Proio: YAIO!

David Blyth

Introduction

- A new IO scheme has been written, and it’s called **proio**.
- Proio is a language-neutral IO scheme that utilizes Google’s Protobuf, and was inspired by ProMC and EicMC.
- This presentation will attempt to *motivate* proio, and describe
  - the way it works,
  - how it’s intended to be used,
  - and the current status of the project.

Why create YAIO (Yet Another IO scheme)?

In descending order of importance:

1. To promote collaboration... It would be great if it were EASY to share “data” at all steps in the simulation/reconstruction chain.
2. Allow physicists freedom of choice when it comes to programming language.
   a. **Critical** to this is having a scheme that maintains consistency between languages. ROOT IO and LCIO pose difficulty in extending to/maintaining in multiple languages.
3. Take advantage of IT industry (use Protobuf)
   a. Let them do the hard coding!
   b. Do more, code less!

Pros and Cons of Protobuf

Pros

- Widely used and actively developed in many languages by IT industry
- As in text formats like JSON, Protobuf uses “field” identifiers, allowing forward and backward compatibility
- Unlike JSON, Protobuf is binary, not text, allowing much greater IO performance
- “Varints” provide intelligent compression of integer numbers, increasing space efficiency

Cons

- Protobuf doesn’t do all the work for us
  - ⇒ ProMC, EicMC, proio
- “Field” identifiers reduce space efficiency, depending on the data format

Options for IO Formats

Consider...

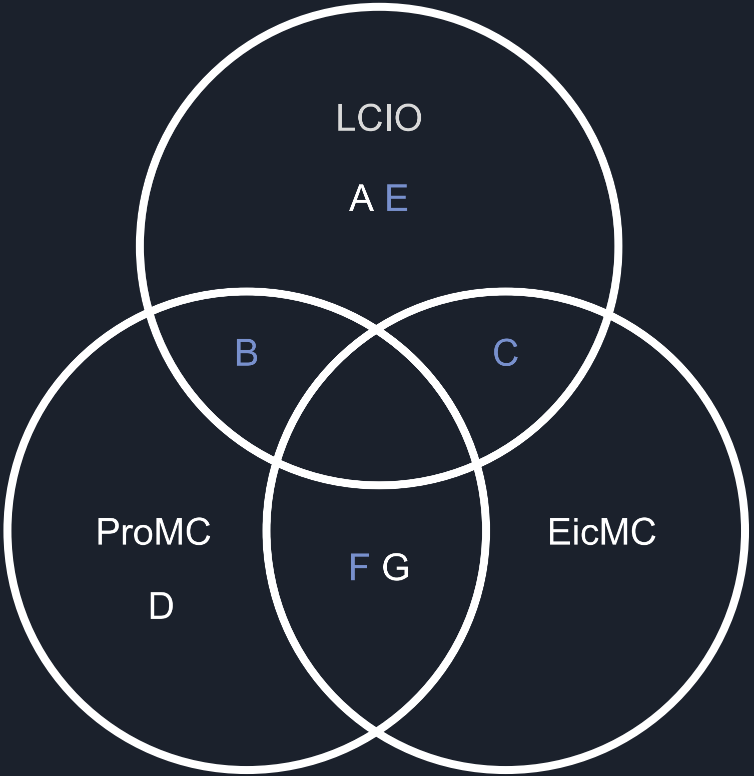
- LCIO
  - Since data model is hard-coded into multiple languages, each implementation becomes fragmented
- ROOT IO
  - Highly flexible
  - Large dependency
  - Further discussion of whether or not ROOT IO is appropriate for us is outside the scope of this presentation
- ProMC (S. Chekanov)
  - Event generator-specific (more discussion to come)
- EicMC (A. Kiselev)
  - Event generator-specific (more discussion to come)

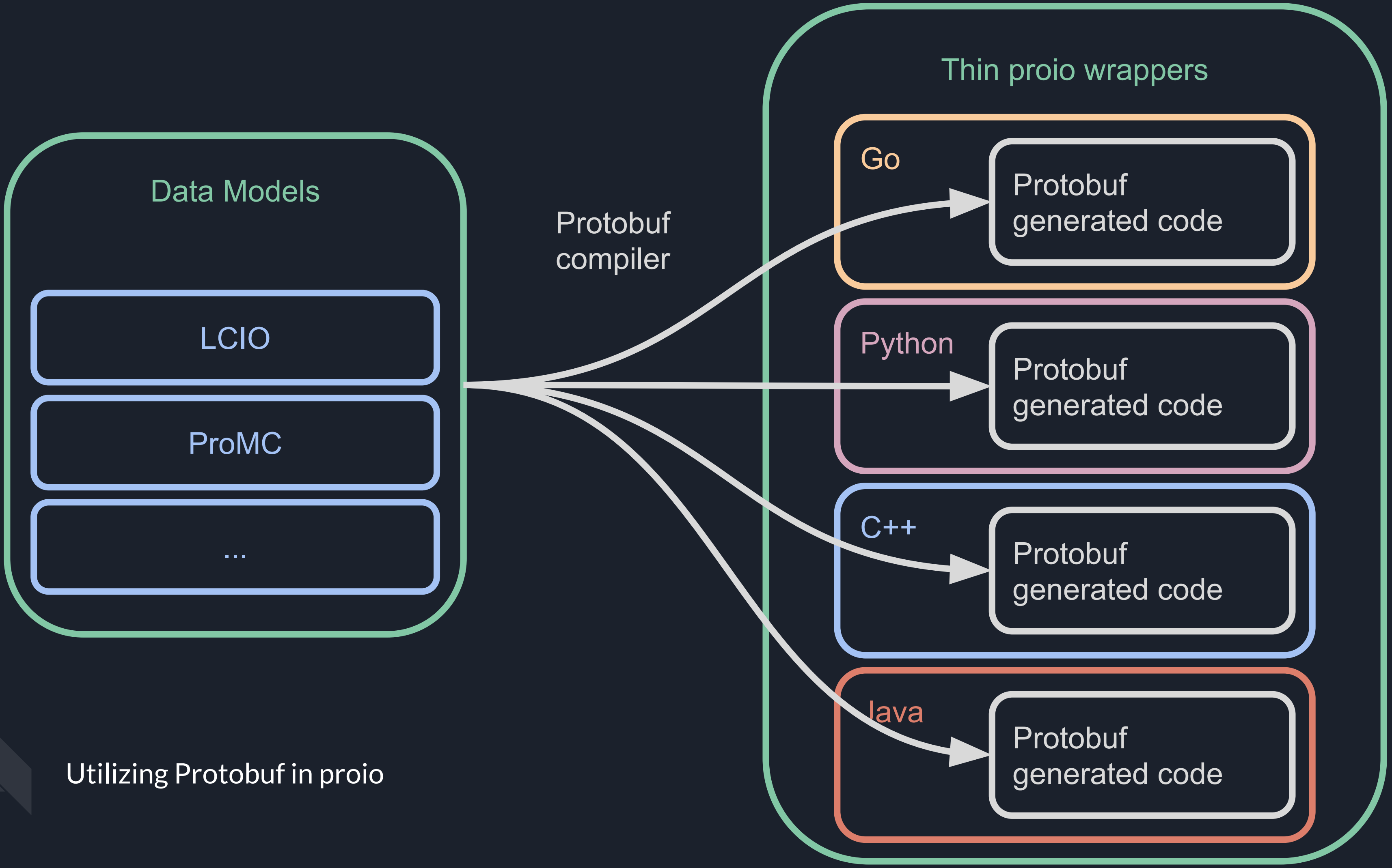
IO Features in a Venn Diagram

- Features
  A. Manually-coded data model
  B. Multiple languages
  C. Compressed stream of events
  D. Events indexed by zip
  E. Allows persistent references between objects
  F. Protobuf
  G. Evgen-oriented

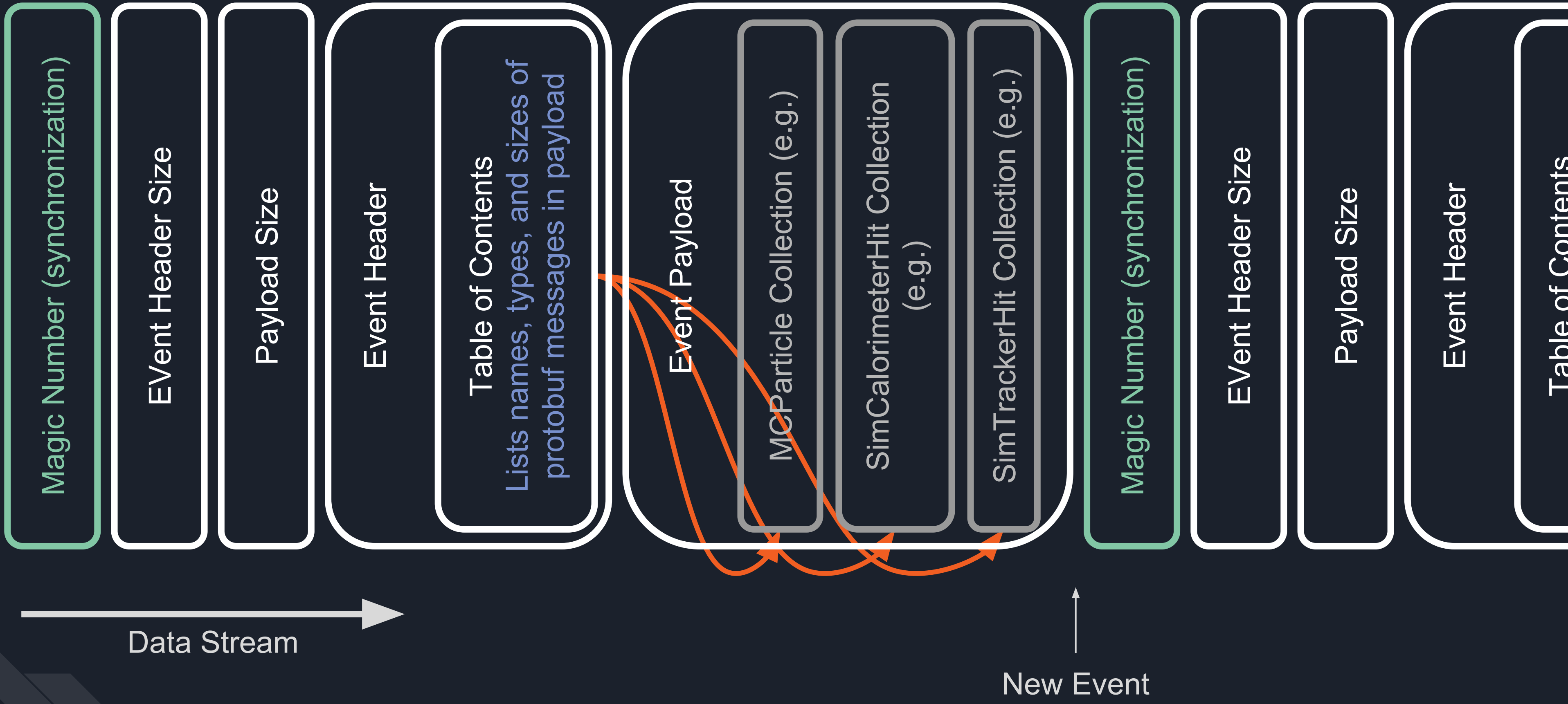
Assert: blue features are desired for forward-looking IO scheme

⇒ proio





Utilizing Protobuf in proio


Magic Number (synchronization)
EVent Header Size
Payload Size
Event Header
Table of Contents
Lists names, types, and sizes of protobuf messages in payload
Event Payload
MCParticle Collection (e.g.)
SimCalorimeterHit Collection (e.g.)
SimTrackerHit Collection (e.g.)
Magic Number (synchronization)
EVent Header Size
Payload Size
Event Header
Table of Contents
Data Stream
New Event

Data format managed by thin wrappers

Proio data structure vs. ProMC/EicMC

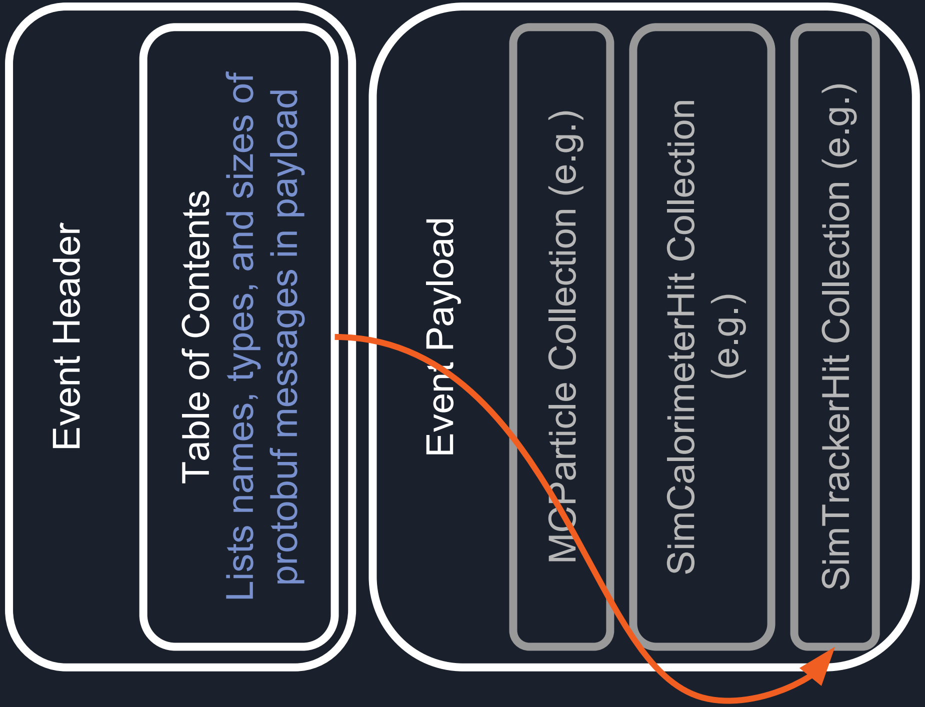
ProMC/EicMC		Proio
MC event oriented	→	Collection oriented
Entire event is deserialized at once	→	Collections are deserialized at once
Contains specific structure for evgens	→	No specific structure beyond basic events/collections

- This difference makes Proio potentially better suited for a broad data model with *multiple interfaces*, because collections can be randomly deserialized.
- Does not make proio better suited as an event generator interface

Example of Random Collection Access

Scenario:

- File/stream is at the output of the simulation
- Contains simulated hits for calorimeters and trackers
  - Calorimeter hit collection is very large compared to tracker hit collection
- Application would like to digitize tracker hits and fit tracks *only*
- In proio, calorimeter hits are not needlessly deserialized


Event Header
Table of Contents
Lists names, types, and sizes of protobuf messages in payload
Event Payload
MCParticle Collection (e.g.)
SimCalorimeterHit Collection (e.g.)
SimTrackerHit Collection (e.g.)

Proio Streams and Files

- Proio creates a stream of events that can be either compressed or uncompressed (similar to EicMC)
- Streams/files can be arbitrarily large (i.e., the size is not limited by proio)
  - However, event sizes are limited to ~2GiB
- Proio is compatible with
  - gzip/gunzip command-line tools
    - A compressed proio file simply has a .proio.gz suffix
  - Unix pipes
  - Concatenation of files
  - Arbitrary cutting of streams/files
    - Currently for uncompressed streams only
    - Enabled in part by magic number synchronization

<u>Piping:</u>

```
lcio2proio sample.slcio | proio-ls -
```

<u>Concatenating:</u>

```
$ cat sample1.proio sample2.proio > allsamples.proio
```

<u>Cutting:</u>

```
dd if=all.proio of=roughtCut.proio bs=1M count=1 skip=1
proio-strip -o cleanCut.proio roughCut.proio
```

Proio Base Tools

- Most tools are written in Go
    - High portability
        - Single command to download and install Go package
        - Statically linked by default (can deploy executables only)
    - High performance

<u>Tools</u>

- proio-ls (Go)
    - Dump events from stream/file
- proio-summary (Go)
    - Read all events and summarize
- proio-strip (Go)
    - Strip collections from event or just reserialize to clean up data
- lcio2proio (Go)
    - Converter from LCIO to proio
- proio2root (C++)
    - Convert to ROOT file
    - Still needs some additional work

Proio Data Models

- Currently LCIO and ProMC data models exist in the proio repository
- Any number of additional parallel models can be added without affecting one another
- Models can be extended without breaking the ability to read older data, or have older libraries read new data
- Changing or adding data models requires no manual coding

```
model/
    proio.proto
    lcio/
        lcio.proto
    promc/
        promc.proto
```

Proio Data Models

- EIC community could, for example
  - Agree to use LCIO as a base model, and rename it eicio
  - Add optional extensions for each effort's needs
  - In this way, the EIC community could share a core data model for interoperability, while allowing extension without breaking forwards or backwards compatibility
- OR
  - Each experiment could maintain a parallel data model within proio
  - At least then we could use the same tool to read/write data for each effort

```
model/
    proio.proto
    lcio/
        lcio.proto
    promc/
        promc.proto
    eicio/
        eicio.proto
        anl.proto
        ...
```

Go installation

```
$ go get github.com/decibelcooper/proio/go-proio/…
```

This single command acquires and builds the Go library along with most of the base proio tools. Provided that $GOPATH and $PATH are set up appropriately, the tools are then immediately available.

Installation for other languages

Canonical build systems chosen for each language:

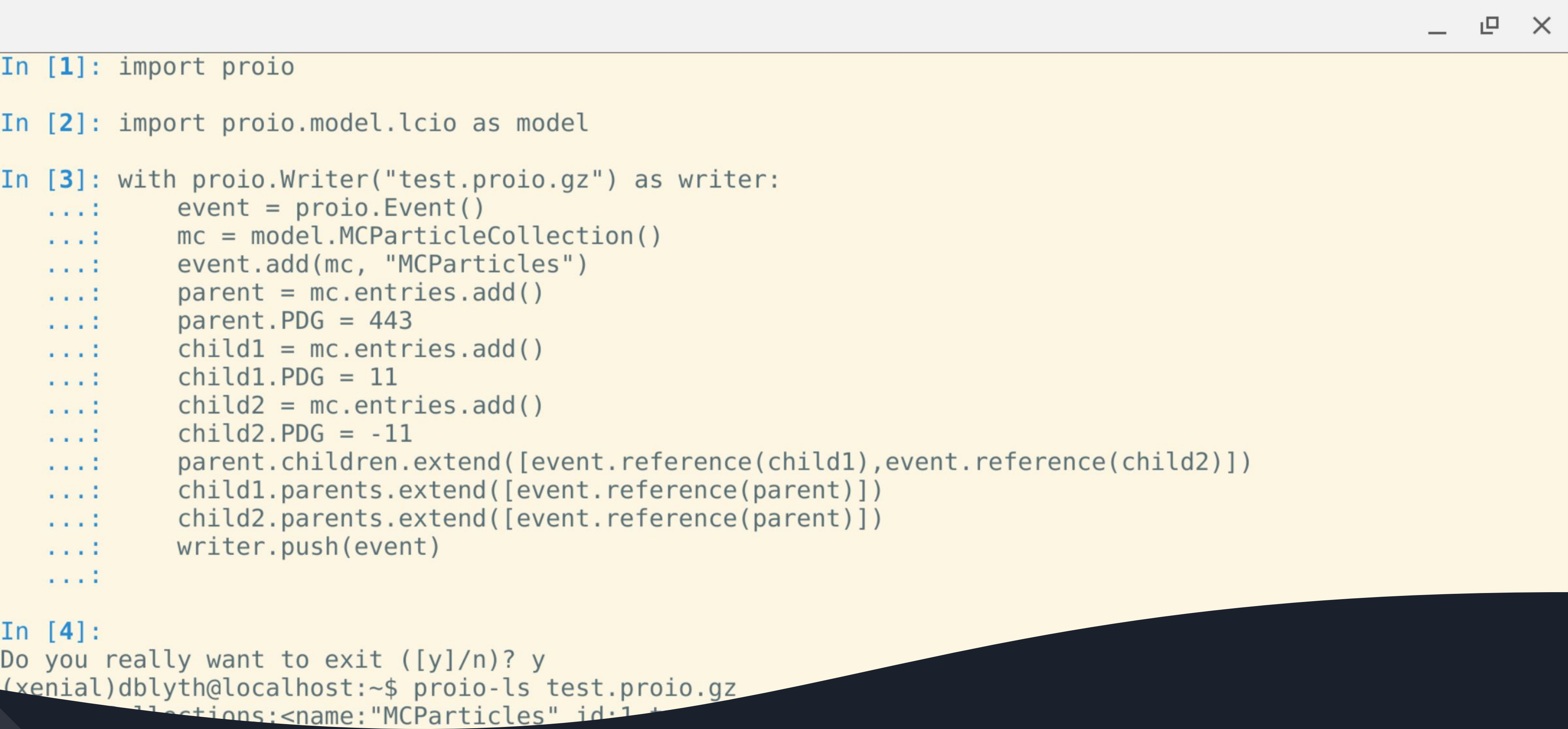
- Go
  - go get
- Python
  - pip install
- C++
  - cmake
- Java
  - mvn install

Please see the appropriate subdirectory in https://github.com/decibelcooper/proio for more details

Python example

Install with…

```
$ pip install --user proio
```



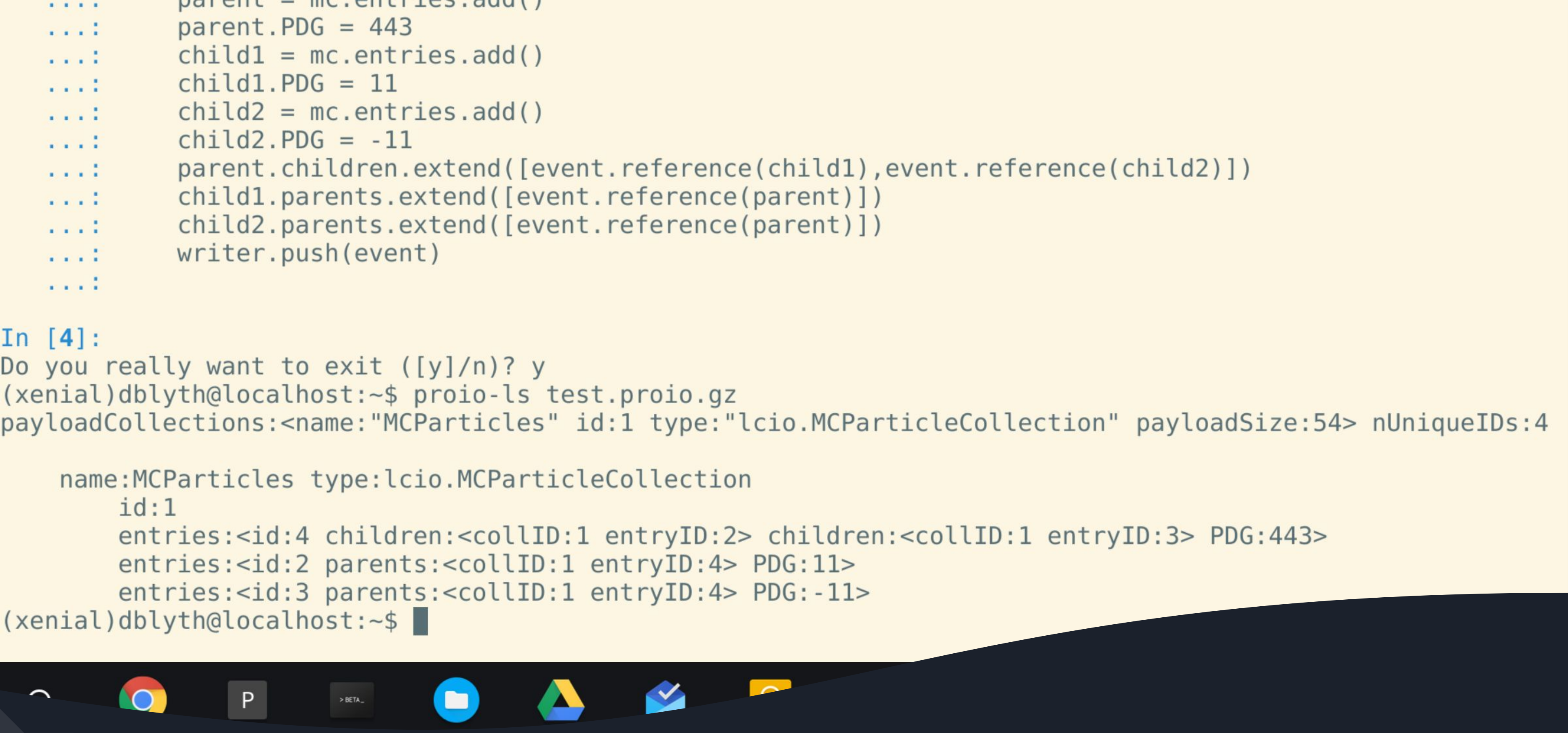
```
In [1]: import proio

In [2]: import proio.model.lcio as model

In [3]: with proio.Writer("test.proio.gz") as writer:
   ...:     event = proio.Event()
   ...:     mc = model.MCParticleCollection()
   ...:     event.add(mc, "MCParticles")
   ...:     parent = mc.entries.add()
   ...:     parent.PDG = 443
   ...:     child1 = mc.entries.add()
   ...:     child1.PDG = 11
   ...:     child2 = mc.entries.add()
   ...:     child2.PDG = -11
   ...:     parent.children.extend([event.reference(child1),event.reference(child2)])
   ...:     child1.parents.extend([event.reference(parent)])
   ...:     child2.parents.extend([event.reference(parent)])
   ...:     writer.push(event)
   ...:

In [4]:
Do you really want to exit ([y]/n)? y
(xenial)dblyth@localhost:~$ proio-ls test.proio.gz
    llections:<name:"MCParticles" id:1
```

Python write example

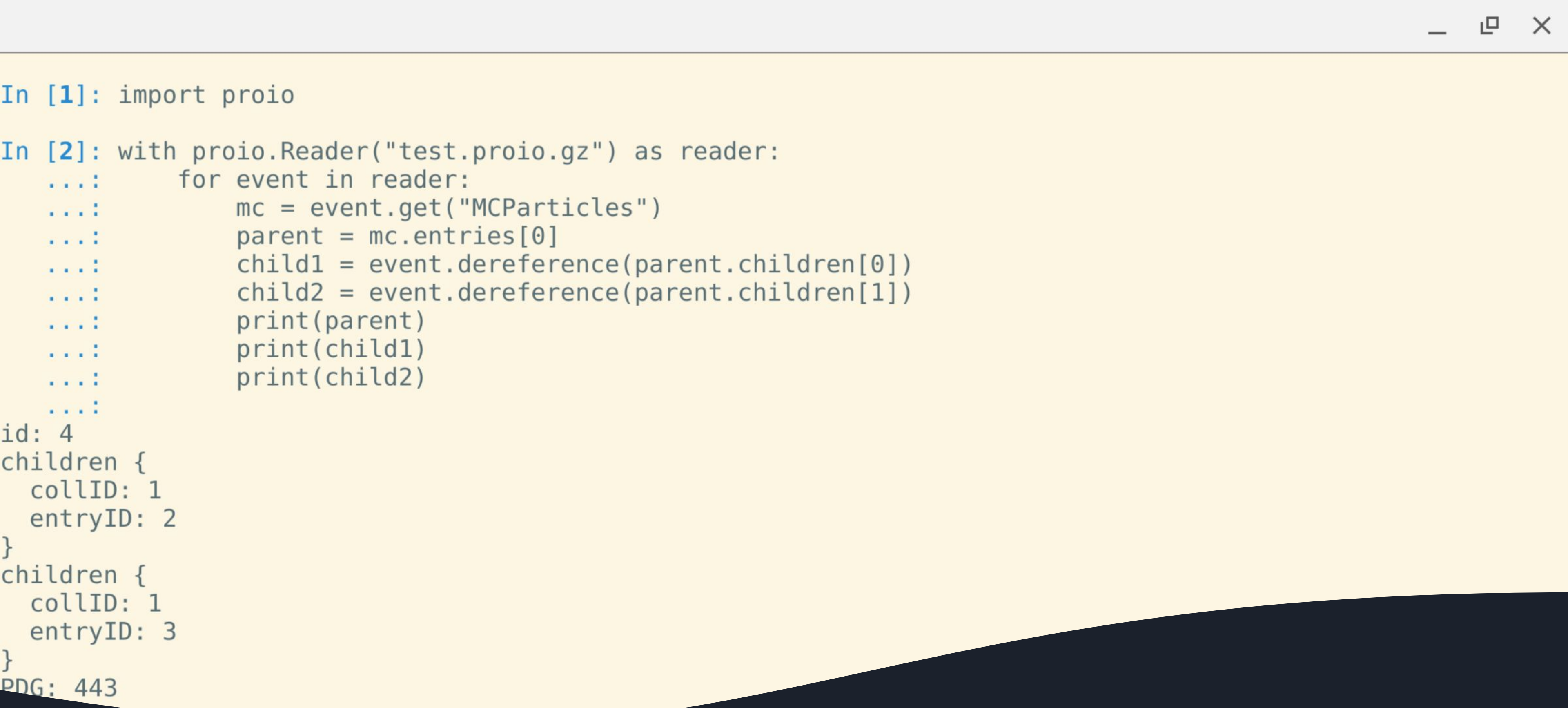


```
   ...:     parent = mc.entries.add()
   ...:     parent.PDG = 443
   ...:     child1 = mc.entries.add()
   ...:     child1.PDG = 11
   ...:     child2 = mc.entries.add()
   ...:     child2.PDG = -11
   ...:     parent.children.extend([event.reference(child1),event.reference(child2)])
   ...:     child1.parents.extend([event.reference(parent)])
   ...:     child2.parents.extend([event.reference(parent)])
   ...:     writer.push(event)
   ...:

In [4]:
Do you really want to exit ([y]/n)? y
(xenial)dblyth@localhost:~$ proio-ls test.proio.gz
payloadCollections:<name:"MCParticles" id:1 type:"lcio.MCParticleCollection" payloadSize:54> nUniqueIDs:4

    name:MCParticles type:lcio.MCParticleCollection
        id:1
        entries:<id:4 children:<collID:1 entryID:2> children:<collID:1 entryID:3> PDG:443>
        entries:<id:2 parents:<collID:1 entryID:4> PDG:11>
        entries:<id:3 parents:<collID:1 entryID:4> PDG:-11>
(xenial)dblyth@localhost:~$
```

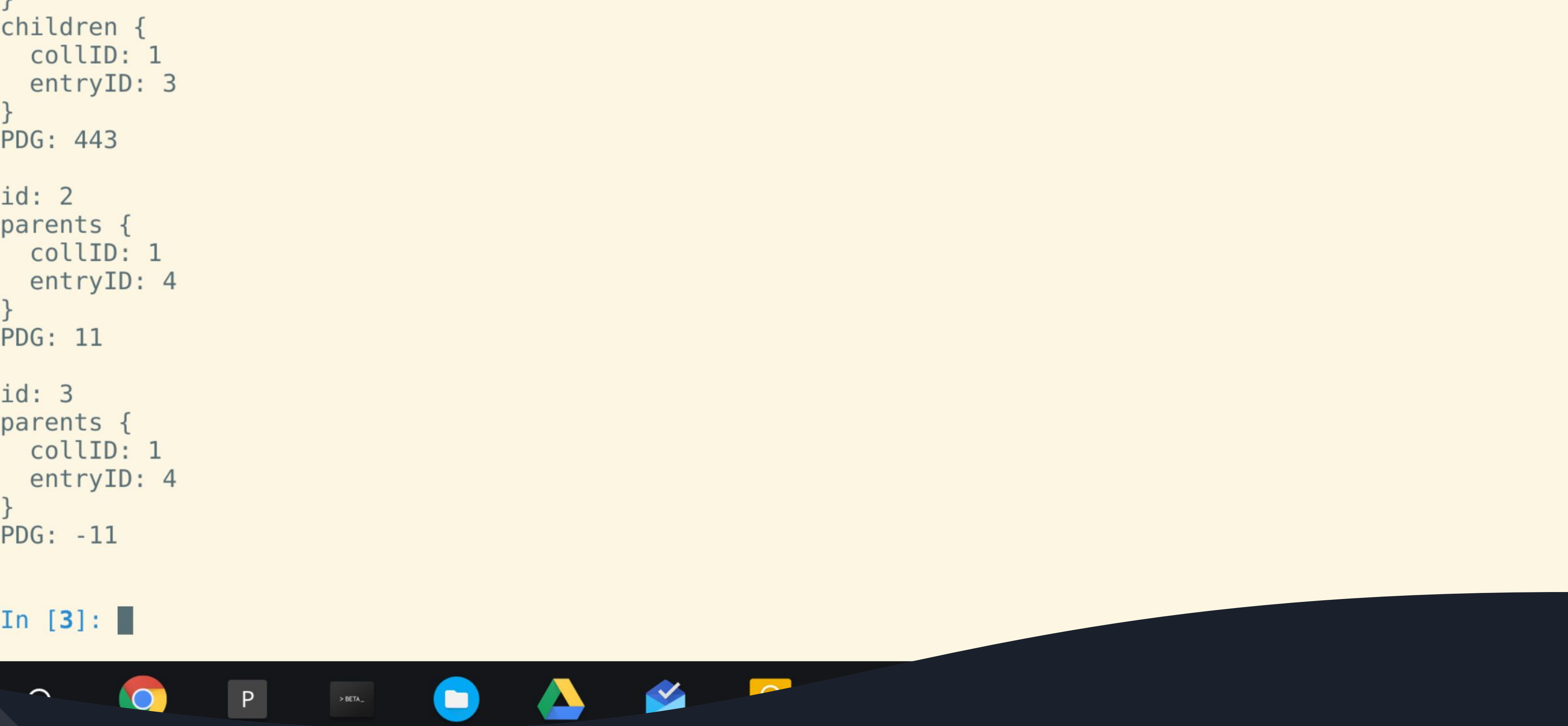
Python write example, cont'd



```
In [1]: import proio

In [2]: with proio.Reader("test.proio.gz") as reader:
   ...:     for event in reader:
   ...:         mc = event.get("MCParticles")
   ...:         parent = mc.entries[0]
   ...:         child1 = event.dereference(parent.children[0])
   ...:         child2 = event.dereference(parent.children[1])
   ...:         print(parent)
   ...:         print(child1)
   ...:         print(child2)
   ...:
id: 4
children {
  collID: 1
  entryID: 2
}
children {
  collID: 1
  entryID: 3
}
PDG: 443
```

Python read example



```
}
children {
  collID: 1
  entryID: 3
}
PDG: 443

id: 2
parents {
  collID: 1
  entryID: 4
}
PDG: 11

id: 3
parents {
  collID: 1
  entryID: 4
}
PDG: -11


In [3]:
```

Python read example

File Size Benchmarks

Data set	Proio size	LCIO size	ProMC size	Comments
Pythia8 35 GeV DIS MC (50K events)	24 MiB	67 MiB	37 MiB	Sparse information (zero-vector position, e.g.)
Lepto 35 GeV DIS MC (50K events)	27 MiB	56 MiB	33 MiB	“”
Pythia8 35 GeV DIS Recon. (500 events)	24 MiB	22 MiB	NA	Dense information
Pythia8 14 TeV t tbar (10K events)	482 MiB	390 MiB	308 MiB	Elaborate ancestry. Many parents/children for some particles.

Performance Benchmarks (Go only)

Scenario:

Analysis routine for calculating track efficiency. Reading from file with full reconstruction data

Time / Event is dominated by event read time in .proio and .slcio cases for this scenario

Caveat: Go LCIO library is likely not as optimized as C++ LCIO library.

File Format	Time / Event
.proio	~200 μs
.proio.gz	~2 ms
.slcio	~45 ms

Future Work

- Continue to clean up build systems for
  - Protobuf code generation
    - Currently a vanilla GNU Make build
    - Will be converted to a more sophisticated CMake build
  - C++ library
    - CMake build needs a bit more fine tuning
- Add ROOT dictionaries for C++ library
  - No desire to alienate people that are comfortable with ROOT
  - Hope to have a high degree of interoperability with ROOT
- Create GTK3 graphical browser
  - Will also be written in Go
- Improve Dereference() performance
- Add write capability to Java library
  - Currently it is read-only