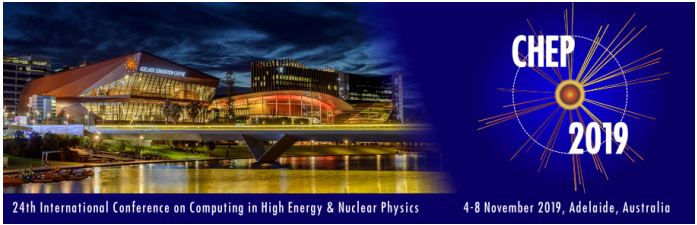
Running synchronous detector reconstruction in ALICE using declarative workflows

Matthias Richter
for the ALICE collaboration

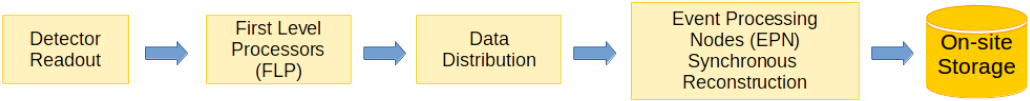
Department of Physics, University of Oslo
CERN - European Organization for Nuclear Research



Online Data Processing in ALICE Run 3 and beyond in a nutshell

Inspect all Pb-Pb collisions at min bias rate of 50 kHz to provide access to rare physics probes.
Continuous processing of data stream of 3.4 TByte/s. No limiting hardware trigger

A logical view of the O^2 online data flow: **synchronous reconstruction**



To cope with the data rate: Detector reconstruction runs on-line in the synchronous stage to provide preprocessed results and compressed/filtered data for intermediate storage.

see Plenary Tue 9:30 ALICE continuous readout and data reduction strategy for Run3

- Distributed, multi-process system with $\sim$ 1800 nodes
- Individual computing tasks are implemented as processes - "**Devices**"
- Processes exchange data entirely via **Messages** using **FairMQ*** as unified data transport

* see Track 5 Mon 11:30 M. Al-Turany: ALFA: A framework for building distributed applications

Combining Online and Offline worlds

The goal: run a large part of the detector reconstruction on-line in the synchronous stage

We need to merge two different domains:

Online

distributed system with all kind of synchronization and communication

Offline

often single-process and for development run-on-your-laptop-approach

▷ maybe we do not need to merge but clearly separate and simplify the way how to integrate "offline habits"

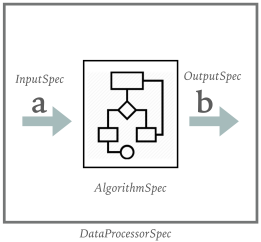
- Detector reconstruction can (should) be agnostic in the actual way the data are provided
- FairMQ as bare transport layer is the foundation for transport
- Boiler plate code supporting O^2 data model and device connectivity to be implemented.
  ▷ exponentially increasing effort when going beyond "hello world" examples

How to simplify the integration of components from the offline world into the online system?
How to simplify the efficient exchange of data?

Workflow-oriented definition of the compute topology

On top of FairMQ as transport layer and the O^2 data model as data layer, a third software layer, the **Data Processing Layer (DPL)** was introduced

- The basic building blocks of DPL workflows are `DataProcessors` defined as entities with **inputs**, an **algorithm**, and **outputs**
- Workflows combine/chain individual DataProcessors
- Multiple workflows can be combined into one workflow



The description is **declarative**: The user describes *what* to achieve in terms of process connectivity and algorithm, the framework takes care of *how* to realize the workflow and the connections.

Broken down to the essentials: What DPL does for the user

- Launching of individual devices or generation of configuration for scheduling/control
- Connectivity: definition of channels between processes with matching outputs and inputs
- Cachelines for synchronization of multiple inputs
- Dispatching of computation once data set is complete according to configurable completion policy
- Input and Output API to easily send and receive data
- Flexible dispatching of output data according to configurable dispatch policy (i.e. all together or as-soon-as-ready)

$\Rightarrow$ the user does not need to care about the complex online system

Running Detector Reconstruction in DPL Workflows ...

... just a few steps

- Inputs and Output are specified in the three coordinates of the O^2 Data Model: *data origin*, *data description* and *sub-specification*, e.g.
- The algorithm can be either state-full or state-less depending on whether e.g. a pre-initialized worker class is invoked
- Optionally, an algorithm initialization can be invoked by DPL in the init phase of the device
- Optional callback for e.g. idle handling or stopping/finishing the device can be registered
- Command line options can be defined and parsing is available in the initialization function
- The underlying transport is flexible and can be shared memory or go over the wire

DataProcessor implementations use one simple, coherent C++ I/O API

API supports different types transparently through implementations of template specializations

- Output:
  - POD data structures, vectors of POD and trivially copyable types
  - ROOT serialization, BOOST serialization
  - Two methods: `make` creates a framework owned object, which can be changed; `snapshot` creates immediate copy

```
template <typename T, typename... Args>
decltype(auto) make(const Output& spec, Args... args);

template <typename T>
void snapshot(const Output& spec, T const& object);
```

e.g.

```
auto& data = outputs().make<std::vector<int>>(...);
data.push_back(42); // sent when computation ends

TH1F* h = ...;
outputs().snapshot({"QA", "HISTOGRAM", 0}, *h);
```

- Input
  - Serialization method is propagated as part of the data annotation (`DataHeader`),

```
template <typename T>
decltype(auto) get(char const* binding) const;
```

e.g.

```
auto object = inputs().get<std::vector<int>>("input1");
```

Container support with polymorphic allocators

Every std container has an underlying allocator, the allocator type as template argument fixes the container type, containers with different allocators can not be converted into each other. Using a polymorphic allocator will fix the container type to this, but the underlying memory resource is flexible.

- The I/O API supports polymorphic allocators
- Polymorphic allocators allow to use different types of underlying memory resources without affecting the container type
- Specific `FairMQMemoryResource` allows allocation directly in the message memory
- Resizable data chunks and standard containers directly in the message memory: at the same time zero copy and standard use of containers in algorithms
- Zero-copy provision of read-only standard containers for input data

Optimization

Intrinsic features of this solution:

- Full separation of algorithm and data transport
- Quick prototypes of detector reconstruction
- Supporting single-machine developer use case and distributed systems at the same time

▷ Decoupled domains: Experts for each domain can work independently on the optimization.

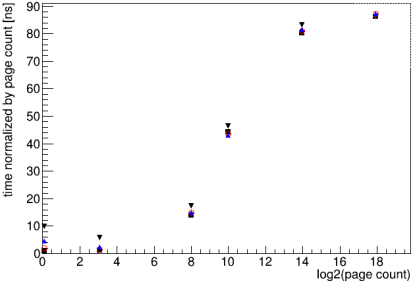
- Optimization of the algorithm itself
- Optimization of serialization and transport of data structures
- Optimization of online data flow

Raw Data Access - Example for tooling

- Data written in *raw page format* into the FLP shared memory
- Each *raw page* starts with the `RAWDataHeader (RDH)`
- Raw pages have fixed size of 8 kB
- Concatenated 8k pages, size of individual pages described in RDH
- Evolution of the RDH requires support of multiple versions

Providing a generic parser for raw pages for multiple header versions using std::variant of multiple concrete implementations for the different header versions

Benchmark of parser implementation, normalized to page count



- ■ direct use of a concrete parser implementation
- ○ parser implementation with iterative element access
- ▲ parser implementation with `std::visit` element access
- ▼ virtual inheritance implementation

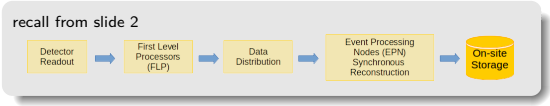
▷ we can provide a generic, version-transparent solution with performance close to single-version parser

▷ for large number of pages, other effects dominate, under investigation

Integration into the O^2 Data Flow

Synchronous reconstruction is running on Event Processing Nodes, data are provided by DataDistribution,

⇒ Track 1 Tue 12:00 G. Neskovic



Emphasis on modularity:

▷ Connection is as easy as running a proxy process for a workflow configuration

Examples:

A process listening to configured channel for specified data and establishing link to reconstruction workflows:

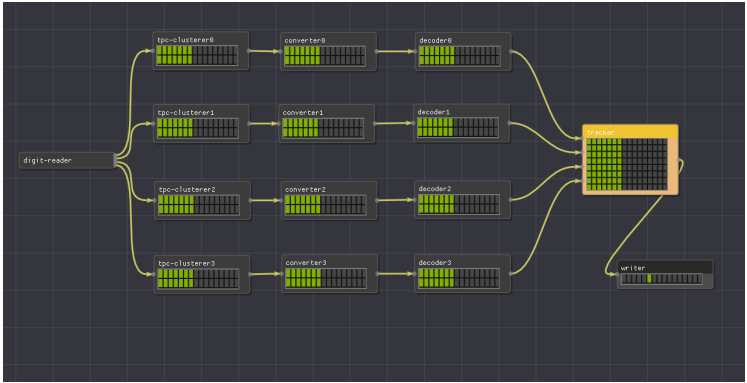
```
o2-dpl-raw-proxy --dataspec "O:FLP/RAWDATA" \\
                 --channel-config "name=readout-proxy,type=pair,method=bind,address=ipc:///tmp/stf-builder-dpl-pipe-0,transport=shmem"
```

Connecting the proxy with e.g. TPC reconstruction workflow:

```
o2-dpl-raw-proxy --dataspec "O:TPC/RAWDATA" --channel-config ... | o2-tpc-reconstruction-workflow
```

Example workflows

Prototype workflows have been implemented e.g reconstruction workflows for TPC, ITS, TOF

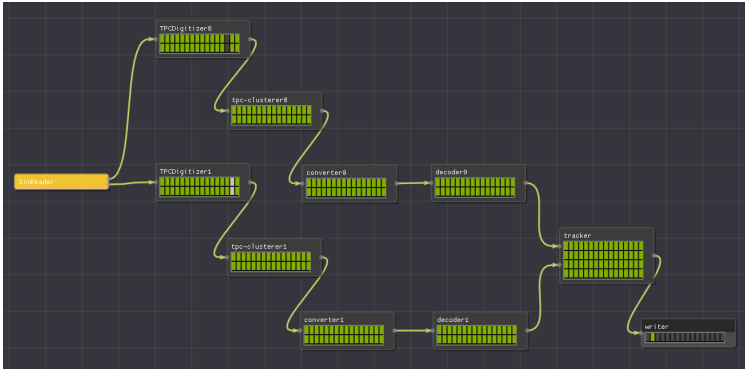


Screenshot of the TPC reconstruction workflow with parallel pipelines

Example workflows

Further prototypes have been implemented for the detector digitization

TPC digitization and connected reconstruction workflow with parallel pipelines



Conclusions

- Alice O^2 uses the FairMQ transport framework and message queue communication to distribute workload among many processes running on multiple compute nodes.
- Using the Data Processing Layer we define computational workflows in a declarative approach
- This approach separates the actual computation (typically "offline") from all the adaption to the online system
- We have many detector workflows now, following our approach of fostering quick prototypes, we are now in the process of optimization, merging and simplification
- Next focus: integration - connecting the reconstruction workflows to DataDistribution in the real system