6 March 2015

Profile report of the reco-2D and reco-3D stages of the MicroBooNE reconstruction software

The purpose of this document is to report on a 2.5 day, informal SCD profiling effort targeted at the MicroBooNE 2D and 3D reconstruction chain that utilizes algorithms within LArSoft. The need for this effort came indirectly through the LArIAT 2015/2016 CPU-hour requests announced during the weekly SCD SPPM meeting and was originally targeted at LArIAT code. The requested per-event CPU time appeared to be extraordinarily high and a team was assigned to take a quick look for possible improvements. The team quickly recognized that the LArIAT estimates were derived directly from MicroBooNE and the effort was redirected there.

We have profiled the reco-2D and reco-3D portions of the production chain using generated Monte Carlo samples that have been propagated through GEANT4 and the detector simulation. For this study, we have produced and used event samples that are intended to represent the environment of the MicroBooNE detector during normal operations. The job configurations to produce such samples were suggested to us by Herb Greenlee of MicroBooNE.

For the reco-2D stage, the median (average) time per event is **12.7 sec** (**32.4 sec)**, whereas for the reco-3D stage, the median (average) time per event is **72.0 sec** (**97.3 sec)**. The lengthy per-event processing times are due in part to the following choices made by the software developers:

- the choices of algorithms used for particular sub-stages of the reconstruction, such as hit finding, clustering, and track finding,
- the implementations of the algorithms themselves, and
- the information stored or kept per event.

Some specific examples are listed below.

- A large percentage of time is spent exercising ROOT's fitting facilities, particularly fitting pulse distributions with Gaussian functions, and even superpositions of Gaussian PDFs.
- A similarly large percentage of time is spent calling functions like `std::pow(…,0.5)` and `std::map::insert(…)`. There are known better ways that ought to be explored.
- Since no event information is dropped throughout the production chain, simulated quantities are passed all the way from the generation stage through the reconstruction stages. By writing the simulated quantities to the output of the reco-3D stage, the virtual memory is increased by 2GB and can result in up to a 30 second increase in job execution time per event.

If effort is going to continue in this area, we recommend that the MicroBooNE software developers and LArSoft developers should:

- Determine if fitting distributions with (superpositions of) Gaussian PDFs, something typically done for physics analysis, is required at the reconstruction level. If so, a dedicated least-squares library may be more effective than ROOT's fitting libraries.
- Determine if some of the perhaps more experimental algorithms (*e.g.* Bezier track fitting) can be dropped in favor of something less time consuming.
- Take care to optimize the use of 'for' loops:
    - Move calls to constant expressions outside of the loop (*e.g.* `std::sqrt(7.)`)
    - For double loops that are commutative (*i.e.* result of `(i,j)` equals that of `(j,i)`), the calculations of off-diagonal elements should not be repeated.

More specific discussions of our setup, findings and comments are given in the following pages.

Allow us to thank you for the opportunity to explore your code and exercise it. We appreciate the chance to learn how FNAL experiments implement their simulation and reconstruction algorithms. We hope that this report will be useful to you as MicroBooNE and LArSoft consider their next steps in improving their software and simulation design.

Please do not hesitate to contact us further.

Best regards,

Kyle Knoepfel
Paul Russo
Jim Kowalkowski

FNAL Scientific Computing Division

I. The setup

The code was built in profile mode using the `e6`-qualified `larsoft_suite` versions `v4_00_00` and `v4_00_01` along with the most recent push to the `uboonecode git` repository, current as of February 23, 2015.

We produced various samples with anywhere from 1 to 500 events, using the production chain:

```
[1] lar -c <some_production_scheme.fcl>
[2] lar -c standard_g4_uboone.fcl prod*_gen.root
[3] lar -c standard_detsim_uboone.fcl prod*_g4.root
[4] lar -c standard_reco_uboone_2D.fcl prod*_detsim.root
[5] lar –c standard_reco_uboone_3D.fcl prod*_2D.root
```

The following production-level FHiCL files were used:

```
[a] prodgenie_common_cosmic_uboone.fcl
[b] prodgenie_bnb_nu_cosmic_uboone.fcl
```

Only stages 4 and 5 were profiled.

We used two profiling programs: `allinea` (v5.00.00) and `valgrind` (v3.10.1). The results from both programs are consistent. For brevity, we primarily refer to the results produced from `allinea`. Both of these tools are generally available to all lab members and can be installed using the usual UPD tool.

All measured runs were performed on develop nodes `woof.fnal.gov` and `cluck.fnal.gov`.

II. Our findings and comments

Although we highlight multiple places where improvements can likely be made, implementing only a few of them is unlikely to achieve significant performance gains. This implies a systemic issue in the overall design of the code: high-powered algorithms have been chosen for reconstruction and high-performance implementations of these algorithms are not utilized. If such algorithms are necessary, then dedicated C++ libraries should be used in favor of ROOT algorithms, which are more appropriate for physics analysis and not optimized for a reconstruction environment. MicroBooNE and LArSoft should investigate whether the chosen set of algorithms are required and should find ways to implement higher performance routines if possible.

In what follows, all percentages are expressed relative to the total CPU running time of the job. It is important to note that time spent doing I/O affects the total running time of

the job, but does not count in the CPU time. This becomes very important when considering the ROOT I/O module.

The tables below show the per-module running times for each module included in the execution path—note that these are wall-clock times, not CPU times.[1] The entries are ordered according to the average time taken for each module. These results were determined by retrieving the *art* `Timing` service printout from the log files. Including all modules, the median (average) time per event is **12.7 sec** (**32.4 sec)** for the reco-2D stage, whereas for the reco-3D stage, the median (average) time per event is **72.0 sec** (**97.3 sec)**.

Reco-2D (in seconds)

module_label	Min	Avg	Max
out1	8.58644	13.3756	32.5296
pandora	0.221644	8.40859	221.102
caldata	1.9309	3.04769	12.6421
fuzzycluster	0.290592	2.5211	11.2051
gaushit	0.547555	2.49191	15.7718
cccluster	0.428353	2.3756	16.7683
opflash	0.0146341	0.0507723	0.862585
rns	8.70228e-05	0.000122139	0.000339985
TriggerResults	4.3869e-05	6.02316e-05	8.98838e-05

Reco-3D (in seconds)

module_label	Min	Avg	Max
beziertrackercc	2.43425	22.3087	60.5369
costrkcc	2.21479	19.2633	42.9008
out1	11.2953	15.6302	24.0332
beziertracker	0.770546	8.62989	23.875
costrk	0.979419	5.19026	10.6814
beamflashcompat	0.772372	2.78226	5.35873
trackkalmanhitcc	0.573231	2.59147	6.6801
stitchcccalo	0.577038	2.36763	6.97095
spacepointfindercc	0.0841441	2.3431	9.70711
trackkalspscccalo	0.472975	2.30428	8.18714
trackkalmanhit	0.528796	1.89992	4.54985
spacepointfinder	0.132835	1.57304	7.0053
trackkalmanhitcccalo	0.812297	1.54655	3.03745
stitchcalo	0.185467	1.46539	6.34125
trackkalspscalo	0.362722	1.30534	6.13064
trackkalsps	0.384797	1.12523	2.94881
costrkcccalo	0.529308	1.04651	2.19475
trackkalmanhitcalo	0.305757	0.951603	2.40347
trackkalspscc	0.394712	0.936043	1.67436
stitchkalmanhitcalo	0.258669	0.927119	2.53201

[1] `TriggerResults` is automatically appended to the module list by *art*.

Reco-3D (in seconds – *continued.*)

module_label	Min	Avg	Max
costrkcalo	0.327197	0.724612	1.83589
featurevtx	0.011682	0.0591867	0.338767
stitchcc	0.0036509	0.0198794	0.0531549
stitch	0.00425005	0.018968	0.0556412
stitchkalmanhit	0.00345302	0.0130652	0.0319519
beziertrackercccalo	0.000331879	0.00141158	0.00225401
costrkcctag	0.000690937	0.0012628	0.003407
trackkalmanhitcctag	0.000537157	0.00120226	0.00262213
costrkccpid	0.000390053	0.00110436	0.00217605
trackkalspscctag	0.000485897	0.00110077	0.00299883
beziertrackercalo	0.000379086	0.00104811	0.00191808
stitchcctag	0.00036788	0.00100462	0.00318193
trackkalspstag	0.000392914	0.000950694	0.00454783
costrktag	0.000488043	0.000936914	0.00300193
trackkalspspid	0.000294924	0.000894427	0.00157714
trackkalmanhittag	0.000365019	0.000885928	0.00232697
trackkalspsccpid	0.000206947	0.000850165	0.00156784
stitchtag	0.000232935	0.000824511	0.00454617
stitchkalmanhittag	0.00027895	0.000723755	0.00218201
costrkpid	0.000325918	0.000694263	0.00112796
trackkalmanhitpid	0.000238895	0.000652337	0.00219488
trackkalmanhitccpid	0.000237942	0.000603557	0.00119996
stitchccpid	0.000203133	0.000550318	0.00114298
stitchpid	0.000194073	0.000530839	0.00151181
stitchkalmanhitpid	0.00019908	0.000443769	0.00097394
beziertrackercctag	7.79629e-05	0.000400949	0.000724077
beziertrackertag	0.000109911	0.000320172	0.000652075
beziertrackerccpid	4.60148e-05	7.15375e-05	0.000183105
beziertrackerpid	4.50611e-05	6.3169e-05	0.000111103
rns	4.19617e-05	5.99861e-05	0.000195026
TriggerResults	1.69277e-05	3.28898e-05	7.60555e-05

The top five modules in execution time have been highlighted in gray in both tables. We will concentrate on these top-five modules and particularly the most time-intensive functions they call.

`valgrind`'s `callgrind`-tool plots of the call trees for the reco-2D and reco-3D stages are available at the following link: https://cdcvs.fnal.gov/redmine/documents/859. These plots are useful for seeing a graphical overview of how the CPU time is split between the module routines.

A. reco-2D

out1 module_type: **RootOutput**

The ROOT I/O is the most expensive part of the reco-2D stage: 41.4% of the total running of the job is spent in the ROOT output module. The profiler shows that 4.0% of the reco-2D CPU time is spent unzipping the input data, and 4.5% of the CPU time is spent zipping the output data, which seems excessive. Not included in the 8.5% data compression/uncompression CPU time is the time spent writing to disk, which accounts for 12.2 seconds of the average 13.4 seconds running time per event spent for this module (the 8.5% of the total job CPU time for zipping/unzipping accounts for the remaining 1.2 seconds of running time).

The 41.4% of the job running time devoted to ROOT I/O points to a larger problem—that the type of information and the way it is stored is suboptimal.[2]

Recommendations

- Evaluate the use of data compression and only compress the data where the loss of performance is justified by the reduction in the cost of storage.
- For data that must be compressed, test the results of choosing different compression levels to find a balance between CPU cost and storage savings. Individual data branches can have specific compression factors associated with them. This is done by specifying in the data-product `xml` file (e.g.):

```
<class name="art::Wrapper<user::CompressedProduct>" compression="9"/>
```

- Evaluate the use of the upcoming *art* feature that allows use of multiple data tiers on input—the raw data could be in one tier, the digitized data in another tier, the simulated data in another, the 2D in another. The advantage here being that data from a given tier is available on request and only transferred from disk on specific request from a module, thereby retaining access to all necessary data but greatly reducing memory usage.

pandora module_type: **MicroBooNEPandora**

`MicroBooNEPandora` inherits from `LArPandoraParticleCreator`, which is where the time is spent, not in `MicroBooNEPandora`. According to `allinea`, 27% of the reco-2D CPU time is spent executing the following statement in `LArPandoraParticleCreator.cxx`:

```
this->RunPandoraInstances(); // line 142 in our checkout
```

[2] See Sect. B, where we discuss the effect of dropping the `simb*` data products in the reco-3D stage and what gains can be expected from this.

The primary contributers to the 27% are:

```
        // ClusterAssociationAlgorithm.cc::47
9.4%    this->PopulateClusterAssociationMap(…)
        // VertexSelectionAlgorithm.cc::60
8.4%    float figureOfMerit(…)
```

Tracing the percentages further down from `this->Populate*Map(…)`, eventually yields `LArClusterHelper::GetExtremalCoordinates(…)` which accounts for 8.7% of the total execution time of reco-2D.

For the `figureOfMerit(…)` calculation, the majority of that time is spent calling histogram-filling functions. A decent percentage of that time is spent using the `std::pow(…)` function, which is not fast.

Recommendations

- For the `PopulateClusterAssociationMap` algorithm, the `LArClusterHelper::GetExtremalCoordinates` processes twice as many cluster pairs as necessary due to overlapping loops. This should be adjusted.
- For the `figureOfMerit` calculation, a higher-performing de-weighting scheme should be considered.

`caldata` `module_type:` **`CalWireROI`**

Many things are going on with `CalWireROI_module.cc`:

1. For line 278, the `evt.getByLabel(…)` call takes 4.5% of the job execution time, which is due almost entirely to the unzipping of the compressed data from the input file.
2. Line 381 is responsible for 3.5% of the execution time.
3. Line 385 takes 1.6% of the job execution time, due to the call to `sqrt(7.)` within the loop.
4. The `sss->SetDecon(transformSize);` call on line 517 comprises 8.7% of the execution time. Of that percentage, 3% is due to evaluating TF1 functions (`SignalShapingServiceMicroBooNE_service.cc:617`).
5. Line 535 is where the deconvolution takes place, which takes 1.7% of the time.

Recommendations

- For the zipping issues, see our comment for the `out1` module above.
- There is a lot of duplication in summing the `rawadc` values (l. 381) with the lines that precede it. Every ADC count is used in the window sum seven times. One can do a rolling sum to avoid so many calls to `operator+=` and `operator-`.

- The `sqrt(7.)` call should be taken out of the loop to make the contribution of line 385 negligible.
- Evaluating the filter functions is taking longer than the deconvolutions. This should be examined.

`fuzzycluster` `module_type:` **`fuzzyCluster`**

This module accounts for 18% of the running time of the reco-2D stage, 13% of which is due to calling `cluster::HoughBaseAlg::Transform(…)`. Tracing this percentage further down, the majority of the time spent here is due to executing `std::map::insert(…)` from calling :

```
cluster::HoughTransformCounters<…>::unchecked_add_range_max(…)
```

Inside that function, calling `Base_t::counter_map.lower_bound(…)` accounts for 9% of the total execution time of the job.

Recommendations

- The design of the sparse Hough counters array is (perhaps overly) complicated, and it may be that the ordering of the entries is important, as indicated by using `lower_bound`. But in the event that the ordering is not (not only here but in other places), it will be more effective to use an `std::unordered_map` object.
- The Hough counters array implementation was chosen to save memory at a rather high cost in CPU time. This should be reconsidered.

`gaushit` `module_type:` **`GausHitFinder`**

According to `allinea`, 2% of the execution time is devoted to constructing the `TF1 Gaus` object on line 731 of `GausHitFinder_module.cc`.

Recommendations

- Fitting ADC distributions with Gaussians is a very high-powered functionality for hit reconstruction. A faster algorithm should be considered.
- If you decide to retain ROOT as the fitting library, to avoid constructing TF1 objects repeatedly, it would be better to create a look-up table of all desired functional forms and to simply use the appropriate one when calling `FitGaussians`.

B. reco-3D

beziertrackercc module_type: **BezierTrackerModule**
beziertracker

These modules are the most time-consuming, with the following breakdowns for total execution time:

	`// BezierTrackerModule_module.cc::167`
14.1%	`fBTrackAlg->FilterOverlapTracks(…)`
	`// BezierTrackerModule_module.cc::169`
13.5%	`fBTrackAlg->MakeOverlapJoins(…)`
	`// BezierTrackerModule_module.cc::172`
4.2%	`fBTrackAlg->FilterOverlapTracks(…)`
32.6%	**Total** (including other `BezierTrackerModule` calls)

20% of the total is due BezierCurveHelper::GetDirectionScales(…) calls—specifically, the three calls to `std::pow(…,0.5)` inside of it.

Recommendations

- `GetDirectionScales` needs to be investigated to see if the use of square roots can be removed, which would result in a 20% savings in reco-3D processing time.

costrkcc module_type: **CosmicTracker**
costrk

Examination of the `produce(…)` routine in `CosmicTracker_module.cc` shows the execution of lines 418-420 comprise 22% of the reco-3D job: 4.6%, 4.5%, and 12.9% respectively.

Recommendations

- Adjust the double loops so that the `(i,j)` and `(j,i)` calculations are not repeated. For example: lines 401 and 402 should become:

  ```
  for (int i = 0; i < nplanes; ++i) {
    for (int j = 0; j < i+1; ++j) {
    // …
  ```

  or something similar instead of repeating the off-diagonal element calculations. The same may hold true for the `c1` and `c2` loops in lines 403 and 404. It looks like something akin to this was explored, based on the comments in lines 399 and 400. If possible, this should be implemented.
- The more general question however, is whether there are more efficient ways of doing integrals and whether a KS test indeed is appropriate at this stage.

out1	module_type: **RootOutput**

This has been discussed in III.A.

Recommendations

- For the reco-3D stage, we did a test by placing a 'drop' directive in the outputCommands portion of the RootOutput ParameterSet.

```
out1: {
  module_type: RootOutput
  fileName:    "%ifb_%tc_reco3D.root"
  outputCommands: [ "drop sim*_*_*_*" ]
  dataTier:    "reconstructed-3d"
  compressionLevel: 1
}
```

- By doing so, the peak virtual memory of the job dropped from roughly 3.5 GB to roughly 1.2 GB. The execution time for the `out1` module also dropped to a negligible amount.
  The request to pass on the simulated quantities to the output of the reconstruction chain is unnecessarily degrading the performance of the reconstruction, which during data collection and data processing will know nothing of these simulated quantities. We can imagine that storing the simulated information may be important for validating reconstruction algorithms against Monte Carlo truth. However, MicroBooNE should decide whether keeping (all of) the `simb*` data products is required.

III. Next steps

To move forward, we suggest meeting with the MicroBooNE and LArSoft developers to discuss the following questions:

- What are the motivations for the chosen set of reconstruction techniques?
- Have other options been tried to achieve equivalent results?
- What tests exist to ensure consistency in physics results upon code modifications?
- What is the time scale for which improvement is necessary?

We also want to discuss mapping out a staged plan with the developers that will make incremental code improvement possible. Here is a start at listing essential pieces of such a plan.

1. Understand the constraints (some examples here):
   - What is the desired reconstruction time? For Monte Carlo? For data?
   - What are the desired physics results?
2. Begin establishing validation processes for:

- Physics - Through representations of quantities (in the form of histograms or statistical measures), which can be influenced by coding changes.
- Coding correctness - Through unit and integration tests to verify intended behavior of implementations.
- Performance - Through profiling metrics, job execution time, and resource usage.

3. Implement the specific modifications for minor performance gains (up to 20% based on this analysis).
4. Replace the more time-consuming code with well-established existing techniques. This step is intended to achieve significant performance gains while retaining equivalent physics results.
5. Iterate the above steps as necessary.