Additional materials

NGT project

Remote Offloading for the High-Level Trigger of the Compact Muon Solenoid Experiment at CERN

Anna Polova < anna.polova@cern.ch >[1,2], Andrea Bocci[1], Mario Gonzalez[1]

[1] CERN, [2] Ukranian Catholic University

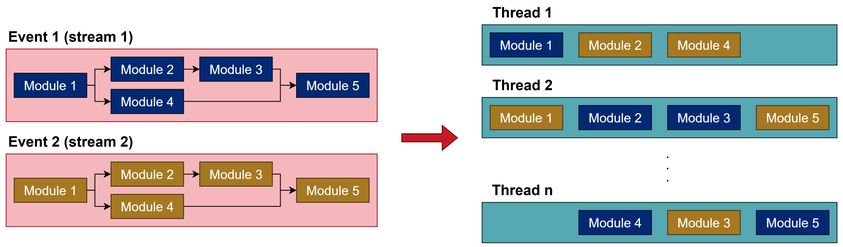
Ukrainian Catholic University · CERN · CMS · NexTGen Next Generation Triggers

Introduction

CERN, the European Organization for Nuclear Research, operates the Large Hadron Collider (LHC), the world's largest particle accelerator, where protons are collided up to 30 million times per second at nearly the speed of light to study the fundamental building blocks of matter. The Compact Muon Solenoid (CMS), records and analyzes the resulting particle interactions using a two-stage trigger system: the hardware-based Level-1 Trigger (L1T) selects about 100,000 events per second, and the software-based High Level Trigger (HLT) further filters this to 30,000 events stored in reduced format and 10,000 in full raw format.

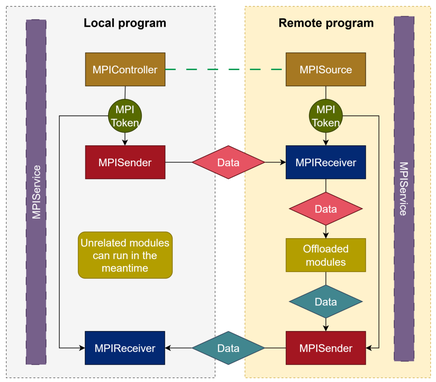
The CMS simulation, reconstruction, and analysis software (CMSSW)

- Modular application (C++ code, Python configuration)
- Central abstraction: the event (a particle collision), stored as a collection of data products.
- Modules:
  - Implement detector simulation, reconstruction, and event selection
  - Declare what data they consume and produce.
  - Can be dynamically loaded and arranged into processing paths.
- Framework automatically determines execution order from data dependencies
- Framework exploits parallelism at multiple levels:
  - Across events (multiple streams)
  - Within events (independent modules run concurrently)
- Worker threads are managed by Intel oneTBB.
- Asynchronous tasks run on a separate, dynamically sized thread pool.



Application of MPI in CMSSW

Our main requirement is to enable remote offloading of selected event-processing steps without changing the core framework. All communication must therefore fit within the concept of loadable modules that explicitly declare their data dependencies.

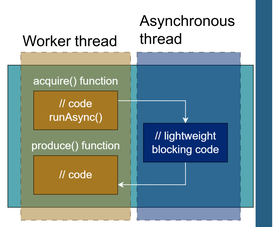


Legend

- MPIController — Prepare an MPI communicator for each event before data transfer begins. MPIController attaches a communicator token to an existing event, while MPISource creates a new event carrying the token (for remote workflows).
- MPISource
- MPI Token — Wapper around an MPI communicator that can be stored as event data. It hides direct MPI calls and provides a framework-friendly handle to the communicator.
- MPISender — Knows which data to transmit from the event based on declared dependencies. Is scheduled as soon as the required data products are available.
- MPIReceiver — MPIReceiver is scheduled once upstream pipeline modules and other MPI communication steps are executed
- MPIService — MPIService is a long-lived service that runs for the entire program. It manages global MPI setup and handles `MPI_Init` and `MPI_Finalize`.

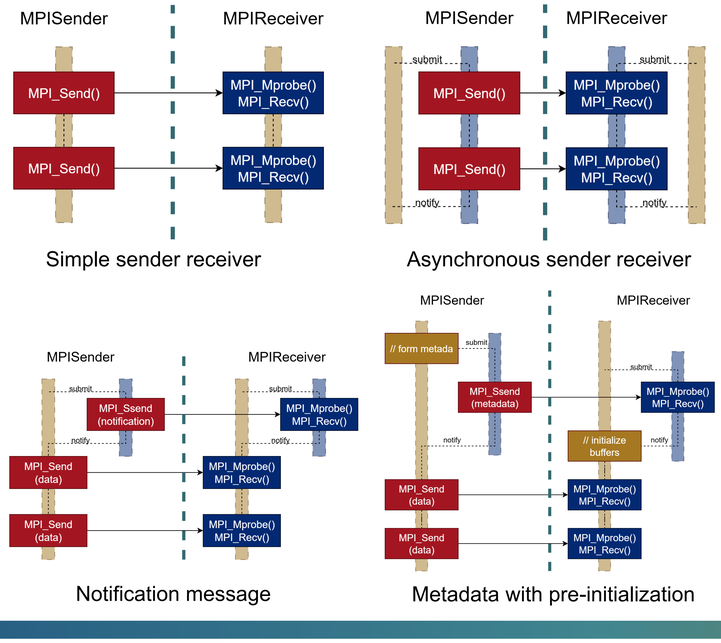
Asynchronous mechanisms in CMSSW

On top of the task-based model CMSSW provides a mechanism for handling asynchronous operations. A module can be split in two parts: the first part can submit asynchronous operations to an external worker; the framework uses a dedicated thread to sleep until these operations are complete, and only then schedules the second part of the module.
This forms the basis for efficient GPU support and is repurposed in our case to handle blocking communications.



Approaches to sender – receiver architecture

Below are the four different approaches to sender-receiver architecture implemented within the described framework. The yellow vertical line indicates the worker thread, while the blue one denotes additional thread created by framework's asynchronous mechanism. Worker thread can do some other tasks while communication code is asynchronously executed.



Experimental setup

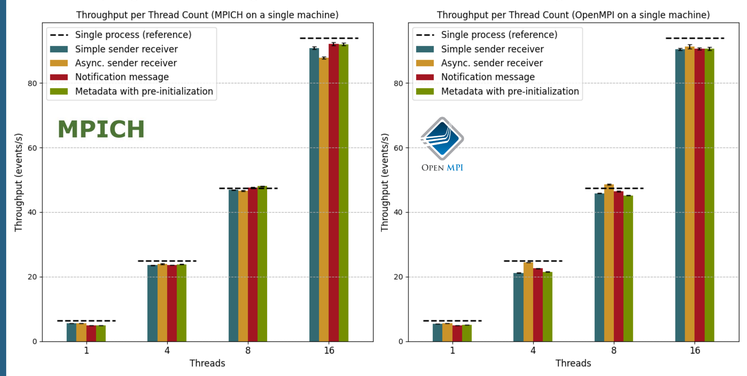
The experiments were performed on two machines connected by a 100 Gbps InfiniBand link, using a realistic HLT pipeline in which the local process hosted a single sender that transmitted raw data, and five receivers that inserted the returned results back into the pipeline. We carried out four sets of measurements:

1. A single CMSSW process (without MPI), used as reference.
2. A CMSSW application split into two processes communicating over MPI on a single machine, using the xpmem backend.
3. A CMSSW application split into two processes communicating over MPI on two machines connected by InfiniBand, using the ucx backend.
4. A simplified set of benchmarks to directly compare the performance of OpenMPI and MPICH on the same hardware.

More details about the setup you can find by "additional materials" QR-code.

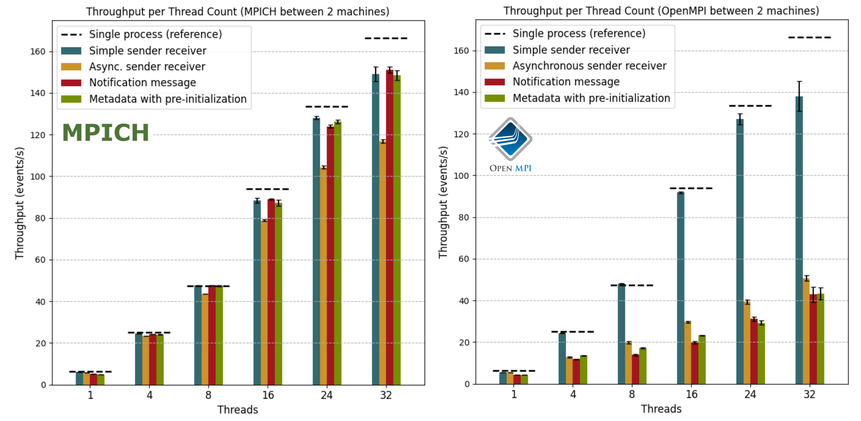
Performance comparison on a single machine

These experiments compare the performance of a single application with that of two MPI programs, running on the same machine within a single CPU socket



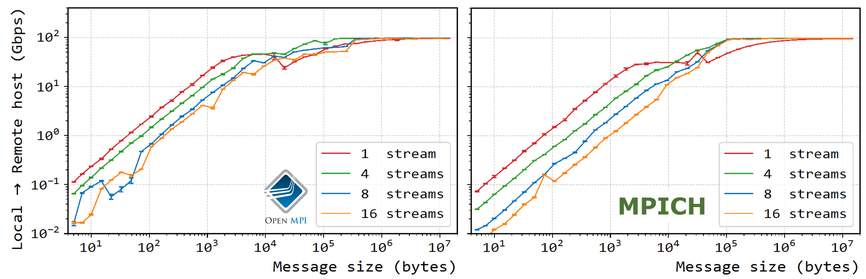
Performance comparison on two machines connected over InfiniBand

The tests below use the same setup between two machines, with blue bar denoting reference single application. Although both MPICH and OpenMPI rely on the same UCX library installation, we observe significant performance differences across most approaches.



Benchmarking outside of CMSSW

We benchmarked OpenMPI and MPICH outside of CMSSW by running two MPI processes, each spawning multiple pinned threads that continuously exchanged data with MPI_Send from one process to MPI_Receive on another. The aggregated bandwidth across all streams was measured to assess whether the communication behaviour observed in CMSSW could be reproduced in isolation.



Conclusions

On a **single machine**, the HLT application split in two process that communicate over MPI achieves **approximately 96%** of the throughput of the reference application for **both OpenMPI and MPICH**, despite the communication overhead, up to **16 threads** per job.
In **cross-machine tests**, the best-performing approaches at **16 threads** reach **97.7% (OpenMPI)** and **94.6% (MPICH)**. At **32 threads**, however, the simple approach yields **83% (OpenMPI)** and **89.5% (MPICH)**. With the preferred metadata-based approach, the gap becomes dramatic: **26% for OpenMPI** versus **89.2% for MPICH**.
Overall, performance on local setups is similar across both libraries and approaches, with communication overhead being much smaller than in inter-machine runs. However, despite showing no significant differences in independent benchmarks, **OpenMPI and MPICH behave very differently in the CMSSW context when inter-machine communication is involved**.

References

- CMS Collaboration, "Development of the CMS detector for the CERN LHC Run 3", Journal of Instrumentation 19 (2024). https://doi.org/10.1088/1748-0221/19/05/P05064
- C.D. Jones and E. Sexton-Kennedy, "Stitched Together: Transitioning CMS to a Hierarchical Threaded Framework", J. Phys. Conf. Ser. 513 (2014) 022034. https://doi.org/10.1088/1742-6596/513/2/022034
- A. Bocci et al., "Bringing heterogeneity to the CMS software framework", EPJ Web Conf. 245 (2020) 05009. https://doi.org/10.1051/epjconf/202024505009
- A. Bocci, C. Jones, and M. Kortelainen, "Performance of Heterogeneous Algorithm Scheduling in CMSSW", EPJ Web Conf. 295 (2024) 11017. https://doi.org/10.1051/epjconf/202429511017
- J. Alawieh et al., "Experience with the alpaka performance portability library in the CMS software", CHEP 2024. URL: https://cds.cern.ch/record/2927051

Supported by the Eric & Wendy Schmidt Fund for Strategic Innovation (grant agreement SIF-2023-004)