CMS Progress on ROOT Multithreaded Output and IMT

Dan Riley (Cornell) & Chris Jones (FNAL)

ROOT I/O Workshop

2018-02-21

ROOT Output Serial Bottlenecks

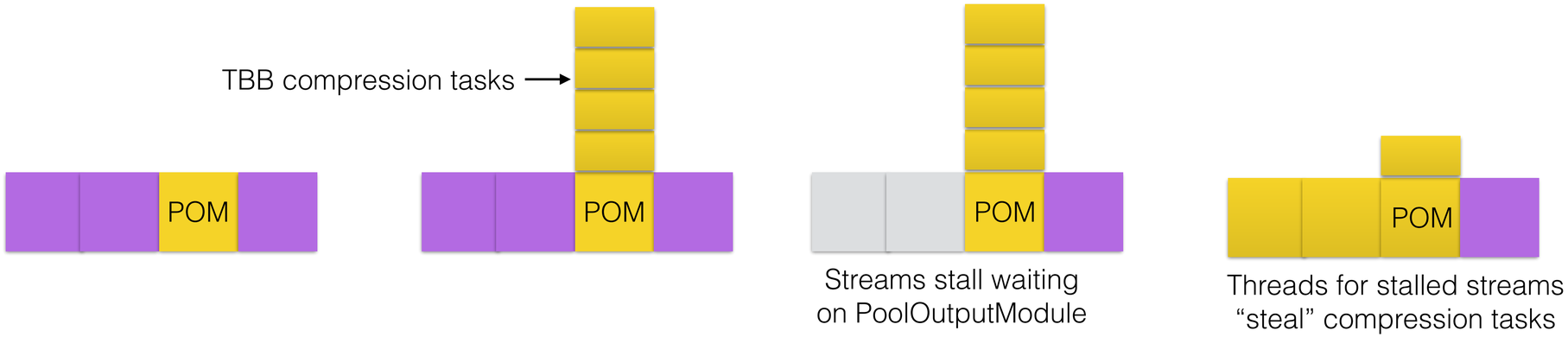
Output file compression is the biggest bottleneck for CMS production jobs

- **ROOT accumulates entries in branch buffers**
- **When the # of entries reaches a configurable threshold, the branch buffers are compressed and written to disk**
- **Compression is, by default, serial in two respects:**
  - Branch buffers are compressed in a single thread
  - The TFile can't be written to while compression and writing is in process
- **Implicit Multi-Threading (IMT) addresses the first**
  - IMT parallelizes branch buffer compression into TBB tasks
- **ParallelPoolOutputModule (PPOM) is meant to address the second**
  - PPOM keeps a pool of output TBufferMergerFiles (derived from TMemFile)
  - Output is written to the available TBufferMergerFile with the most entries
  - Full TBufferMergerFiles are copied to a buffer and written by an auxiliary thread

IMT in Schematic Form

IMT takes advantage of threads that would otherwise stall

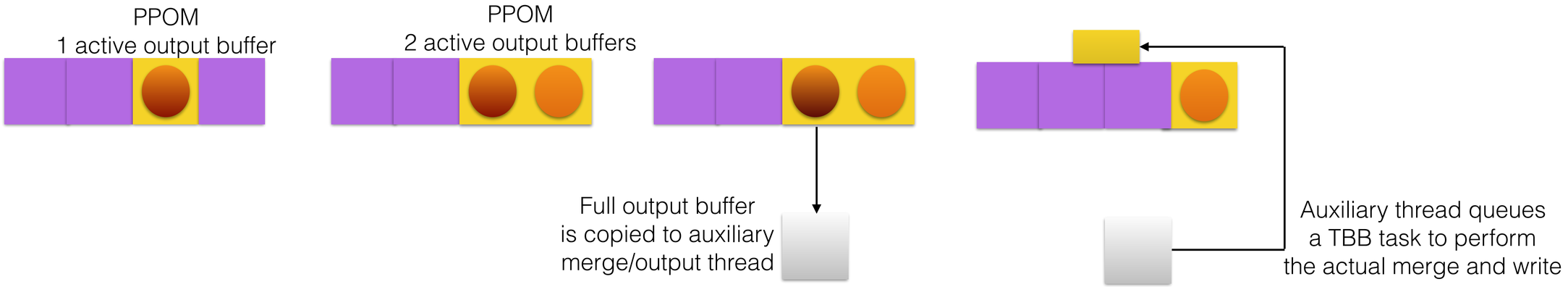
- **IMT creates TBB tasks for compressing branch buffers**
- **TBB tasks are queued on the PoolOutputModule's thread's task queue**
- **If another thread has no work on its task queue, it will "steal" work from the PoolOutputModule queue**
  - This invisible is to the framework & stall monitor—they can't distinguish idle threads from threads gainfully employed compressing branch buffers
  - IMT can't use threads that are blocked (e.g., on a mutex)



ParallelPoolOutputModule Schematic

ParallelPoolOutputModule creates TBufferMergerFiles on demand

- **limited::OutputModule to limit the # of TBufferMergerFiles created**
  - Framework needs to know about the limit so it can schedule accordingly
- **Always fill the available TBufferMergerFile with the most entries**
  - Avoids synchronization effects, minimizes tail effects, approximates serial ordering
- **Branch buffer compression happens on the PPOM thread**
  - Possibly using IMT—can lead to non-trivial interactions



Potential TBufferMerger Issues

Several issues were found from a design review:

- **The output thread is not managed by the framework/TBB**
- **If the merge thread can't keep up, the queue can grow without bound**
  - Nothing stops the worker threads from adding new entries onto the queue

These are all OK **if** the merge operation takes negligible time

- **OK for MINIAOD, issues found with AOD & RECO**
- **Due to TTree AutoSave, which allows partial recovery of files from crashed jobs**
  - AutoSave rewrites the list of branch keys, which grows with the number of branch baskets
  - Compression of the branch keys can take significant CPU time
- **PoolOutputModule turns off AutoSave entirely**
  - But apparently this isn't currently possible for the TFileMerger
  - This is wasted CPU time for us since we don't try to recover output files from crashed jobs

Issue Mitigation

Several steps taken to mitigate the issues:

- **ROOT made some internal changes to reduce the AutoSave frequency and remove mutexes**
- **AutoSave frequency depends on the merge operation frequency**
  - Merge the TMemFiles in bunches
  - Increase the event AutoFlush size (also improves compression)
  - Both these measures increase memory “hoarding”
- **Use a faster compression algorithm for AutoSave**
  - Data use the TMemFile compression settings
  - AutoSave uses the the TMergeFile compression settings—doesn’t have to be the same!
- **Wrap the merge operation in a lambda that executes in the framework TBB task arena**
  - Execute the CPU-intensive compression into a framework TBB thread for compatibility with framework scheduling
  - Avoid using more resources than we were allocated
  - Lets the framework monitor the queue size and respond if the backlog is too large
  - Need to submit a PR with this change!

Implementation Status

ParallelPoolOutputModule is mostly implemented

- Refactored PoolOutputModule so most bookkeeping is shared
- Some metadata merging is still incomplete

TBufferMerger still needs the merge callback change

- Change allows a wrapper around the merge operation
- Results shown include this change so the CPU accounting is accurate

Comparison Tests

Test setup:

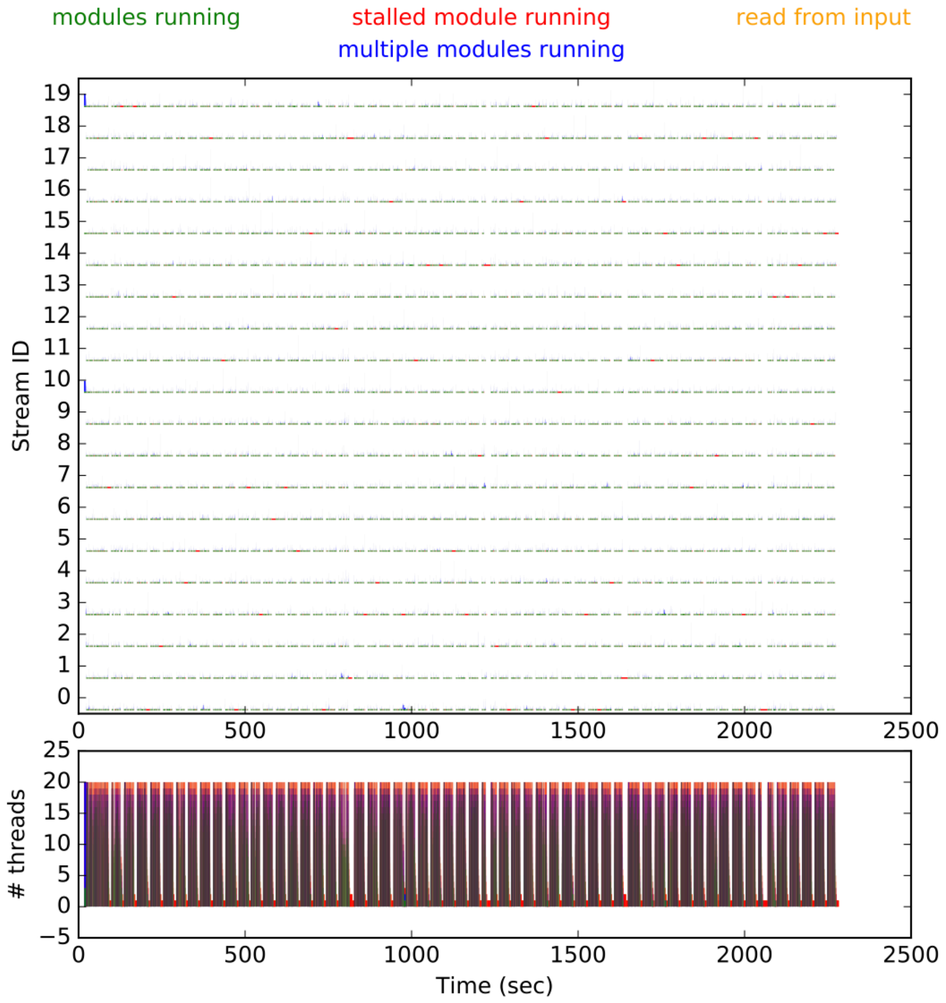
- **Realistic pileup**
  - Samples with more realistic pileup, including tests with realistic summer 2017 PU
  - Previous tests used simple no-pileup events that exaggerated output stalls
- **Modified TBufferMerger with wrapper around merge operation**
- **Writing RECO, AOD and MINIAOD, standard compression levels**
  - Also tested just AOD and MINIAOD
- **20-core Haswell E5-2620v3, 8 & 20 threads & streams**

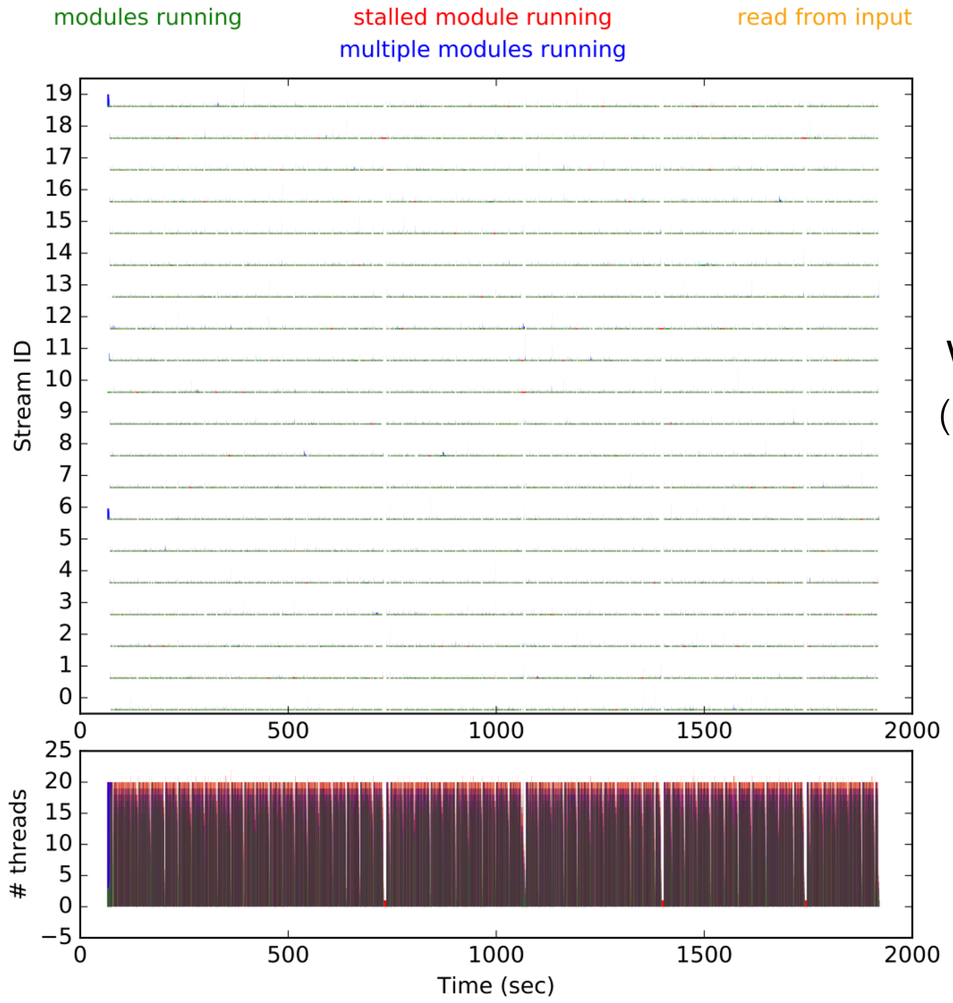
Tests:

- **Normal PoolOutputModule with and without IMT**
- **ParallelPoolOutputModule with and without IMT**
  - RECO output concurrency 3, AOD 3, MINIAOD 2

Standard output, no IMT vs w/IMT

No IMT

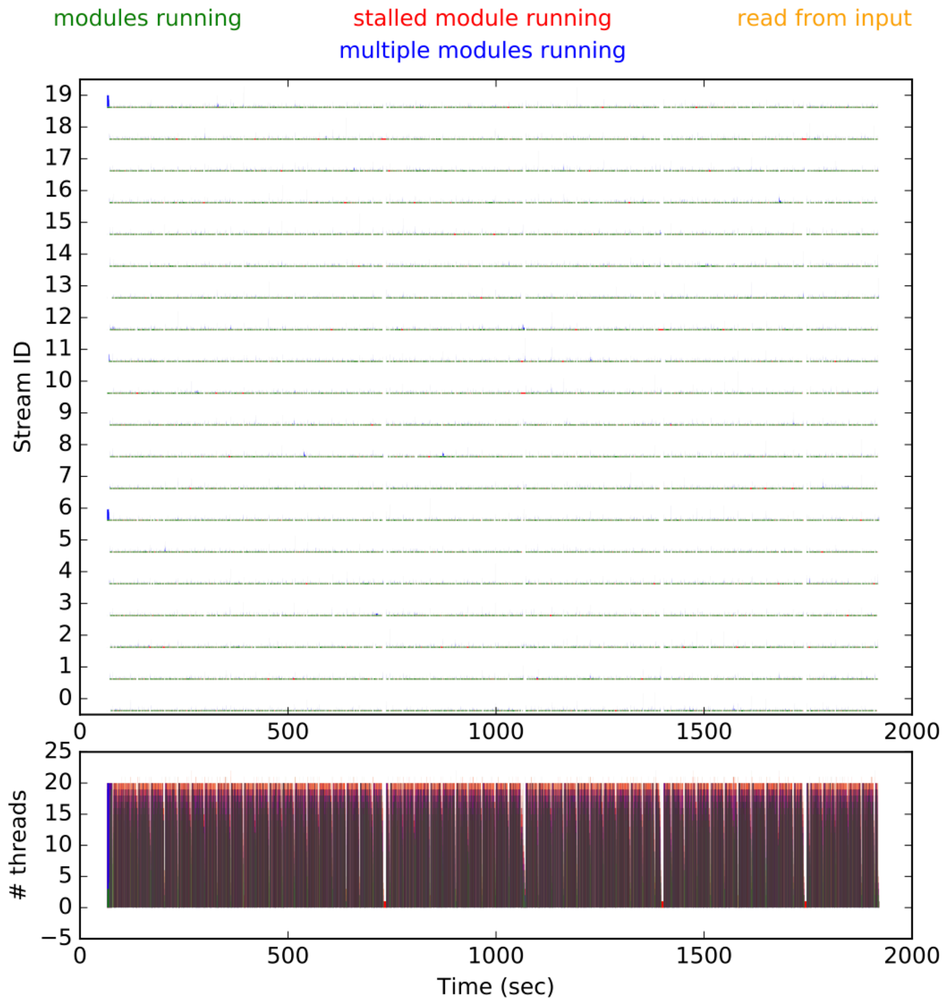


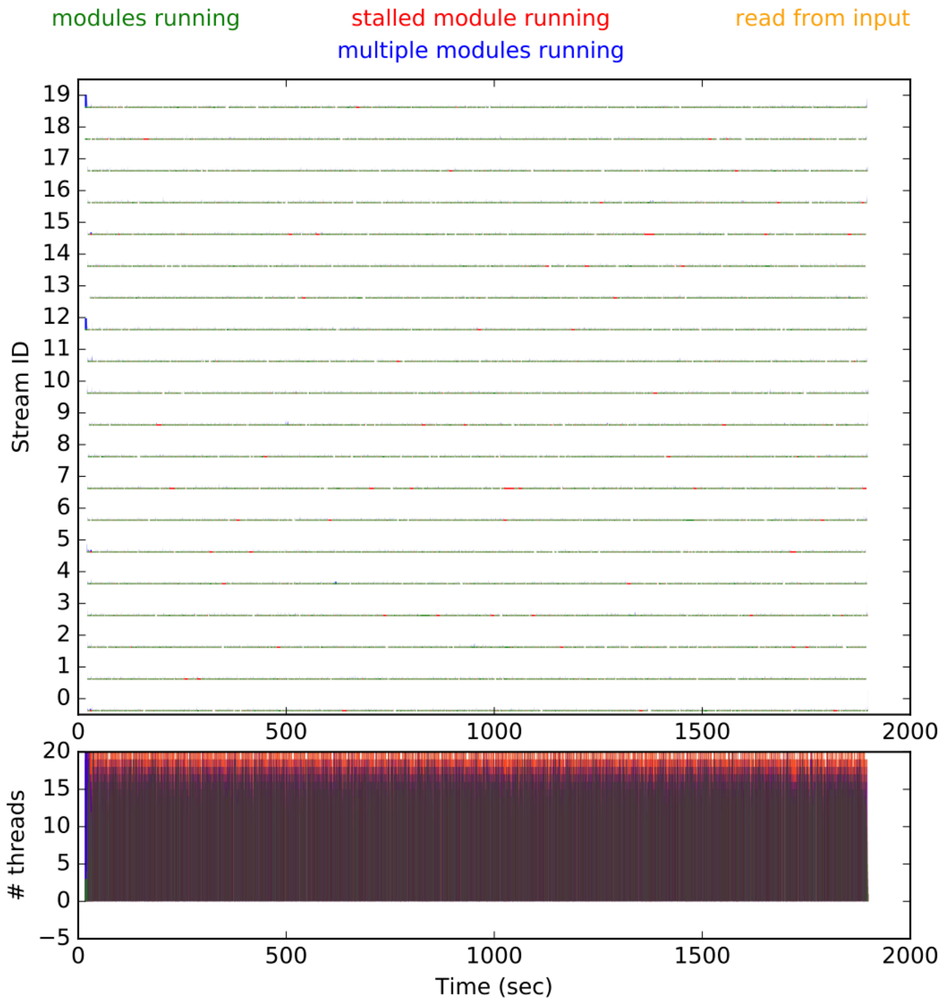


w/IMT
(note scale change)

Standard output w/IMT vs. parallel merger w/o IMT

IMT





Parallel Merger

8 thread RECO-AOD-MINIAOD

Module	Total Loop Time	Total Loop CPU	Efficiency CPU/Real/ thread	Events/ Second	RSS	# of AOD stalls	Total AOD stall time
Standard w/o IMT	4534	33248	0.913	1.10	5244	605	3493
Standard w/IMT	4197	32878	0.978	1.19	4940	564	976
Parallel Merger w/o IMT	4292	33810	0.980	1.16	21754	409	205

Most CPU

Best Throughput

20 thread RECO-AOD-MINIAOD

Module	Total Loop Time	Total Loop CPU	Efficiency CPU/Real/ thread	Events/ Second	RSS	# of AOD stalls	Total AOD stall time
Standard w/o IMT	2283	36688	0.798	2.19	6675	1799	10486
Standard w/IMT	1921	36158	0.937	2.60	6989	1577	2392
Parallel merger w/o IMT	1900	37393	0.971	2.63	22487	1035	309

20 thread RECO-AOD-MINIAOD, realistic 2017 PU

Module	Total Loop Time	Total Loop CPU	Efficiency CPU/Real/ thread	Events/ Second	RSS	# of AOD stalls	Total AOD stall time
POM	4009	72144	0.894	1.25	10103	1312	6459
POM w/ IMT	3760	69997	0.928	1.33	10162	1033	1377
Parallel POM no IMT	3825	71833	0.932	1.31	27949	789	317

20 thread AOD-MINIAOD, realistic 2017 PU

Module	Total Loop Time	Total Loop CPU	Efficiency CPU/Real/ thread	Events/ Second	RSS	# of AOD stalls	Total AOD stall time
POM	4018	68462	0.847	1.24	10207	1413	6687
POM w/ IMT	3875	68676	0.884	1.29	9611	1070	5507
Parallel POM no IMT	3846	69900	0.902	1.30	11963	523	159

Conclusions

With realistic pileup

- **IMT and parallel merger give comparable throughput and scaling up to 20 threads**
  - Fewer, shorter stalls
- **IMT reduces CPU time**
  - TBB will busy wait during stalls; by stealing those cycles, IMT improves CPU utilization
- **Parallel merger increases CPU time slightly**
  - Due to extra compression in merger
- **Extrapolate that IMT probably will not provide enough tasks much past 20 threads**
  - Expect parallel merger to provide better throughput at more than ~32 threads
- **CMS has turned on IMT by default**
  - It's a clear win for our current 4 and 8 thread production jobs

TODO

- **More threads, test on KNL**
- **Mixed output modules to optimize performance and memory usage**
  - Standard module w/IMT for sparse RECO + parallel merger for AOD & MINIAOD
- **Larger autoflush size to improve compression and reduce autosave overhead**
- **Finish off implementation loose ends (metadata merging, throttling) and submit ROOT PR for merge wrapper**

BACKUP SLIDES

Merge Wrapper

```
void TBufferMerger::RegisterMergerExec(const std::function<void(std::function<void()>)> &f)
{
   if (nullptr == f) {
      fMergerExec = [](const std::function<void()> &f){ f(); }; // just execute
   } else {
      fMergerExec = f;
   }
}

In the TBufferMerger::WriteOutputFile() loop:

   auto mergefn = [&merger] { merger.PartialMerge(); merger.Reset(); };
   fMergerExec(mergefn);
```

CMS wrapper:

```
mergeExec_ = [this](const std::function<void()> &f){
  std::promise<void> barrier;
  auto fwrap = [&]() {
    auto set_value = [](decltype(barrier)* b) { b->set_value(); };
    std::unique_ptr<decltype(barrier), decltype(set_value)> release(&barrier, set_value);
    f(); // execute the merge, promise barrier is set on exit from the wrapper
  };
  taskArena_->enqueue(fwrap);  // queue the merge operation to the CMS TBB task arena
  barrier.get_future().wait(); // wait for the promise to complete
};
```