Event Building Performance Assessment through OpenMPI Network Benchmarking

Lana Miličević, *University of Rijeka*
Supervisors: Flavio Pisani, Niko Neufeld

October 2023

1 Abstract

The Event Building (EB) system is a crucial component of the LHCb experiment at CERN, used to assemble data fragments from experiment runs into events. High-speed interconnects such as InfiniBand are vital for its optimal performance. A standalone OpenMPI-based benchmark was developed with the primary goal of creating a flexible benchmarking program, capable of adjusting to various configuration and run parameters. Subsequent tests were conducted on real servers. The benchmark can aid in network diagnostics, as well as enhancing its efficiency and reliability.

Keywords: data acquisition, LHCb experiment, event building, OpenMPI, InfiniBand, benchmarking

2 Background

The LHCb experiment at CERN focuses on studying the properties of rare B meson decays produced in the Large Hadron Collider (LHC). The experiment's data acquisition (DAQ) system plays a crucial role in collecting and processing vast amounts of data generated by the collisions. The new LHCb DAQ system incorporates a high-speed InfiniBand network, capable of handling a data rate of approximately 32 Terabits per second. The data is collected from various sub-detectors through optical fiber links and processed using Tell40 FPGA-based cards. The Event Building (EB) process, responsible for assembling complete events from data fragments, utilises a push DAQ model with Readout Units (RUs) and Builder Units (BUs) to efficiently aggregate the data. To manage the network traffic efficiently, a linear shifting traffic shaping algorithm is employed, enabling the EB nodes to balance the data distribution and avoid over-subscription. [1]

The EB process relies on a folded architecture, where a BU and an RU share the same network port to optimise bidirectional bandwidth usage. This architectural choice reduces the number of required network ports at the cost of sharing other resources like CPU and RAM. To handle the challenge of transmitting small-sized data fragments efficiently, multiple fragments are packed into a multiple-fragment packet (MFP), while complete events are assembled into multiple-event packets (MEPs) for transmission over the InfiniBand network. The use of multiple-fragment and multiple-event packets optimises the transmission of small-sized data fragments.

To achieve optimal data aggregation, the EB process utilizes a push DAQ model, employing RUs for reading and BUs for receiving and assembling the data. These units are the building blocks of the EB, implemented as a modular C++ application integrated into the Online framework, DataFlow. The DataFlow framework provides essential functionalities, and a setup which allows for easy configuration and customisation of the EB behavior. The low-level communication with the InfiniBand network is managed through a custom interface library called InfiniBuilder, with complete control over the InfiniBand network. The EB application, when using InfiniBuilder, operates as separate processes.

To ensure correct execution of the linear shifting scheduling, the EB process implements a synchronisation barrier using collective communication. This two-step procedure involves all processes reaching the barrier, followed by a notification to release all processes once the barrier is successfully reached.

In order to benchmark the network's preformance, the EB system's setup was mirrored into the benchmarking implementation. The benchmark program encompasses its network layout, communication patterns, and linear shifting logic. Following this setup, various types of simulated data were communicated through the system to assess its performance.

3 Development

3.1 Communication Basics

To establish the foundational framework of the project, a rudimentary Open MPI application for interprocess communication between two MPI ranks was initially implemented. This preliminary application was designed with the following core functionalities which were in most part kept as a basis for development of the final product:

- Utilisation of two MPI processes, each occupying a distinct rank on a TDEB server, simulating a basic client-server architecture.
- Transfer of data with user-configurable specifications, including the quantity of data to be transmitted, the size of the data buffer, and the number of iterations between the communicating processes.
- Throughput and average communication time calculation and analysis, serving as a benchmark for assessing the communication infrastructure.

3.2 Unit-based Communication

Following the successful implementation of the project fundamentals, its capabilities have been enchanced to support interprocess communication across a larger number of distinct computational ranks. The initialisation of the ranks is designed to distribute them across multiple servers, using both of their NICs in such a way that a BU and an RU share the same network port, as aligned with the architecture detailed in Section 2. Elevating the project's foundational aspects to this development stage, employing an object-oriented methodology and adaptable variables, allows for the scalability in the benchmark and facilitates possible future work on its development.

In this phase of development, the primary newly introduced features involve transitioning from a fixed 2-rank structure to adopting a unit-centric perspective when considering hosts in the context of BUs and RUs. Consequently, each process is characterized by both the benchmark it executes (either scan or continuous, with choices for fixed or variable message size and blocking or non-blocking communication) and the unit it utilises (BU or RU). All processes within the benchmark share the same benchmark type, and they collectively access information on the other units participating in the benchmark run.

Given that communication operates according to a linear shift, such that during phase n, RU i communicates with BU $(n+i)$ `mod` N and vice versa, each unit initialises a shift vector specifying its communication partners for each phase. Additionally, certain nodes can be designated as dummies, and as a result, communication with these nodes is omitted.

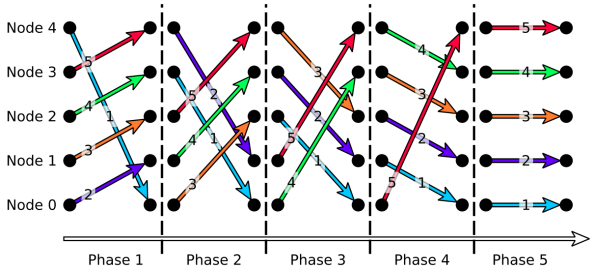

Figure 1: Example of data exchange between EB nodes using a linear shifting scheme [1]

3.3 Testing

In the final phase of development, test runs were conducted on the tdeb servers within the CERN network. They were conducted by executing non-blocking runs with a consistent message size. Firstly the experiment was initiated with two servers and progressively expanded by one, up to a total of ten servers, all under the same set of parameters. Each parameter set consisted of three messages, each sized at 5 MB per phase, with a total of 100,000 iterations per phase. Testing results are presented in Section 5.

The primary objective of these tests was to evaluate the network's performance and to validate our hypotheses regarding throughput. These tests serve a dual purpose: first, to affirm that the network operates within the expected capacity (about 200 Gbit/s) for the given parameter set, and second, to validate the robustness of the program's construction. Notably, any anomalies in the results would indicate potential issues with program implementation. Furthermore, higher bandwidth outliers are expected in scenarios where data remains within the same host rather than being transmitted over the network. All the observed evidence attests to the proper implementation of both the program and the run configuration in the short term, and lays the foundation for assessing network performance in real-world applications in the long term.

4 Implementation

An Open MPI-based benchmarking program was created, leveraging the Open MPI Project — an open-source implementation of the Message Passing Interface (MPI). The Open MPI communications library provides highly efficient communica-

tions support, tailored to specific communications context. [2], [3]
The program supports various communication types including scan runs, fixed message size runs, and variable message size runs, with options for non-blocking communication. The main script parses run arguments, initializes the benchmark according to specified run type, and conducts warmup and benchmark runs.

4.1 Project Hierarchy

A structured class hierarchy is employed to represent key concepts involved in the communication processes. The core class, `Benchmark` serves as the parent class for two specialized subclasses: `ScanBenchmark` and `ContinuousBenchmark`. The `ScanBenchmark` class is tailored for conducting scan runs to gain insight into the relation of orders of message and throughput sizes, systematically measuring message throughput for various sizes in powers of 2 across two machines. On the other hand, the `ContinuousBenchmark` class encompasses both fixed and variable message size scenarios, accommodating the dynamic nature of continuous communication. Derived from this abstract parent class, `BenchmarkFixedMessage` and `BenchmarkVariableMessage` offer size-specific functionalities for run configuration and throughput analysis. This class structure enhances the modularity and flexibility of the benchmarking framework, facilitating comprehensive performance assessments in diverse communication contexts.

Additionally, the `Unit` class is employed to represent distinct communication entities, each equipped with member variables for efficient buffer memory management. To streamline this process, every benchmark object is associated with a corresponding `Unit` object, which possesses its own dedicated buffer and is endowed with rank and identifier information inherited from the process. Communication operations within the project are performed utilising the functions encapsulated within the `CommunicationInterface`, implemented for better code accessibility and organisation.

4.2 Messages

Continuous benchmarks differ in their approach to sending messages, which can either have a fixed or variable size. In both cases, these benchmarks populate their send buffers with placeholder data, each element of which is a byte.
In the case of a fixed-size benchmark, the amount of data sent over the network remains constant throughout the benchmark run, commencing from a specific offset within the send buffer.

Conversely, in a variable-size benchmark, the approach is more dynamic. Initially, a vector of modifiable size is initialised with message sizes, generated randomly within a predefined size range. These generated sizes adhere to a uniform distribution within the specified range, and ensure that they do not exceed the buffer's capacity. During communication, a random index within the vector of message sizes is selected, and the size value at that index is sent over the network to inform the receiver of the impending data transfer. The actual message, of the specified size, is then transmitted between sender and receiver, using the send and receive buffers.

4.3 Buffers

Buffers are an integral part of the `Unit` class and serve as either send or receive buffers depending on the unit type. Each unit belongs to a process and is responsible for memory allocation to create its own buffer. To optimise memory management, the unit first identifies the system's page size and allocates memory aligned to this page size to accommodate the necessary buffer size. Once the memory allocation is successful, it proceeds to initialise the allocated memory by populating it with byte-sized zeros as placeholder data.

4.4 Communication Modes

Warmup

Before initiating the communication runs, a warmup phase is conducted on the buffer memory space of the processes, ensuring enhanced throughput. This warmup phase is performed for two distinct benchmarking scenarios: `ScanBenchmark`, which employs a fixed two-process configuration, and `ContinuousBenchmark`, adaptable to multiple processes. In the case of `ScanBenchmark`, a sender and receiver rank are predefined, establishing a static communication pattern. In contrast, `ContinuousBenchmark` operates with multiple ranks, with each rank sending data to its subsequent counterpart (e.g., 0 to 1, 1 to 2, and so forth).

The warmup function performs data exchanges between processes located on different servers by employing a set of predetermined subarray indices. These indices strategically partition the buffer memory into segments, each constituting 32% of the buffer's total size. Additionally, each segment is intentionally overlapped with its neighboring segments by an extent equal to 15% of the buffer size (Figure 2). Note that these numbers were

chosen experimentally. This overlapping mechanism ensures data consistency, minimises fragmentation, and enhances subsequent communication efficiency during the warm-up phase.

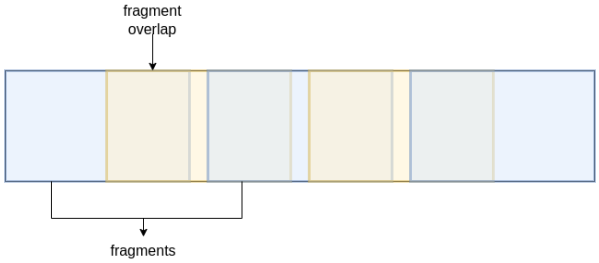

Figure 2: Warmup buffer fragments

Blocking

On the sender side, chunks (messages) are taken out of a circular buffer and sent over network using `MPI_Send`. The `MPI_Send` function executes a standard-mode, blocking send operation within the Message Passing Interface (MPI). When this routine is invoked, it will enter a blocking state and will not return until the entire message has been successfully sent to the specified destination. In cases where the intended message size for communication surpasses the boundaries of the buffer, a straightforward approach is adopted for ease of implementation. The buffer offset is reset to its initial position, and communication continues from that point.

```
for (std::size_t i = 0; i < iterations; i
    ++)
{
    if (sendOffset + messageSize >
    sndBufferBytes)
        sendOffset = 0;
    MPI_Send(bufferSnd + sendOffset,
    messageSize, MPI_BYTE,
             buRank, 0, MPI_COMM_WORLD);
    sendOffset = (sendOffset + messageSize)
     % sndBufferBytes;
}
```

Listing 1: Blocking send

On the receiver side, the BU utilises a circular buffer for receiving, employing the same underlying logic for circular buffer and offset handling as the RU. While the offset handling logic remains the same, for simplicity of implementation a message that is larger than the entire receive buffer will be cut off at its end. The reception process involves an `MPI_Recv` operation, blocking in nature. This means that it will only return once the receive buffer holds the newly received message. A receive operation can complete before the corresponding send operation has finished (although only after the matching send operation has started).

```
MPI_Recv(bufferRcv + recvOffset,
    messageSize, MPI_BYTE, ruRank, 0,
         MPI_COMM_WORLD, &statuses[i]);
```

Listing 2: Blocking receive

Non-blocking

Non-blocking communication in follows the same logic to the previously described blocking communication, but it instead utilises `MPI_ISend` and `MPI_IRecv`. These non-blocking calls allocate a communication request object associated with a request handle, allowing for later status queries or completion waits. Overall, non-blocking communication offers performance benefits by allowing for concurrent data transfers, likely leading to faster execution times.

For non-blocking sends, it signifies that data send may start from the send buffer, and it's essential not to modify the buffer until the send operation is complete. Similarly, for non-blocking receives, the system can start writing data into the receive buffer, and it's crucial not to access it until the receive operation concludes. To ensure synchronisation and completion of these non-blocking operations, `MPI_Waitall` is used. It blocks until all communication operations linked to active handles are completed, returning their statuses. When there are errors in any of the communications, `MPI_Waitall` returns an error status, aiding in error handling.

```
if (processRank == ruRank)
{
    // ...
    MPI_Isend(bufferSnd + sendOffset,
    messageSize, MPI_BYTE, buRank, 0,
              MPI_COMM_WORLD, &sendRequests
    [i]);
    // ...
    MPI_Waitall(iterations, sendRequests.
    data(), statuses.data());
}
```

Listing 3: Non-blocking send

```
if (processRank == buRank)
{
    // ...
    MPI_Irecv(bufferRcv + recvOffset,
    messageSize, MPI_BYTE, ruRank, 0,
              MPI_COMM_WORLD, &recvRequests
    [i % syncIterations]);
    // ...
    MPI_Waitall(iterations, recvRequests.
    data(), statuses.data());
}
```

Listing 4: Non-blocking receive

4.5 Run Initiation

To facilitate the benchmarking application run and make it more modular, two Python scripts are employed to set the host and run parameters.

Configuration

The configuration Python script's task is configuring the network parameters for the benchmark application run. Its core functionality is parsing a hosts file, which serves as a repository of data concerning network nodes and their associated devices. The script provides a range of utility functions designed to interpret the input file, as well as perform host and device verification to ensure that it aligns with specific run and architecture requirements.

Once the hosts file has been processed, the script undertakes two actions. Firstly, it generates a configuration JSON file and a host file, both of which are used in setting the run parameters such as proper allocation of NICs and noting the presence of dummy nodes within the network. Furthermore, the script constructs a shift pattern that defines the communication pathways between nodes. This shift pattern takes into account each node's unique index, device, and whether it functions as a dummy node or not.

Run

The purpose of the run Python script is to initiate an `mpirun` command for conducting the benchmark. It achieves this by parsing input files and parameters to generate a tailored MPI command. To do this, it must perform parsing of a hosts file generated by the configuration script, containing information regarding network nodes and their associated devices. Each line in this file corresponds to a host and device name, which the script extracts for subsequent use.

The script is highly flexible, allowing users to specify the benchmarking mode (scan, fixed, or variable), non-blocking mode support, and a multitude of parameters such as message size, buffer size, logging intervals, and other. Upon processing the hosts file and user-defined parameters, the script generates a customised `mpirun` command, and executes it as a process.

5 Discussion

The benchmark underwent testing on the tdeb servers on the InfiniBand network, and the results were subjected to analysis using the generated logs of the average throughput for each phase and its evolution over time across all phases. The results are analysed from two perspectives: (i) throughput analysis per number of hosts involved in communication, and (ii) overall throughput comparisons across configurations featuring different host counts, presented in Figure **??**.

The average throughput observed over a 5-second interval exhibited consistency, ranging from approximately 190 Gbit/s to 196 Gbit/s, with any observed discrepancies attributed to random variations rather than variations in the number of nodes.

When examining throughput on a per-phase basis, two distinct patterns emerged. Firstly, phase 0 consistently exhibited the highest throughput, primarily owing to local data copying within the same host rather than data transmission over the network. Secondly, both phase 1 and the final phase consistently demonstrated significantly higher throughput compared to the intermediate phases. This heightened throughput in these phases is attributed to partial presence of data transfer through memory copying.

6 Conclusion

In conclusion, this paper outlined the development and testing of a standalone OpenMPI-based benchmark application. The developed program serves as a versatile and modular tool for evaluating the EB system under varying configuration and runtime parameters. The benchmark's multifaceted capabilities extend beyond real-server performance evaluation; it can serve as a valuable tool for network diagnostics, enhancing the overall efficiency and reliability of the EB system, underscoring the role of benchmarking in optimising the performance of critical components in large-scale experiments.

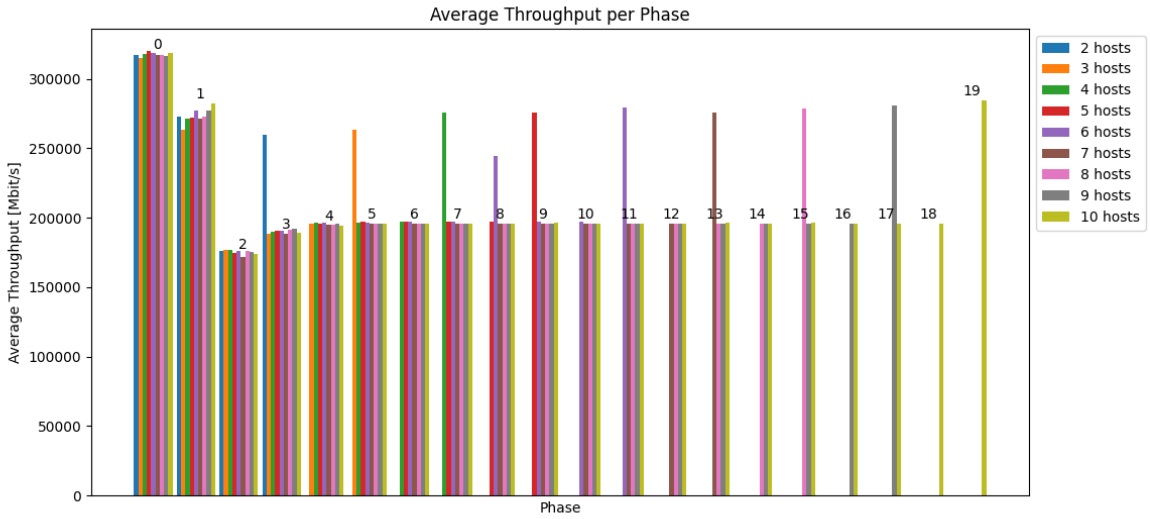

Figure 3: Average throughput per phase

References

[1] F. Pisani, T. Colombo, P. Durante, *et al.*, "Design and commissioning of the first 32-tbit/s event-builder," *IEEE Transactions on Nuclear Science*, vol. 70, no. 6, pp. 906–913, 2023. DOI: 10.1109/TNS.2023.3240514.

[2] Open MPI Project, *Open MPI*, version 4.1, Jul. 4, 2023. [Online]. Available: https://www.open-mpi.org/.

[3] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine, "Open mpi: A high-performance, heterogeneous mpi," in *2006 IEEE International Conference on Cluster Computing*, Sep. 2006, pp. 1–9. DOI: 10.1109/CLUSTR.2006.311904.