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

M. Jackson (EPCC, The University of Edinburgh) 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. MAUS consists of four key components: inputters to handle input of live, archived or simulated data; transforms to analyse data; mergers to summarise transformed data in the form of histograms and tables; and outputters to save this summarised data as images or tabular data. MAUS supports parallel data transformation in a map-reduce-inspired approach, however, the requirements of online reconstruction precluded adoption of a traditional map-reduce solution. A document-oriented database is used to cache transformed data, prior to merging, to support concurrent merging and visualisation of data. In MAUS, both offline and online analysis are implemented and executed in the same way, thereby removing the need for MICE to maintain and use two sets of analysis software, a common requirement elsewhere in experimental particle physics.

1. **Introduction**

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

In addition to the accelerator physics equipment, there has been a considerable investment in detectors to measure the manipulated muon beam: there are four detector technologies across eight detectors. This creates a software complexity problem: the number of technologies and detectors – by corollary, the number of calibrations and reconstruction algorithms – is comparable to a collider physics experiment like ATLAS [1, 2]. However, the size of the experiment and collaboration is more akin to a smaller neutrino experiment like MiniBooNE [3, 4].

2. **The MAUS software framework**

The MICE Analysis User Software (MAUS) [5, 6] addresses the software complexity problem. MAUS is a Python framework, which also supports C++ components via SWIG interfaces [7].
MAUS executes four types of component (Fig. 1). Each component creates or manipulates spills. In this context, a spill is a JSON document ([8]) representing information about a collection of trigger events.

- **Inputters** read live or archived DAQ data and convert this into spills, or, alternatively, generate simulated spill data.
- **Transforms** analyse the data within a spill in various ways, derive new data and add this to the spill.
- **Mergers** summarise data within a series of successive spills and output new spills with the summarised data. This summarised data may be in the form of histogram image data or tables.
- **Outputters** manage the storage of the spills output from mergers, for example saving them as JSON files or, where applicable, extracting image data from them and saving this as image files.

In essence, MAUS repeats the following loop until there are no more spills:

- **Input** a spill by reading it from an inputer.
- **Transform** the spill by passing it to a transform.
- **Merge** the spill with previously-transformed spills by passing it to a merger.
- **Output** a merged spill by passing it to an outputter.

### 2.1. MAUS and map-reduce architecture

The architecture of MAUS is very similar to a map-reduce architecture [9]. Transforms can act on spills independently and so can be viewed as analogous to maps. Mergers consume multiple spills and output summaries derived from these, similar to reduce steps. However, map-reduce typically assumes that all map operations terminate before a reduce operation is performed, whereas with online reconstruction, there is a requirement to reduce (merge) before the transform (map) operations have terminated. In effect, the reduction is done on ever larger sets of mapped data. This is necessary so that information can be presented to users in near-to-real-time as data is streamed in from detectors. As a consequence it was necessary to develop a custom framework, instead of using an off-the-shelf map-reduce solution.

### 3. MAUS and online reconstruction

In this section the components of MAUS that support online reconstruction are described.
3.1. Parallel transformation of spills

MAUS can be used for online data quality (DQ) purposes, where the status of the experiment and quality of the data needs to be monitored in real-time. This requires spills to be rapidly and continuously processed.

As each spill is independent, spills can be transformed in parallel. Celery [10], a distributed asynchronous task queue for Python, is used to implement parallel transformation of spills. Celery uses RabbitMQ [11] as a message broker to handle communications between clients, which submit tasks, and Celery workers, which execute these tasks in parallel. When a Celery worker starts up it registers with a RabbitMQ task broker, which can be located on a local or remote host. By default, each Celery worker spawns $N$ sub-processes where $N$ is the number of CPUs available, though $N$ can be explicitly set by a user.

In MAUS, each Celery worker runs transforms. Each sub-process can execute one transform upon a spill at a time. As multiple Celery workers can be deployed, each of which running one or more sub-processes, Celery allows MAUS to transform spills in a highly-parallelised way.

Once the Celery workers are running, the MAUS framework can dynamically configure the workers to execute the transforms required by a user. This configuration is done as follows (Fig. 2):

(i) A user writes a program which specifies the transforms they want to apply and passes these to MAUS, along with configuration information. Examples include performing detector digitisation, creating a beam, or running the simulation.

(ii) MAUS creates a transform specification.

(iii) A Celery broadcast is invoked, to communicate the transform specification, the configuration and a unique configuration ID (e.g. the client’s process ID) to each Celery worker.

(iv) Celery broadcasts are received by all Celery workers registered with the RabbitMQ message broker.

(v) On receipt of the broadcast, each Celery worker:

(a) Checks that the version of MAUS it is running is the same as that the user is running. If not then an error is returned to the user.

(b) Forces the transform specification down to each sub-process.

(c) Waits for the sub-processes to confirm that they have received the transform specification and have updated themselves to apply these transforms. If all sub-processes update correctly then a success message is returned to the user. Otherwise, a failure message, with details, is returned to the user.
(vi) Each Celery worker sub-process:
   (a) Invokes a death operation on the existing transforms, to allow for any clean-up to be done.
   (b) Updates its configuration.
   (c) Creates new transforms as specified in the transform configuration.
   (d) Invokes a birth operation on these transforms with the new configuration.
   (e) Confirms with the Celery worker that the update has been done.

Celery workers and sub-processes catch any exceptions 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:

(i) MAUS reads the next spill from an inputter.
(ii) A Celery client-side proxy is used to submit the spill to Celery. This proxy gives MAUS an identifier which it can use to poll the status of the transform.
(iii) The client-side proxy forwards the spill to RabbitMQ.
(iv) RabbitMQ forwards the spill to an available Celery worker. If none are available then the spill is queued.
(v) The Celery worker picks an available sub-process.
(vi) The sub-process executes the current transform on the spill.
(vii) The result spill is returned to the Celery worker and there back to RabbitMQ and, from there, back to MAUS.

The framework regularly polls the status of the transform job until its status is successful, in which case the result spill is available, or failed, in which case the error is recorded but execution continues. It is acceptable for an occasional spill to be lost for online running as long as the data exists to reprocess the event offline.

3.2. Caching transformed spills in a document-oriented database
After a spill has been transformed, MAUS stores the transformed spill in a document-oriented database, MongoDB [12]. This database represents the interface between the input-transform and merge-output phases of spill processing.

A document oriented database was chosen rather than a relational database because they allow more flexibility for changing the schema. The speed of this database, which just serves as an interface, is not a bottleneck for the whole system. Accordingly, it was found that it was quicker to prototype with document oriented NOSQL databases then apply schemas once the system has matured and there are more developers.

As MongoDB stores documents in collections, MAUS 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 in turn to a merger and then takes the output of the merger and passes it to an outputter.

Use of a database allows the input and transformation of spills to execute separately from their merging and the output of merged spills. This, in turn, allows for concurrent execution – one process can input and transform spills, another can merge transformed spills and output the merged results. In addition, many merge-output processes can use the same transformed data, for example to generate multiple types of histogram from the same data.
3.3. Histogram and scalar mergers

Histogram mergers are a type of merger which take in spills and, from the data within the spills, update histograms. They output 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 e.g. “TOF”, “ADC”.
- 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. Base64 encoding is used to avoid problems with control characters in string versions of JSON documents.

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

Example histogram mergers currently exist for histograms drawn using PyROOT [13] (e.g. time of flight plots) and matplotlib [14] (e.g. TDC and ADC counts).

Scalar mergers maintain counts and averages of values in spills (for example trigger requests, GVA, or time-of-flight). They output JSON documents with tabular data on a spill-by-spill basis. The JSON document includes:

- A list of keywords e.g. “Scalars”.
- A description of the tabular data.
- A list of table column headings e.g. “scalar name”, “last read value”, “average of last 10 values”, “average over run”.
- A list of lists, with the current values for each count and average.

As for histogram mergers, scalar mergers do not display or save tabular data.

3.4. Image and file outputters

An image outputter 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 into the file. It also saves the JSON document (minus the image data) in an associated meta-data file.

A file outputter allows JSON documents to be saved in files. As for images, the user can specify the directory where the files are saved and a file name prefix for the files. The file outputter is used to save tabular data from scalar mergers.

3.5. Web front-end

A web front-end allows histogram images and tabular data to be viewed. The web front-end is implemented in Django [15], a Python web framework. Django ships with its own lightweight web server or can be run under Apache web server [16].

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. Similarly, it serves up tabular data from this directory as HTML tables.

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

The interface between MAUS and the web front-end is just a set of tabular data files, image files and their accompanying JSON meta-data documents (though the web front-end can also render images without any accompanying JSON meta-data).
Figure 3. Processing steps for an individual spill (or rather collection of events). Each spill starts at Go.py, where Go.py is the initial python class that is called by the user’s macros. Go.py uses the Python Celery task queue to send the spill for processing. Celery uses RabbitMQ to communicate with worker nodes and send the spill to a worker node. At the worker node, the spill is processed, with the result being returned to Go.py via RabbitMQ.

4. Design details and Implementation

The processing for an individual spill can be seen in Fig. 3.

4.1. 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 $N$

4.2. 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 sync with the order in which they were originally read from the input stream.

4.3. 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:

```python
run_number = None
d_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 less than 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.

4.4. Document store

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

Documents are of form:

```json
{''_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.
4.4.1. 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”.

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.

5. Conclusions

Using a number of free, established, supported open source products – including Celery, RabbitMQ, MongoDB and Django – allowed the MAUS online data quality application to be rapidly developed. MAUS solves the computational challenges associated with continuous data streaming and live processing within MICE and MAUS has been successfully deployed and run in the MICE control room (Fig. 4). Future computational needs of MICE can be addressed by adding more Celery worker nodes.

MAUS has been designed in such a way that the algorithms that run online are the same as the algorithms that run offline. They are configured and used in the same way. This removes the need for MICE to maintain and use two sets of analysis software, a common requirement.

Figure 4. MAUS online data quality application running in the MICE control room. On the left screen are terminals for monitoring the various processes needed to run the application. On the right screen is a web browser that refreshes periodically to display histograms as they are updated.
elsewhere in experimental particle physics. It also removes from users the need to understand how to run two types of software.

The use of established open source products removes from MICE the requirement to develop, and to maintain and support, custom solutions for distributed transformation, document caching and web presentation.

Future work will investigate how to extend the set of mergers and the web front-end to ensure that control room staff have the information required to monitor the live execution of the MICE detectors, and how best to present this.

6. Acknowledgements
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.

References
[1] G. Aad et al. The ATLAS Experiment at the CERN Large Hadron Collider. Journal of Instrumentation. 3. August 2008. doi:10.1088/1748-0221/3/08/S08003
[2] ATLAS Experiment. http://atlas.ch/
[3] A.A. Aguilar-Arevalo et al. The MiniBooNE Detector. Nucl. Instr. Meth. 599(1), pp28–46. 2009.
[4] MiniBooNE Experiment. http://www-boone.fnal.gov/
[5] MICE Analysis User Software, C.D. Tunnell and C. Rogers. IPAC 2011.
[6] MAUS. http://maus.rl.ac.uk
[7] Simplified Wrapper and Interface Generator. http://www.swig.org/
[8] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). Internet RFC 4627. July 2006.
[9] J. Dean J and S. Ghemawat. MapReduce: simplified data processing on large clusters. Commun ACM. 51(1). pp107-113. January 2008. doi:10.1145/1327452.1327492
[10] Celery Distributed Task Queue. Celery Project. http://celeryproject.org/
[11] RabbitMQ. VMware, Inc. http://www.rabbitmq.com/
[12] MongoDB. 10gen, Inc. http://www.mongodb.org/
[13] PyROOT. http://root.cern.ch/drupal/content/pyroot
[14] matplotlib. http://matplotlib.sourceforge.net/
[15] Django. Django Software Foundation. https://www.djangoproject.com/
[16] Apache HTTP Server Project. The Apache Software Foundation. http://httpd.apache.org/