Small tweaks to improve throughput

Kyle J. Knoepfel
9 April 2019

Small tweaks

I've been profiling some MicroBooNE workflows; there are places in LArSoft that have come up as taking a lot of unnecessary CPU time.

Only "breaking" changes I present today:

```
     if (( numberPhotons < std::numeric_limits<double>::epsilon() )
         || ( energy<=std::numeric_limits<double>::epsilon()      ))
     {
-      // will throw
-      MF_LOG_ERROR("OpDetBacktrackerRecord")
+      MF_LOG_DEBUG("OpDetBacktrackerRecord")
       << "AddTrackPhotons() trying to add to iTimePDclock #"
```

Files:

- `lardataobj/Simulation/OpDetBacktrackerRecord.cxx` (shown above)
- `lardataobj/Simulation/SimChannel.cxx`

Feature branch at `lardataobj:feature/knoepfel_suppress_logging`.

Improvements in exponentiation

A lot of uses of `pow`, where other options are more efficient.

- Prefer `std::sqrt` over `std::pow(..., 0.5)`

```
- geom->WirePitch(0)/cosgamma * pow(dir.Mag2(),0.5) / pow(dir_tmp.Mag2(),0.5);
+ geom->WirePitch(0)/cosgamma * dir.Mag() / dir_tmp.R();
```

- Prefer compile-time expressions over run-time

```
#include "cetlib/pow.h"

- double distance = pow((trkw[i]-w0)*wire_pitch,2)+pow(trkx0[i]-x0,2);
+ double distance = cet::sum_of_squares((trkw[i]-w0)*wire_pitch, trkx0[i]-x0);
```

art::Ptrs (1)

art::Ptrs provide a persistable means of referring to product elements. **They are expensive. If you do not need to persist a reference, do not use an art::Ptr.** In larreco:

```
double dist(art::Ptr<Pos> const& a, art::Ptr<Pos> const& b) {
  double const dx = b->x - a->x;  // 2 expensive dereferences
  double const dy = b->y - a->y;  // 2 more expensive dereferences
  double const dz = b->z - a->z;  // Yet 2 more expensive dereferences
  return std::sqrt(dx*dx + dy*dy + dz*dz);
}
```

art::Ptrs (1)

art::Ptrs provide a persistable means of referring to product elements. **They are expensive. If you do not need to persist a reference, do not use an art::Ptr.** In larreco:

```
double dist(art::Ptr<Pos> const& a, art::Ptr<Pos> const& b) {
  double const dx = b->x - a->x;  // 2 expensive dereferences
  double const dy = b->y - a->y;  // 2 more expensive dereferences
  double const dz = b->z - a->z;  // Yet 2 more expensive dereferences
  return std::sqrt(dx*dx + dy*dy + dz*dz);
}
```

Better:

```
double dist(art::Ptr<Pos> const& a, art::Ptr<Pos> const& b) {
  auto const& a_pos = *a; // 1 expensive dereference
  auto const& b_pos = *b; // 1 expensive dereference
  double const dx = b_pos.x - a_pos.x;
  double const dy = b_pos.y - a_pos.y;
  double const dz = b_pos.z - a_pos.z;
  return std::sqrt(dx*dx + dy*dy + dz*dz);
}
```

`art::Ptr`s (2)

`art::Ptr`s provide a persistable means of referring to product elements. **They are expensive. If you do not need to persist a reference, do not use an `art::Ptr`.** In `larreco`:

```
double dist(art::Ptr<Pos> const& a, art::Ptr<Pos> const& b) {
  double const dx = b->x - a->x;  // 2 expensive dereferences
  double const dy = b->y - a->y;  // 2 more expensive dereferences
  double const dz = b->z - a->z;  // Yet 2 more expensive dereferences
  return std::sqrt(dx*dx + dy*dy + dz*dz);
}
```

The best option is to not use `art::Ptr`s at all:

```
double dist(Pos const& a, Pos const& b) {
  double const dx = b.x - a.x;
  double const dy = b.y - a.y;
  double const dz = b.z - a.z;
  return std::sqrt(dx*dx + dy*dy + dz*dz);
}
```

Chose the one-time dereference option, change already pushed to `develop`.