ROOT I/O

PH. CANAL, D. PIPARO

FOR THE I/O TEAM.

COLLABORATORS

- Ph Canal - FNAL
- P. Van Gemmeren - ANL
- G. Amadio, J. Blomer, E. Guiraud, D. Piparo, E. Tejedor - CERN
- S. Linev - GSI
- B. Bockelman, J. Pivarski, O. Shadura, Z. Zhang - UNL/Princeton

RECENT IMPROVEMENTS

- `TBufferFile` has been split in `TBufferText` and `TBufferIO`
  - Will allow to remove the lower level virtual function call
- `StreamerInfo` record no longer processed if (hash) already seen [hash == sha256]
  - save time and global lock contention (3x runtime 8 logical cores with small files, ~1MB, added to a TChain)
- `StreamerInfo` record now compressed with same level (or lack thereof) as the rest of the file.
- Allow writing temporary objects (with same address) in the same `TBuffer`(s).
  - new parameter cacheReuse added to `TBuffer*::WriteObject`
- Switch (and fix) Parallel unzipping to use TBB tasks.
  - mutually exclusive with `TTree::GetEntry` parallelised with TBB
- Added `TKey::ReadObject<typeName>`

MORE CHANGES

- LZ4 compression is now the default and can be read by:
  - v5.34/38
  - v6.08/06 [not yet released]
  - v6.10/08
  - v6.12/02
  - v6.14/00
- Revised treatment of tree name in `TChain` URLs
  - Support for the ill-defined way to pass the tree name in the url as '/tree_name' is limited to cases where the substring '.root' is contained in the file name but not in the tree name.
  - This syntax is now deprecated.

JSON / XML

- Implemented reading of objects data from JSON
- Provide `TBufferJSON::ToJSON()` and `TBufferJSON::FromJSON()` methods
- Provide `TBufferXML::ToXML()` and `TBufferXML::FromXML()` methods
- Converts NaN and Infinity values into null in JSON, there are no other direct equivalent
- Compaction of arrays within the JSON output

DATA FRAME: READING ARBITRARY FORMATS WITH DATA SOURCES

- `RDataSource` interface available to the user to develop input sources for `RDataFrame`
- Potentially any columnar dataset can be read
  - Current examples: CSV, Arrow, in-memory, LHCb MDF format (demostrator), Atlas XAOD (demostrator)
- Same analysis, different source dataset
- Can always convert to `TTree` in `TFile` thanks to "dataset snapshot capability" of `RDataFrame`!

ROOT GLOBAL LOCK: NOW A READ-WRITE LOCK

- Resolved the race conditions inherent to the use of the **RecursiveRemove** mechanism.
- Introduced `ROOT::TReentrantRWLock`,
  - Reentrant read-write lock with a configurable internal mutex/lock and a condition variable to synchronize readers and writers when necessary.
- Allows a single reader to take the write lock without releasing the reader lock. It also allows the writer to take a read lock. In other words, the lock is re-entrant for both reading and writing.
- Tries to make faster the scenario when readers come and go but there is no writer. In that case, readers will not pay the price of taking the internal mutex.
- Tries to be fair with writers, giving them the possibility to claim the lock and wait for only the remaining readers, thus preventing starvation.

ROOT GLOBAL LOCK: NOW A READ-WRITE LOCK

- Switched the ROOT global to be a `ROOT::TReentrantRWLock` and renamed it `ROOT::gCoreMutex`. The old name `gROOTMutex` and `gInterpreterMutex` are deprecated and may be removed in future releases.
- Added `TReadLockGuard`, `TWriteLockGuard`, R__READ_LOCKGUARD and R__WRITE_LOCKGUARD to take advantage of the new lock. The legacy `TLockGuard` and R__LOCKGUARD use the write lock.
- Improved scaling of `TROOT::RecursiveRemove` in the case of large collection.
- Added a thread safe mode for the following ROOT collections: `THashList`, `THashTable`, `TList` and `TObjArray`. When ROOT's thread safe mode is enabled and the collection is set to use internal locks by:
  - collection->UseRWLock();
- all operations on the collection will take the read or write lock when needed, currently they shared the global lock (`ROOT::gCoreMutex`).

TBRANCH AND TREE

- `TBranch::BackFill`
- Use this instead of `TBranch::Fill` to create a well clustered file
- to catch up with the number of entries already in the TTree.
  - for(auto e = 0; e < tree->GetEntries(); ++e) { // loop over entries.
  - for(auto branch : branchCollection) {
  - ... Make change to the data associated with the branch ...
  - branch->BackFill();
  - }
  - }

TTREE 2 NUMPY

- Convert in PyROOT a TTree to a numpy array: TTree.AsMatrix

```
myTree # Contains branches x and y of type float

# Convert to numpy array and apply numpy methods
myArray = myTree.AsMatrix()
m = np.mean(myArray, axis = 0)

# Read only specific branches, specify type
xAsInts = myTree.AsMatrix(columns = ['x'], dtype = 'int')
```

SECONDARY TTREE CACHE MISS

- Off by default.
- Tracks all branches used **after** the learning phase (cache misses).
- On cache miss, prefetch one basket for each of those branches

- Helps in the case when infrequently accessed branches are accessed together.
  - For example: select and skim

- NOTE - when this mode is enabled, the memory dedicated to the cache will up to double if there cache misses.

PRELOADING AND RETAINING UNCOMPRESSED CLUSTERS

- Can prevent additional reads from occurring when reading events out of sequence.
- To decompress a full cluster at time
  - Use TTree::SetClusterPrefetch.
- Retain previous cluster by setting MaxVirtualSize to a negative value
  - Absolute value indicates how many additional clusters will be kept in memory

SIGNIFICANT REVAMP OF TTREECACHE::FILLBUFFER

The new scheme insures a much more stable and efficient behavior in case of low memory given by the user compared to the size of the buffers or 'odd' basket layout.

The basket collection is now done in 4 phases:

1. One basket per branch, basket must contain the requested entry and is not yet loaded or used,
2. Even out by adding baskets so that all branches reach the same entry (or close)
3. Add the remaining branches from the current cluster.
4. Add the basket from the beginning of the cluuster to the current entry (if any)

then repeat the 4 steps for the next cluster.

SIGNIFICANT REVAMP OF TTREECACHE::FILLBUFFER

The iteration is stopped as soon as the cache is 'full' as defined by these rules:

- During step 1 of the first cluster, continue up to 4 times the user requested cache size
- During steps 2 to 4 of the first cluster, continue up to 2 times the user requested cache sizep
- During steps 2 to 4, the 'first' basket of a branch is accepted up to 4 times the user requested cache size (i.e as if it had been selected during the 1st step)
- During the other clusters, continue up to the user requested cache size
- A basket if rejected/skipped if its individual size is larger than the user requested cache size

In addition, upon seeing a cache miss, `FillBuffer` now detects if all the baskets in the cache have already been used (read from the cache) in which case we can discard them and load the next set of baskets.

TTREE PERF STATS

As a side effect, we now keep a record of which baskets are in the cache and which of those baskets have been used. The `TTreePerfStats` now keeps a complete log of all the baskets that are:

- loaded in the main cache (and how many times)
- loaded in the 'misss' cache (and how many times)
- used
- read directly (complete cache miss)

This will be helpful in understanding situation of over-read or slow operations.

2018 PRIORITIES

- Tweaks to compression algorithm choice, e.g. **LZ4, studying Zstd**
- Increasing C++ standard support: `std::shared_ptr, std::variant`
- Increase internal parallelism: **unzipping, branch reading**
- Increase parallelism support, e.g. `RDataFrame` and **writing same tree from many threads**
- Increase read performance, e.g. with "bulk I/O"
- Reduce file size: **work on redundant info, re-use compression dictionaries**
- Parallel `TTree/TFile` Merging over MPI on-HPC (Summer work @ ANL).

TTREE/TFILE FOR V7: "FOREST" INITIAL PROTOTYPES

- Why is it needed:
  - provide a thread-safe I/O
  - offer modern C++ iterator/cursor interfaces to ROOT data sets
  - offer a well-defined bulk I/O interface
  - allow for arbitrarily nested split collections
  - open the door to (more easily) store time series data in ROOT
- Who can use it:
  - other ROOT code and ROOT users
- Why now:
  - I/O capabilities and interfaces are the foundation of other tasks, e.g. parallel histogram filling
  - New column-wise storage formats/libraries are rising (e.g. Parquet, Arrow), ROOT should not fall behind in terms of performance/features.

FURTHER DOWN THE LINE

- Pass-though I/O
  - Reduce copy to a minimal especially for 'very high speed device'
  - Challenges: compression, platform independence, padding and alignment rules
  - Enables also: faster inter-process communication.
- Object Stores
  - Main advantages: data de-duplication and "cache" sharing (and maybe indexing)
- Improve efficiency for many-core/many-processor architecture
  - i.e. when all processes should not (can not) be writing to the (same) disk(s)

UPCOMING WORKSHOPS

- HEP-CCE: Scalable IO for Energy and Intensity Frontier Experiments – Argonne – August 23, 24
  - http://hepcce.org/?p=2726
- ROOT User's Workshop – Sarajevo – September 10-13
  - https://indico.cern.ch/event/697389

CONCLUSION

- ROOT parallelism greatly improved for 6.14 release
  - In several ways: RW lock, shorter critical sections, optimised
  - Clear benefits for IO: cache unzipping, faster data analysis, writing
- More interoperable
  - ROOT datasets convertible to Numpy natively in ROOT
  - RDataFrame can potentially read any format
- Continuously assessing impact of compression algorithms and strategies
- Working hard on evolution of ROOT I/O "Forest"