MICE Online Data Quality *Journal of Physics: Conference Series*

M. Jackson (EPCC)[1] **and C.D. Tunnell (U. Oxford)**

E-mail: `michaelj@epcc.ed.ac.uk, c.tunnell1@physics.ox.ac.uk`

Abstract. Within the Muon Ionization Cooling Experiment (MICE), the MICE Analysis User Software (MAUS) framework performs both online analysis of live data and detailed offline data analysis, simulation, and accelerator design. The MAUS Map-Reduce-inspired API parallelizes computing in the control room, ensures that code can be run both offline and online, and displays plots for users in an easily extendable manner. The classic Map-Reduce design can be advantageous for offline computing but cannot be used in online settings. It expects all map operations to terminate before running the reduction; however, the data flow for online analysis requires the continuous updating of live plots as data arrives. For online running, the "map" and "reduce", which we call transform and merge, steps must happen concurrently; therefore, new parallelisation routines were developed specifically for this use. The transform step is parallelized using a Python-based distributed task queue called Celery, and output from these tasks is then written into the NoSQL database CouchDB. As the transformer writes output, the plotting mergers query the database, request data from a user-specified window in time, and make plots using Matplotlib or PyRoot. The mergers serialize the plots into the data stream after which all the data is written to the database by the output routines. Finally, plots are displayed on the web using the Django platform, which queries the database and displays the plots to the control room and the world. By maintaining the API and modifying the data flow, MICE is able to use identical analysis software in both offline and online scenarios, thus avoiding a common issue in experimental particle physics.

1. Background

The MICE experiment is based at the Rutherford Appleton Laboratory and is an R&D experiment intending to reduce the phase space of muon beams. This accelerator physics R&D is a muon linear accelerator and will help determine the feasibility of muon colliders and neutrino factories towards probing new physics in the post-LHC era. The experiment includes a *cooling channel* that performs the phase space reduction, scintillator fibre detectors upstream and downstream of the cooling channel, as well as upstream time-of-flight and Cherenkov detectors and a downstream totally active scintillator calorimeter.

In addition to the accelerator physics equipment, there has been considerable investment into detectors to measure the manipulated muon beam. There are four detector technologies across eight detectors which provides a considerable software complexity problem: the number of technologies and detectors – by corollary, the number of calibrations and reconstruction

[1] We would like to acknowledge the assistance of the Software Sustainability Institute. The work carried out by the SSI is supported by EPSRC through grant EP/H043160/1.

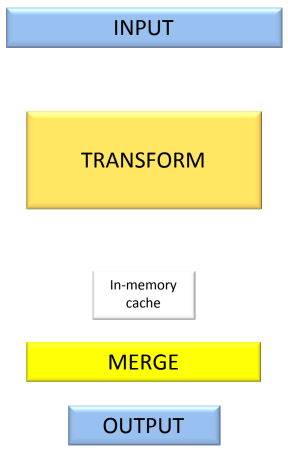

Figure 1. The generic data flow within MAUS.

algorithms – is comparable to a collider physics experiment like ATLAS however the size of the experiment and collaboration closer resembles a smaller neutrino experiment like MiniBooNe.

2. Introduction

This software challenge is addressed by the MICE Analysis User Software (MAUS) software [1, 2], which is a Python-driven application that uses SWIG to interface with various C++ components. The MAUS framework executes four types of component (Fig. 1). Each component creates or manipulates spills. In this context, a spill is a JSON document representing information about a collection of trigger events.

- Inputters - these input or generate spills e.g. read live DAQ data and convert into spills, read archived DAQ data and convert into spills, or generate simulated spill data.
- Transforms - these analyse the data within the spill in various ways, derive new data and add this to the spill.
- Mergers - these summarise data within a series of successive spills and output new spills with the summarised data.
- Outputters - these take in merged spills and output spills, for example saving them as JSON files or, where applicable, extracting image data from them and saving this as image files.

The framework repeats the following loop until there are no more spills:

(i) Reads a spill from an inputter.
(ii) Transforms the spill.
(iii) Passes the transformed spill to a merger.
(iv) Passes a merged spill to an outputter.

3. Parallel transformation of spills

The MAUS work flow is able to be used for Online Data Quality (DQ) purposes (Fig. 2), where the status of the experiment and quality of the data needs to be quickly and continuously processed and displayed. However, parallelisation is required to meet the speed requirements of this task.

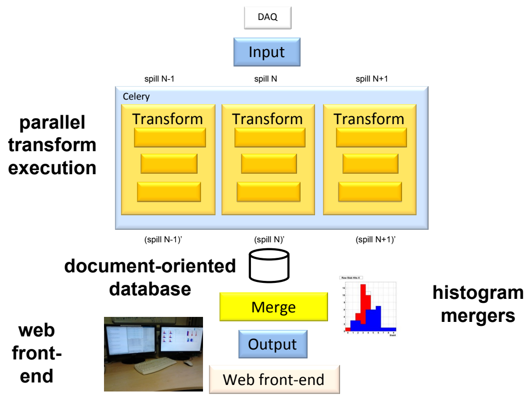

Figure 2. Details of the DQ data flow with distributed computing and a web interface.

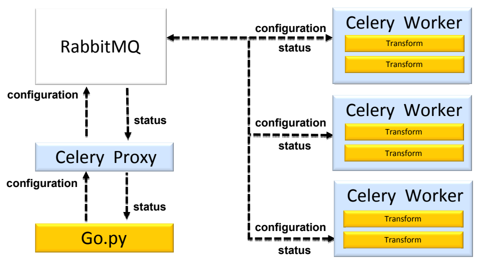

Figure 3. The configuration procedure for worker nodes. Configuration information includes the run number, calibration constants, and the choice of reconstruction algorithm.

As each spill is independent, spills can be transformed in parallel. Celery [3], a distributed asynchronous task queue for Python is used to implement parallel transformation of spills. Celery uses RabbitMQ [4] as a message broker to handle communications between clients and Celery worker nodes.

A Celery worker executes transforms. When a worker starts up it registers with RabbitMQ – the RabbitMQ task broker can be local or remote. By default Celery spawns N sub-processes where N is the number of CPUs on a node, though N can be explicitly set by a user. Each sub-process can execute one transform at a time. Multiple Celery workers can be deployed, each of which with one or more sub-processes. Celery allows MAUS to execute highly-parallelised transforming of spills.

For MAUS, each Celery worker needs a complete MAUS deployment running on the same node as the worker. The MAUS distributed execution framework, configures Celery as follows (Fig. 3):

- The framework gets the names of the transforms the user wants to apply e.g. a MapPyGroup containing MapPyBeamMaker, MapCppSimulation, and MapCppTrackerDigitization. This is termed a transform specification.
- A Celery broadcast is invoked, passing the transform specification, the MAUS configuration and a configuration ID (e.g. the client's process ID).

- Celery broadcasts are received by all Celery workers registered with the RabbitMQ message broker.
- On receipt of the broadcast, each Celery worker:
  Checks that the framework's MAUS version is the same as the worker's. If not then an error is returned to the client.
  Forces the transform specification down to each sub-process.
  Waits for the sub-processes to confirm receipt.
  If all sub-processes update correctly then a success message is returned to the framework.
  If any sub-process fails to update then a failure message, with details, is returned to the framework.
- Each Celery sub-process:
  Invokes *death()* on the existing transforms, to allow for clean-up to be done.
  Updates their configuration.
  Creates new transforms as specified in the transform configuration.
  Invokes *birth()* on these with the new configuration.
  Confirms with the Celery worker that the update has been done.
- Celery workers and sub-processes catch any exceptions they can to avoid the sub-processes or, more seriously, the Celery worker itself from crashing in an unexpected way.

MAUS uses Celery to transform spills as follows:

- The framework gets the next spill from its input.
- A Celery client-side proxy is used to submit the spill to Celery. It gets an object which it can use to poll the status of the "job".
- The client-side proxy forwards the spill to RabbitMQ.
- RabbitMQ forwards this to an available Celery worker. If none are available then the job is queued.
- The Celery worker picks an available sub-process.
- The sub-process executes the current transform on the spill.
- The result spill is returned to the Celery worker and there back to RabbitMQ.
- The framework regularly polls the status of the transform job until it's status is successful, in which case the result spill is available, or failed, in which case the error is recorded but execution continues.

4. Document-oriented database

After spills have been transformed, a document-oriented database, MongoDB [5], is used by the framework to store the transformed spills. This database represents the interface between the input-transform and merge-output phases of a spill processing workflow.

The framework is given the name of a collection of spills and reads these in order of the dates and times they were added to the database. It passes each spill to a merger and then takes the output of the merger and passes it to an outputter.

Use of a database allows the input-transform part of a workflow to be separate from the merge-output part. It also allows them to operate in concurrently – one process can input and transform spills, another can merge transformed spills and output the merged results. This also allows many merge-output workflows to use the same transformed data, for example to generate multiple types of histogram from the same data.

5. Histogram mergers

Histogram mergers take in spills and, from the data within the spills, update histograms. They regularly output one or more histograms (either on a spill-by-spill basis or every N spills, where N is configurable). The histogram is output in the form of a JSON document which includes:

- A list of keywords.
- A description of the histogram.
- A tag which can be used to name a file when the histogram is saved. The tags can also be auto-numbered if the user wants.
- An image type e.g. EPS, PNG, JPG, or PDF. The image type is selected by the user.
- The image data itself in a base64-encoded format.

Histogram mergers do not display or save the histograms. That is the responsibility of other components.

Example histogram mergers, and generic super-classes to build these, currently exist for histograms drawn using PyROOT [6] (*ReducePyTOFPlot* and *ReducePyROOTHistogram*) and matplotlib [7](*ReducePyHistogramTDCADCCounts* and *ReducePyMatplotlibHistogram*).

6. Saving images

An outputter (*OutputPyImage*) allows the JSON documents output by histogram mergers to be saved. The user can specify the directory where the images are saved and a file name prefix for the files. The tag in the JSON document is also used to create a file name.

The outputter extracts the base-64 encoded image data, unencodes it and saves it in a file. It also saves the JSON document (minus the image data) in an associated meta-data file.

7. Web front-end

The web front-end allows histogram images to be viewed. The web front-end is implemented in Django [8], a Python web framework. Django ships with its own lightweight web server or can be run under Apache web server.

The web front-end serves up histogram images from a directory and supports keyword-based searches for images whose file names contain those key words.

The web pages dynamically refresh so updated images deposited into the image directory can be automatically presented to users.

The interface between the online reconstruction framework and the web front-end is just a set of image files and their accompanying JSON meta-data documents (though the web front-end can also render images without any accompanying JSON meta-data).

8. Design details

The processing for an individual spill can be seen in Fig. 4.

8.1. Run numbers

Each spill will be part of a run and have an associated run number. Run numbers are assumed to be as follows:

- $-N$: Monte Carlo simulation of run N
- 0 : pure Monte Carlo simulation
- $+N$: run N

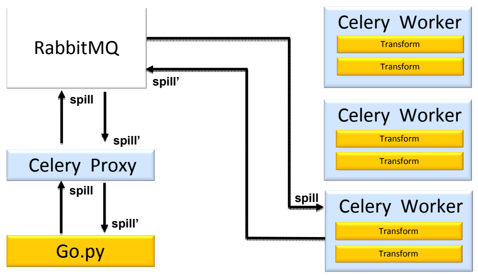

Figure 4. The processing steps for an individual spill (or rather collection of events). The spill will start at Go.py where Go.py will use the Python Celery API to send the spill for processing. Celery uses RabbitMQ to communicate with worker nodes and distribute the spill to a worker node. At the worker node, the spill will be processed before being sent back to RabbitMQ then Go.py.

8.2. Transforming spills from an input stream (Input-Transform)

This is the algorithm used to transform spills from an input:

```
CLEAR document store
run_number = None
WHILE an input spill is available
  GET next spill
  IF spill does not have a run number
    # Assume pure MC
    spill_run_number = 0
  IF (spill_run_number != run_number)
    # We've changed run.
    IF spill is NOT a start_of_run spill
      WARN user of missing start_of_run spill
    WAIT for current Celery tasks to complete
      WRITE result spills to document store
    run_number = spill_run_number
    CONFIGURE Celery by DEATHing current transforms and BIRTHing new transforms
  TRANSFORM spill using Celery
  WRITE result spill to document store
 DEATH Celery worker transforms
```

If there is no initial *start_of_run* spill (or no *spill_num* in the spills) in the input stream (as can occur when using *simple_histogram_example.py* or *simulate_mice.py*) then *spill_run_number* will be 0, *run_number* will be None and a Celery configuration will be done before the first spill needs to be transformed.

Spills are inserted into the document store in the order of their return from Celery workers. This may not be in synch with the order in which they were originally read from the input stream.

8.3. Merging spills and passing results to an output stream (Merge-Output)

This is the algorithm used to merge spills and pass the results to an output stream:

```
run_number = None
end_of_run = None
is_birthed = FALSE
last_time = 01/01/1970
WHILE TRUE
  READ spills added since last time from document store
  IF spill IS ``end_of_run''
    end_of_run = spill
  IF spill_run_number != run_number
    IF is_birthed
      IF end_of_run == None
          end_of_run = {``daq_event_type'':''end_of_run'', ``run_num'':run_number}
      Send end_of_run to merger
      DEATH merger and outputter
    BIRTH merger and outputter
    run_number = spill_run_number
    end_of_run = None
    is_birthed = TRUE
  MERGE and OUTPUT spill
Send END_OF_RUN block to merger
DEATH merger and outputter
```

The Input-Transform policy of waiting for the processing of spills from a run to complete before starting processing spills from a new run means that all spills from run N-1 are guaranteed to have a time stamp ¡ spills from run N.

is_birthed is used to ensure that there is no BIRTH-DEATH-BIRTH redundancy on receipt of the first spill from the document store.

8.4. Document store

Spills are stored in documents in a collection in the document store.

Documents are of form:

```
{``_id'':ID, ``date'':DATE, ``doc'':SPILL}
```

where:

- ID: index of this document in the chain of those successfully transformed. It has no significance beyond being unique in an execution of the Input-Transform loop which deposits the spill. It is not equal to the *spill_num* (Python string)
- DATE: date and time to the milli-second noting when the document was added. A Python timestamp.
- DOC: spill document. A Python string holding a valid JSON document.

8.4.1. Collection names For Input-Transform,

- If configuration parameter *doc_collection_name* is None, an empty string, or “auto” then *HOSTNAME_PID*, where *HOSTNAME* is the machine name and *PID* the process ID, is used.
- Otherwise the value of *doc_collection_name* is used.
- *doc_collection_name* has default value “spills”.

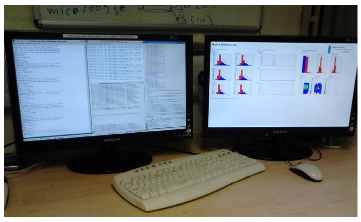

Figure 5. A photograph of the MAUS DQ running in parallel in the control room. On the left screen are terminals for monitoring the various processes needed to run the DQ application. On the right screen is a full-screen web browser that updates periodically with new plots.

For Merge-Output,

- If configuration parameter *doc_collection_name* is None, the empty string, or undefined then an error is raised.
- Otherwise the value of *doc_collection_name* is used.

9. Conclusions

The MAUS Online DQ application has been successfully implemented and run in the MICE control room (Fig. 5). By using various "off the shelf" Python-interface like Celery, MongoDB, and Django, a DQ tool has been developed where the algorithms that run online are the same as the algorithms that run offline and the various computational challenges associated with continous data taking and live processing are solved. Future computational needs can be addressed by adding more Celery worker nodes. MAUS has been extended to online running.

References

[1] MICE Analysis User Software, C.D. Tunnell and C. Rogers. IPAC 2011.
[2] maus.rl.ac.uk
[3] http://celeryproject.org/
[4] http://www.rabbitmq.com/
[5] http://www.mongodb.org/
[6] http://root.cern.ch/drupal/content/pyroot
[7] http://matplotlib.sourceforge.net/
[8] https://www.djangoproject.com/