Bulk IO Update for PPP

Brian Bockelman

14 November 2018

Bulk IO Recap

- The bulk IO interface to a TTree (ROOT PR #2519) provides the caller with the ability to ask for objects corresponding to an array of events (implementation: returns TBuffer from a TBasket).
  - Tradeoff is that a limited set of types can be supported by bulk IO.
  - Lowest-level bulk IO interface is exported by `ROOT::Experimental::Internal::TBulkBranchRead`, accessible by `TBranch::GetBulkRead()` method.
    - Bulk IO object exports `GetEntriesSerialized(Int_t event_num, TBuffer&)`; on success, buffer is filled with event data.
    - **Challenge**: caller must handle transform from buffer to C++ objects. Simpler in Python as this converts very naturally to a NumPy array.
- PR contains a `TTreeReader`-like interface, but this only works for types that work with bulk IO. "Exercise for user" to determine this!

Q: What's the best approach for using Bulk IO from C++?

A: `RDataSource`!

- `RDataSource` has type information prior to the execution of the data frame. *Hence, there's opportunity to determine whether bulk IO can be used.*
  - We can fallback to “normal IO” in the case it can't.
- Accordingly, I went ahead and did a prototype RRootBulkDS to determine whether `RDataFrame` applications could benefit from bulk IO.
  - Take-home #1: RDataFrame can benefit from bulk IO.
  - Take-home #2: Not as fast as “raw” bulk IO, but there are opportunities for improvement.

Let's see what we can do!

Implementation Details

- See implementation for more details than I can fit in slide.
- The data source internally has a "buffer manager" object that keeps track of a `TBuffer` per branch.
- When SetEntryRange is called, we invoke the bulk IO API to prepare the buffer per branch.
- When SetEntry is called, we iterate through all the active branch buffers, advance the pointers within the buffer, and perform the correct deserialization operation (e.g., byteswap).
- **Limitations of prototype** (not fundamental, just needs implementation):
  - Assumes basket size == cluster size.
  - Only a small number of types implemented.

Aside on test methodology

- All numbers presented here are based on this branch:
  - https://github.com/bbockelm/root/tree/rrootbulkds
  - Code samples shown here are cleaned-up / simplified from this branch.
- In particular, they can be reproduced by running build target `tree/treeplayer/datasource_rootbulk` from that branch.
- For this test:
  - Numbers were run on a 2.3 GHz Haswell-class Xeon processor.
  - Release build with debug symbols.
  - Input dataset is ~430MB: too big for the processor's L3 cache, but small enough to stay in page cache.
- I expect the ratios between cases to be consistent but absolute numbers to vary based on the test setup.

Test #1: Raw Bulk IO

- Code:
  - Iterates through all the events
  - Calls GetEntriesSerialized to receive a buffer of objects.
  - Deserializes the objects inline.
  - Does “something silly” with the data.
  - Bumps the index counter.
- Extremely fast: 450MHz.

```
while (events) {

   auto count = branchI->GetBulkRead().\
       GetEntriesSerialized(evt_idx, branchbuf);
   events = events > count ? (events - count) : 0;
   int *entry = (int *)branchbuf.GetCurrent();

   for (Int_t idx=0; idx < count; idx++) {
      Int_t tmp = *(Int_t*)(&entry[idx]);
      char *tmp_ptr = (char *)&tmp;
      int val;
      frombuf(tmp_ptr, &val);
      if (val > max_bulk) {max_bulk = val;}
   }

   evt_idx += count;
}
```

Test #2: Invoke RDataSource Directly

- Code:
  - Directly creates a `RRootBulkDS`.
  - Performs appropriate initialization.
  - Iterates through each “range” (here, each basket is a range).
  - Sets entry for each event.
- Very fast: 160MHz.
- Opportunities for speedup:
  - Compiler can’t currently inline `SetEntry` implementation.   Appears fixable.
  - This introduces a function call per event - opportunities for a function call per range?

```
RRootBulkDS tds(treeName, fileName);
tds.SetNSlots(1);
auto vals = tds.GetColumnReaders<int>("myInt");
tds.Initialise();
auto ranges = tds.GetEntryRanges();
Int_t max3 = 0;
for (auto &&range : ranges) {
    tds.InitSlot(0U, range.first);
    for (int i = range.first;
         i < range.second;
         i++)
    {
       tds.SetEntry(slot, i);
       auto val = **vals[slot];
       if (val > max3) {max3 = val;}
    }
 }
```

Test #3: RDataFrame with RRootBulkDS

- Code:
  - Creates a data source
  - Creates a data frame
  - Triggers computation on one branch.
- Fast: 42MHz.
- Opportunities for speedup:
  - Can it effectively devirtualize the data source?!?
  - Any way to JIT larger parts of the event loop in the loop manager?
- Potential for bad measurements:
  - Is there a better way to measure event rate *minus* startup costs?

```
std::unique_ptr<RDataSource>
    rds2(new RRootBulkDS(treeName, fileName));
RDataFrame rdf2(std::move(rds2));
auto max2 = rdf2.Max<int>("myInt");
```

And everything else

- Standard `RDataFrame` (no bulk IO) executes at 14MHz; bulk data source sees immediate significant improvement.
    - Compared to this reference, RDF + bulk DS is 3x faster; invoking bulk DS directly is 11x faster; raw bulk IO is 32x faster.
    - Further opportunities exist: what's the end-goal? *Seems unrealistic to expect it to be comparable with low-level code…*
- Is time better spent "making it faster" or "making it more feature complete"?