the host machine. The expected average event size is 124 kB. Simulation data courtesy of LHCb.

MFP and MEP are used to increase the message size to efficiently use the network, since single fragments are $\mathcal{O}(100)$ B and single events are ~ 100 kB on average, as shown in Table 2.1. The network performance with different packet sizes has been measured, suggesting a message size of > 1 MB, as shown in Figure 2.4. The MEPs are then transferred to the High Level Trigger 1 (HLT1). The LHCb trigger scheme is summarised in Figure 2.5. A more detailed description of the EB is available in section 3.1.

2.1.3 High Level Trigger 1

The High Level Trigger 1 is the first software trigger that data encounter (the framework name is Allen [28]). It is executed on the same servers used for the EB and runs on Graphical Processor Units (GPUs). Modern GPUs are highly parallel multiprocessors that can be used to accelerate the reconstruction process. The HLT1 sequence relies on 3 sub-detector systems in order to select events. The tracking system is used to reconstruct tracks and vertices, while also evaluating clustering and momentum. Muon stations and calorimeters data integrates the particle identification. Selections are made using the reconstructed momentum, displacement, and muon identification (Figure 2.6). While the minimum time between collisions is 25 ns (40 MHz rate), the real average collision rate is ~ 30 MHz due to geometric constraints of the LHC. HLT1 achieves the 30 MHz goal with ~ 200 GPUs.

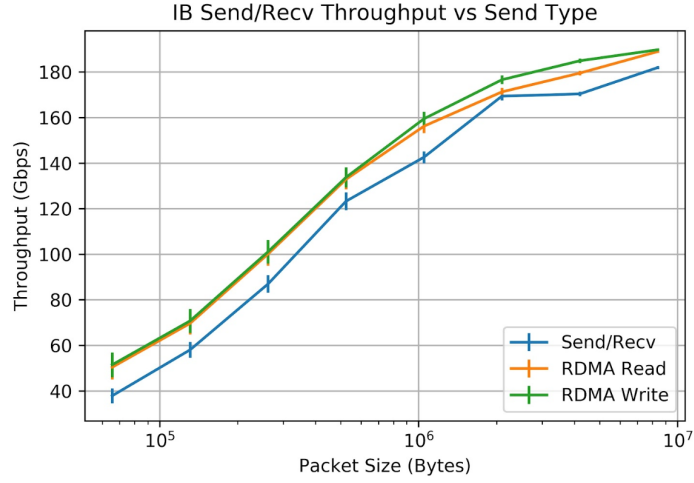


Figure 2.4: Throughput measurements with different packet sizes. These measurements were taken between two nodes of the test cluster of the EB. The different data series in this plot will be explained in detail in section 3.1.

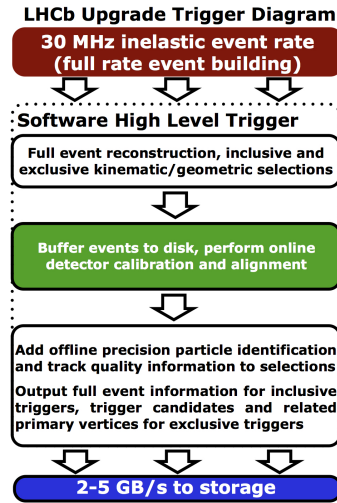


Figure 2.5: LHCb trigger diagram for Run 3. Courtesy of LHCb.

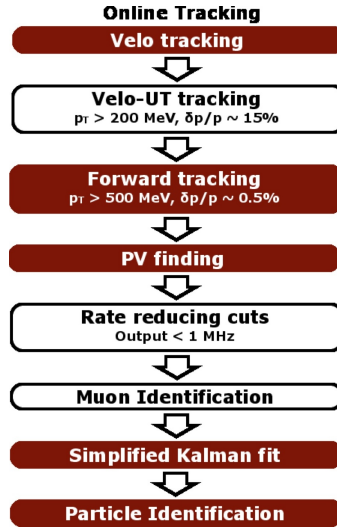


Figure 2.6: HLT online selection scheme for the Run 3. Courtesy of LHCb.

2.1.4 High Level Trigger 2

The last element in the online data flow is the High Level Trigger 2 (HLT2). The detector is aligned and calibrated in near real-time and the remaining events are reconstructed with offline quality, thanks to a storage buffer placed between HLT1 and HLT2. HLT2 classification outputs the events in a custom format called *Turbo Stream* [29] that preserve only the candidates reconstructed by the trigger discarding the rest of the event, thus reducing the event size of an order of magnitude. This optimises the use of disk and tape storage and computing resources, discarding raw data and removing the offline reconstruction. This technique was used extensively during Run 2 and will be used for more than 70% of the events in Run 3. A second storage buffer is present between HLT2 and tape storage.

Chapter 3

Design of the Event Builder Infrastructure

In this chapter, the hardware, the architecture, and the software used for the Event Builder (EB) will be discussed. A detailed view of Message Passing Interface (MPI) is presented to introduce the key concepts underlying parallel programming. A brief overview of common parallel computing terms and principles used in the next sections is available in Appendix A.

3.1 The LHCb Event Building Process

3.1.1 Detailed Overview of the Event Building Process

As described in the overview of the DAQ system, the data from the LHCb detector is divided across ~ 11000 optical links connected to ~ 500 Tell40 readout cards.

The readout cards are placed in groups of up to three in the Event Building servers. This implies that data is fragmented over the entire cluster. Data from all sub-detectors has to be collected and all the data fragments of a single event have to be assembled in the same place to proceed with the reconstruction and selection process. This process is the so called Event Building (Figure 3.1). This implies that servers in the Event Building cluster must be connected by an interconnection network in order to send and receive data fragments. The network is realised using the InfiniBand technology and is further described in the next subsections.

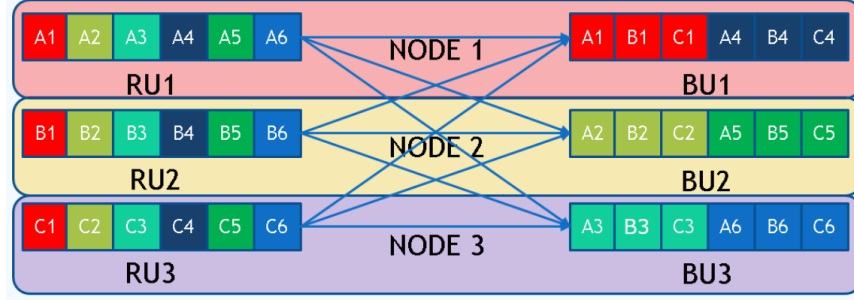


Figure 3.1: Example of the reordering done by the EB nodes. Fragments are sent in an all-to-one way from RUs to BUs in order to build the full events. Courtesy of Flavio Pisani (CERN).

To provide a simple description of the Event Building task, the process can be divided in two groups of logical units:

- **readout Units (RUs)** collect the fragments from the Tell40 boards and send them to the Builder Units (BUs);
- **Builder Units (BUs)** receive and assemble fragments into full events.

All the EB nodes (servers that are hosting the required hardware and running the EB software) run multiple instances of RUs and BUs based on the hardware installed. The events produced by the LHCb detector in pp collisions have a nominal size of ~ 100 kB, with fragments of $\mathcal{O}(100)$ B. As explained in section 2.1.2, modern interconnection technologies are not designed to transfer just a few hundreds of bytes efficiently, so fragments are grouped into Multi Fragment Packets (MFPs) and consequently events into Multi Event Packets (MEPs). The packing factor of the LHCb detector is configurable and tests were made using a value of 30000.

3.1.2 The Event Building Network

The design of the Event Building network reflects the nature of the data flow. Since the generated traffic comes from multiple sources going to a single destination, the overloading of specific links could lead to network congestion and the consequent degradation of the performance. The strategy put in place to avoid these issues in the EB software is called *linear shifting scheduling* [30].

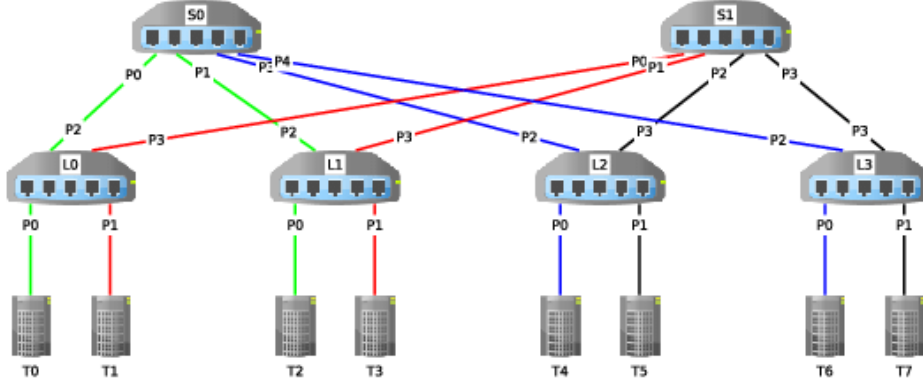


Figure 3.2: Example of the network layout between the EB nodes. Data paths are highlighted with different colors to show that there is no sharing of the links during the phase n of the linear shifting. Original image from LHCb.

Linear shifting scheduling organizes the exchange of data in a way that links are never overloaded. In a network with N data sources, the processing of N events is divided in $N - 1$ phases:

- In any phase, each RU sends data to one BU and each BU receives data from one RU;
- During phase n , the i -th RU sends data to $\text{BU} \bmod(i + n, N)$;
- All the units switch synchronously from phase n to $n + 1$.

The process is illustrated in Figure 3.3 for a network of $N = 4$ nodes. Using this scheduling technique, the destination conflict issue is solved. On the other hand, this requires synchronisation between nodes and the processing of multiple events in parallel. Moreover, the network needs to be *non-blocking*, as it must be always possible to establish a free path between a pair of unused nodes regardless of the other traffic. Since the traffic generated by the linear shift is one-to-one, this guarantees that the full exchange is conflict-free. To ensure the scheduling, ghost nodes are added to balance the algorithm and the distribution over the cluster.

The EB linear shifting scheduling has been designed to use full synchronisation between all the nodes in order to have a fully deterministic conflict-free scheduling. This alignment is done by an exchange of synchronisation messages over the network and involves all the nodes. This procedure has

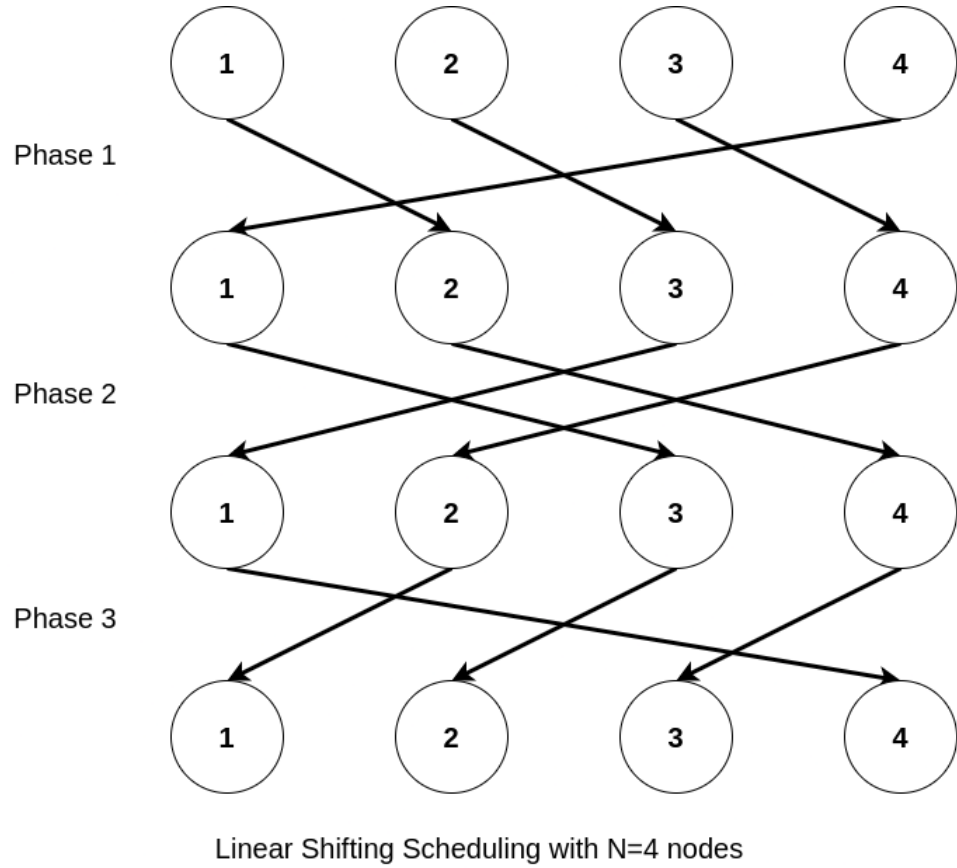


Figure 3.3: This illustration shows the steps of *linear shifting scheduling* [30] with $N = 4$ nodes. Between each phase, nodes are fully synchronized to guarantee that the exchange is conflict-free. In the full EB deployment, N is ≈ 300 .

a complexity of $\mathcal{O}(N)$ with a total of $N(N - 1)$ messages; in the full EB deployment with $N \approx 300$ nodes, $\mathcal{O}(10^5)$ messages are exchanged.

The network architecture used to connect the EB nodes is a *folded Clos* network, also known as *fat tree*. By looking at Figure 3.2, the switches connecting to the nodes are called *leaves*, their uplinks are then connected to the other row of switches which are called *spines*. This network connects a node to another in maximum three hops, no matter which nodes are considered. This configuration is highly scalable, since if the hardware has enough ports, it is possible to add *leaves* and *spines* without modifying the existent layout. Moreover, each *leaf* has m equal cost paths of the inter-*leaf* traffic, where m is the number of *spines*.

3.1.3 The Event Builder Finite State Machine

A Readout Unit or Builder Unit is structured as a Finite State Machine and is managed by the Experiment Control System (ECS). Here is presented the flow of a RU:

1. The network library is initialized;
2. Buffers are allocated and configured to receive data from the PCIe40 cards;
3. Information about the data sources is exchanged (`src_ids`, `src_names`) with the BUs using *broadcast* operation;
4. Fragments are read from the PCIe40 cards;
5. Processes are synchronised using a *barrier*;
6. MFPs' sizes are sent (*one-to-one send*) to the BUs using *scatter* operation;
7. Linear shifting is used to send MFPs to BUs (*one-to-one send*), synchronizing each step with a *barrier*;
8. The flow is repeated from step 4.

The highlighted operations will be described in detail in the next section. Steps from 1 to 3 are part of the START state, while steps for 4 to 8 are part of the RUN state of the FSM. The flow of a BU is similar to the one showed above, using the opposite operations.

3.2 Message Passing Interface (MPI)

The early versions of the EB software used MPI to implement the process management and the network communication. MPI is presented in detail to give the reader the basic concepts around which the EB software and the InfiniBuilder library are designed, and to provide a reference for comparison with the new versions of the EB with the InfiniBuilder library.

MPI is a message-passing standard designed for parallel computing architectures. It offers syntax, semantics and library routines to cover a wide range of applications and is available in C and Fortran. MPI is the *de facto* standard for communication among processes in a distributed memory system. MPI is meant to provide to the user an high level environment with virtual topology, synchronization and communication routines between processes. MPI applications are usually written using the Single Program Multiple Data (SPMD) paradigm, as the programmer writes a single program that gets distributed on multiple nodes by MPI and uses MPI built-in functions to know its role and behave as intended.

3.2.1 Structure of an MPI Application

In order to describe the pros and cons of MPI, an MPI example is analysed.

Listing 3.1: Example of an MPI application

```
1 #include <mpi.h>
2 #include <stdio.h>
3
4 int main(int argc, char** argv) {
5     // Initialize the MPI environment
6     MPI_Init(NULL, NULL);
7
8     // Get the number of processes
9     int world_size;
10    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
11
12    // Get the rank of the process
13    int world_rank;
14    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
15
16    // If it is the server, send to process 1
17    int number;
18    if (world_rank == 0) {
```

```
19     number = -1;
20     MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
21 } else if (world_rank == 1) {
22     // If it is the client, receive from process 0
23     MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
24             MPI_STATUS_IGNORE);
25     printf("Process 1 received number %d from process 0\n",
26           number);
27 }
28 // Finalize the MPI environment.
29 MPI_Finalize();
30 }
```

MPI is initialised by calling the `MPI_Init` function, which defines some global environment variables such as `MPI_COMM_WORLD`.

`MPI_COMM_WORLD` is called the default communicator and gathers all the processes where the program is run. Its dimension can be retrieved by the `MPI_Comm_size()` method, and the rank, which is the unique identifier of the process in the communicator, can be obtained by using `MPI_Comm_rank()`.

After this initial setup, nodes can start to talk with each other using the `MPI_Send()` and `MPI_Recv()` routines. The behaviour of the process is switched at runtime by looking at the rank. When all of the operations are completed, the program is terminated by calling `MPI_Finalize()`, which takes care of deallocating resources and closing the communications.

An MPI program is usually launched using its `mpirun` command from the terminal. `mpirun` connects to the remote nodes via SSH, defines the needed environmental variables and sets what is needed to run the application. The remote nodes can be specified in a file and multiple processes can be instantiated per node. Connections are specified at high-level in the configuration of `mpirun`: the user does not need to care about the technology used to interconnect the nodes, as MPI automatically sets it up. MPI is designed to be used with static allocation of processes: the process number cannot be changed at runtime, which means that if a process dies, the MPI application exits ungracefully.

3.2.2 Data Exchange

MPI defines multiple methods of exchanging data between processes in order to cover most of the use-cases. For one-to-one communication, six types

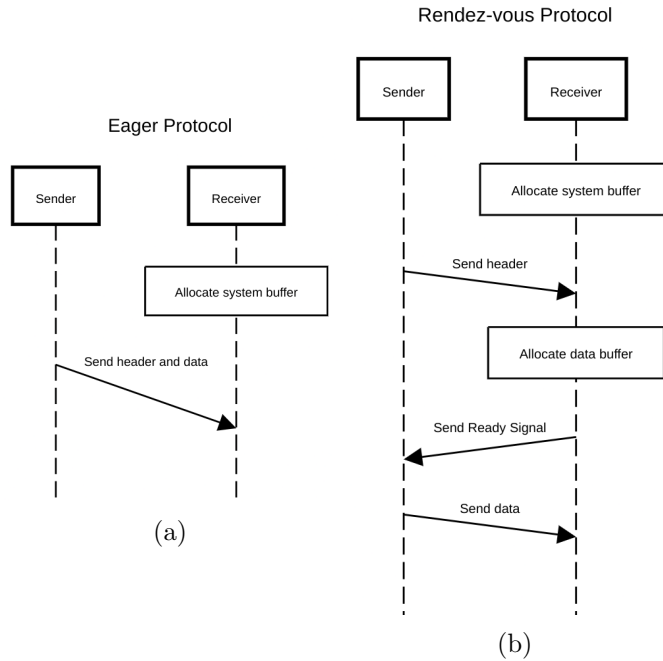


Figure 3.4: Message sequence diagrams for the send/receive protocols: **(a)** Eager, **(b)** Rendez-vous.

of send and receive routines are defined. To better understand the applications, some basic concepts are illustrated. When the execution of a function requires the application to wait until the operation completes, it is called a *blocking* function. If the application can proceed without waiting for completion, it is called *non-blocking*. Non-blocking functions require the user to check the completion of the operation.

The other concept is *synchronisation*: synchronous **send** can complete only if the **receive** has completed the reception of the message. Asynchronous routines only guarantee that the send buffer (i.e. the memory location where the message is stored) can be reused safely without compromising the exchange, but they do not say anything about the receiving process.

MPI implements various combinations of these two concepts to satisfy every use-case.

Another interesting point in the MPI communication concerns the message protocols. A message consists of an header, which stores tag, communicator, length, and other informations, and the data itself. Two different

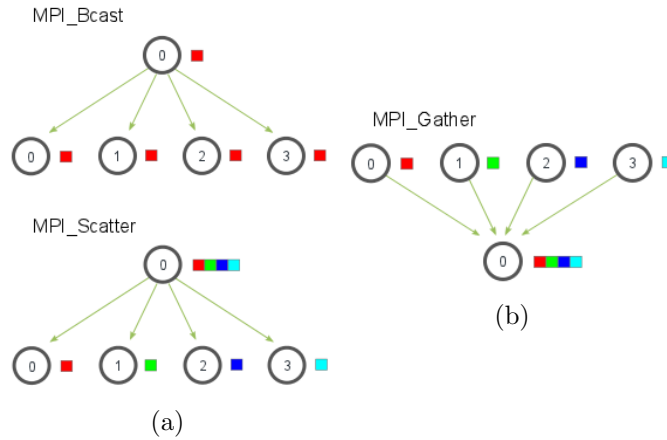


Figure 3.5: Illustrations of collective functions: **(a)** Broadcast and Scatter, **(b)** Gather. Original diagrams from [31].

protocols can be used:

- The **Eager** protocol assumes that the destination has enough space to store the message sent: this reduces the delays required by synchronisation and simplifies programming, but requires significant buffering and can introduce additional copies (an explanation of the role of copies is available in Appendix A.5).
- The **Rendez-vous** protocol delivers first the header, the receiver then sets up the space required for the reception of data, and eventually tells the sender that it is ready for the transfer. This approach can relax the requirements on copies and buffer allocation, but also adds complexity and synchronization delays.

MPI chooses which protocol to use depending on the message size. The message sequence diagrams for the two protocols are shown in Figure 3.4. The high-level programming and versatility of MPI shows up in the collective methods. Collective methods cover a set of the mostly used collective operations, as all-to-one, one-to-all, and all-to-all communications. MPI abstracts the algorithms needed to do this kind of operations by offering a simple interface to the programmer. Some of the functions often used are (also depicted in Figure 3.5):

- **MPI_Bcast** broadcasts the same data from one process to all;

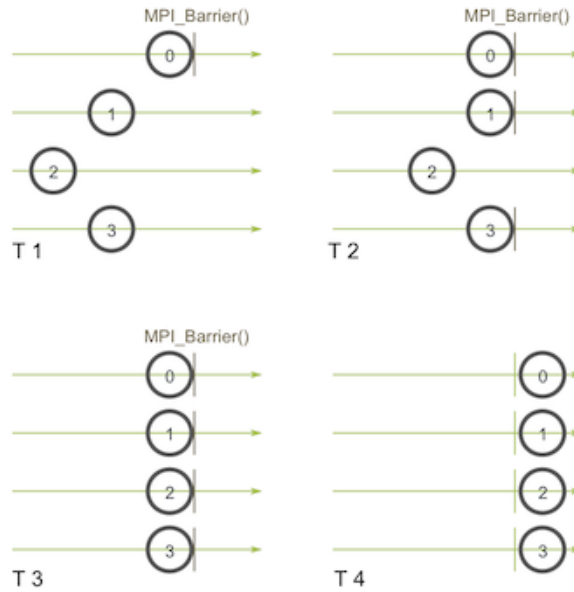


Figure 3.6: Illustration of the different phases when using the `MPI_Barrier`. Processes go idle after reaching the barrier (T1 and T2), waiting for all to arrive. Once all the processes have reached the barrier (T3), they are released and can proceed with their operations (T4). Original diagrams from [31].

- `MPI_Scatter` sends chunks of an array of data from one process to all. The difference between this method and the broadcast is that `MPI_Scatter` sends different data to different processes;
- `MPI_Gather` gathers data from all the processes to one and puts all of the data in one single array;
- `MPI_Reduce` works as gather, but instead of putting data into an array it performs a specified operation to it and returns the result;
- `MPI_Alltoall` is a combination of scatter and gather. It sends specific data to specific processes and collects specific data from specific processes. The Event Builder process is based on this function;
- `MPI_Barrier` is a synchronisation procedure, no exchange of data occurs. Any node reaching a barrier is stopped until all of them reached it. They are then released all together (Depicted in Figure 3.6).

Other collective methods, being a combination of the above, are available.

3.3 InfiniBand Network Technology

This section focuses on the specifications of the InfiniBand network technology and aims at providing a detailed overview that will help understand the design choices for the InfiniBuilder library.

InfiniBand is a computer networking standard defined by the InfiniBand Trade Association (IBTA)¹ usually used in high-performance computing that features high throughput and low latency. It is used to interconnect computers both directly or with a switched interconnect. InfiniBand has been chosen because of *Remote Direct Memory Access* (RDMA) read and write operations and since it is cheaper and more reliable than Ethernet at the required specifications of the EB [32].

RDMA is a direct memory access from the memory of one computer into that of another without involving the CPU, cache and operating systems, as shown in Figure 3.7. This implies that no copy is required, hence the *zero-copy* naming. The advantage of RDMA is the reduction in latency and the improvement of throughput, while freeing CPU cycles and memory bandwidth for computing. To achieve the requirements of the EB software without using RDMA, it would be necessary to have much more servers, which would be less cost effective.

InfiniBand transmits data in packets with a 4 kB payload that can be chained to form a message with a maximum size of 2 GB. It supports sends, receives, multicast, and atomic operations.

The InfiniBand interface is inherently **non-blocking**: this is necessary because the operations are offloaded to the Host Channel Adapter, reducing the load on the host. The Host Channel Adapter and the main processor exchange operations by using *queues*. Queues are First-In First-Out (FIFO) buffers which hold a finite number of objects; these objects can be transfer operations, completion events, and more. The use of queues guarantees that the Host Channel Adapter and the main system can work asynchronously and will not block each other. The queues support multiple producers and consumers and are thread-safe.

Communication happens using *work queues*. Send and Receive Queues are used in pairs, called Queue Pairs. Queue Pairs are destination and sources of all messages. To send and receive messages, Work Requests are

¹<https://www.infinibandta.org/>

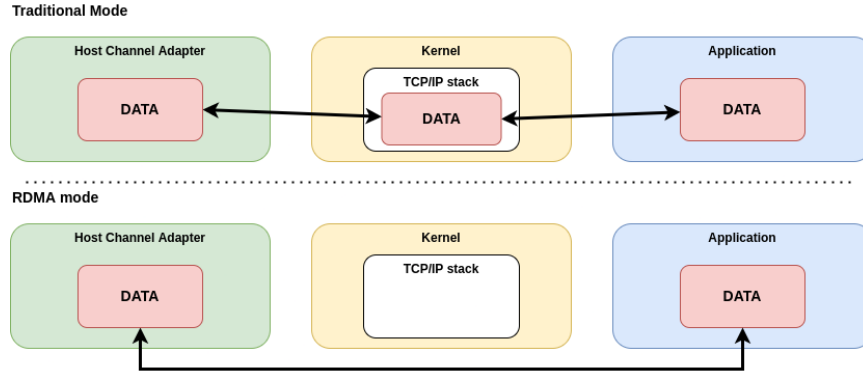


Figure 3.7: Difference between traditional way of transferring data and Remote Direct Memory Access. The network device can directly access the application memory, freeing resources and reducing copies.

posted in a Queue Pairs. Work Requests contain information on the type of transfer (send or receive), the memory location pointer, and other message information. Once a Work Request is posted, the Host Channel Adapter will manage the transfer on its own and when a Work Request is completed, a Work Completion entry is posted onto a Completion Queue associated to the work queue. The host system can then look for the Work Completion entry by polling the Completion Queue.

All of these operations are hidden under the hood when using high-level libraries such as Message Passing Interface (MPI) or can be tuned and programmed by using the low-level Application Programming Interface called *InfiniBand verbs*.

The network cards used in the EB cluster are Single Port NVIDIA Connect-X 6 Smart Host Channel Adapters² supporting 200 Gbps bandwidth per port (Figure 3.8). Two Host Channel Adapters are installed in each EB server, for a total of two 200 Gbps links available.

3.4 Requirements from the EB Software

The EB software was developed using MPI as the network communication stack and process manager. Adequate performance was achieved while testing and simulating the EB software and hardware structure. Further testing showed a number of inconveniences in the MPI implementation that will now

²NVIDIA Product Page

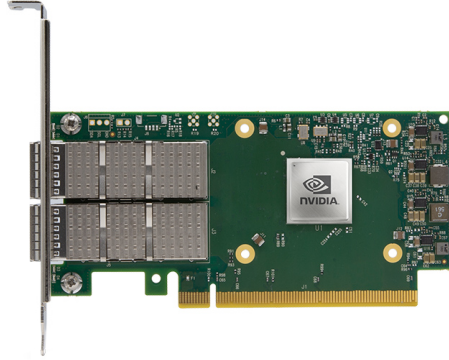


Figure 3.8: The Host Channel Adapter (HCA) used in the EB cluster. Courtesy of NVIDIA Corporation.

be discussed.

3.4.1 Warm-Up

Warm-up is defined as the time needed by the program to reach its optimal performance. MPI has a long warm-up when used in the EB, since large messages (~ 4 MB) use the rendez-vous protocol and buffers have to be allocated. This means that the system needed to exchange empty messages during the startup in order to reach the expected performance. In an on-line DAQ, this span of time when the system cannot register any data is considered dead time.

3.4.2 Dead Processes

Dead processes are processes that for some reason do not respond to the system anymore.

When this happens - and it is expected to happen with $\mathcal{O}(1000)$ processes - the only way to reset the system to a known state with MPI is killing the whole application and restarting it from the ground up. Some sort of runtime restart is not possible with MPI due to its static process allocation design. This means that all of the initial setup has to be repeated, including the warm-up, contributing to the additional dead time.

3.5 The Choice of a Custom Design

The inconveniences due to the use of MPI lead to the decision of evaluating the migration to a custom solution, designing a new network communication library. The main features of this library are:

- Little to no warm-up needed to reach full performance levels
- Reset and reconfiguration at runtime without killing the EB processes
- Lighter and specialised implementation with just the main functions optimised for the EB software
- High-level API as a drop-in replacement for MPI calls, reducing the effort needed to transition from one system to the other
- Fine tuning possible using the *verbs* low-level interface

The custom library, named InfiniBuilder, will be presented in the next chapter.

Chapter 4

The InfiniBuilder Network Library

4.1 Design Choices

The InfiniBuilder network library has been designed following the specifications and requirements of the EB. InfiniBuilder is implemented using the InfiniBand *verbs* low-level programming interface to precisely control the InfiniBand hardware and leverages all of its features. The library is written using the C++ programming language version 11, the same used for the EB software to simplify adoption and compilation. A detailed description of the inside mechanisms of the InfiniBuilder library is discussed in the next sections. This description, while being very technical, is necessary to understand the issues found in the implementation and how the methods and calls were modified to solve and prevent those problems, such as synchronisation.

4.2 Setup

Prior to being able to communicate over InfiniBand, it is necessary to setup the hardware and tell the system how the network is configured. This configuration takes place at runtime. The process can be divided in three steps:

- Read the configuration file to know which processes to connect;
- Startup of the local hardware and InfiniBand environment;
- Exchange of InfiniBand setup information over a TCP connection.

4.2.1 Configuration Parser

Configuration is done using a JSON file. JSON is a human and machine-readable file format used to store information. The configuration holds the information needed to setup the InfiniBuilder communication between all the processes.

The file is structured as an array of objects. Each object is composed of:

- **hostname:** The hostname of the server where the process is hosted. This is used for the out-of-band communication, that is not communicating over InfiniBand but over TCP on a separate Ethernet network in this case;
- **port:** the port of the TCP socket opened to exchange configurations;
- **ibdev:** the name of the InfiniBand Host Channel Adapter in the Linux Operating System;
- **UTGID:** The unique identifier string of a process, this is set in an environment variable by the Experiment Control System (ECS) when starting the processes.

Listing 4.1: Example configuration file for 2 processes.

```
1 "hosts": [  
2     {  
3         "hostname": "sceb02",  
4         "port": 10000,  
5         "ibdev": "mlx5_0",  
6         "UTGID": "EB_SCEB02_RU"  
7     },  
8     {  
9         "hostname": "sceb03",  
10        "port": 10001,  
11        "ibdev": "mlx5_0",  
12        "UTGID": "EB_SCEB03_BU"  
13    }  
14 ]
```

The parsing of the JSON file is done using the helper class `Parser`. The `Parser` class constructor takes the path to the configuration file as an argument. The decoding of the JSON format is done using the `ArduinoJSON` library, converting the data to a C++-like object. Parsing takes care of

checking the information provided by the file with the corresponding values on the machine, selecting the right entry by matching the UTGID. If the UTGID is not found, an exception is thrown to signal the mismatch. The UTGID is the reference used to identify the specific process and to set the behaviour of the EB software. In the example above, the UTGID `EB_SCEB02_RU` indicates that the process is part of the EB cluster, and is the readout unit (RU) of the second node of the SciFi sub-detector readout (SC stands for SciFi side C),

The parsing library also populates a `std::vector` containing the network information of each process in a `struct sockaddr_in`. This format is POSIX-specific and is used for the later communication over a TCP socket. `Parser` gives the user some getter functions, to retrieve information about processes, network and hardware devices.

4.2.2 Local InfiniBand Setup

Prior to initialization of the InfiniBuilder library, the `Parser` object is passed as a reference via the method `ibSetHostList(Parser& hostParser)` to give the necessary information about the processes. Since the system is expected to have multiple NUMA¹ nodes, the NUMA node ID corresponding to the specific InfiniBand device is retrieved.

Once these steps are completed, the `ibInit()` function can be called safely. In this function, the InfiniBand Context is set up according to the following steps [33]:

- *Opening the device*: gets and opens the InfiniBand device, which gives a *context* used to complete the next steps;
- *Allocation of a Protection Domain (PD)*: a PD protects resources from being arbitrarily accessed from a remote devices;
- *Creation of Send and Receive Completion Queues (CQs) for data exchanges*: CQs are used to store the results of non-blocking Send and Receive operations;
- *Creation of Send and Receive Completion Queues (CQs) for synchronization messages*: these CQs are used by the synchronization mech-

¹Non-Uniform Memory Access, see Appendix A.

anism that will be described in detail in Section 4.5. Data and Sync CQs are independent for performance reasons;

- *Creation of Data and Sync Queue Pairs (QPs) for every process:* QPs are objects that manages send and receive operations over the RDMA architecture. They can be considered as similar to sockets. Here, one QP for data and one QP for sync are created to keep the traffic separated.

QPs have many options to modify their behaviour. The type of QPs used in the library is *Reliable Connected* (RC, which can be compared to the equivalent of TCP in sockets). This guarantees that messages are delivered at most once, in the right sequence and without corruption. Moreover, in this configuration the requester considers a message operation completed once the responder sends an ACK signal that the message was read/written to its memory, while the responder considers a message operation completed once the message is read/written to its memory.

Once the device context is set up on the local machine, the next phase is the exchange of QP info with the remote processes in order to instantiate the InfiniBand communication. Since there is no way of transmitting data over InfiniBand before the exchange is completed, it is necessary to find another way: the InfiniBuilder library uses TCP/IP as an out-of-band (i.e. not using InfiniBand) connection².

4.2.3 Queue Pair Detailed Description

A Queue Pair is an object that combines Send and Receives Queues. There are several types of QPs that can be represented in:

- **Reliable:** messages are guaranteed to be delivered at most once, in order and without corruption;
- **Unreliable:** there is no guarantee that messages will be delivered or about the order of packets;
- **Connected:** the requester QP is associated to one and only one responder QP;
- **Unconnected:** the QP can receive and send from/to any QP.

²There is an alternative called *Internet Protocol over InfiniBand* (IPoIB) which uses the HCAs as traditional Ethernet interfaces and can be used for the configuration exchange.

Only the *Reliable Connected* QP supports all the operations, while also having a maximum message size of 2 GB (which is split in packets of MTU³ size).

In the InfiniBand verbs context, a QP is created using the `ibv_create_qp` function, passing the Protection Domain and the initialization attributes which contain the send and receive CQs pointers, the maximum number of Work Requests (WRs) for each CQ, the QP type, and other specific parameters.

When a QP is created, a pointer is returned and used for further configuration. A QP is built as a Finite State Machine, which means that it can be in exactly one of a finite number of states at any given time and the transition from one state to another is possible as a response to some inputs. In the case of QPs, state transitions are handled by the `ibv_modify_qp` function. Here is how the Finite State Machine works:

- A QP is created in the RESET state: in this state the QP cannot send and receive packets;
- The next state is the INIT state: during INIT it is possible to post receive WRs before transitioning to Ready To Receive; this helps prevent Receiver Not Ready (RNR) errors on the requester side;
- QP is then modified to Ready To Receive (RTR): in this state, the QP can be used only as a responder using the receive queues;
- The main working state is Ready To Send (RTS): in RTS a QP can have WRs posted to both send and receive queues;
- The Send Queue Drained (SQD) state is used to modify attributes in the send queue and only receive WRs will be processed;
- The Send Queue Error (SQE) state is an error state where all QP types, except RC, transit to automatically. Here, send WRs are flushed with error, while receive WRs are processed.
- The ERROR state is the last state of a QP Finite State Machine. The transition can happen automatically if a send WR in a RC QP was completed with error, a receive WR was completed with error, or by

³Maximum Transferrable Unit. It is the maximum size of a packet that can be sent without splitting. In InfiniBand the default MTU is 4096 B.

explicitly calling `ibv_modify_qp` from any state. Incoming packets will be silently dropped and no packets will be sent from this QP.

4.2.4 Out-of-Band Communication

Once the QPs are declared, each process creates a TCP socket using the port specified in the configuration file. The socket is then opened and listens for connections from the other processes. This part is multi-threaded: the receiving part is done by a second thread, while the send is done by the main thread.

When a remote process connects, it sends its info in a `struct` which holds the Local Identifier (LID), a 16 bit address assigned to end nodes by the subnet manager⁴ (equivalent to the IP address in InfiniBand), the QP number of the data QP, and the QP number of the sync QP. Once all the processes have exchanged the data, the QP Finite State Machine state is set to Ready to Send (RTS).

4.3 Basic Exchange

4.3.1 Memory Registration

The main advantage of RDMA is to operate on the remote system memory without involving the operating system and the resources. To do that, it is necessary to declare a memory region and give its address to the Host Channel Adapter. A memory region (MR) is a virtually contiguous set of buffers that has been registered. The OS provides the virtual-to-physical mapping of that region and pins the memory, which means that virtual memory swap operations are prohibited. The memory registration process is time-consuming due to pinning and address translations, thus it can impact the performance if not organised properly [34].

In the InfiniBand context, multiple non-contiguous MRs can be defined and operations on single MR can use up to the whole size of the MR (or the maximum message size if MR is larger). Each MR is identified by two keys in the InfiniBand context: `R_key` and `L_key`. In order to complete a RDMA operation using the verbs opcodes `IBV_WR_RDMA_READ` and `IBV_WR_RDMA_WRITE`,

⁴The subnet manager is a centralised entity running on the switches. It manages network discovery, device configuration, and traffic flow.

the requester must know the `R_key` and the address of the memory region of the responder before submitting the operation. By definition, the responder is **not** notified of the completion of the operation. Using the `IBV_WR_SEND`, there is no need for the requester to know the MR information of the responder, since the responder has to post a receive WR with its MR linked in order to complete the operation. This way, the responder is aware of the completion of the operation.

In the InfiniBuilder library, MRs can be registered by calling the `ibAddMR` function, which takes the pointer to the buffer and its size. This method registers the memory using `ibv_reg_mr` and creates a new entry in the internal MR look-up table, storing the MR object pointer, the buffer pointer, and its size. This routine is made thread-safe using mutual exclusion locks. MRs can be registered with access control, allowing only a set of opcodes; in the InfiniBuilder library MRs are configured to be used with all opcodes. Removing a MR is possible using the `ibRemoveMR` method, passing the buffer pointer and the size.

The `ibAddMR` method is called whenever an operation with memory is done (send, receive, RDMA). This way if the message pointer is already inside of a MR, it returns the MR pointer from the look-up table, or if it requires a new MR to be registered. This is done to simplify the use of the InfiniBuilder library and make it more similar to the socket/MPI calls.

To improve the performance, it is best practice to declare a sufficiently sized MR (or MRs if needed) during the initialization and reuse these regions during the exchanges. This prevents from the resource-intensive and time-consuming process of registering memory.

4.3.2 Send and Receive

Send and receive operations are the core of the network communication and will be discussed thoroughly in this section.

In InfiniBand, different operations are available:

- **Send:** the send operations allow to send data to a remote QP's receive queue. The receiver *must* have previously posted a receive WR specifying the receive buffer. The sender has no control over where the data will reside in the remote host;
- **Receive:** it is the corresponding operation to a send operation. The

receiving host is notified that data has been received;

- **RDMA read:** a memory region is read from the remote host. The requester has to specify the remote MR to read from and where to copy the result in local memory. RDMA read operations are conducted with no notification whatsoever to the responder;
- **RDMA write:** similar to RDMA read, but data is written to the remote host.

More operations are available, but will not be covered as they were not implemented in the InfiniBuilder library. *immediate data* is a specific way of sending small messages: this data is a 4 byte value that can be sent in the header with Send and RDMA write operations. It is not part of the message - thus not requiring a MR to work - but is part of the receive notification.

In InfiniBand verbs, a send operation is posted to the corresponding QP by using the `ibv_post_send` function. A send WR requires the following attributes:

- **wr_id:** a 32-bit unsigned integer unique identifier of the WR;
- **sg_list** and **num_sge:** a list of the elements to send, by specifying the MR L_key, the buffer pointer and the buffer size. In the InfiniBuilder library, a single Scatter-Gather Element (SGE) is used;
- **opcode:** the operation code is set to `IBV_WR_SEND` for send operations;
- **send_flags:** it describes the properties of the send request. In the InfiniBuilder library, it is set to `IBV_SEND_SIGNALED`, which means that a Work Completion (WC) will be generated when the processing of this WR has ended.

The most important part is the **wr_id**: this ID is used to check the completion of the send operation. Since operation are offloaded to the HCA, the **wr_id** is used by the host system to track the WR and the consequent Work Completion (WC). WCs are posted by the HCA in the corresponding Completion Queue (CQ) and the host system can check if there are any of them by using the `ibv_poll_cq` call, which populates an array of struct `ibv_wc`. An `ibv_wc` item contains the **wr_id** correspondent to the WR, and

the opcode, and the status of the WR, which can be completed with success or with errors.

The InfiniBuilder library offers simplified calls that hide most of the above from the end user. Sending and receiving are managed using the methods in Listing 4.2.

Listing 4.2: Send, receive, and RDMA operations

```

1 uint32_t ibRecv(char* bufPtr, uint32_t bufSize, uint32_t sender
   );
2 uint32_t ibSend(char* bufPtr, uint32_t bufSize, uint32_t dest);
3 int ibBlockRecv(char* bufPtr, uint32_t bufSize, uint32_t sender
   );
4 int ibBlockSend(char* bufPtr, uint32_t bufSize, uint32_t dest);
5 int ibBlockRDMAWrite(char* bufPtr, uint32_t bufSize, uint32_t
   dest, bool sendComplete);
6 uint32_t ibRDMAWrite(char* bufPtr, uint32_t bufSize, uint32_t
   dest);
7 int ibBlockRDMARead(char* bufPtr, uint32_t bufSize, uint32_t
   dest, bool sendComplete);
8 uint32_t ibRDMARead(char* bufPtr, uint32_t bufSize, uint32_t
   dest);
9 int ibRecvRDMAInfo(char* bufPtr, uint32_t bufSize, uint32_t src
   );
10 int ibSendRDMAInfo(char* bufPtr, uint32_t bufSize, uint32_t
   dest);

```

All of the operations simply take as arguments the pointer to the local memory buffer, the size of the buffer, and the responder rank id. Non-blocking operations (`ibRecv`, `ibSend`, `ibRDMAWrite`, `ibRDMARead`) register the MR if needed, prepare the message, and post it to the corresponding QP. They return the `wr_id` correspondent to the WR and leave the user the responsibility to check for completion. Blocking operations (with `Block` in the method name) wrap the non-blocking calls and the polling of the CQs. They return 0 on success and `-1` on error. Two additional methods (`ibRecvRDMAInfo`, `ibSendRDMAInfo`) are used to exchange MR info and have to be used before any RDMA Read/Write calls.

Checking for work completion is done using the following methods (Listing 4.3).

Listing 4.3: CQ polling methods

```

1 int ibWaitSendCQ(uint32_t wrId);

```

```

2  int ibWaitRecvCQ(uint32_t wrId);
3  int ibTestSendCQ(uint32_t wrId);
4  int ibTestRecvCQ(uint32_t wrId);
5  int ibWaitAllSends(std::vector<uint32_t>& wrs);
6  int ibWaitAllRecvs(std::vector<uint32_t>& wrs);
7  std::vector<int> ibTestAnySends(std::vector<uint32_t>& wrs);
8  std::vector<int> ibTestAnyRecvs(std::vector<uint32_t>& wrs);
9  int ibTestAllRecvs(std::vector<uint32_t>& wrs);
10 std::vector<int> ibWaitAnySends(std::vector<uint32_t>& wrs);
11 std::vector<int> ibWaitAnyRecvs(std::vector<uint32_t>& wrs);

```

All methods are based on the use of `wrId`, the first four check for a single WC, while the remaining check for multiple WCs. Methods are divided in **Send** and **Recv** since WRs are posted in different CQs. **Wait** methods are blocking, meaning they will wait until the WR is completed: **WaitAll** will wait for every WR to be completed and will return 0 on success and -1 on error, while **WaitAny** will wait for the first one and return a vector with the corresponding state of each WR (0 for success, -1 for error, 1 for not found).

Test methods are non blocking and return immediately. Return codes are 0 for success, -1 for error, and 1 for not found. Collective **Test** methods can test for any WR in a group and all WRs in a group and are non-blocking.

Listing 4.4: The InfiniBuilder Send method

```

1  uint32_t IB_verbs::IB::_ibSend(ibv_mr* mr, char* bufPtr,
    uint32_t bufSize, uint32_t dest)
2  {
3      Proc& host = _hosts.at(dest);
4      ibv_send_wr* bad_send_wr;
5      ibv_sge list;
6      memset(&list, 0, sizeof(list));
7      list.addr = (uintptr_t) bufPtr;
8      list.length = bufSize;
9      uint32_t wrId = _ibGetNextWrId(); // get random wr_id
10     int nsge = 1;
11     if (bufPtr == NULL && bufSize == 0) {
12         nsge = 0; // zero-byte msg
13         list.lkey = 0;
14     } else {
15         list.lkey = mr->lkey;
16     }
17     ibv_send_wr send_wr;
18     memset(&send_wr, 0, sizeof(send_wr));
19     send_wr.wr_id = wrId;

```

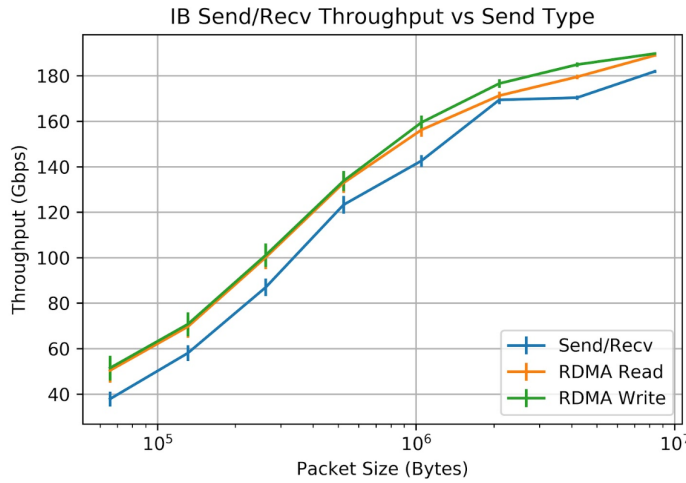


Figure 4.1: Point-to-point throughput benchmarks with different packet sizes of RDMA Read, RDMA Write and Send/Recv operations using blocking operations. The EB packet size is estimated to be ~ 3 MB. These measurements were taken between two servers of the EB cluster. Measurement procedures are explained in chapter 5.

```

20  send_wr.sg_list = &list;
21  send_wr.num_sge = nsge;
22  send_wr.opcode = IBV_WR_SEND;
23  send_wr.send_flags = IBV_SEND_SIGNALED;
24  int ret = ibv_post_send(host.qp, &send_wr, &bad_send_wr);
25  if (ret < 0) return 0;
26  return wrId;
27 }

```

The Listing 4.4 above shows the send procedure extracted from the InfiniBuilder library source code. This is what a user should do in order to post a send WR using Verbs directly, while in the InfiniBuilder library is just a single, simple line of code.

For this thesis, a first set of benchmarks was done to measure the differences between RDMA Read/Write operations and Send/Recv operations. The results in figure 4.1 show that both operations have similar performance. While the InfiniBuilder library implements both methods, the Send/Recv method has been chosen for the EB because it is easier and safer to use as it notifies both the sender and the receiver, eliminating the need of synchronization. However, synchronization is not ruled out and plays a crucial role

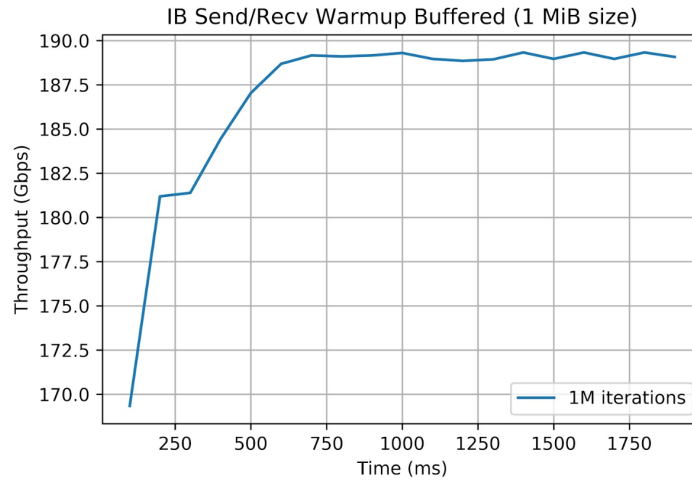


Figure 4.2: Benchmark of the initial warm-up phase using InfiniBuilder’s non-blocking Send/Recv operations. It takes less than 700 ms to achieve the full performance. Measurements were taken as a reference during the design phase of the InfiniBuilder library. Measurement procedures are explained in chapter 5.

to achieve full performance even with Send/Recv, as it will be analyzed in section 4.5.

Benchmarks of the initial warm-up phase were also taken (Figure 4.2), showing that the elapsed time before reaching the full throughput is around 700 ms, which is more than acceptable.

4.4 Barrier

In parallel computing, *barriers* play a key role in synchronising the whole application and in keeping its behaviour under control. A barrier is a point in the program execution where every process involved has to stop and wait until all the processes have reached that point: only then are all of them released and can continue with the execution. Since a barrier involves a group of processes and inherently makes some of them wait for the others, those idling processes are not doing anything, wasting computation time. This means that a barrier is always expensive and its use must be reduced to a minimum. While we cannot control where the processes are in their execution and we know for sure that they are not synchronized, it is impor-

tant to tune the barrier operation in order to waste less time and resources as possible.

In the InfiniBuilder library, two barrier algorithms are implemented and can be selected by the user. The simplest algorithm is a *centralized barrier*[35]:

- Process 0 waits for a **BARRIER** message from each process except itself;
- Each process, when reaching the barrier, sends the message to Process 0 and then waits for a release message from Process 0;
- Once Process 0 has received all **BARRIER** messages, it sends a release message to each process and then continues with its execution;
- When each process receives the release message, it continues with the execution.

While this barrier implementation is simple to understand and implement, it greatly impacts performance due to its $\mathcal{O}(N)$ time complexity, where N is the number of processes ($N \sim 800$ for the EB). Since its all-to-one and one-to-all communication patterns can create bottlenecks on the network, it may happen that processes will be idle for a longer period waiting for messages to arrive, introducing latency.

To reduce the time spent in the barrier and balance the load between processes, another algorithm has been used. This algorithm is called *decentralized tree barrier*[35] and is a hierarchical way to improve the scalability.

- All processes are divided in subgroups and synchronise with the other processes in that subgroup in a central way;
- Once all subgroups have done the synchronisation, the master process of that subgroup enters a further level of synchronisation, where it synchronises with a group of master processes;
- The algorithm proceeds until it reaches the last level, where there is only one process left;
- After the final synchronisation, the release signal goes through the whole tree, releasing all of the processes.

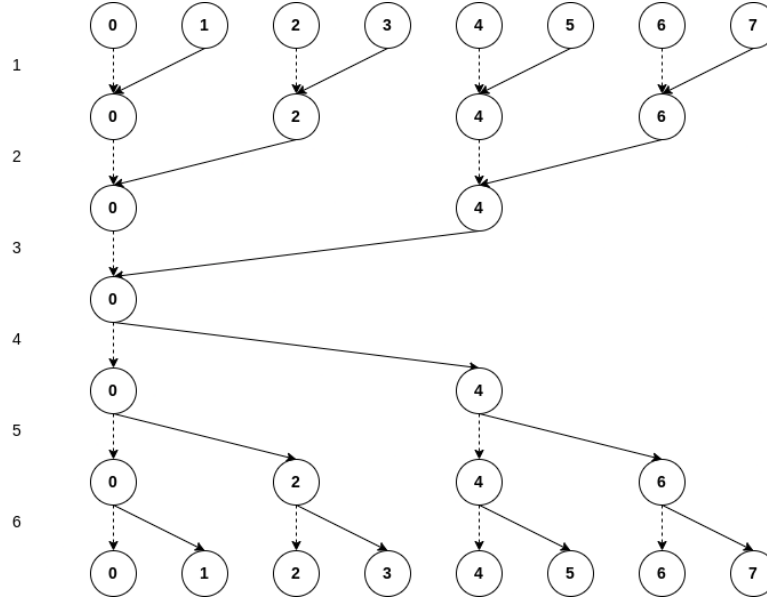


Figure 4.3: Step diagram of a tree barrier with $N = 8$ processes. Solid arrows indicate an exchange of messages over the network, whereas dashed arrows are local operations. Rows represent sequential steps in the algorithm. Process 0 is called the *root* node.

It is easier to understand the steps by looking at figure 4.3, which depicts the phases of the algorithm in detail. The figure also suggests how traffic is distributed over the network: having point-to-point communication in each step guarantees load balancing between links and processes. Moreover, the time complexity of the tree barrier is $\mathcal{O}(\log_2 N)$, since it takes $2\log_2 N$ steps to complete.

In the InfiniBuilder library, the tree barrier pattern is built during the initialisation phase. Each process calculates from which processes it has to receive messages and to which process it has to send messages. The results are the rank IDs of the processes stored in a vector and the send rank ID stored in a separate variable. During the running phase, each process reads from these look-up tables to set up the correspondent send and receive operations.

4.5 Synchronisation

During the first benchmarks using the InfiniBuilder library in the EB software, performance issues were observed. With $N = 10$ nodes, the MPI

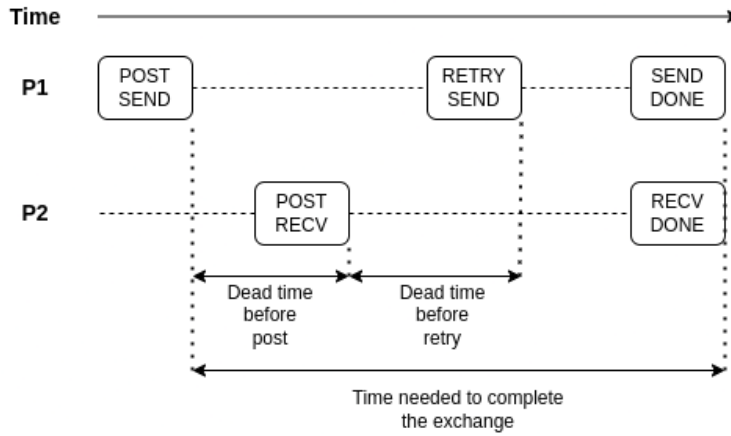


Figure 4.4: Time diagram showing the synchronization issue observed with the first version of the EB developed with the InfiniBuilder library. It shows how important is dead time over performance.

version was used as a reference, measuring an average of 165 Gb/s per node. The IB version was under-performing, reaching only 47 Gb/s using simple `ibSend` and `ibRecv` operations.

This lead to further investigations to track down the issue. Extensive measurements were made over all of the methods of the InfiniBuilder library, paying particular attention to the `WaitAll` collectives.

In the benchmarks, the polling of CQs using `Test` methods took 50 to 120 ns to complete, depending on the presence of the message in the completion table. To implement `WaitAll` procedures, a `while` cycle was used to continuously check the CQs with `Test` methods. The number of iterations was measured and it was found that the average value was around 5k iterations, but with spikes of up to 14k iterations.

This was a clear indication that the time taken to complete the Send/Recv operation was higher than expected. Two hypotheses were made: it could either be a synchronisation issue or a CQ problem. Further analyses suggested that it was indeed a synchronisation issue.

When a process tries to send data, it posts a send WR. The send WR is processed and needs the presence of a receive WR on the remote machine to proceed. If the receive WR is not present, the send WR will be posted again as many times as the library specifies and each time it will wait some time before retrying. The time spent retrying is dead time, since the process is waiting for the send WR to complete. If the time gets larger, it could lead

to timeout issues killing the whole EB software. A time diagram of the issue is shown in figure 4.4.

To test the impact of this behaviour, the `MPI_Barrier` function was used to synchronise the exchanges. While this adds more constraint than needed since it synchronise all nodes, this improved performance, as reported in the measured throughput of 130 Gb/s with $N = 10$ nodes. With this knowledge, it was time to build a synchronisation mechanism in the InfiniBuilder library.

4.5.1 sync Procedures

The synchronisation in the InfiniBuilder library has been designed to be fast, flexible, and independent from the data exchange. To do so, a separate set of QP and their relative CQs is used. This way the data exchange is not affected by sync messages, and sync is not slowed down by data transfers. Sync QPs use buffered requests to overcome sync issues: a number of receive WRs is always posted on each QP, so send WRs are consumed immediately. Since sync messages are used for different purposes at the same time, there should be a way of filtering them. This is done using *immediate data*. As illustrated in section 4.3.2, this is a 4-byte message stored in the header, which can be read directly in the WC object. Thanks to this, no MR is needed for sync purposes and zero-byte messages are used, reducing furthermore the time needed to process synchronisation procedures. To make it all completely independent, the sync polling is done on a concurrent thread.

Sync methods are the same used in data exchange, except they present the `Sync` keyword in the methods' name. InfiniBuilder library has predefined sync messages under the `immediate_t` enumeration which are:

- `NO_IMM_DATA`: for simple sync messages and debug purposes;
- `BARRIER`: messages used for barrier synchronisation only;
- `ONE_TO_ONE`: synchronisation of send, receive, and other point-to-point messages;
- `RDMA_OP_COMPLETE`: used to signal the completion of an RDMA operation to the responder.

The InfiniBuilder library does not implement directly the sync procedures in its send and receive methods, but rather prefers a more flexible approach leaving the use of sync to the end user.

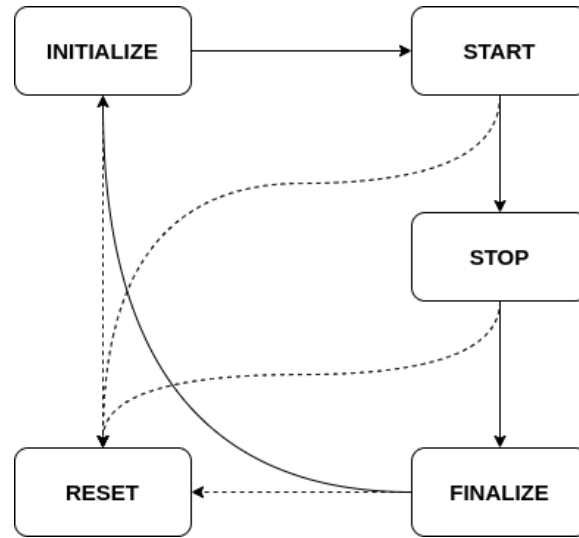


Figure 4.5: Finite State Machine of the Transport unit.

4.6 Interfacing with the EB

With all the ingredients present in the library, it was time to test it in a real benchmark by including it in the EB software. In the first version, only the one-to-one send and receive operations were included, while the rest was still managed by MPI. After six releases, all procedures were migrated to the InfiniBuilder library including barriers and scatter/gather functions.

The EB software comes with two classes responsible of network communication: `Transport_unit` and `Parallel_comm`. The `Transport_unit` class is the high-level network stack manager and is built as a Finite State Machine. The `Parallel_comm` class acts as a middle layer customizing the behaviour of the InfiniBuilder library, such as implementing collective communication functions and signaled operations.

4.6.1 `Transport_unit` Class

The class `Transport_unit` is a Finite State Machine composed of five states (Figure 4.5):

- **INITIALIZE**: the host configuration file is read, rank ids are checked and stored, the number of sources is initialised and the linear shifting pattern is computed;

- **START**: during the start, the `Parallel_comm` object is initialised, which calls the `ibInit` method from the InfiniBuilder library;
- **STOP**: in the stop procedure, the `Parallel_comm` object is destructed, which deallocates and reset the resources;
- **FINALIZE**: during finalize, the configuration parser is destructed and reset;
- **CANCEL**: the cancel state is used whenever the system needs to be reset, stopping all pending operations.

4.6.2 `Parallel_comm` Class

The `Parallel_comm` class is a middle layer between the EB software and the InfiniBuilder library. Synchronisation procedures are built for each network operation and are hidden from the end user. As shown in figure 4.6, a synchronised operation is done in five steps:

1. The receiver posts a Receive WR and then sends a SYNC message to the sender;
2. The sender waits for the SYNC message from the receiver;
3. The two processes are synchronised and can move forward;
4. The Send WR is posted by the sender and is consumed immediately, since the corresponding Receive WR was posted in step 1;
5. Sender and receiver exchange data and complete the operation.

The SYNC operation guarantees that the Receive WR is always posted before the Send WR, acting as a barrier between the two processes. It is possible that step 1 and 2 happen in different order, but after the synchronisation they follow the correct order. It is also clear how important it is for SYNC procedures to be fast and independent, because the traffic could greatly impact performance.

4.6.3 Collective Functions

The EB software makes use of *scatter*, *broadcast*, and *gather* operations to exchange event fragments with other processes. Scatter and broadcast are

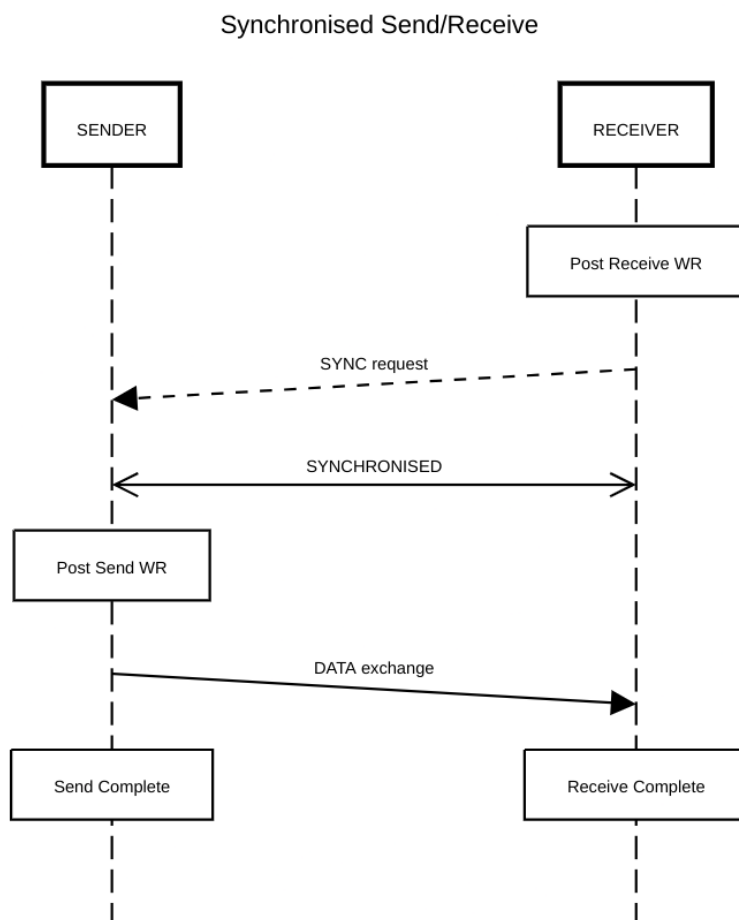


Figure 4.6: Sequence diagram of a Synchronised Send/Receive operation, as implemented in the `Parallel_comm` class.

used on the sender side together with the gather operation on the receiver side. Collective functions are implemented in the `Parallel_comm` class using InfiniBuilder methods.

The Scatter operation is implemented using a for loop in which each destination is synced and then the send WR is posted. Send operations are non-blocking and get checked for completion after the for loop is completed, so they can be consumed in parallel.

While being identical to the Scatter function, the Broadcast operation it sends the same array of data to different destinations.

In the Gather operation, receive WRs are posted for each source and then sync is sent to all sources. Gather operation is available in both blocking and non-blocking form: the blocking gather waits for completion of all receive operations, while the non-blocking gather returns an array containing receive WRs' ids that can be checked by the user for completion.

Chapter 5

Measurements and Benchmarks

5.1 Experimental Setup

This chapter describes the measurements taken using the actual servers of the Event Builder (EB) cluster. The first set of measurements was performed on a subset of the EB cluster made of 10 nodes. These tests were used to evaluate the InfiniBuilder performance during the design phase and to compare the InfiniBuilder-based EB to the MPI version. The scalability of the implementation was checked later performing measurements on the complete EB cluster.

5.1.1 Hardware

The computer cluster is composed of 170 servers equipped with dual AMD Epyc 7502 32 cores/64 threads CPUs with a base clock of 2.5 GHz and 512 GB DDR4-ECC memory. Each server hosts two NVIDIA Connect-x 6 InfiniBand HDR Host Channel Adapters, three PCIe40 readout cards, and one NVIDIA RTX A5000 GPU. Each InfiniBand card is connected to a separate CPU so as to avoid crossing over the interconnection for network exchanges. Each CPU is considered a separate NUMA node. In the following tests, only the InfiniBand cards and the readout cards were used.

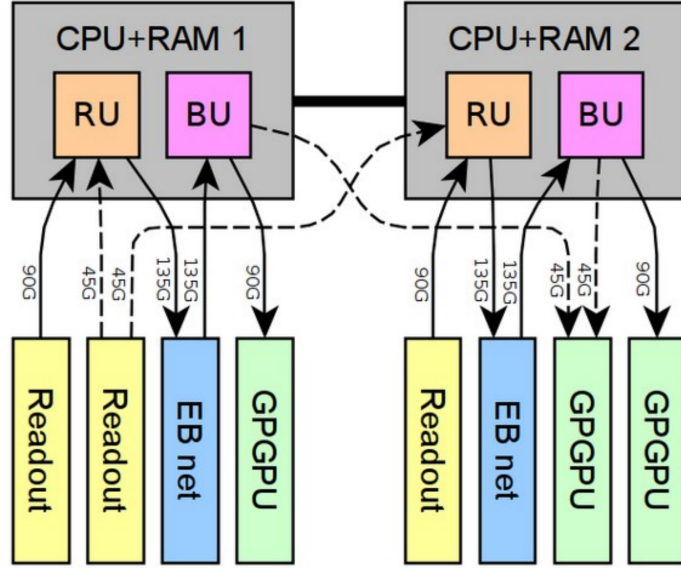


Figure 5.1: Hardware layout of an Event Builder server. Each InfiniBand card (EB net in the figure) is connected directly to the PCIe bus of the respective CPU. This prevents the crossing of the interconnection between CPUs. Original image from LHCb.

5.1.2 Software

Servers are equipped with Linux CentOS 7.9.2009 distribution and kernel 3.10. The NVIDIA version of Open Fabrics Enterprise Distribution (OFED) drivers (release 5.1) is installed to run the InfiniBand network, enabling configuration, management, and more importantly InfiniBand verbs programming.

5.1.3 Measurement of Throughput

Throughput is the ratio between the size of data exchanged and the time spent to exchange it. The data size is known and set by the user in the custom programs. The time spent to exchange the data is more difficult to measure. Since exchanging data involves two computers, it is not possible to start a timer on the sender node and stop it on the receiver node. Computer clocks are not synchronised, so it is not possible to make reliable measurements across the two. Measurements have to be carried on a single server using its clock, counting the time passed between the post of the respective

WR and its completion.

Time measurements have to be reliable and computer clocks have different levels of reliability that can be selected by the user. The programs written for this project use the POSIX `<time.h>` library, which extends the ISO C library. This library offers the `CLOCK_MONOTONIC` identifier, which refers to a system-wide monotonic clock, and it is not possible to set using `clock_gettime` and cannot have negative jumps, thus being more stable than the other available clocks. Since it is system-wide, its value is not CPU, process, or thread specific. Time values are registered in a `struct timespec` with ns precision via the `clock_gettime` method.

5.2 Point-to-Point Communication

The aim of these reference measurements was to evaluate the performance of the interconnection before writing the custom implementation.

The reference measurements for point-to-point communication were taken using the benchmark tools provided with OFED. These benchmarks implement the opcodes `IBV_POST_SEND`, `IBV_RDMA_READ`, and `IBV_RDMA_WRITE` and offer the setting of most of the parameters available in InfiniBand verbs to the user. The tools used are named `ib_send_bw` and `ib_write_bw` respectively for Send and RDMA operations. The parameters used in both tools were:

- Variable message size from 10 KiB to 100 MiB¹;
- Variable number of QPs (1 or 2);
- Unidirectional transfer, since in bidirectional mode the tool was found to be not reliable;
- Maximum Transferrable Unit (MTU) set to 4096 B;
- Reliably Connected (RC) connection type;
- Number of messages exchanged set to 1000.

¹Units such as KiB and MiB are part of the binary unit system and defined by the IEC60027-2 international standard. A kilobinary-byte (KiB) corresponds to 1024 B (2^{10}), 1 MiB = 1024 KiB = $(2^{10})^2$ B.

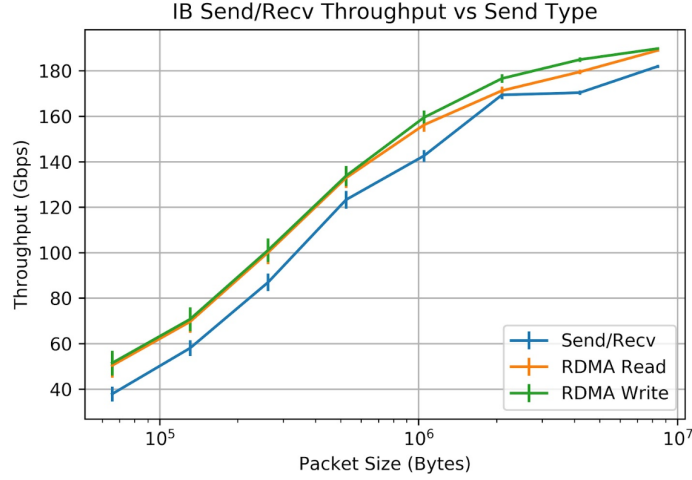


Figure 5.2: Point-to-point throughput benchmarks with different packet sizes of RDMA Read, RDMA Write, and Send/Recv operations using blocking operations. The packet size is estimated to be $\mathcal{O}(1)$ MB considering the event size times the packing factor. These measurements were taken between two servers of the EB cluster as a reference during the design phase of the InfiniBuilder library.

Moreover, tests were carried on binding to the NUMA node of the HCA. During the first development phase, custom test programs were written to understand the key steps in the communication and the results were compared with the reference values until they matched. Most of the preliminary tests were also carried on to evaluate which parameters played a key role in performance: for convenience the plots (Figures 5.2, 5.3) of these preliminary tests are also reported in section 4.3.2.

Throughput was measured with a sampling rate of 300 ms and a total of 100k iterations for each packet size. The results presented in the figures are the corresponding average and the error bars report the standard error calculated as $SE = \sigma/\sqrt{N}$, with standard deviation σ and the number of measurements for each packet size N .

By looking at different tests it was evident how important the use of buffering is. This is because having a set of Send WRs always posted keeps the device busy, reducing the dead time between successive postings and improving throughput stability. In Figure 5.4, buffered requests show an improvement of $\sim 30\%$ with 1 MiB size messages. This chart also shows how the dead time is higher for smaller messages, since larger messages need

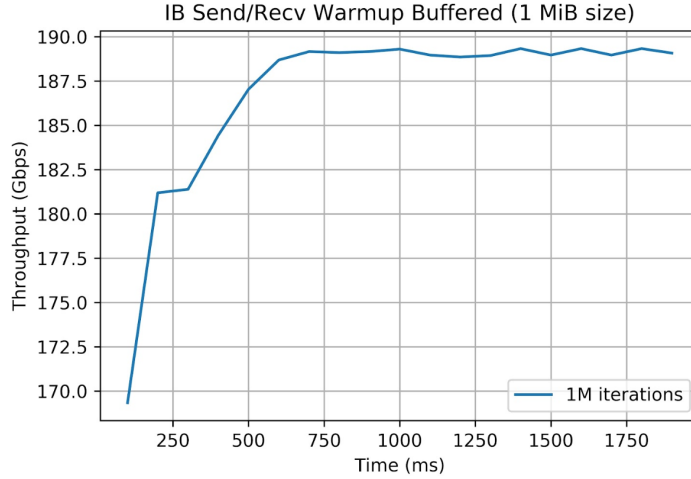


Figure 5.3: Benchmark of the initial warm-up phase using InfiniBuilder’s non-blocking Send/Recv operations. It takes less than 700 ms to achieve the full performance. Measurements were taken as a reference during the design phase of the InfiniBuilder library.

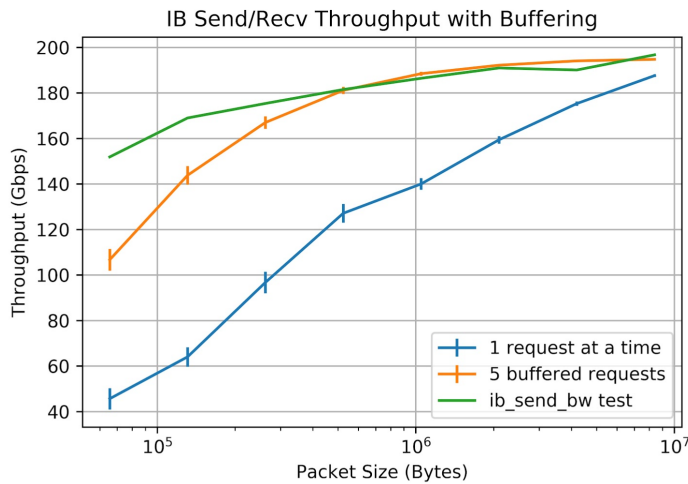


Figure 5.4: Comparison of the use of buffered requests to single blocking requests. The measurements taken with custom programs are presented together with the reference from `ib_send_bw`. For messages larger than 500 KiB, the custom program using buffered requests has similar performance to the reference.

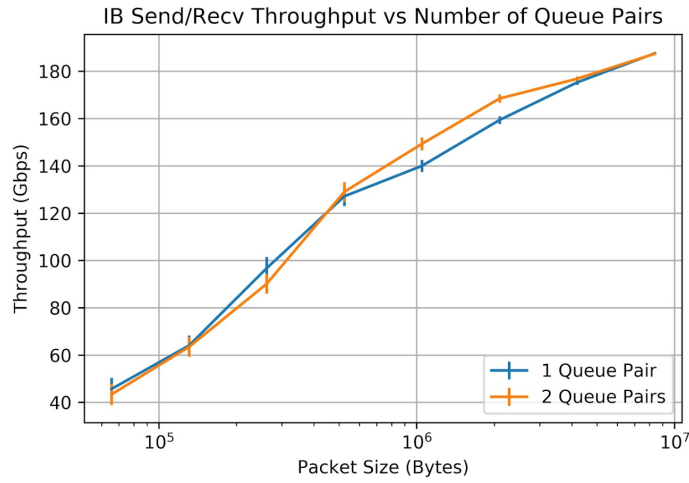


Figure 5.5: Comparison of the throughput measurements of blocking request using one and two QPs. There is no significant performance difference between the two, while the use of two QPs complicates the program. This result was at the base of the decision of using a single data QP in the InfiniBuilder library.

more time to be transferred. Moreover, the comparison with `ib_send_bw` shows that the custom program has similar performance when using buffered requests and messages larger than 500 KiB.

Another parameter to be evaluated is the number of QPs to use for data transfers. Adding QPs should help keep the device busy even with small size messages, improving throughput. Tests were taken using one and two QPs, one single request at a time. The posting on two QPs uses a round-robin simple algorithm, distributing the WRs alternately. The results in Figure 5.5 show that there is no significant improvement in using a multiple QPs, while the algorithm that distributes the WRs over the QPs makes the functions more complex and more prone to errors.

From these preliminary tests it was established that the InfiniBuilder would use a single QP for data transfers and its recommended use would include buffered requests and `IBV_POST_SEND` operations.

5.3 Event Builder

The integration of the InfiniBuilder library in the Event Builder was gradual, replacing the MPI calls in the software.

- In the first phase, point-to-point communication in the linear shift step was replaced with InfiniBuilder send and receive operations;
- In the second phase, barrier synchronisation was moved from MPI to InfiniBuilder library;
- In the last phase, collective methods were replaced with scatter, broadcast, and gather operations implemented in the `Parallel_comm` library.

As already explained in section 4.5, the InfiniBuilder library, initially, severely under-performed compared to the MPI version due to sync issues. Results of these first tests reported 47 Gbps of throughput per node, while the MPI version reported 165 Gbps per node: 3.5 times higher.

Measurements taken with the synchronisation mechanism in place are reported in the next subsection.

5.3.1 Small Batch Size

The deployment of the new Event Builder software was done on a smaller number of servers to test the integration and simplify debugging before scaling up to the full size cluster. Measurements were taken on a cluster of 10 servers with a batch size varying between 4 and 16 nodes. These tests involved only the EB process and used dummy data in input without using the PCIe40 cards. Dummy data are packets with no specific physics content.

In figure 5.6, the throughput measurements on the test cluster showed that the InfiniBuilder version of the Event Builder software is outperforming the reference MPI version with a 14% increase with 4 nodes and a 9% increase with 16 nodes. These positive results demonstrate that the InfiniBuilder library achieves the requirements and can be used as the network communication library for the Event Builder software, replacing MPI completely. The measurements presented in the next subsections were done using the InfiniBuilder version of the EB software and the MPI version was not developed further since it was dismissed.

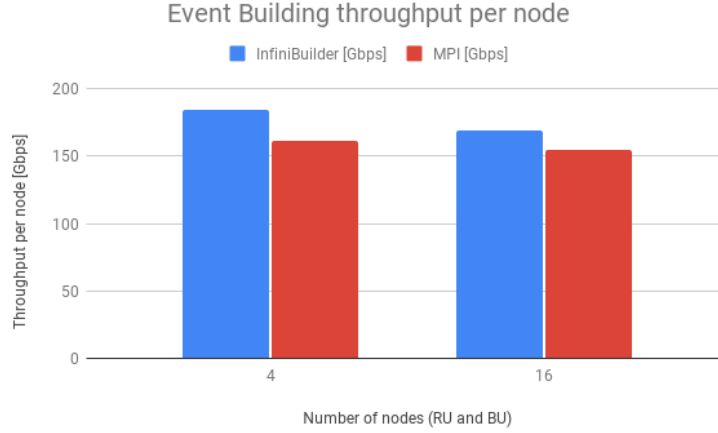


Figure 5.6: This histogram shows the throughput measurements of the MPI version and the InfiniBuilder version of the Event Builder with 4 and 16 nodes. The InfiniBuilder version, in blue, has 14% higher throughput with 4 nodes and a 9% increase with 16 nodes. Measurements were taken using dummy data as input and using the test cluster.

5.3.2 Full Size Scalability Tests

The next step in testing the EB software was to evaluate its performance and reliability when the number of nodes involved increases. The EB software was deployed on the full cluster with dummy data as input. Measurements were taken with a variable number of nodes N up to 300.

As shown in Figure 5.7, the system is scaling well with the increase of nodes involved. The event rate is over 30 MHz for each value of N , achieving the set requirements. The aggregated total throughput of the system is over 30 Tb/s. The event rate slightly decreases with the increase of nodes and this behaviour is related to the presence of ghost nodes, which are nodes used to balance the linear shifting scheduling, but do not contribute to the Event Building. The throughput per node can be calculated considering the slope of the throughput series and its average value is 133 Gb/s. This result, while being less than the values shown in Figure 5.6, can be explained with the increased number of nodes, which impacts collective functions, and the presence of ghost nodes. Moreover, the EB will settle to the lowest throughput node, due to its synchronous design.

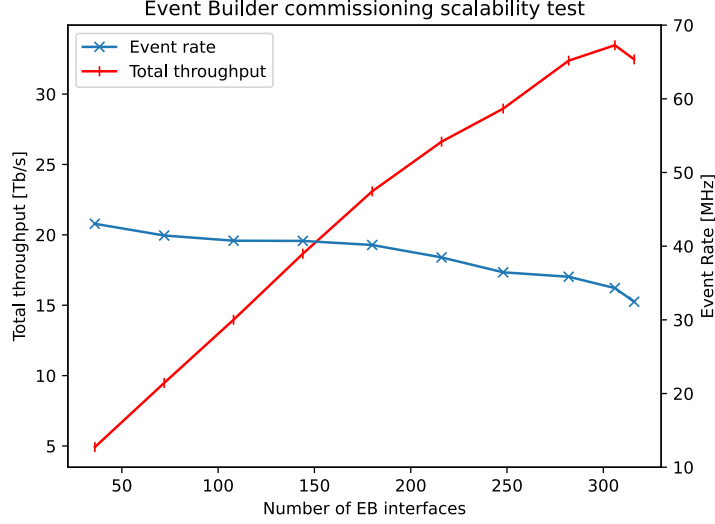


Figure 5.7: This plot shows the aggregated throughput (in red) and the event rate (in blue) as a function of the number of nodes in the EB cluster. These tests show that the system is achieving the 30 MHz event rate requirement over the full cluster.

5.3.3 Full Size Test

In this subsection, the results of the test at full size of the EB cluster were reported. The sub-detector groups used were: ECAL, HCAL, PLUME, RICH1, RICH2, SciFi side A, MUON side A and C, UT side C, and Test Detector (a partition of UT side A). In this test, the data in input was generated by a special Tell40 firmware and the number of nodes used was $N = 220$. This test involved the complete Online chain in order to test each step in the data flow. During the test, the profiling was run for 800 s. Small oscillations in measurements could be tracked down to delays between the reading of counters in processes and their reception by the ECS. The following plots report the measurements as a function of the elapsed time to show the stability and reliability of the system.

Event rate, total throughput, dead time Figure 5.8 shows the event rate measured in input and output of the EB BUs. Figure 5.9 shows the aggregated throughput in input and output of the EB Builder Units (BUs), which has the same behaviour of the event rate. The throughput is lower

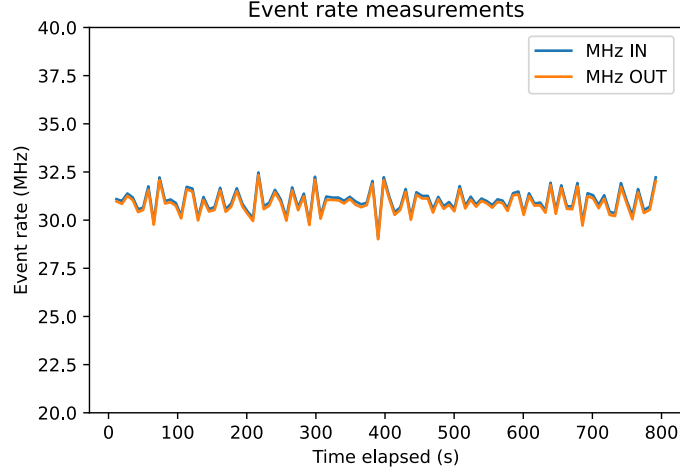


Figure 5.8: Measured event rate in input (in blue) and in output (in orange) of the EB Builder Units (BUs). The slight difference between input and output rate shows the HLT dead time. The horizontal axis represent the time elapsed from the start of the test. Small oscillations in measurements could be tracked down to delays between the reading of counters in processes and their reception by the ECS

than the theoretical 32 Tb/s because some sub-detectors were not capable of generating sufficient data, since the test was carried on in conditions of no real collisions. This limits the whole EB software, as it is synchronous, and some links are not running at their full potential.

In both plots there is a slight difference between input and output. This difference is related to the dead time of the High Level Trigger. Dead time is the fraction of time in which valid events cannot be processed due to several reasons:

$$R_{\text{dead}} = \frac{f_{\text{in}} - f_{\text{out}}}{f_{\text{in}}} \quad (5.1)$$

where R_{dead} is the dead time, f_{in} the input event rate, and f_{out} the output event rate. Figure 5.10 shows the dead time during the test. Dead time is consistent with the event rate, reporting an average value of $R_{\text{dead}} = 0.46\%$ at 30 MHz.

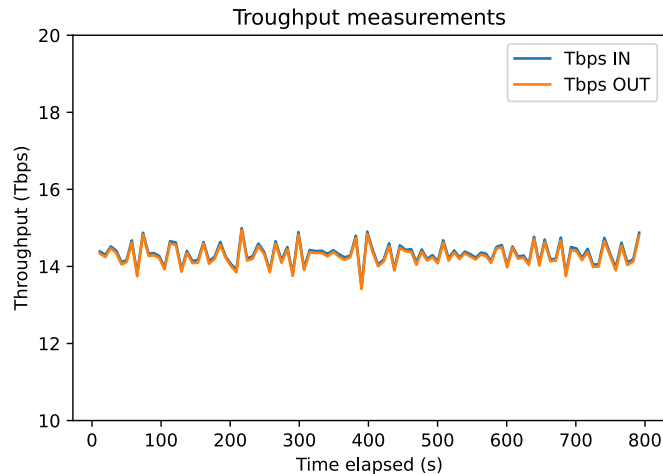


Figure 5.9: Measured aggregated throughput in input (in blue) and in output (in orange) of the EB software. Input and output counters of the Builder Units (BUs) are used. The horizontal axis represent the time elapsed from the start of the test. Small oscillations in measurements could be tracked down to delays between the reading of counters in processes and their reception by the ECS

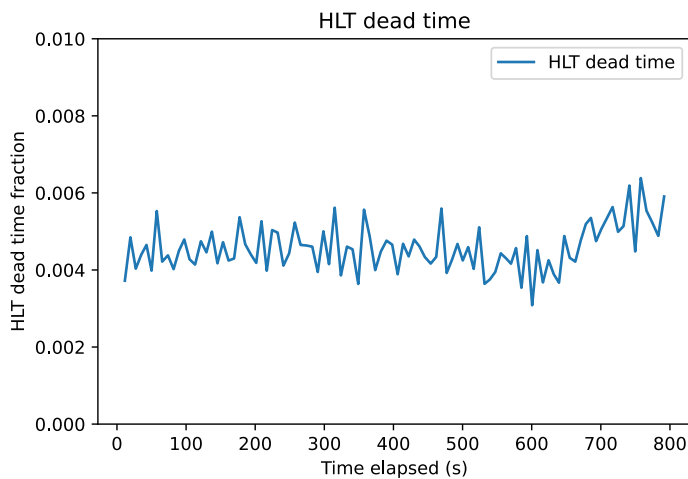


Figure 5.10: High Level Trigger (HLT) dead time trend during the test. The horizontal axis represent the time elapsed from the start of the test. Small oscillations in measurements could be tracked down to delays between the reading of counters in processes and their reception by the ECS

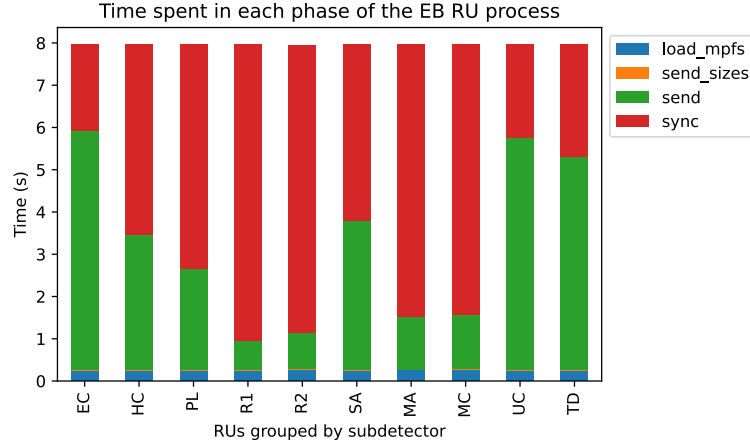


Figure 5.11: Average time distribution in a EB RU (Readout Unit) cycle, divided by sub-detector group. 96.7% of the total time is spent in the linear shifting procedure (**send** and **sync**). EC is the Electromagnetic Calorimeter, HC the Hadronic Calorimeter, PL is the PLUME detector, R1 and R2 are the RICH detectors, and SA is the side A of SciFi, MA and MC are the two sides of the MUON, UC is the UT side C, and TD is the test detector.

Time counters The EB software exports several time counters that measure different sections of the program. It is possible to isolate and benchmark single operations using these counters.

Figure 5.11 shows how time is distributed in a EB RU (Readout Unit) cycle. The columns are the different sub-detectors and show the time cost of each operation. The plot shows the key role of the barrier (**sync** in the plot), which guarantees that all sub-detector groups are synchronised regardless of the data transferred. The time spent sending data is the **send** series in the plot. The sum **sync** and **send** is the time spent during the linear shifting procedure. Processes with less data to send will wait longer in the barrier than processes with more data.

Figure 5.12 shows the time distribution in a EB BU (Builder Unit) cycle. The columns are the different sub-detectors and are divided by program sections. This plot highlights that all the different sections take the same time since they are synchronised. The time spent in barrier is significant since BU processes synchronise with RU processes and RU operations take more time to complete.

Both plots highlight the cost of communication, which is 96.7% of the total time of an RU cycle and 99.6% of a BU cycle.

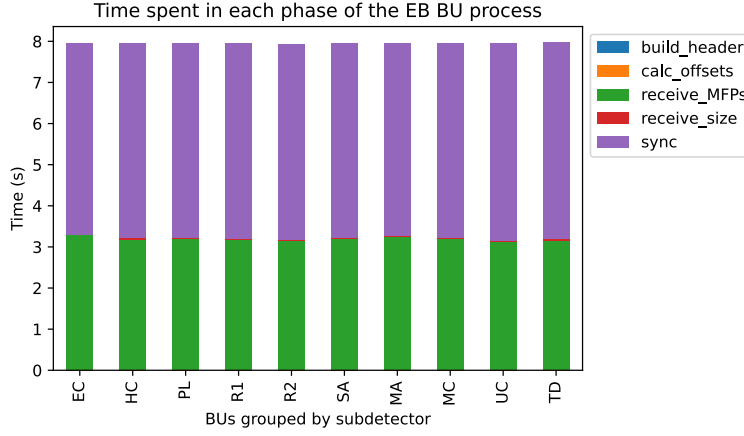


Figure 5.12: Average time distribution in a EB BU (Builder Unit) cycle, divided by sub-detector. 99.6% of the total time is spent in the linear shifting procedure (**send** and **sync**). EC is the Electromagnetic Calorimeter, HC the Hadronic Calorimeter, PL is the PLUME detector, R1 and R2 are the RICH detectors, and SA is the side A of SciFi, MA and MC are the two sides of the MUON, UC is the UT side C, and TD is the test detector.

5.4 Conclusions

The results reported in this thesis showed that the current Event Builder software is capable of achieving the requirements using the InfiniBuilder library. The InfiniBuilder library resolved the issues found with MPI and was capable of outperforming the MPI version of the EB software. The design of a custom network library turned out to be successful and the current production version of the EB software implements it.

The InfiniBuilder-based EB is now running in the Online system and is currently acquiring data from Run 3 collisions.

5.4.1 Future Developments

Based on the measurements, communication is the most expensive part of the EB software. More efficient barrier algorithms that could take advantage of the network topology could be tested and developed. Since the barrier is a building block of the linear shifting scheduling, this will reduce the time needed significantly, resulting in a performance increase.

Appendix A

Principles of Parallel Computing

There is an ongoing demand for higher computational power than what it is available with single computers using current technologies. Some workloads need to be completed in a definite period (such as weather forecast) or use more resources (such as neural networks datasets). This leads to the use of multiple computer together, aggregating their computer power to achieve the problems' requisites. The idea is that if a single computer takes T time to elaborate the task, N computers should be able to do it in T/N time. This theoretical value is only reachable if the program is completely parallel, but in reality there is always a component that cannot be parallelised. This is taken care in the definition of Amdahl's law, which says that the maximum *speed-up* is:

$$S(N, F) = \frac{1}{(1 - F) + \frac{F}{N}} \quad (\text{A.1})$$

where F is the fraction of the code that can be parallelised. When $N \rightarrow \infty$, the speed-up tends to $1/(1 - F)$.

There are two types of parallel computers: *shared memory* multiprocessors and *distributed memory* multi-computers. A simple computer can be described as a logical processor that executes the program from a system memory. A unique address is associated to each memory location. In a shared memory system, multiple processors and memories are connected together via an interconnection network, such that each processor can access any memory. This layout permits to have CPUs that are capable of execut-

ing multiple processes in parallel in a single system. This approach is used in most of the CPUs available on the market.

In a distributed memory system, simple computers are connected through an interconnection network which can be direct or switch-based. In data centers, this is the standard configuration, as it is cheaper and more scalable.

In current systems, a combination of the two is always used. Each computer is capable of handling multiple processes, while also being connected to an interconnection network.

A.1 Distributed Shared Memory and Non Uniform Memory Access

In high performance servers, there are multiple CPUs on the same machine, which are connected via a dedicated high speed interconnection. This design can be defined as *distributed shared memory*: each CPU has its own memory and can be thought of as a simple computer, but the system sees a single combined memory with a single address space, thus enabling shared memory programming techniques. Even though this simplifies the programmer's work, it is important to be aware of the limitations of this design to achieve the best performance.

From a performance point of view, it is faster for a CPU to access its own local memory than going over the interconnection network and fetch data from another CPU's memory. This aspect is called Non-Uniform Memory Access (NUMA). Each set of processes with the same access speeds (i.e. on the same CPU) is grouped under a NUMA node. Executing a program specifying a particular NUMA node can lead to large improvements in performance. Moreover, CPUs have also other devices embedded such as PCIe controllers. They are dependent on NUMA nodes as well, since devices are connected to a specific CPU in the system.

This is why the InfiniBuilder library suggests the selection of the NUMA node associated with the InfiniBand PCIe card.

A.2 Programming models

Computer are classified based upon instruction streams and data streams:

- **Single instruction stream, single data stream (SISD)** computers use a single stream of instruction generated from the program. The instructions operate upon a single stream of data items;
- **Multiple instruction stream, multiple data stream (MIMD)** computers are the designed used in general-purpose multiprocessor systems, where each processor has a separate program and one instruction stream is generated from each program for each processor. Each instruction operates upon different data. Both shared memory and message passing systems are in this classification;
- **Single nstruction stream, multiple data stream (SIMD)** computers are specially designed to take the instruction from the program and broadcast it to more than one processor. Each processor executes the same instruction in synchronism, but using different data. These computers have important applications where large arrays and vectors of data are used.

Extending the classification further, two types of structures can be defined in the MIMD computers. The **Multiple Program, Multiple Data (MPMD)** structure indicates that each processor will have its own program to execute, with its own stream of data. The **Single Program, Multiple Data (SPMD)** structure provides a single source program and each processor executes a personal copy of it, indipendently and not in synchronism. The source program can be constructed such that parts of the program are executed by certain computers and not others upon the identity of the computer.

The SPMD programming model is the standard way of creating programs for message passing (MPI). Control statements in the program select different part for each processor to executed. All executables started together, so the creation of processes is static with all the processes starting and ending together. In listing A.1, a SPMD pseudo-code is presented: master or slave role is selected at runtime using the rank number of the process.

The MPMD programming model provides separate programs for each processor. One processor executes the master process. Other processes are

started from within the master process: this is called dynamic process creation and is the model used in shared memory applications using *threads*.

A.3 Message Passing

Message Passing is the term used for two processes which exchange data via messages over an interconnection network. Message Passing libraries provide the user two fundamental mechanisms: a method of creating separate processes for execution on different computers and a method of sending and receiving messages. In MPI, the creation of separate processes is static and processes change their behaviour using control statements in the program. MPI uses communicators to define the scope of a communication operation. Processes have ranks, which are numeric identifiers, associated with a communicator. All processes are enrolled in a *universe* called `MPI_COMM_WORLD`, and each process is given a unique rank, a number for 0 to $N - 1$, with N processes. Other communicators can be established by the user for groups of processes. Only the processes in the communicator are allowed to communicate between themselves. This permits to separate communication domains of different part of code from that of a user program.

Listing A.1: SPMD pseudo-code

```
1 main (int argc, char *argv[]){
2     MPI_Init(&argc, &argv);
3     .
4     .
5     MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /*find process rank
6     */
7     if (myrank == 0)
8         master();
9     else
10        slave();
11    .
12    MPI_Finalize();
13 }
```

The sending and receiving of messages is more complicated. In a basic point-to-point send and receive exchange, processes can exchange data using *send* and *receive* library calls. This exchange can be either synchronous or asynchronous.

In synchronous mode, the *send* routine waits until the complete message can be accepted by the receiving process before sending the message, while the *receive* routine waits until the message it is expecting arrives. As the mode imply, these routines perform two actions: they transfer data and synchronise processes.

A common way of implementing a synchronous exchange is using a 3-way protocol:

- Process 1 sends a request to send signal to process 2;
- Process 2 answers with an acknowledgement (ACK) signal;
- Process 1 receives the ACK signal and sends the message.

This way, process 1 waits until the ACK signal is received and process 2 waits until it receives the request to send. At the end of the exchange, both processes are synchronised.

In asynchronous mode, routines do not wait for actions to complete before returning and usually require local buffers to store messages. They do not synchronise processes, allowing them to proceed sooner.

In MPI routines are divided in *blocking* and *non-blocking*. *Blocking* operations return after their local actions complete, though the message transfer may not have been completed. For example, a *blocking send* returns when the message buffer is safe to be reused, but this does not guarantee the transfer completion.

Non-blocking operations return immediately and assume that data buffers used for transfer are not modified by subsequent statements before being used for transfer. It is left to the programmer to ensure this.

It is important to not mistake blocking and synchronous operations: blocking only ensures local completion, while synchronous ensures completion also on the remote side.

A.4 Threads

A common technique of programming shared memory systems is the use of *threads*. A thread is a component of a process with all the information a CPU needs to execute a stream of instructions. It shares the same memory space and global variable between routines.

Threads programming is done using libraries such as `Pthreads` (POSIX standard) and `std::thread` in C++. Threads are created using the relative routine in the main process, which points to a method that has to be executed in parallel, they can return when completed and their return value is received using the `join` method in the main process. Join is not needed when it is not necessary to know if the thread created terminates. Those threads are called *detached threads* and when they terminate, their resources are released and they get destroyed. An important point in multi-threaded programming is the statement execution order. In the case of a single processor we do not have to worry since typically threads are executed until blocked, in a sequential way. However, in multi processor systems, instructions of threads are interleaved in time. This does not guarantee that the ordering of instructions is the same as sequential order, which could lead to modification of resources shared between instructions, producing different results than expected. The *thread safety* concept describes objects that can be called from multiple threads simultaneously and always follow the expected behaviour. Routines that access shared data require careful control to be made thread safe.

Consider two processes each of which adds one to a shared variable x . A process read the content of x , computes $x + 1$, and then writes the result back to the location of x . With more than one process, undefined behaviour is expected since there is a conflict in accessing the shared variable.

A mechanism for ensuring that only one process accesses a particular resource at a time is to define sections of code called *critical sections* involving the shared resource and assures that there section cannot be executed simultaneously. This mechanism is known as *mutual exclusion*, **mutex**.

The simplest mechanism to ensure mutual exclusion is the *lock*. A lock is a 1-bit variable that stores a 1 to indicate that a process has entered the critical section and a 0 to indicate that no process is in the critical section. Before acquiring the lock, the thread checks its value and then proceeds if it is 0 or waits until it becomes 0.

There are different techniques used in thread safety such as semaphores, but they will not be discussed in this chapter.

A.5 Side Note on Copies

Virtual memory The most common way to manage physical memory in computing is to use the technique of *virtual memory*. The operating system maps memory addresses used by a program, which are called virtual memory addresses, into physical addresses in computer memory. Programs see the memory as a contiguous address space. The operating system manages virtual address spaces and the assignment of physical memory to virtual memory. The use of virtual memory has different benefits, such as increased security thanks to the isolation of address spaces, freeing the applications from managing shared memory space, and use the memory shared used by libraries between processes.

Kernel and user spaces A modern operating system usually divides virtual memory into *user space* and *kernel space*. This separation provides memory protection and hardware protection from malicious or unexpected software behaviour. The user space is the memory area dedicated to the applications and to some drivers, while the kernel space is strictly reserved for running the operating system kernel, kernel extensions, and most device drivers.

Traditional TCP/IP stack behaviour The TCP/IP stack in the Linux kernel is taken as an example to explain the role of memory spaces and copies.

In user space, the abstraction of network communication is the *socket*. The socket is the interface to interact with the kernel-based TCP/IP stack. The network communication is handled via **send/recv** calls in the user space and their related message buffers are also allocated in user space. The kernel-based stack takes care of the required operations to transmit the message over the network and requires its own buffers this time allocated in kernel space. The device driver which manages the network interface and allocates, in turn, buffers.

The data flow of a message in traditional mode follows:

- In user space, the application allocates and populates a buffer;
- The application calls the **send** function with a reference to the buffer;

- In kernel space, the TCP/IP stack allocates a buffer in kernel space and **copies** the message from the user space buffer to it;
- After the TCP/IP stack has prepared the message to be sent, the message is **copied** to the device buffer;
- The device sends the message.

As highlighted, two copies are required to send the message in the traditional way. Since copies require space and time, this can be sub-optimal for high-performance environments. There are many workarounds and optimisations available, but they will not be described here.

Direct Memory Access As explained in the previous paragraph, copies are expensive in high-performance computing. *Direct Memory Access* (DMA) is a technique used to reduce the number of copies and the load over the system by letting a device to directly access the main memory. Using DMA the transfer of data can be offloaded to reduce the cost on the CPU. In the example above, DMA is used between the network device and its driver to offload the control of the transmission from the CPU to the network controller of the device.

Network technologies such as InfiniBand and RoCE¹ go a step further, implementing **Remote Direct Memory Access** (RDMA). RDMA enables a server to access the main memory of another server without involving the operating system and the resources of the target. RDMA is also known as a *zero-copy* transfer since the sender can directly transfer data to or from the application memory, eliminating the need of copies and buffers in the operating system. This results in a reduction in latency, improvement of throughput, and better performance overall, since the CPU is freed from managing the transfers.

¹RDMA over Converged Ethernet

Bibliography

- [1] F. J. Hasert, S. Kabe, W. Krenz, *et al.*, “Observation of neutrino-like interactions without muon or electron in the gargamelle neutrino experiment,” *Physics Letters B*, vol. 46, no. 1, pp. 138–140, Sep. 3, 1973, ISSN: 0370-2693. DOI: 10.1016/0370-2693(73)90499-1. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/0370269373904991> (visited on 05/16/2022).
- [2] G. Arnison, A. Astbury, B. Aubert, *et al.*, “Experimental observation of isolated large transverse energy electrons with associated missing energy at s=540 GeV,” *Physics Letters B*, vol. 122, no. 1, pp. 103–116, Feb. 24, 1983, ISSN: 0370-2693. DOI: 10.1016/0370-2693(83)91177-2. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/0370269383911772> (visited on 05/16/2022).
- [3] D. Decamp, B. Deschizeaux, J. -J. Lees, *et al.*, “A precise determination of the number of families with light neutrinos and of the z boson partial widths,” *Physics Letters B*, vol. 235, no. 3, pp. 399–411, Feb. 1, 1990, ISSN: 0370-2693. DOI: 10.1016/0370-2693(90)91984-J. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/037026939091984J> (visited on 05/16/2022).
- [4] G. B. Andresen, M. D. Ashkezari, M. Baquero-Ruiz, *et al.*, “Trapped antihydrogen,” *Nature*, vol. 468, no. 7324, pp. 673–676, Dec. 2010, Number: 7324 Publisher: Nature Publishing Group, ISSN: 1476-4687. DOI: 10.1038/nature09610. [Online]. Available: <https://www.nature.com/articles/nature09610> (visited on 05/16/2022).
- [5] J. R. Batley, R. S. Dosanjh, T. J. Gershon, *et al.*, “A precision measurement of direct CP violation in the decay of neutral kaons into two pions,” *Physics Letters B*, vol. 544, no. 1, pp. 97–112, Sep. 19, 2002,

- ISSN: 0370-2693. DOI: 10.1016/S0370-2693(02)02476-0. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0370269302024760> (visited on 05/16/2022).
- [6] G. Aad, T. Abajyan, B. Abbott, *et al.*, “Observation of a new particle in the search for the standard model higgs boson with the ATLAS detector at the LHC,” *Physics Letters B*, vol. 716, no. 1, pp. 1–29, Sep. 17, 2012, ISSN: 0370-2693. DOI: 10.1016/j.physletb.2012.08.020. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S037026931200857X> (visited on 05/16/2022).
- [7] S. Chatrchyan, V. Khachatryan, A. M. Sirunyan, *et al.*, “Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC,” *Physics Letters B*, vol. 716, no. 1, pp. 30–61, Sep. 17, 2012, ISSN: 0370-2693. DOI: 10.1016/j.physletb.2012.08.021. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0370269312008581> (visited on 05/16/2022).
- [8] LHCb Collaboration, R. Aaij, B. Adeva, *et al.*, “Observation of $J/\psi p$ resonances consistent with pentaquark states in $\Lambda_b^0 \rightarrow J/\psi K^- p$ decays,” *Physical Review Letters*, vol. 115, no. 7, p. 072001, Aug. 12, 2015, Publisher: American Physical Society. DOI: 10.1103/PhysRevLett.115.072001. [Online]. Available: <https://link.aps.org/doi/10.1103/PhysRevLett.115.072001> (visited on 05/16/2022).
- [9] LHCb Collaboration, R. Aaij, C. Abellán Beteta, *et al.*, “Observation of CP violation in charm decays,” *Physical Review Letters*, vol. 122, no. 21, p. 211803, May 29, 2019, Publisher: American Physical Society. DOI: 10.1103/PhysRevLett.122.211803. [Online]. Available: <https://link.aps.org/doi/10.1103/PhysRevLett.122.211803> (visited on 05/16/2022).
- [10] “LHC design report vol.1: The LHC main ring,” O. S. Bruning, P. Collier, P. Lebrun, *et al.*, Eds., Jun. 2004. DOI: 10.5170/CERN-2004-003-V-1.
- [11] S. Fartoukh, I. Efthymiopoulos, R. Tomas Garcia, *et al.* “LHC configuration and operational scenario for run 3,” CERN Document Server. Number: CERN-ACC-2021-0007. (Nov. 12, 2021), [Online]. Available: <https://cds.cern.ch/record/2790409> (visited on 05/18/2022).

- [12] C. Elsasser. “LHCb bb production angle plots.” (), [Online]. Available: https://lhcb.web.cern.ch/speakersbureau/html/bb_productionangles.html (visited on 06/30/2022).
- [13] LHCb Collaboration, R. Aaij, B. Adeva, *et al.*, “Measurement of the b -quark production cross section in 7 and 13 TeV pp collisions,” *Physical Review Letters*, vol. 118, no. 5, p. 052002, Feb. 3, 2017, Publisher: American Physical Society. DOI: 10.1103/PhysRevLett.118.052002. [Online]. Available: <https://link.aps.org/doi/10.1103/PhysRevLett.118.052002> (visited on 05/18/2022).
- [14] R. Aaij *et al.*, “Measurements of prompt charm production cross-sections in pp collisions at $\sqrt{s} = 13$ TeV,” *JHEP*, vol. 03, p. 159, 2016, *eprint* : 1510.01707. DOI: 10.1007/JHEP03(2016)159.
- [15] L. Collaboration. “LHCb tracker upgrade technical design report,” CERN Document Server. ISBN: 9789290833970 Number: CERN-LHCC-2014-001. (Feb. 21, 2014), [Online]. Available: <https://cds.cern.ch/record/1647400> (visited on 05/21/2022).
- [16] J. Albrecht, D. Bourgeois, V. Coco, *et al.*, “New approaches for track reconstruction in LHCb’s vertex locator,” *EPJ Web of Conferences*, vol. 214, p. 01042, Jan. 1, 2019. DOI: 10.1051/epjconf/201921401042.
- [17] E. Buchanan, “The LHCb vertex locator (VELO) pixel detector upgrade,” *JINST*, vol. 12, p. C01013, Jan. 2, 2017, Number: LHCb-PROC-2017-002. DOI: 10.1088/1748-0221/12/01/C01013. [Online]. Available: <https://cds.cern.ch/record/2243156> (visited on 07/05/2022).
- [18] T. Bird, “The upgrade of the LHCb vertex locator,” *JINST*, vol. 9, no. 12, p. C12041, 2014, *eprint* : 1410.0812. DOI: 10.1088/1748-0221/9/12/C12041.
- [19] L. M. Garcia, L. Henry, B. Kishor, and A. Oyanguren, “Tracking performance for long-lived particles at LHCb,” *Journal of Physics: Conference Series*, vol. 1525, no. 1, p. 012095, Apr. 2020, Publisher: IOP Publishing, ISSN: 1742-6596. DOI: 10.1088/1742-6596/1525/1/012095. [Online]. Available: <https://doi.org/10.1088/1742-6596/1525/1/012095> (visited on 07/04/2022).

- [20] M. Adinolfi, C. Gotti, S. Playfer, *et al.*, “Performance of the LHCb RICH detector at the LHC,” CERN-LHCb-DP-2012-003, LHCb-DP-2012-003, Nov. 30, 2012, Publication Title: Eur. Phys. J. C Volume: 73, p. 2431. DOI: 10.1140/epjc/s10052-013-2431-9. [Online]. Available: <https://cds.cern.ch/record/1495721> (visited on 06/30/2022).
- [21] R. Aaij, C. A. Beteta, T. Ackernley, *et al.*, “Test of lepton universality in beauty-quark decays,” *Nature Physics*, vol. 18, no. 3, pp. 277–282, Mar. 2022, Number: 3 Publisher: Nature Publishing Group, ISSN: 1745-2481. DOI: 10.1038/s41567-021-01478-8. [Online]. Available: <https://www.nature.com/articles/s41567-021-01478-8> (visited on 06/20/2022).
- [22] L. collaboration, R. Aaij, A. S. W. Abdelmotteleb, *et al.*, “Observation of an exotic narrow doubly charmed tetraquark,” *Nature Physics*, Jun. 16, 2022, ISSN: 1745-2473, 1745-2481. DOI: 10.1038/s41567-022-01614-y. arXiv: 2109.01038[hep-ex]. [Online]. Available: <http://arxiv.org/abs/2109.01038> (visited on 06/20/2022).
- [23] S. Cadeddu, P. Dalpiaz, Z. Guzik, *et al.*, “LHCb trigger system : Technical Design Report,” CERN, CERN-LHCC-2003-031, 2003, ISBN: 9789290832089 Publication Title: CERN Document Server. [Online]. Available: <https://cds.cern.ch/record/630828> (visited on 07/03/2022).
- [24] L. collaboration. “Letter of Intent for the LHCb Upgrade,” CERN Document Server. Number: CERN-LHCC-2011-001. (Mar. 1, 2011), [Online]. Available: <https://cds.cern.ch/record/1333091> (visited on 07/03/2022).
- [25] G. Liu and N. Neufeld, “DAQ architecture for the LHCb upgrade,” *Journal of Physics: Conference Series*, vol. 513, no. 1, p. 012027, Jun. 2014, Publisher: IOP Publishing, ISSN: 1742-6596. DOI: 10.1088/1742-6596/513/1/012027. [Online]. Available: <https://doi.org/10.1088/1742-6596/513/1/012027> (visited on 05/20/2022).
- [26] M. Bellato, G. Collazuol, I. D’Antone, *et al.*, “A PCIe gen3 based readout for the LHCb upgrade,” *Journal of Physics: Conference Series*, vol. 513, no. 1, p. 012023, Jun. 2014, Publisher: IOP Publishing, ISSN: 1742-6596. DOI: 10.1088/1742-6596/513/1/012023. [Online]. Available: <https://doi.org/10.1088/1742-6596/513/1/012023> (visited on 05/20/2022).

- [27] P. Moreira, K. Wyllie, B. Yu, *et al.* “The GBT Project,” CERN Document Server. Publisher: CERN. (2009), [Online]. Available: <https://cds.cern.ch/record/1235836> (visited on 07/03/2022).
- [28] R. Aaij *et al.*, “Allen: A high level trigger on GPUs for LHCb,” *Comput. Softw. Big Sci.*, vol. 4, no. 1, p. 7, 2020, *eprint* : 1912.09161. DOI: 10.1007/s41781-020-00039-7.
- [29] S. Benson, V. Gligorov, M. A. Vesterinen, and J. M. Williams, “The LHCb turbo stream,” *Journal of Physics: Conference Series*, vol. 664, no. 8, p. 082004, Dec. 2015, Publisher: IOP Publishing, ISSN: 1742-6596. DOI: 10.1088/1742-6596/664/8/082004. [Online]. Available: <https://doi.org/10.1088/1742-6596/664/8/082004> (visited on 06/21/2022).
- [30] E. Zahavi, G. Johnson, D. Kerbyson, and M. Lang, “Optimized InfiniBand™ fat-tree routing for shift all-to-all communication patterns,” *Concurrency and Computation: Practice and Experience*, vol. 22, pp. 217–231, Nov. 17, 2009. DOI: 10.1002/cpe.1527.
- [31] W. Kendall, *MPI tutorial*, original-date: 2012-11-29T04:53:19Z, Jul. 2, 2022. [Online]. Available: <https://github.com/mpitutorial/mpitutorial> (visited on 07/04/2022).
- [32] R. D. Krawczyk, T. Colombo, N. Neufeld, F. Pisani, and S. Valat, “Feasibility tests of RoCE v2 for LHCb event building,” *EPJ Web of Conferences*, vol. 245, p. 01011, 2020, Publisher: EDP Sciences, ISSN: 2100-014X. DOI: 10.1051/epjconf/202024501011. [Online]. Available: https://www.epj-conferences.org/articles/epjconf/abs/2020/21/epjconf_chep2020_01011/epjconf_chep2020_01011.html (visited on 07/04/2022).
- [33] NVIDIA, *RDMA Aware Networks Programming User Manual*.
- [34] F. Mietke, R. Rex, R. Baumgartl, T. Mehlan, T. Hoefler, and W. Rehm, “Analysis of the memory registration process in the mellanox InfiniBand software stack,” in *Euro-Par 2006 Parallel Processing*, W. E. Nagel, W. V. Walter, and W. Lehner, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2006, pp. 124–133, ISBN: 978-3-540-37784-9. DOI: 10.1007/11823285_13.

- [35] B. Wilkinson and C. M. Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Pearson/Prentice Hall, 2005, 467 pp., Google-Books-ID: Uh92ngEACAAJ, ISBN: 978-0-13-191865-8.