



Introducing heterogeneous farms in the CMS framework

AUGUST 2018

AUTHOR:

Serena
Ziviani

CERN IT-CF-FPP

SUPERVISORS:

Luca Atzori
Felice Pantaleo





Project Specification

In High-Energy Physics, experiments' trigger and reconstruction run on homogeneous computing systems. Machines which are not specialized can run different work-flows with average performance. However, the LHC will enter an era of long luminosity doubling times with the expectation that small signals will gradually emerge from backgrounds. Any physics that is discovered will need very high statistics to fully explore its properties, hence requiring large samples of data. The experiments' computing infrastructure has grown, but because of the non-unlimited funding, will not be as much as needed to keep up. Making computing systems like the trigger farms heterogeneous would help in optimizing resources, by offloading different types of computation to the machines whose topology and architecture are the best match for the job. The purpose of this project is to develop a small scale demonstrator with a software framework that can be used to offload specific type of computation to another better suited node



Abstract

¹⁵ The High Luminosity upgrade scheduled for 2026 will greatly increase the number of events per collision. Moore's law will optimistically get a factor 4 performance gain, not enough to handle the luminosity increase without incurring significant costs and logistic issues as the actual system is already deeply optimized.

²⁰ A possible solution is the creation of a *heterogeneous cluster*, a cluster in which different nodes have different kinds of co-processors available. This study tackles the problem of moving efficiently a task that can be executed on a co-processor to a different node in the cluster.

²⁵ As a first step, a standalone application to study the communication protocol between the three kinds of nodes has been implemented. Then, the protocol has been introduced into the framework by implementing new kinds of EDProducers and EDAnalyzers. Finally, the code has been benchmarked and the results analyzed.



Contents

Contents	iv
List of Figures	v
Listings	vi
30 1 Introduction	1
2 Motivation	2
2.1 Cluster architecture	2
2.1.1 A novel approach: heterogeneous clusters	2
2.2 Compact Muon Solenoid (cms) Trigger and Data Acquisition System	4
35 2.3 Environment: the CMS! (cms!) Software (cmssw) framework	4
2.3.1 EDM: Event Data Model	4
2.3.2 Module types	6
2.3.3 Services	6
2.3.4 Configuration files	6
40 2.4 The offloading model	6
2.4.1 Communication protocol	7
3 Implementation	9
3.1 Message Passing Interface (MPI)	9
3.2 Intel Threading Building Blocks (TBB)	9
45 3.3 Proof of Concept	9
3.3.1 Scheduler	10
3.3.2 Offloader	10
3.3.3 Worker	10
3.4 MPICore	10
3.4.1 Initialization	11
3.4.2 Offloader	11
3.4.3 Worker path	12
4 Results	14
4.1 Test environment	14
55 4.2 Overhead	15
5 Conclusions	17
Bibliography	18



List of Figures

60	2.1	A standard, homogeneous cluster	2
	2.2	A cluster for heterogeneous computing	3
	2.3	A heterogeneous cluster	3
	2.4	Difference between execution on the same machine (2.4a) and offload to a node that can exploit a much more efficient implementation for the same task (2.4b). Note that offloading becomes even more beneficial the more modules are of- floaded (2.4c)	4
65	2.5	An example of a cmssw path	5
	2.6	Communication protocol	7
70	3.1	The two paths representing an Offloader (left side) and a Worker (right side). Mod- ules in blue are aware of the MPI communication, while modules in green commu- nicate directly with the Graphics Processing Unit (GPU).	11
	4.1	The probe points inserted in MPICore	14
	4.2	Distribution plot of overhead for different vector lengths	16
	4.3	Overhead mean for 1 offloader, 1 worker for each vector length, absolute (4.3a) and relative to vector length (4.3b)	16



75 Listings

1	OffloadProducer	12
2	FetchProducer	13
3	WorkProducer	13
4	SendAnalyzer	13



80 List of acronyms

- GPU** Graphics Processing Unit
CPU Central Processing Unit
CUDA Compute Unified Device Architecture
FPGA Field-Programmable Gate Array
85 **MPI** Message Passing Interface
EDM Event Data Model
CMS Compact Muon Solenoid
CMSSW CMS! Software
MPI Message Passing Interface
90 **TBB** Threading Building Blocks
ASIC Application-Specific Integrated Circuit
HLT High-Level Trigger

1. Introduction

The High Luminosity upgrade scheduled for 2026 will greatly increase the number of events per collision.⁹⁵ Moore's law will optimistically get a factor 4 performance gain, not enough to handle the luminosity increase and consequently the data acquisition chain must be able to process even more data per second. The actual system is already deeply optimized, so there is the need to either increase the cluster/datacenter size or to explore new solutions.

Right now the cluster used by the CMS experiment is homogeneous, that is it's entirely composed of "normal" servers that execute the work on Central Processing Units (CPUs), even for workloads that could be executed in a more optimized way on a different architecture as, for example, GPUs, Field-Programmable Gate Arrays (FPGAs) and other kinds of accelerators.¹⁰⁰

The idea is to explore the creation of a *heterogeneous cluster*, in other words a cluster with different kind of nodes/architectures available, and to offload the computation to the most efficient node available.¹⁰⁵

Nodes are differentiated into three kinds based on their capabilities, with the three categories being:

- A *Scheduler* that knows the infrastructure of the cluster and can choose the most suitable node for the computation;
- *Offloaders* that execute a chain of modules, offloading the ones that can be executed faster on a different machine;
- *Workers* that are equipped with an accelerator of any kind (e.g GPU, FPGA|).

As a first step, a standalone application to study the communication protocol between the three kinds of nodes has been implemented. Then, the protocol has been introduced into the framework by implementing new kinds of EDProducers and EDAnalyzers. Finally, the code has been benchmarked and the results analyzed.¹¹⁵



2. Motivation

2.1 Cluster architecture

The intent of this chapter is to cover the concepts and the ideas that motivate the introduction of a *heterogeneous cluster* in the CMS experiment.
120

A *computer cluster* is defined as a set of computers, tightly or loosely coupled, that execute the same task to reach a common goal.

Usually the nodes of a cluster are connected with each other with an Ethernet connection in a LAN. In most cases, all of the nodes have the same or equivalent hardware.

125 When heterogeneous computing is introduced in such a cluster every node is equipped with the same co-processor, for example one or more GPUs, FPGAs or many-core architectures like the Xeon Phi. This allows for a great performance boost but, even with medium sized clusters, it can become an important expense.

2.1.1 A novel approach: heterogeneous clusters

130 A heterogeneous cluster is a cluster that has different sets of capabilities between each node. This can be due to many factors:

- the addition of co-processors only on certain nodes;
- the addition of different kinds of co-processor on different nodes;
- the use of different CPU architectures;
- the use of nodes with drastically different computing power between each other.

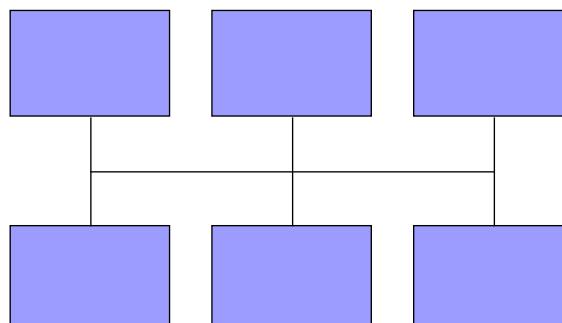


Figure 2.1: A standard, homogeneous cluster

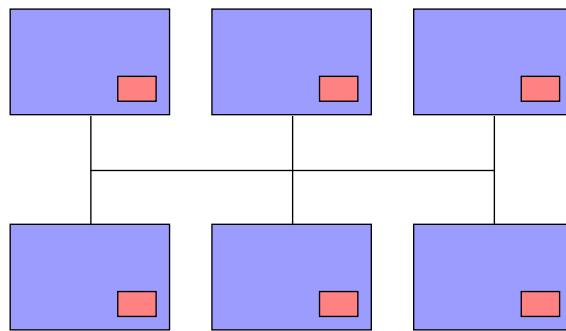


Figure 2.2: A cluster for heterogeneous computing

This approach offers greater flexibility both in the creation, modification and update of the cluster and in its use, but it creates the necessity of a different programming paradigm: programs now are made only for parallel computing, heterogeneous computing or a combination of the two.

Differently from a homogeneous cluster that exploits heterogeneous computing, where the number and the type of coprocessor are the same for every node and therefore are known, a node in a heterogeneous cluster knows neither if there exists a node with a specific accelerator nor how to locate it. To be able to fully exploit the cluster it is then vital to develop a system that allows to offload specific tasks to the most suitable node.

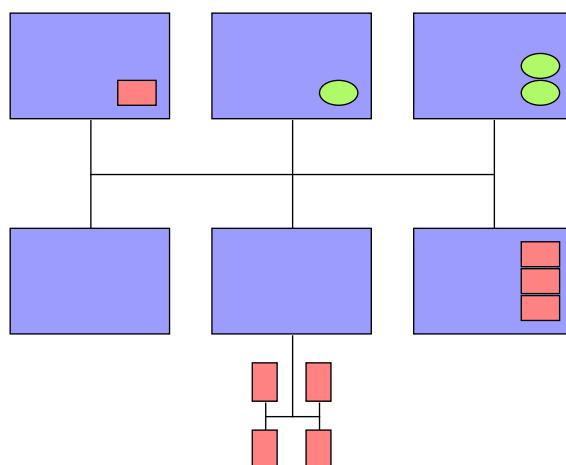


Figure 2.3: A heterogeneous cluster

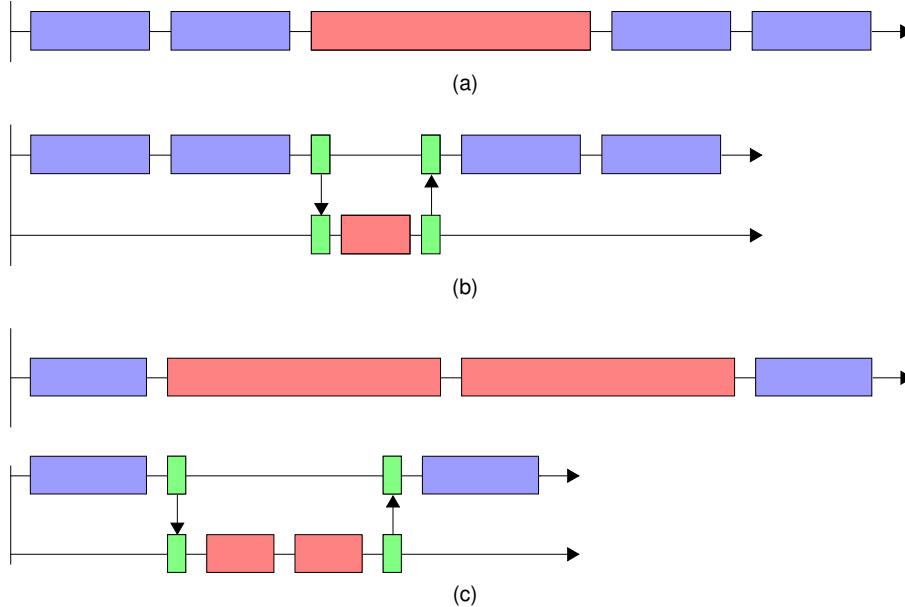


Figure 2.4: Difference between execution on the same machine (2.4a) and offload to a node that can exploit a much more efficient implementation for the same task (2.4b). Note that offloading becomes even more beneficial the more modules are offloaded (2.4c)

2.2 CMS Trigger and Data Acquisition System

- ¹⁴⁵ The Trigger and Data Acquisition system [8] of the CMS experiment is composed of two parts:
- **Level 1 Trigger**, which implements the first filter after the data acquisition and is based on FPGAs and Application-Specific Integrated Circuits (ASICs);
 - **High-Level Trigger** [2], which takes as input the events accepted by the L1 trigger and reconstructs, analyzes and filters them.
- ¹⁵⁰ As stated in [5], in 2016 the HLT runs on a cluster of commercial computers, made of ~22000 Intel Xeon cores. This cluster runs a streamlined version of CMSSW [1], the C++ offline reconstruction software, processing many events in parallel [6].

2.3 Environment: the CMSSW framework

¹⁵⁵ The cmssw framework is used in the cms experiment for the High-Level Trigger (HLT), data analysis and reconstruction. It is a module-based framework written in C++ and configured via Python scripts. It revolves around the concept of *Event*. An event is defined as a set of data resulting from an interaction, or several interactions, originated from a collision.

2.3.1 EDM: Event Data Model

¹⁶⁰ Events are processed by passing them through a sequence of modules. Every EDProducer can add data to the Event. Every module can read from the Event every object available at execution time.

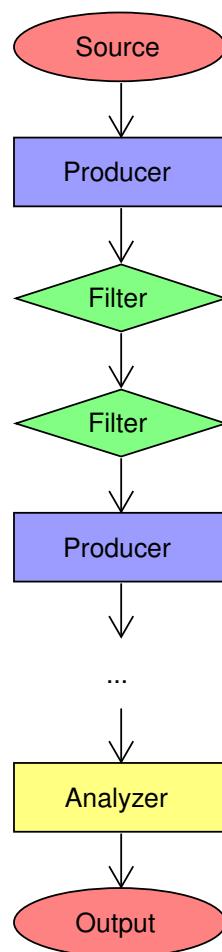


Figure 2.5: An example of a cmssw path



2.3.2 Module types

A module is a component of the CMSSW framework that encapsulates a unit of clearly-defined event-processing functionality. Different categories of modules are available:

¹⁶⁵ **Source**

Creates the Event and adds data to it.

EDProducer

Takes data from the Event in one format, elaborates it and produces data in another format to be put into the Event.

¹⁷⁰ **EDFilter**

Takes data from the Event and returns a boolean value used to determine if the processing of the current Event for the current path should continue or not.

EDAnalyzer

Takes data from the Event and analyzes it. It is not allowed to add new data to the Event.

¹⁷⁵ **2.3.3 Services**

2.3.4 Configuration files

A single executable, cmsRun, is used to execute every module. Its execution is configured via a configuration file written in Python, which defines which modules are loaded, which configurable parameters they have to use, and in which exact order they are run. This last specification is called a *path*.

2.4 The offloading model

A model and a protocol for the offloading of tasks to a different node has been studied. Three different node categories have been identified:

- ¹⁸⁵
- *Offloaders*, which execute a chain of modules, offloading on another node the ones that can be executed faster on a different architecture;
 - *Workers*, which wait for incoming tasks from any Offloader;
 - *Scheduler*, which orchestrates between the requests of Offloaders and the availability of Workers.

The term *task* is used to describe a single module or a set of modules that are offloaded to a Worker.

Each node type has a different task and behaves accordingly, coordinating with the other nodes to produce the expected result.



The *Scheduler* node is the centralized "brain" of the model: it acts as a source of information for the other nodes, ultimately deciding which node to use for a specific task. It has a mapping of all the *Worker* nodes, with the following details for each Worker:

- total number of accelerators;
- number of available accelerators at any specific moment;
- Worker's coordinates.

Upon request from an Offloader node, the Scheduler chooses a Worker for the task and sends its coordinates to the Offloader.

Workers wait for incoming messages from any Offloader. Upon receiving an incoming task description (an identifier for the task and the input data) it is in charge of finding the right modules to be executed and to manage the interactions with the chosen co-processor.

Offloaders are in charge of executing the main algorithm (the main path). When an Offloader encounters a module or a set of modules that can be executed to another node with an advantage, they are offloaded, with the modality described in 2.4.1, to a Worker node.

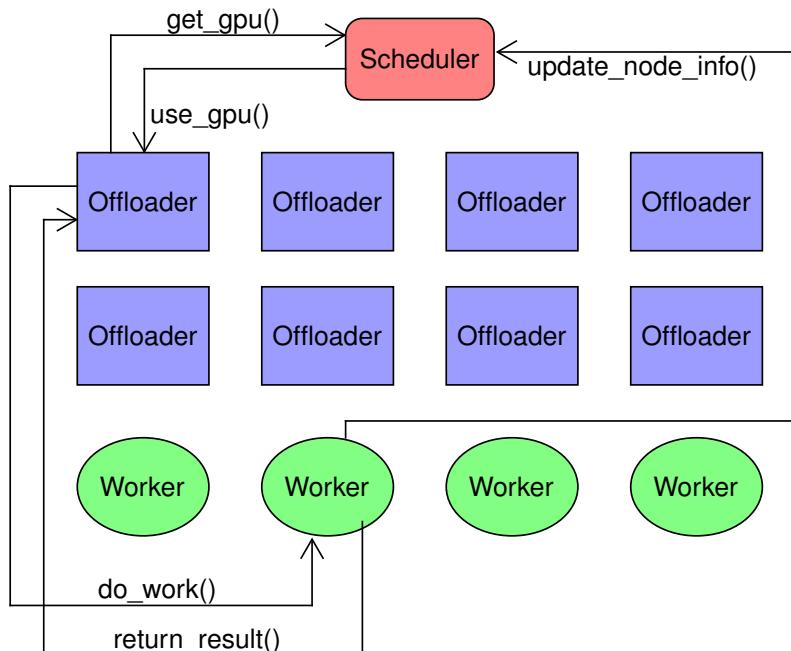


Figure 2.6: Communication protocol

2.4.1 Communication protocol

Initialization

At the program startup every Worker node must send its information (number and type of accelerators, coordinates) to the Scheduler. In this way only the Scheduler needs to be aware of the position, the type and availability of the Workers nodes.



Execution

If, during the execution of the code, an Offloader encounters a task that can be offloaded to a different architecture it performs the following steps:

- 215 1. inquire the Scheduler for a Worker node (`get_worker()`);
 2. send the task description and parameters to the chosen Worker node (`do_work()`);
 3. execute other modules while waiting for the answer to come back from the Worker node;
 4. continue the execution of the workload.

Upon reception of a message from an Offloader, a Worker node performs the following steps:

- 220 1. retrieve the set of modules associated with the task description;
 2. offload the computation on the chosen co-processor;
 3. return the results of the computation to the Offloader that requested it.

Termination

225 Both the Scheduler and the Workers are executing an infinite loop and must be stopped by an external condition to close the program gracefully. All Offloaders get synchronized to make sure that every task is terminated and one Offloader, designated as a master node, send a termination message to every Worker and the Scheduler.



3. Implementation

3.1 MPI

²³⁰ MPI [9] is a standard to facilitate message passing between processes. It is the industry-standard specification for writing message passing programs.

There are several implementations of this standard, for example MPICH [4] and OpenMPI [3]. For the purpose of this document we will reference MPI version 3 and the OpenMPI implementation.

3.2 Intel TBB

²³⁵ TBB [7] is a C++ library developed by Intel for enabling parallelism in C++ applications and libraries. It is designed, in particular, for applications that are loop- or task-based. CMSSW uses TBB internally.

3.3 Proof of Concept

²⁴⁰ As a first stage, the communication protocol has been implemented and tested in a standalone C++ program, using MPI for the interprocess communication and Intel TBB for creating and handling multithreading.

It is available at github.com/sere/the_offloadinator. It is compiled using `mpicc -ltbb -g the_offloadinator.cpp -o the_offloadinator` and run with `mpirun -n N the_offloadinator`, with N defined as an arbitrary value greater than 3.

²⁴⁵ It features 3 classes, one for each kind of node described in Section 2.4: Scheduler, Worker and Offloader.

Only one Scheduler is instantiated while Offloaders and Workers are instantiated on multiple processes (using MPI) and on multiple threads (using TBB).

The MPI id `mpi_id` determines the nature of the node (n = number of nodes):

- ²⁵⁰
- Workers belong to the range [0, `MPI_FIRST_OFF` - 1];
 - Offloaders to the range [`MPI_FIRST_OFF`, n - 2];
 - the Scheduler is node n - 1.

The value for `MPI_FIRST_OFF` can be changed programmatically, making sure to have at least one Offloader and one Worker. It is set to n/2.



²⁵⁵ Four threads are created on each Offloader node, while two threads are created on each Worker node. These numbers are non-binding and can be changed programmatically.

Every thread has its private Offloader/Worker class instance and runs the function `offload()`/`work()`

3.3.1 Scheduler

²⁶⁰ The Scheduler class manage the MPI requests for nodes in the method `schedule()`.

Initialization

The Scheduler class is initialized with only one parameter: the number of Workers available in the system.

²⁶⁵ During initialization the Scheduler receives, from every Worker present in the system, the information about the Worker itself: MPI rank and number of co-processor available on the Worker node. These are stored into a map, with the key being the MPI rank.

schedule()

This method waits indefinitely on incoming MPI messages from any source and with any tag. Three kinds of message are possible:

- ²⁷⁰ • a request from an Offloader for a Worker, with tag `SCHED_REQ + mpi_id`;
- updated information about a Worker, with tag `SCHED_UPD + mpi_id`;
- termination message, with tag `SCHED_DIE = mpi_id`.

The `mpi_id` has been added to the MPI tag to facilitate debugging and traceability.

3.3.2 Offloader

²⁷⁵ To simplify the proof of concept, every Offloader executes only one task, then exits. Tasks can be offloadable or not. In the first case, the Offloader instance asks the Scheduler for a Worker and proceeds with sending the task to this Worker. If no Worker is available or the task is not offloadable, this will be executed directly by the Offloader.

3.3.3 Worker

²⁸⁰ Workers wait for incoming messages from any Offloader. Upon reception of a message, the counter `gpu_available_` is decremented and the task is executed. After execution, the number of available co-processors is updated again and a message is sent to the Scheduler with the state of the Worker.

3.4 MPICore

²⁸⁵ MPICore is a first implementation of the offloading mechanism inside CMSSW. It is structured as 6 modules, 3 for the Offloader implementation and 3 for the Worker one.

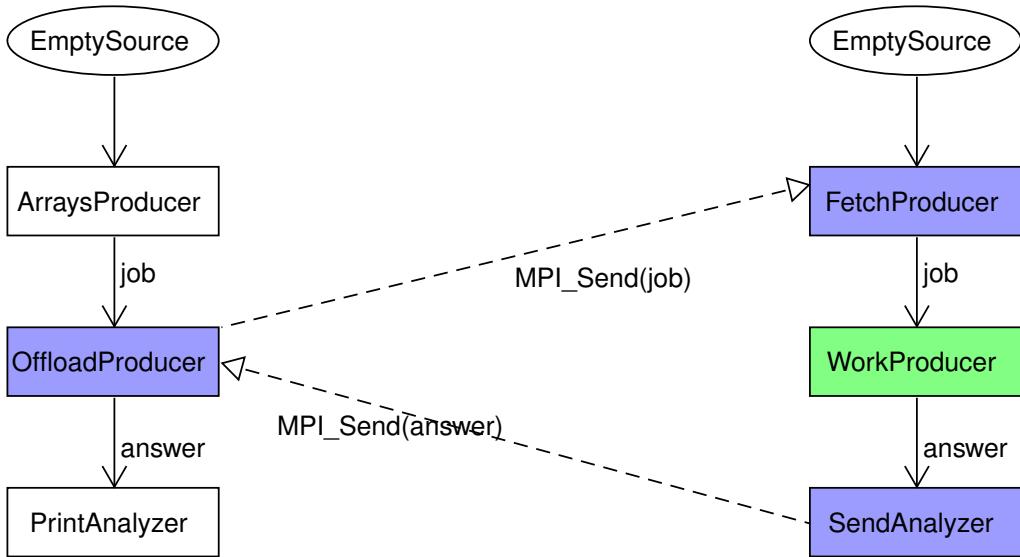


Figure 3.1: The two paths representing an Offloader (left side) and a Worker (right side). Modules in blue are aware of the MPI communication, while modules in green communicate directly with the GPU.

Offloaders and Workers are run by independent cmsRun commands but both under the same mpirun command, with this format:

290 `mpirun -n N cmsRun MPIWorker.py <params> : -n M cmsRun MPIOffloader.py <params>`

The syntax `-n N cmsRun executable_1 : -n M executable_2` is used to run both executables under the same MPI communicator. The code performs a root of sum squares between two vectors. The two vectors are built in the Offloader, sent via MPI_Send to the Worker and the result is sent back to the Offloader to simulate a complete interaction.

295 Every module belongs to the interface `edm::stream`. Using this interface every Stream is associated with its own instance of a Stream module. This is needed to allow for true parallelism between different Events. As modules in MPICore don't need to coordinate between different streams, this interface can be used.

3.4.1 Initialization

300 MPI is initialized with `MPI_Init_thread` with `MPI_THREAD_MULTIPLE` as the required level of thread support. This is needed because the MPICore uses Streams and therefore it's multithreaded. `MPI_THREAD_MULTIPLE` allows MPI calls by any thread, with no restriction. This, in conjunction with the use of the interface `edm::stream`, allows the full exploitation of multithreading.

305 The initialization code has been written in a new Service called `MPIService`.

3.4.2 Offloader

Offloader nodes are configured in `MPIOffloader.py`, which takes the following command-line parameters:



- vlen, the length of the vector (default: 1)
- maxEvents, the number of events to run (default: -1)

310 Multiple instances of it can be executed on different cluster nodes. The number of nodes is configured via the -n parameter of mpirun.

It is composed by the following 3 modules:

ArraysProducer

315 Subclass of edm::stream::EDProducer.

Create a vector of $2 * \text{vecLen}$ elements, initialize it with random `double` elements and put it into the Event.

OffloadProducer

Subclass of edm::stream::EDProducer, handles the communication with the Worker.

320 Consume the `std::vector<double>` created by ArrayProducer, send its raw data as a serialized object of type `char[]` to a Worker via MPI_Send, receive the result of the computation, a serialized object of type `char[]`, from the same Worker via MPI_Mprobe and MPI_Mrecv and produce this array as `std::vector<double>`.

```
64     MPI_Ssend(buffer.data(), buffer.size(),
65                 MPI_CHAR, workerPE, WORKTAG + mpiID,
66                 MPI_COMM_WORLD);
81     MPI_Mprobe(workerPE, WORKTAG + mpiID, MPI_COMM_WORLD, &message, &
82                 status);
83
88     int size;
84     MPI_Get_count(&status, MPI_CHAR, &size);
85     io::unique_buffer write_buffer(size);
86     MPI_Mrecv(write_buffer.data(), size, MPI_CHAR, &message, &status);
```

Listing 1: OffloadProducer

PrintAnalyzer

335 Subclass of edm::stream::EDAnalyzer.

Consume the result of the computation and, if debugging is on, print it.

3.4.3 Worker path

Worker nodes are configured in MPIWorker.py. They take the following command-line parameters:

- streams, the number of CMSSW Streams to use (default: 1)
- maxEvents, the number of events to run. To allow the program to terminate cleanly, it must be equal to the number of Offloaders times the number of events for the Offloaders. (default: -1)



- 345 • runOnGpu. If this boolean is True the task will be run on the GPU using Compute Unified
Device Architecture (CUDA), otherwise it will be run on the CPU.

It is composed by the following 3 modules:

FetchProducer

Subclass of `edm::stream::EDProducer`.

- 350 Receive a serialized vector of type `char[]` from a Producer via `MPI_Mprobe` and `MPI_Mrecv`,
transform it into a `std::vector<double>` and produce it.

```
49     MPI_Mprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &message,  
50                 &status);  
53     MPI_Get_count(&status, MPI_CHAR, &size);  
54  
55     std::map<std::string, double> times;  
56     io::unique_buffer write_buffer(size);  
57     MPI_Mrecv(write_buffer.data(), size, MPI_CHAR, &message, &status);
```

Listing 2: FetchProducer

WorkProducer

Subclass of `edm::stream::EDProducer`.

- 360 Consume the `std::vector<double>` gathered by FetchProducer, offload the task on the
GPU if `runOnGPU` is True else run it on the CPU and produce the result of the computation as a
`std::vector<double>`.

Listing 3: WorkProducer

SendAnalyzer

Subclass of `edm::stream::EDAnalyzer`.

- 365 Consume an `std::vector<double>`, send it serialized as a `char[]` to the same Producer via
`MPI_Send`.

```
64     MPI_Ssend(buffer.data(), buffer.size(),  
65                 MPI_CHAR, *offloaderIDHandle, *mpiTagHandle,  
66                 MPI_COMM_WORLD);
```

Listing 4: SendAnalyzer



4. Results

370

4.1 Test environment

The machine used to run the tests features:

- MPI version 3, OpenMPI version 2.1.5
- CentOS Linux release 7.5.1804
- CMSSW 10_2_4_Patatrack
- 375 • CPU: Intel Xeon ES-2650 with 2 thread per core, 8 core per socket and 2 socket
- GPU: Tesla K20Xm

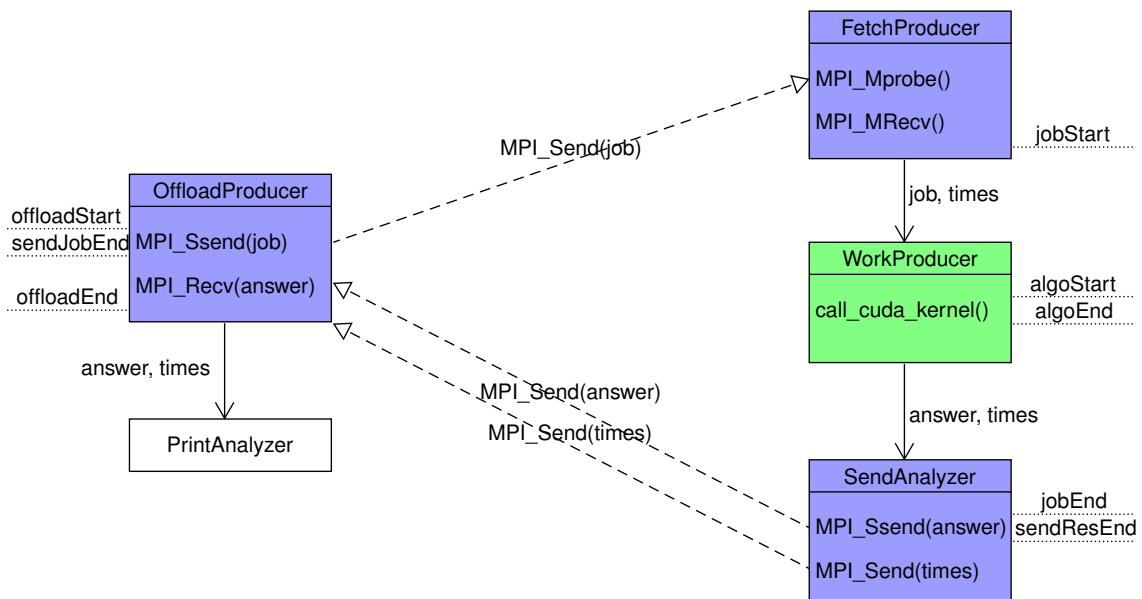


Figure 4.1: The probe points inserted in MPICore

A number of different probe points were set up to analyze the behaviour of the framework in different places:

380

- *offload time*, the amount of time spent on the whole offloading, from `offloadStart` to `offloadEnd`;
- *sendData*, the amount of time used to send the arrays to the Worker. It goes from `offloadStart` to `sendJobEnd`;



- 385
 - *job time*, the amount of time used by the Worker upon receiving the request and until the shipping of the response. It goes from jobStart to jobEnd;
 - *algorithm time*, the amount of time spent in the algorithm execution. It goes from algoStart to algoEnd;
 - *sendRes*, the amount of time used to send the results to the Offloader. It goes from jobEnd to sendResEnd.

390 A number of derived timing were created:

- *overhead*. It is defined as offload time - algorithm time.
- *job_time*. It is defined as job time - algorithm time

Tests have been made varying the vector length and the device type.

4.2 Overhead

395 Figure 4.2a depicts the raw overhead created by the offloading code. Running the code on a GPU has an additional initialization overhead, even though an attempt has been made to exclude initialization costs from the metric. This becomes negligible with arrays of size greater than 100000, which is a more likely use-case.

400 Increasing the vector length makes the overhead grow in absolute terms, but the cost per vector element shrinks considerably.

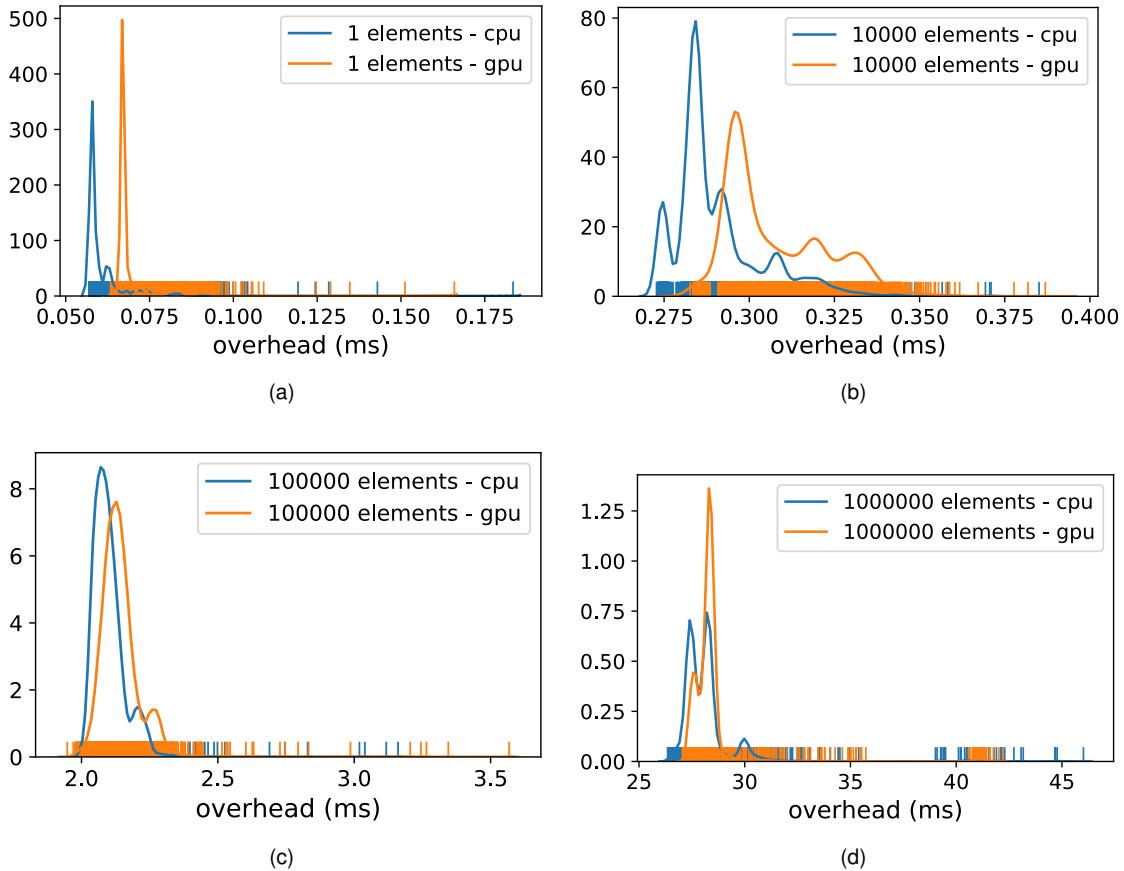


Figure 4.2: Distribution plot of overhead for different vector lengths

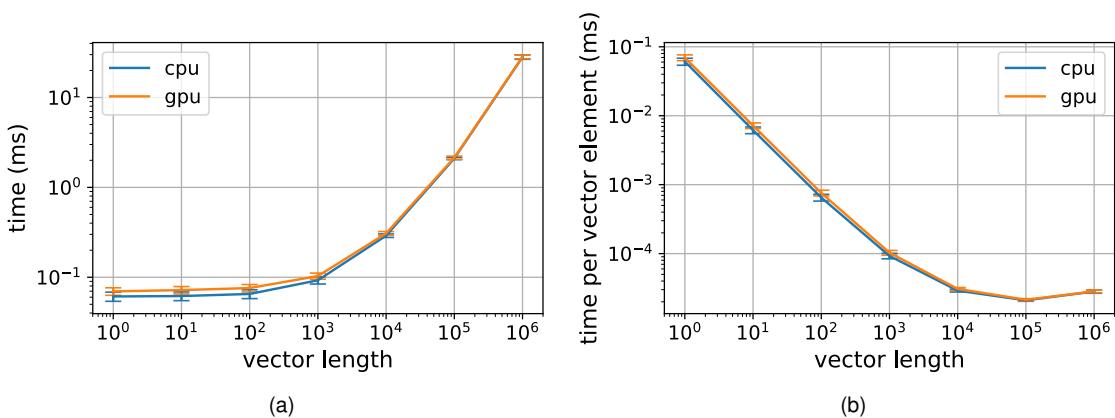


Figure 4.3: Overhead mean for 1 offloader, 1 worker for each vector length, absolute (4.3a) and relative to vector length (4.3b)



5. Conclusions

The use of MPI has been proven a viable and efficient way to offload tasks on another node. The infrastructure and protocol has been defined and its stability has been assessed. The benchmarks have shown that the overhead that MPI introduces is reasonable. However, the decision to offload or not a certain task cannot be based only on the existence of tasks with a co-processor implementation. but it must take into account both the network latency and the overhead caused by MPI.
405



Bibliography

- [1] CMS Physics TDR, Volume 1, CERN-LHCC-2006-001, 2 February 2006. 4
- 410 [2] CMS collaboration et al. The CMS high level trigger. *The European Physical Journal C-Particles and Fields*, 46(3):605–667, 2006. 4
- [3] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004. 9
- 415 [4] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing*, 22(6):789–828, 1996. 9
- 420 [5] Felice Pantaleo. *New Track Seeding Techniques for the CMS Experiment*. PhD thesis, CERN, 2017. 4
- [6] Andrea Perrotta. Performance of the CMS high level trigger. In *Journal of Physics: Conference Series*, volume 664, page 082044. IOP Publishing, 2015. 4
- 425 [7] Chuck Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008. 9
- [8] Attila Rácz and Paris Sphicas. CMS The TriDAS Project: Technical Design Report, Volume 2: Data Acquisition and High-Level Trigger. Technical report, CMS-TDR-006, 2002. 4
- 430 [9] David W Walker and Jack J Dongarra. MPI: A standard message passing interface. *Supercomputer*, 12:56–68, 1996. 9