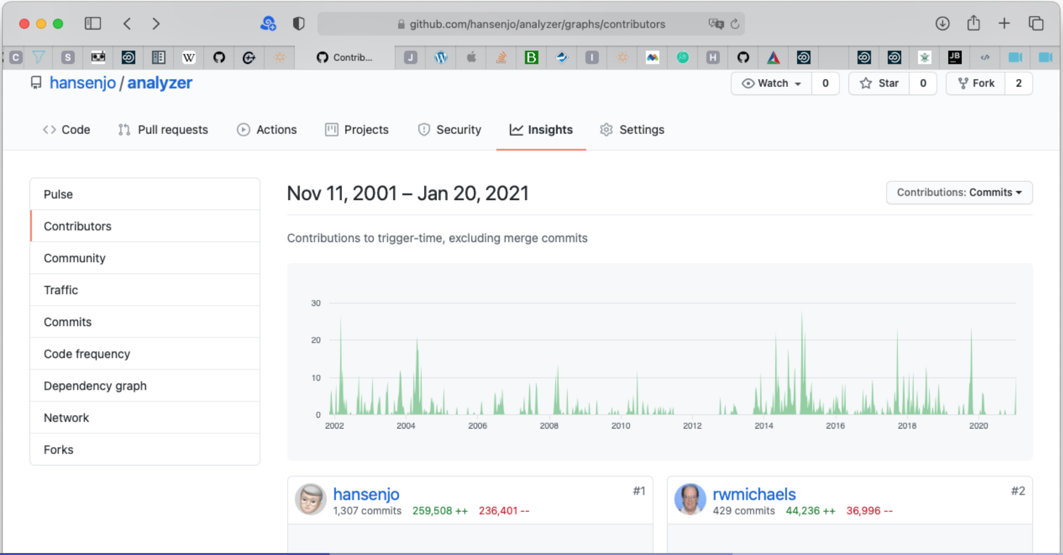
Hall A Analysis Software Update

Ole Hansen

Jefferson Lab

Hall A Collaboration Meeting
January 21, 2021

C++ Analyzer Project "Podd" — Over 20 Years! (started ∼April 2000)



Podd Development Status & Plans

- Current release: 1.7.0 ($\sim$ Feb 2021, sorry for the delay)
  - Many updates and new features (see next page)
  - Requires `C++11` compiler
  - Drops support for obsolete ROOT 5
  - Final testing & documenting in progress
- Priority development: 2.0-devel (hopefully Summer 2021)
  - Multithreading
  - Intended for SBS
  - Will require `C++17` (e.g. `gcc 9+`, available on `ifarm`)
  - Existing code will need minor modifications
- Auxiliary development: 1.8-devel (if time permits)
  - Include features missed in 1.7
  - Maintain system requirements and API of version 1.7 as much as possible

New in Podd 1.7

- Decoder upgrades
  - Support for CODA 3 data format, bank data and event block decoding (Bob Michaels)
  - EVIO upgraded to version 5.2 (better I/O performance and many bugfixes)
  - Includes FADC decoders developed for Tritium experiments, to be reused in SBS
- New module type: "InterStageModule"
  - May combine information from arbitrary detectors after each processing stage
  - Needed for coincidence time correction in Tritium ΛN
  - Removes a *significant limitation*[1] of Podd; many other possible uses
- Build system overhaul
  - `CMake` build system added (used by SBS, for example)
  - `SCons` build system significantly improved (used by `hcana`)
  - Old `make` system removed
- Extensive code cleanup & reorganization
  - Libraries split into core and Hall A parts: `libPodd` and `libHallA`

[1] Too many such limitations? See later for discussion

Code Cleanup Example — Old (Left) vs. New (Right)

Podd – Podd/THaCherenkov.cxx

analyzer 〉 Podd 〉 THaCherenkov.cxx   nnL replay | Debug   Git:

Podd/THaCherenkov.cxx (3941956)

```cpp
//____________________________________________________________________________
Int_t THaCherenkov::Decode( const THaEvData& evdata )
{
  // Decode Cherenkov data, correct TDC times and ADC amplitudes, and copy
  // the data into the local data members.
  // This implementation assumes that the first half of the detector map
  // entries corresponds to ADCs, and the second half, to TDCs.

  const char* const here = "Decode";

  // Loop over all modules defined for Cherenkov detector
  bool has_warning = false;
  for( Int_t i = 0; i < fDetMap->GetSize(); i++ ) {
    THaDetMap::Module* d = fDetMap->GetModule( i );
    bool adc = (d->model ? d->IsADC() : i < fDetMap->GetSize()/2 );
    bool not_common_stop_tdc = (adc || d->IsCommonStart());

    // Loop over all channels that have a hit.
    for( Int_t j = 0; j < evdata.GetNumChan( d->crate, d->slot ); j++) {

      Int_t chan = evdata.GetNextChan( d->crate, d->slot, j );
      if( chan < d->lo || chan > d->hi ) continue;     // Not one of my channels

      Int_t nhit = evdata.GetNumHits(d->crate, d->slot, chan);
      if( nhit > 1 || nhit == 0 ) {
        ostringstream msg;
        msg << nhit << " hits on " << (adc ? "ADC" : "TDC")
            << " channel " << d->crate << "/" << d->slot << "/" << chan;
        ++fMessages[msg.str()];
        has_warning = true;
        if( nhit == 0 ) {
          msg << ". Should never happen. Decoder bug. Call expert.";
          Warning( Here(here), "Event %d: %s", evdata.GetEvNum(),
                   msg.str().c_str() );
          continue;
        }
#ifdef WITH_DEBUG
        if( fDebug>0 ) {
          Warning( Here(here), "Event %d: %s", evdata.GetEvNum(),
                   msg.str().c_str() );
        }
#endif
      }

      // Get the data. If multiple hits on a TDC channel, take
      // either first or last hit, depending on TDC mode
      assert( nhit>0 );
      Int_t ihit = ( not_common_stop_tdc ) ? 0 : nhit-1;
      Int_t data = evdata.GetData( d->crate, d->slot, chan, ihit );

      // Get the detector channel number, starting at 0
      Int_t k = d->first + ((d->reverse) ? d->hi - chan : chan - d->lo) - 1;

      if( k<0 || k>= fNelem ) {
        Error( Here(here), "Illegal detector channel: %d", k );
        continue;
```

Podd/THaCherenkov.cxx

```cpp
//____________________________________________________________________________
Int_t THaCherenkov::Decode( const THaEvData& evdata )
{
  // Decode Cherenkov data, correct TDC times and ADC amplitudes, and copy
  // the data into the local data members.
  // This implementation assumes that the first half of the detector map
  // entries corresponds to ADCs, and the second half, to TDCs.

  const char* const here = "Decode";

  // Loop over all modules defined for Cherenkov detector
  bool has_warning = false;

  auto hitIter = fDetMap->MakeIterator(evdata);
  while( hitIter ) {
    const auto& hitinfo = *hitIter;
    if( hitinfo.nhit > 1 ) {
      // Multiple hits in a channel (usually noise)
      MultipleHitWarning(hitinfo, here);
      has_warning = true;
    }

    // Get the data for this hit
    OptInt_t data = LoadData(evdata, hitinfo);
    if( !data ) {
      // Data could not be retrieved (probably decoder bug)
      DataLoadWarning(hitinfo, here);
      has_warning = true;
      continue;
    }

    // Store hit in fDetectorData
    StoreHit(hitinfo, data.value());

    // only add channels with signals to the amplitude sums
    const auto& PMT = fPMTData->GetPMT(hitinfo);
    if( PMT.adc_p > 0 )
      fASUM_p += PMT.adc_p;              // Sum of ADC minus ped
    if( PMT.adc_c > 0 )
      fASUM_c += PMT.adc_c;              // Sum of ADC corrected

    // Next active channel
    ++hitIter;
  }
  if( has_warning )
    ++fNEventsWithWarnings;

#ifdef WITH_DEBUG
  if ( fDebug > 3 )
    PrintDecodedData(evdata);
#endif

  return fPMTData->GetHitCount().tdc;
}

//____________________________________________________________________________
```

Building with CMake

Prerequisites:

- Install ROOT (`root-config` should be in `PATH`, or set `$ROOTSYS`)
  - Farm: run `setroot_CUE.csh`. RHEL: install from EPEL. macOS: install from Homebrew.
  - See also `https://redmine.jlab.org/projects/podd/wiki/ROOT_Installation_Guide`
- Ensure you have CMake >= 3.5 (`cmake --version`. `cmake3` on RedHat)

Building the Hall A analyzer with CMake

```
$ git clone https://github.com/JeffersonLab/analyzer.git
$ cd analyzer && mkdir build && cd build
$ cmake ..
$ make [-j4]
$ ./apps/analyzer
```

Notes:

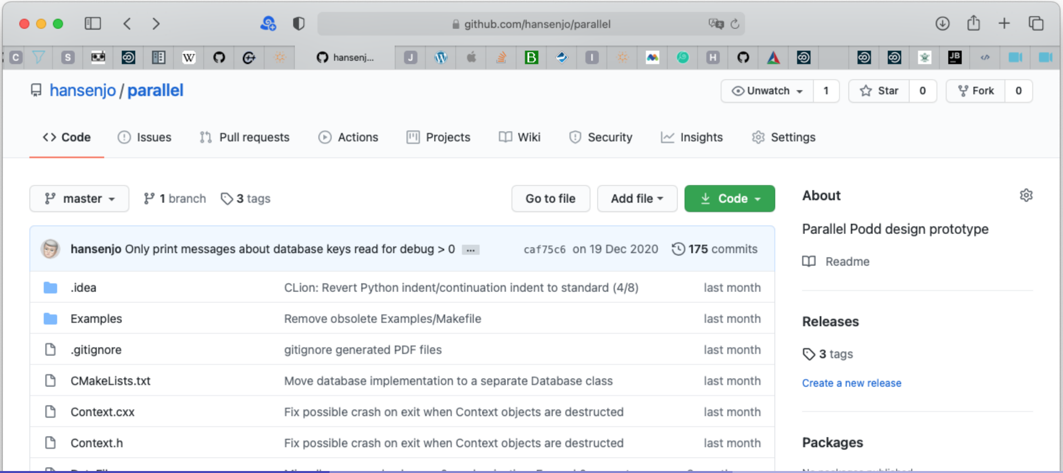
- Installing recommended (`make install`): Set `CMAKE_INSTALL_PREFIX`
- For debug build, set `CMAKE_BUILD_TYPE`
- Will phase out aging SCons build system (too many limitations)

Podd 2.0

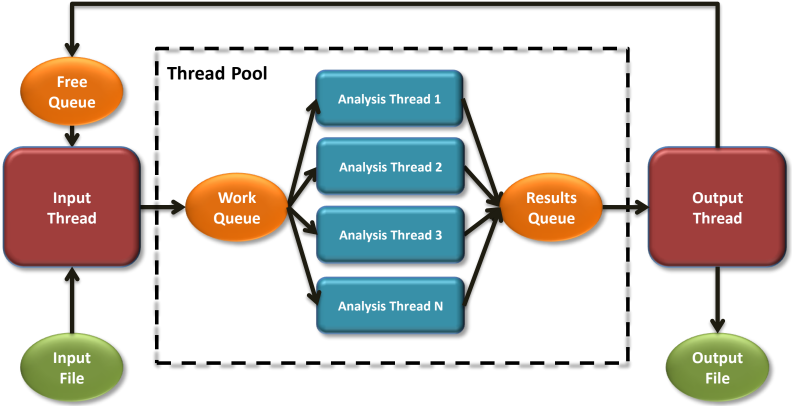
- Event-based parallelization/multithreading
  - Important for SBS online replay
  - Reduced memory footprint compared to multiple individual jobs
  - Requires thread safe user code ($\rightarrow$ only const or protected globals, statics)
- I/O improvements
  - Output system upgrade (full set of data types, object variables)
  - TBD: HIPO output file format support
  - TBD: EVIO 6 input format support (HIPO-like raw data files)

ToyPodd Parallel Processing Prototype

- Small standalone toy analyzer with hand-implemented multithreading
- Mimics main components of Podd (*e.g.* decoder, analysis variables, output)
- A few example “detectors” included whose processing is intended to burn CPU cycles



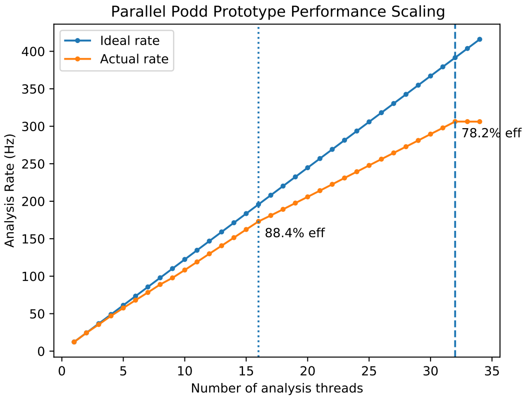
Podd Parallelization Design

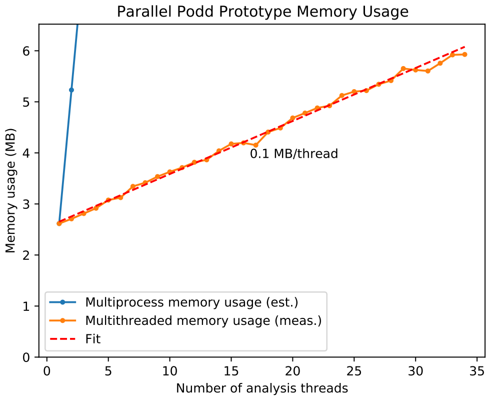


- Thread Pool with three thread-safe queues
- Queues hold working sets: raw event buffer, analysis modules, event-by-event results
- Options
  - Sync event stream at certain points (*e.g.* scaler events, run boundaries)
  - Preserve strict event ordering (at a considerable performance penalty)

ToyPodd Performance Scaling Benchmark

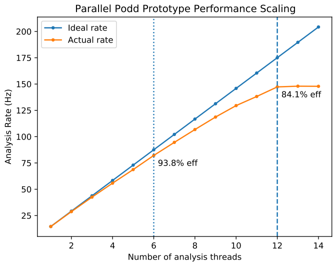
- Benchmark processing rate as function of number of analysis threads
- Run on `aonl1` (16 hyperthreaded cores, Intel Xeon E5-2650 v2 @ 2.60GHz), RHEL 7.9, idle
- Admittedly extreme example: maximally CPU-bound (negligible I/O & memory use)



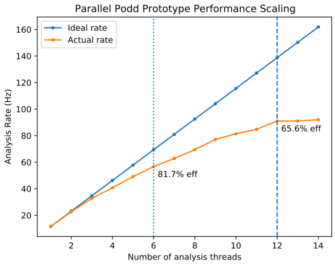


More ToyPodd Benchmark Results

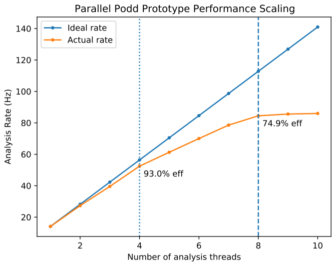
AMD Ryzen 5 3600 (6C/12T), macOS 11.1



Intel i9-8950HK (6C/12T), MacBook Pro, macOS

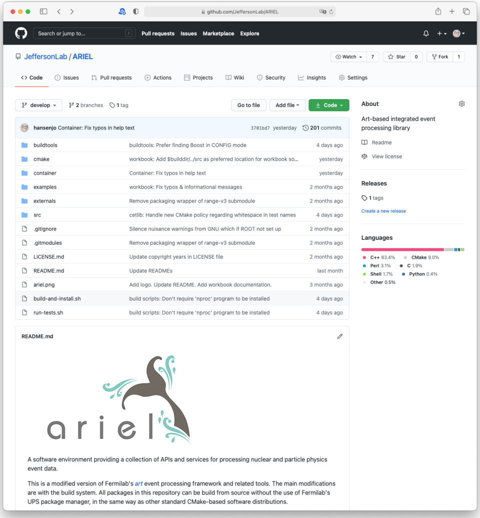


Apple M1 (4C/8T), Mac Mini, macOS



A Future Hall A Framework Candidate: ARIEL

- `https://github.com/JeffersonLab/ARIEL`
- Repackaged version of Fermilab's *art* framework w/custom build system. Based on `CMSSW` (LHC).
- Intended as base software for SoLID, but completely experiment-agnostic.
- "*art* made usable": Easy-to-install bundle of entire *art* suite, independent of custom Fermilab package manager. Installable from source.
- Most recent *art* version 3.06.03 (Aug 2020) plus dependency packages, integration tests, examples (toyExperiment, art-workbook)
- Task-based event-level multithreading (TBB)
- Supported on Linux & macOS w/`C++17`



ARIEL Singularity Container

Singularity Container on CUE

```
ifarm1901> module load singularity
ifarm1901> singularity run /group/solid/apps/ARIEL.sif
Singularity> art --version
art 3.06.03
Singularity> ^D
ifarm1901>
```

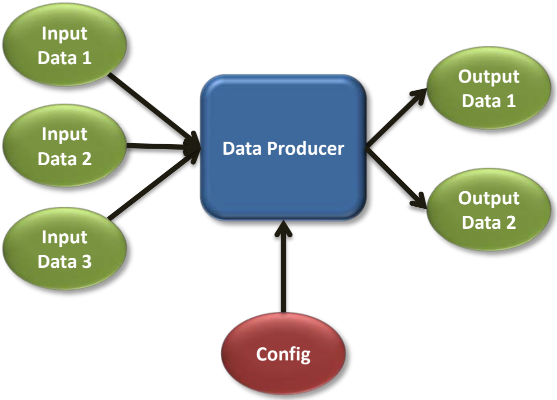
- NB: Singularity currently only works on Linux (but basically any Linux)
- Download: `https://solid.jlab.org/files/ARIEL.sif` (775 MB)
- Container software: Ubuntu 20.04 LTS base w/gcc 9.3.0. ROOT 6.22.06 w/C++17
- Built-in help/documentation:

```
singularity run-help ARIEL.sif
```

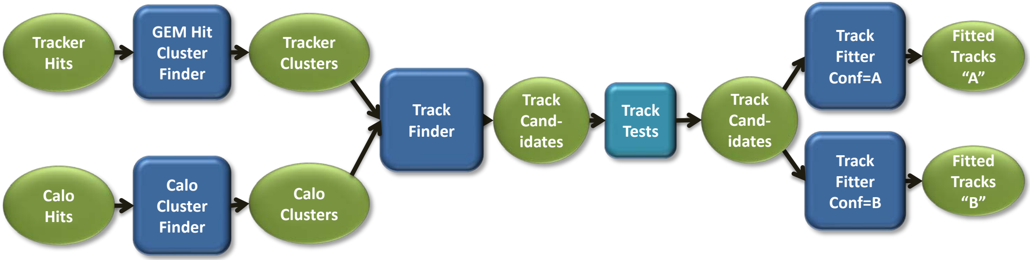
- Docker version planned

Why a New Framework? Decoupled Algorithms & Data Objects

- Very successful computing paradigm in HEP for past 20+ years
- Data objects (inputs & results)
  - Mostly "dumb data" (structs)
  - May reference other data objects (with or without framework support)
  - Persistable on disk (ROOT)
  - Streamable via message services (*e.g.* `protobuf`, `zeromq`)
- Data consumers/producers (algorithms)
  - Single algorithm per module
  - Input configurable at run-time →
    - ★ modules are reusable
    - ★ multiple module instances possible



Analysis Flow Becomes Flexible: Analysis Chains



- Modules communicate exclusively via data objects
- Module relationships configurable at run time
- Multiple chains per job
- Support for condition testing modules
- Output modules (not shown) for DST and histogram/ntuple files

A Simple Prototype Module

- Ported benchmarking algorithm from my "parallel Podd" toy analyzer.
- Minimal framework overhead (see screenshot). <1 hour of beginner-level work.
- This example algorithm implements a CPU-intensive calculation of π [1]

[1] Rabinowitz and Wagon, American Mathematical Monthly, 102 (3), 195-203 (March 1995), doi:10.2307/2975006

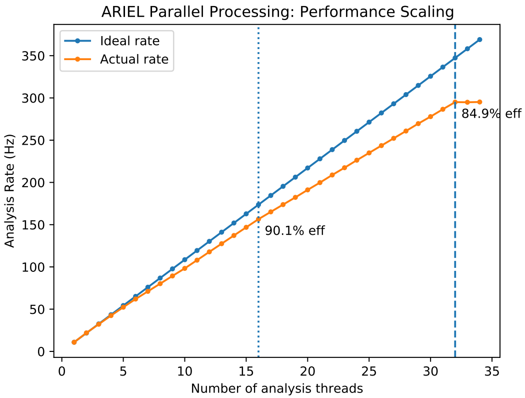
DetectorTypeC_module.cc - emacs@mackinley.redvw.com

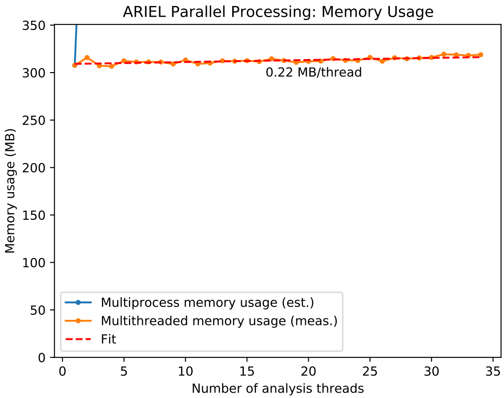
```
// DetectorTypeC: demonstration of a detector with a time-consuming
// algorithm, simulated here by a calculation O(1000) digits of pi

#include "DetectorTypeCResults.h"
#include "detail/util.h"
#include "art/Framework/Core/ReplicatedProducer.h"
#include "art/Framework/Core/ModuleMacros.h"
#include "art/Framework/Principal/Event.h"

#include <iostream>
#include <vector>
#include <string>
#include <memory>
#include <cstdlib>

class DetectorTypeC : public art::ReplicatedProducer {
public:
  DetectorTypeC(fhicl::ParameterSet const& pset, art::ProcessingFrame const& frame );
  void produce( art::Event& event, art::ProcessingFrame const& ) override;
private:
  std::vector<int> m_a;      // Workspace
  std::string      m_result; // Result as string representation of a decimal number
  double           m_scale;  // Scale factor
};

parallel::DetectorTypeC::DetectorTypeC(fhicl::ParameterSet const& pset,
                                       art::ProcessingFrame const& frame ):
  art::ReplicatedProducer{pset,frame},
  m_scale(pset.get<double>("scale", 1.0))
{
  produces<DetectorTypeCResults>();
}

void parallel::DetectorTypeC::produce( art::Event& event, art::ProcessingFrame const& )
{
  // This detector type computes n digits of pi

  //......... algorithm goes here ............

  // Create data product
  auto output = std::make_unique<DetectorTypeCResults>(some_result,another_result);
  // Add the product to the event
  event.put( std::move(output) );
}

DEFINE_ART_MODULE(parallel::DetectorTypeC)
```

U:--- DetectorTypeC_module.cc   All of 1.5k (47,0)   (C++//l Abbrev)

ARIEL Parallel Processing Benchmark

- Exact same algorithm & hardware (`aonl1`) as for ToyPodd benchmark
- Full framework w/ROOT output backend: higher base memory usage
- Run in Singularity container: much newer compiler (`gcc 9.3` vs. `gcc 4.8` for ToyPodd)





ARIEL for SBS?

- Pros
  - State-of-the-art multithreading readily available
  - Custom analysis flows readily configurable
  - Multi-pass analysis readily supported, will save much analysis time with high data volumes
  - Consistent Hall A environment
  - Infrastructure additions developed for SBS will benefit everyone in the *art* community
  - Will build expertise with HEP-style framework software, which helps inform development for SoLID, EIC etc.
- Cons
  - Learning curve
  - Must add support for reading CODA data format (significant work)
  - Must add some sort of conditions database support
  - Must port existing reconstruction algorithms (fairly easy)
  - Should add support for reading `g4sbs` file format (fairly easy)
  - Runtime-configurable ntuple output module, like Podd's, would be nice (moderate work)
- Could deploy at later stage of SBS program

Scientific Computing Status

- Farm is now entirely running CentOS 7.7
- Batch system has been transitioned to Slurm
- `swif2` workflow software being rolled out
- Significantly increased farm resources over past year
  - Disk: Lustre: 3.8 PB, Work: 465 TB
  - CPU: 12330 cores / 24660 threads. Total capacity 215 M-core-hours/year
  - Almost half the capacity is on AMD EPYC 7502 64C/128T systems (speed demons!)
- Mass storage system
  - Throughput $\approx$ 7 GB/s (uncompressed, theoretical)
  - $\approx$ 150 PB capacity (LTO-8, uncompressed)
  - Significant capacity headroom (more frames, LTO-9) with current silo, up to $\approx$ 325 PB.
- **Tape issue**
  - LTO-8 tapes written between $\approx$ August and December 2020 may be corrupted
  - Mostly raw data!
  - Duplicates written to M-8 tapes appear OK. Recovery underway.

Next Analysis Workshop?

Results of 2019 survey re topics for next analysis workshop

	1	2	3	4	5	Score
Advanced ROOT	1		4	5	6	3.94
Hall A simulations	1	2	2	4	4	3.62
Analysis in Python		2	5	3	3	3.54
Cross section analysis	1	2	2	5	3	3.54
Hall C simulations	1	2	4	3	3	3.38
Batch farm usage	1	3	3	3	3	3.31
Example analyses	1	5	1	2	4	3.23
Plugin modules	1	4	3	2	3	3.15
Replay scripts	1	5	2	2	3	3.08
Optics optimization	3		6	1	3	3.08
Counting house computing	3	3	2	2	3	2.92
Detector calibration	3	1	5	2	2	2.92
Asymmetry analysis	3	2	2	5	1	2.92
Intermediate ROOT	3	3	7			2.31
Basic ROOT	7	5			1	1.69

Standouts

- Advanced ROOT (e.g. dataframes)
- Python analysis (e.g. PyROOT, uproot)
- Simulations (not quite sure what to cover)
- Actual physics analyses, esp. cross-sections

We should start planning. Date, length, format, contents . . .

Summary

- "Podd" analysis software continues to be actively maintained and used by current experiments
- Significant development work (multithreading etc.) underway for SBS
- Hall A analysis may migrate to a new framework, e.g. ARIEL, in the medium term as our demands on flexibility and performance rise
- Another analysis workshop will be coming, perhaps this summer