Doing the IceTray Limbo

Getting parallel

Nathan Whitehorn

Michigan State

October 21, 2020

Problem

- Traditional IceTray parallelism is event-based: one file/core
- This provides maximum possible performance
- It may use RAM inefficiently in two cases:
  1. RAM per event is large, such that available RAM/processes is too low, leaving some CPUs idle
  2. Modules have tables that could be shared between events but require an instantiation per process

Which?

Are we in one of these cases? Which one? The solutions are rather different in the two cases.

Open Questions for Needs

- What is the marginal cost (in RAM) of 1 more in-flight event?
- Are per-event temporary costs dominated by the frame, or frame-derived intermediates in modules?
- How much RAM do we use in static tables?
- How much RAM do we use in quasi-static tables (calculated from GCD data)?
- How much would we gain in resource availability today by reducing the RAM footprint to tables + 1 event per node? To tables + N_{cores} events?
- How much would we gain in 5 years?

We can't coherently decide our approach without knowing the answer to every question above. I at least don't know the answer to any of them.

Strategy 1: Event-level parallelism

- Events or blocks of events handled per core
- Basically what we do now, but could be per-event rather than per-file
- Event-wise would reduce latency (e.g. for PNF), otherwise no change in resource usage or results from what we do now
- Fills CPUs well; high efficiency
- Some reassembly overhead, especially if per-event

Room for Improvement

We could do a better job sharing initialization time data (photospline tables, etc.) and GCD-time data (PMT simulation), addressing problem #2. More on this later.

Strategy 2: Module-level parallelism

- Module (or group of contiguous modules) per core/thread
- Fills CPUs well, so long as no single module uses more than $1/N_{cores}$ of CPU time
- No reassembly overhead, but requires care in work division
- Solves problem #2 (copies of quasi-static data)
- Solves a limited number of instances of problem #1 involving large per-event temporaries in a small ($< N_{cores}$) number of modules (does this happen?)

This is quite similar to how we use GPUs.

Strategy 3: Intra-module parallelism

- One event at a time, one module at a time
- Modules internally break up work into threads
- Solves problems #1 and #2
- Easiest way to do this is per-DOM loops (calibration, likelihoods) – limits parallelism to N_{chan}, which is > 8, so might be fine if per-DOM loops dominate time.
- Bad news:
  - Guaranteed to lower throughput, potentially drastically: not every module parallelizable – is it worth it?
  - Maximally invasive to existing code
  - Significant task spawning and synchronization overhead

Intermediate Thoughts

- Event-level: Low-hanging fruit mostly gone, the remainder is sharing photospline tables etc.
- Module-level: Can improve things in circumstances we may not have.
- Intra-module: Even in principle, likely to result in significant throughput loss unless we already have KNL-style suffering. Doing it well requires a complete bottom-up restructuring of all our code and may still lower efficiency.

Low-hanging fruit largely picked already. Returns in all cases may be small except in the most RAM-constrained systems.

Processes and Threads

Threads:

- Light-weight
- Share memory (lowers memory use)
- Requires some care in synchronization
- Overhead in `malloc()` and friends
- There is the GIL

Processes:

- Heavier-weight
- Share only memory allocated before `fork()` (includes spline tables)
- Frame exchange involves serialization (amortized by lazy serdes)
- There is <u>not</u> the GIL!

The Freaking GIL

Python has a “global interpreter lock”:

- <u>Must</u> be held when entering Python
- We call Python at unpredictable times (e.g. shared pointer destruction, logging)
- Cannot be acquired automatically without LOR-induced deadlocks

The Freaking GIL

Python has a "global interpreter lock":

- Must be held when entering Python
- We call Python at unpredictable times (e.g. shared pointer destruction, logging)
- Cannot be acquired automatically without LOR-induced deadlocks
- Well, shit
- Mostly ruins threads for us

How to Solve Some of It

I started some code to keep a deletion queue in another thread, but this requires patching Boost. Makes things better, doesn't solve the whole issue and will take a while to get in.
But basically we have to use multi-process.

IceTray Invariants

I3Modules need the following:

- Within one module, events need to be strongly ordered with respect to metadata (GCD, S, M, etc.)
- Certain modules (rare) need strong ordering of events as well
- No requirements that tray-wide services are actually global (though see enxt slide)
- No requirements that modules process events in any order with respect to each other (the tray can process event 2 in module A before event 1 in module B so long as each module sees events in order and B runs after A)
- No requirements that modules be in the same process
- Modules interact with each other only via the frame (which is serializable!)

We are incredibly lucky to have these semantics for trying to parallelize IceTray internally.

Deterministic and Parallel Services

- Need to make sure services are parallel-safe (thread-safe if threads)
- IceTray has no requirement that modules see the same instance of services
- Subtle point is that RNGs need to stay deterministic
  - Key is that each module interacts with services without other modules in the middle
  - Module-level parallelism with processes inherently safe
  - Some event-level parallelism (I3MPI) safe
  - Threads are dicy – probably need TLS and deterministic assignment of (module, event) pairs to threads, which complicates worker pools

Technical Strategy 0: Making What We Have Better

Some possible gains with static tables and not much work:

- Chris wrote code we aren't using to share spline tables in SYSV shared memory
- Also could `mmap()` all tables and then let the kernel VM pager handle this
  - Would have to pre-compute convolutions for photospline tables
- Also could call `os.fork()` in production scripts after tray initialization and before configuring `I3Reader` – all init-time memory is then shared
- Another area is that we can queue-and-return more in modules using accelerators (e.g. GPUs) rather than blocking. <u>This can break determinism in random services and needs care</u>

Technical Strategy 1: Per-event Parallelism

- Usually have a mid-tray "balloon":
  1. Serial pipeline reads data
  2. Round-robin distributor
  3. *N* copies of set of modules sees event streams with gaps
  4. A reserializer
  5. Serial pipeline finishes processing
- Potential memory balloon in reserializer
- Breaks some modules that need continuous data
- Need to take care of strong ordering/broadcast of meta-data – works well if it doesn't change much (SnowStorm?)
- Remember that this basically doesn't help us: only gain relative to per-file is that we can make modules with big GCD-dependent tables part of the serial pipeline and share them

Worked Example: I3MPI

- Multi-process (and/or multi-node model)
- `I3MPIDispatch` module implements round-robin dispatch and reserialization, inserts another script (parallel) into the serial tray
- Inner script starts with `I3MPIReceiver`, which takes frames from `I3MPIDispatch` and starts tray
- Ends with `I3MPIReply`, which sends them back
- Does some internal evil with non-printable shadow frames to handle dropped events

`http://code.icecube.wisc.edu/svn/sandbox/nwhitehorn/i3mpi`

Technical Strategy 2: Per-module Parallelism

- Limiting case is one module per thread
- Input/output queue design makes this easy, handles all synchronization with clean boundaries
- Without the GIL, could easily do this with threads and all locking in I3Tray
- With the GIL, works well with multi-process since frames can be serialized
- Need to think about how to chunk up the tray into processes to amortize serialization overhead
- Obeys all I3Module invariants, requires zero changes outside of I3Tray
- Reminder: identical number of in-flight events as now, but reduces tables to a single copy per node
- NB: If one module is 90% of CPU time, doesn't help

Technical Strategy 3: Hybrid Work-Queue Systems

- The sneaky option: have a thread pool that iteratively clears input queues
- Dynamically-chunked per-module parallelism
- Could add a module flag that event-parallel is allowed and dynamically do that too
- Obeys all invariants
- By far the cleanest option: least breakage, highest throughput
- (Will require care to maintain ordering of RNG calls to keep simulation reproducible)

The Bad News

This is totally unworkable with multi-process and requires threads. Also, like all these strategies, may not solve any real problems.

Technical Strategy 4: Intra-module Parallelization

- Could do threading inside modules
- If it doesn't touch frames, or logging, doesn't hit GIL problems
- Could amortize thread start-up costs by putting a thread-pool service into I3Context
- How many modules can usefully parallelize internally? If this isn't large ($>$ 80%?), this <u>makes things worse instead of better</u>
- Task queue entries appearing/disappearing at least at $N_{modules} \times N_{cores}$ per event – this could easily be tens of thousands of synchronization operations per event
- Requires rewriting every module using non-negligible CPU

The Ugly

This has the virtue of actually solving both problems, but is hugely invasive and will need to be treated with great care.

Conclusions

- We have a lot of freedom in IceTray to parallelize efficiently without breaking APIs
- What we need depends a lot on which issues we're having, which I at least don't know
- It's not clear (to me) that we don't already have the best strategy
- I am deeply suspicious that intra-module parallelism will reduce throughput.
- My favorite strategies if we need to do something are the per-module ones (#2 and #3)