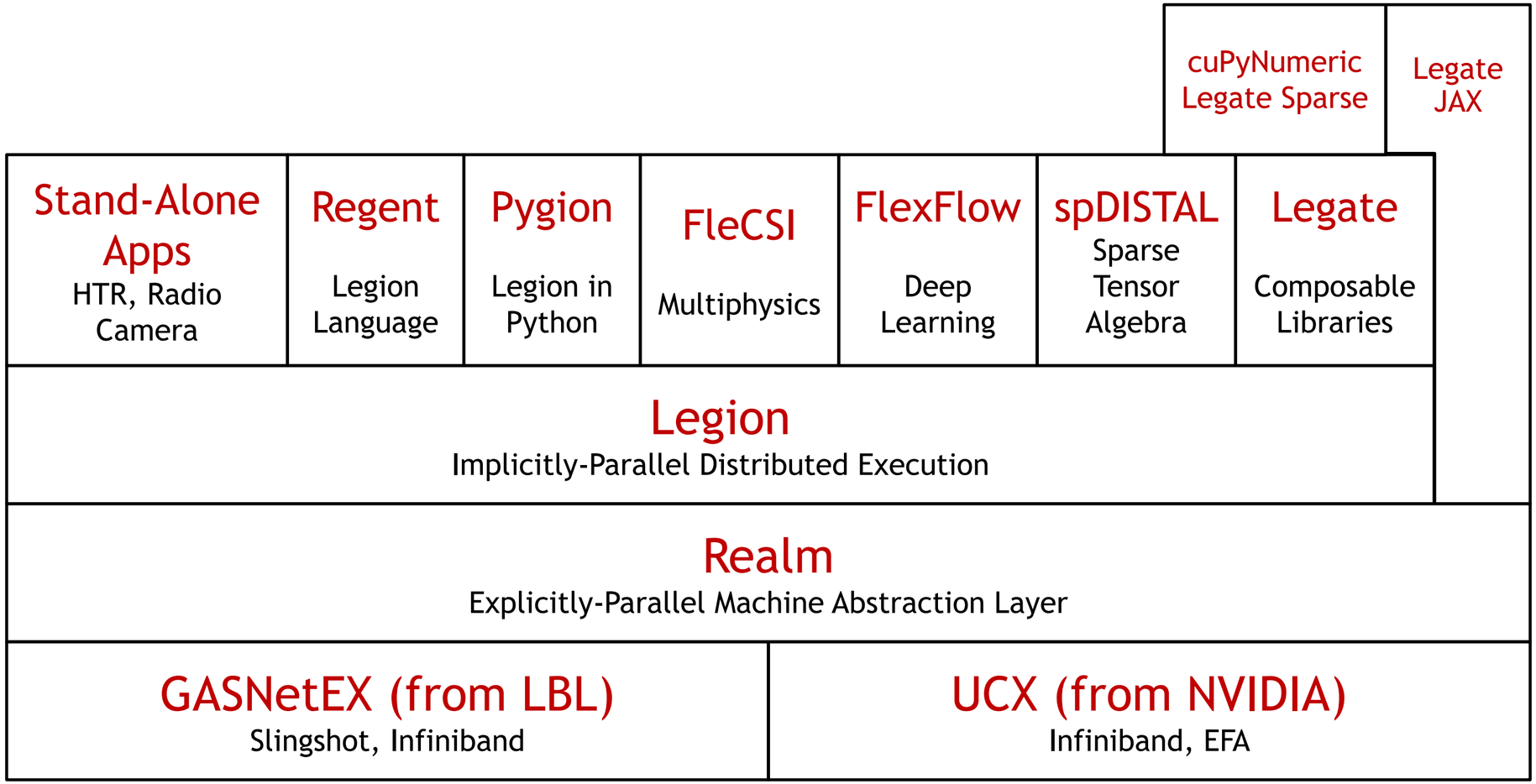
STATE OF

Legion

Michael Bauer, 12/04/24

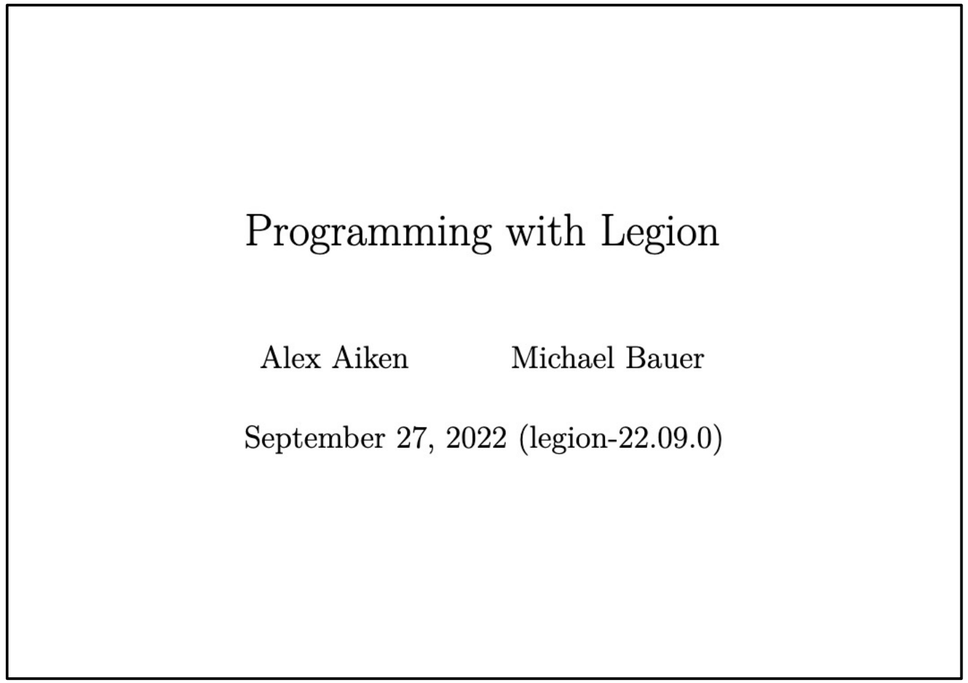
CURRENT LEGION CLIENTS

Balancing Lots of Competing Interests



DOCUMENTATION

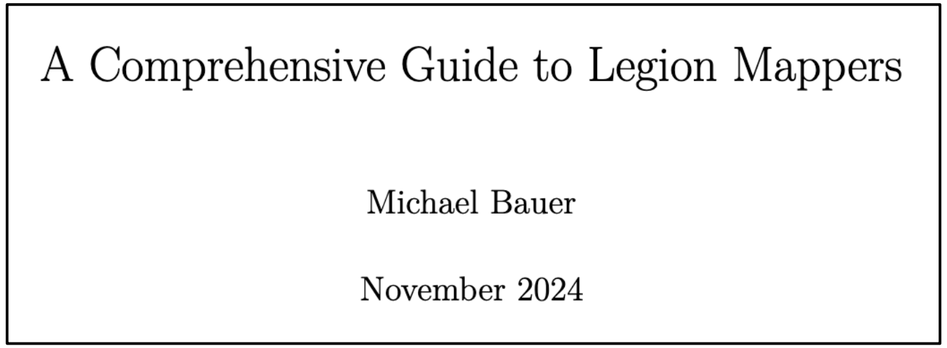
We've fallen off the horse on this one



Programming with Legion

Alex Aiken Michael Bauer

September 27, 2022 (legion-22.09.0)



A Comprehensive Guide to Legion Mappers

Michael Bauer

November 2024

Legion manual is now two years old

https://legion.stanford.edu/pdfs/legion-manual.pdf

No updates since…

Most requested kind of documentation is the mapper

Working on a comprehensive guide to the mapper interface and how to write mappers

Is this the most pressing need or something else?

CONTROL REPLICATION

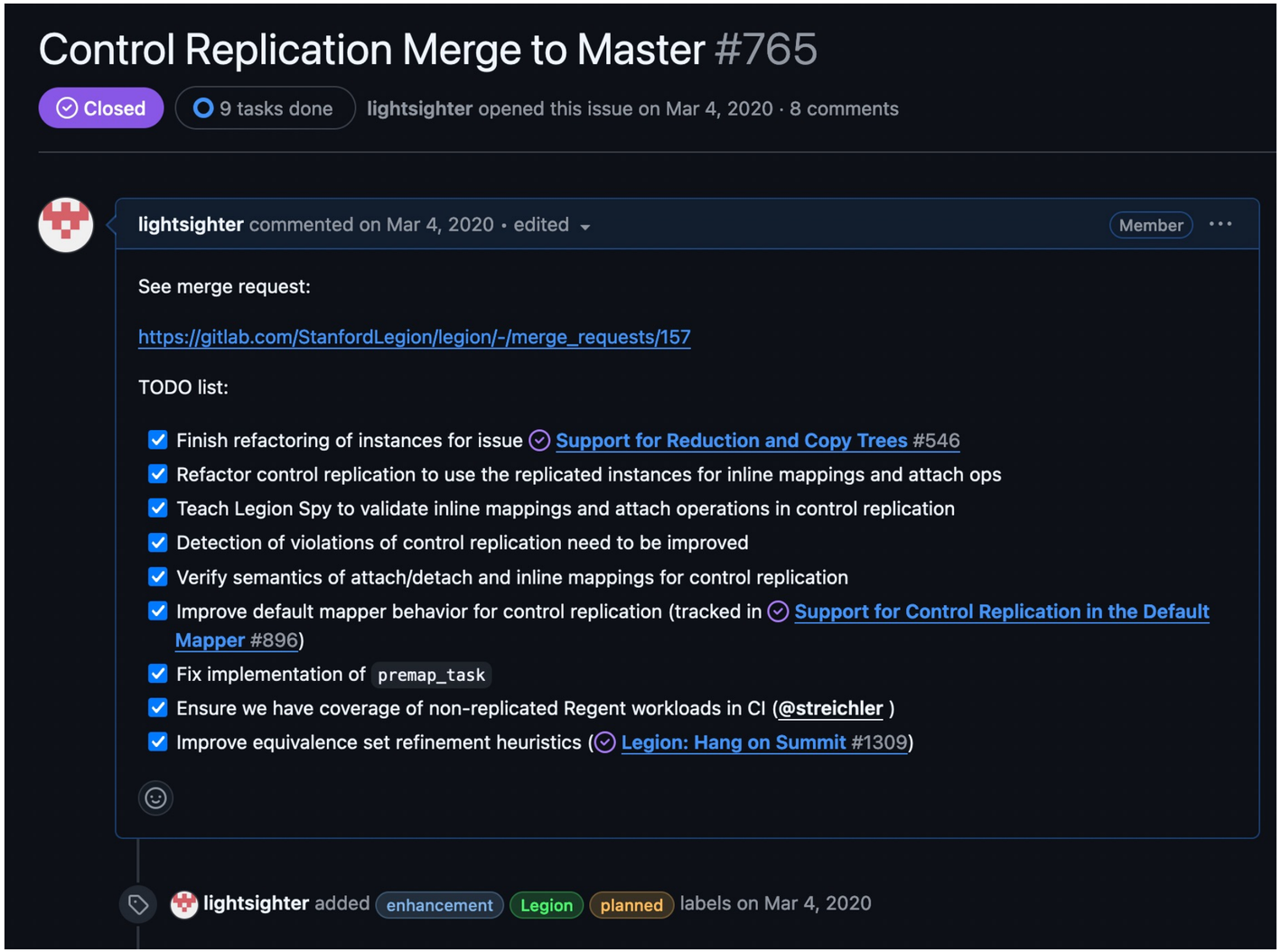
7 Years Later

Finally Done!

Took an extra year 🙄

Comes with a new equivalence set refinement heuristic

Also supports non-control-replication of leaf tasks



LEGION ROBUSTNESS

Reducing the likelihood of Legion bugs

Legion Spy verification on both single-node and multi-node CI jobs

Legion Spy verifies more conditions than it use to (e.g., sequential semantics, race-free)

Now detects use-after-free on physical instances and atomic coherence violations

We now “fuzz” Legion’s logical and physical dependence analyses

Test lots of unusual patterns of region/field usage

Done periodically and before every release

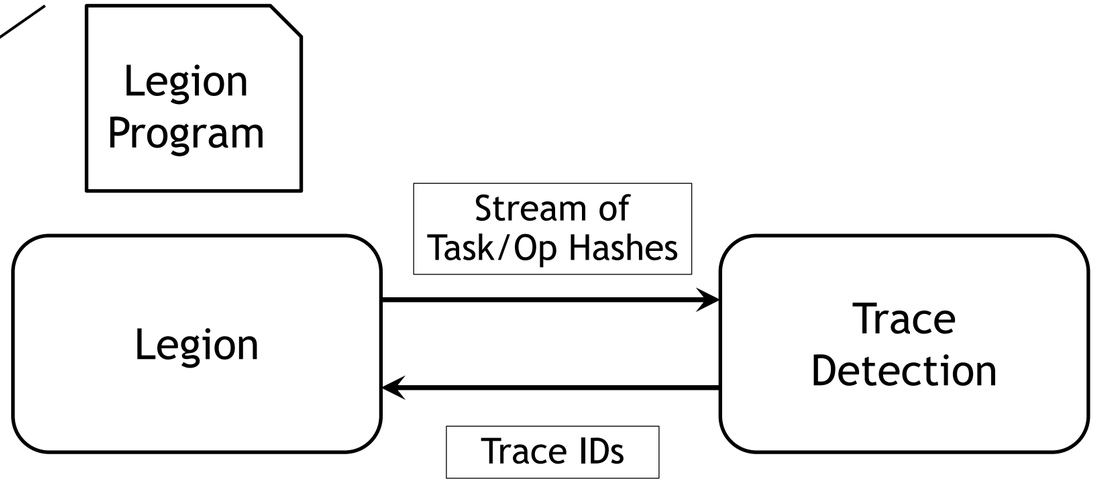
TRACING IMPROVEMENTS

Legion as a JIT-ing Interpreter

Done:

- Non-idempotent traces
- Traces release memory when not in use
- Checks for safe traces

Imminent: automatic tracing



- Extremely low-cost: ~5us/task launch
- Does not interfere with explicit traces
- Can be disabled with a command line flag
- Is subject to “turbo-lag”

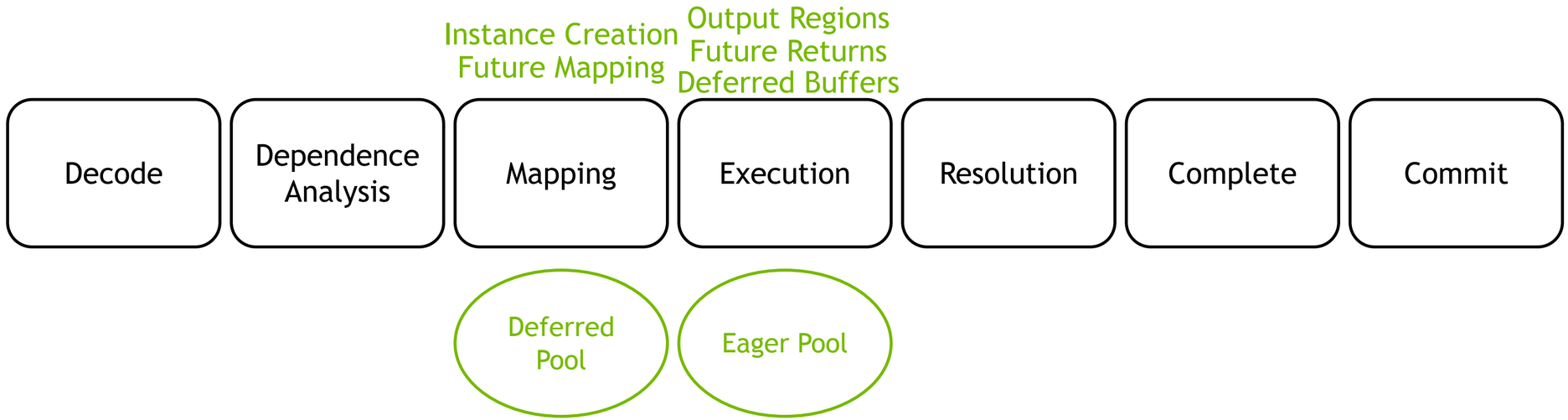
TODO:

- Trace compilation + optimization
- Support for predication
- Lowering to Realm Graphs (to CUDA Graphs)

ONE POOL MEMORY MANAGEMENT

Fixing the “two pools” problem

When are instances allocated in the pipeline:



Today we use two pools: deferred allocations in mapping and eager allocations during execution

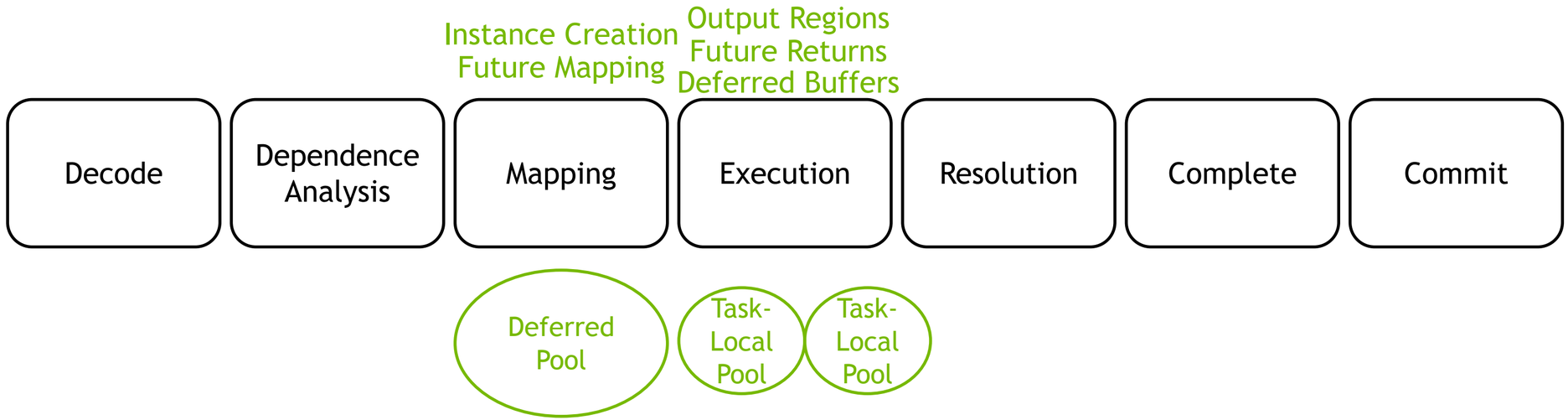
Necessary to avoid deferred allocation deadlocks

How to set the dreaded -lg:eager_alloc_percentage for each memory

ONE POOL MEMORY MANAGEMENT

Fixing the “two pools” problem

When are instances allocated in the pipeline:



If you do allocations during execution, you have three options:

1. Legacy Mode: try to do eager allocations unsafely, but detect when they *may* cause deadlock (sound but not precise)
2. Bounded: create a task-local pool during the mapping stage to use for allocations (can be static or dynamic)
3. Unbounded: block later tasks from allocating in a memory until execution is complete

    Three scopes for unbounded: restricted, index task, and permissive

RELIGHT

Automatic Checkpointing and Fast-Forward Replay for Legion

Relight is a library for Legion + Regent that greatly simplifies checkpointing and restart

Use Relight namespace for Legion and annotate where checkpoints should be taken

No need to specify what to checkpoint!

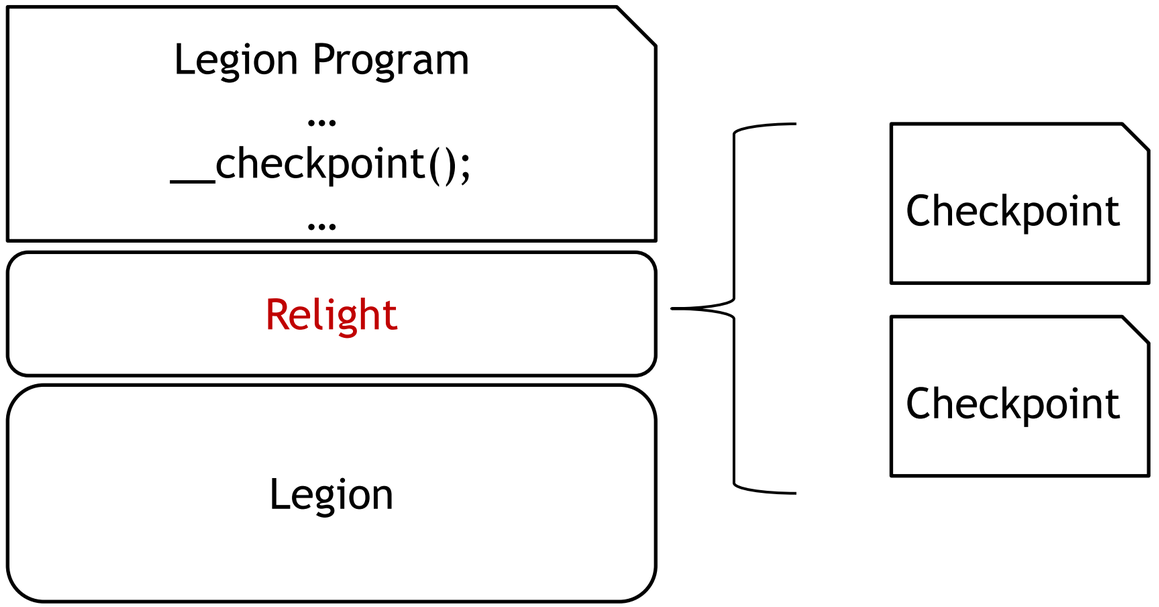
Automatically uses partitions and asynchronous data movement to create checkpoints

To resume just replay from the start

Relight “fast-forward replays” skipping tasks until reaching last checkpoint (10-100K tasks/sec)

All done transparently and automatically

https://github.com/StanfordLegion/resilience



INDEX SPACE TASK ENHANCEMENTS

Concurrent Index Space Task Launches

Many clients wanting to use collective communication between tasks (e.g., MPI All-to-All, NCCL All-Reduce)

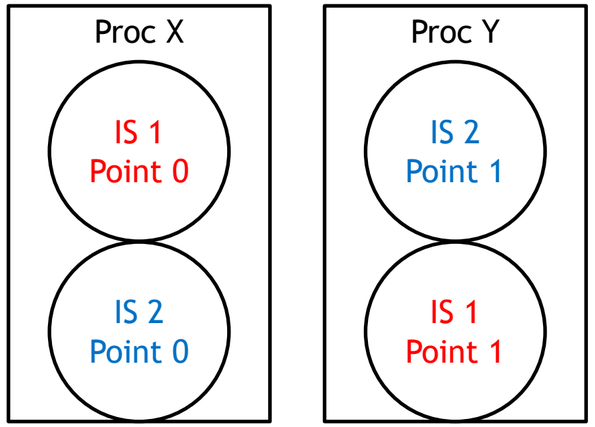
Need to avoid deadlocks due to dynamic task scheduling

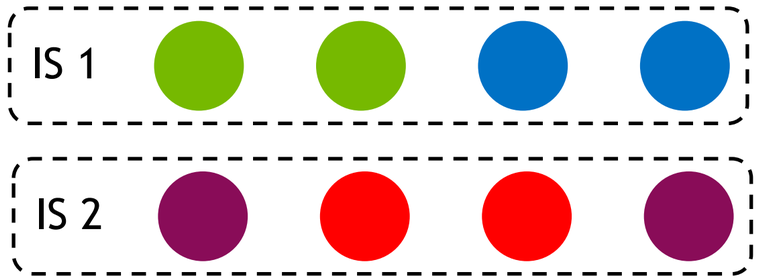
If you mark a task as concurrent and pick a concurrent task variant, Legion guarantees tasks will begin without deadlock

Uses a dynamic, distributed protocol to guarantee this

Introduces latency of a max all-reduce between participating processors before tasks can start

New version allows concurrency to be scoped to subset of points in an index space task launch





INDEX SPACE TASK ENHANCEMENTS

Inter-Index-Space Point-Wise Dependence Analysis

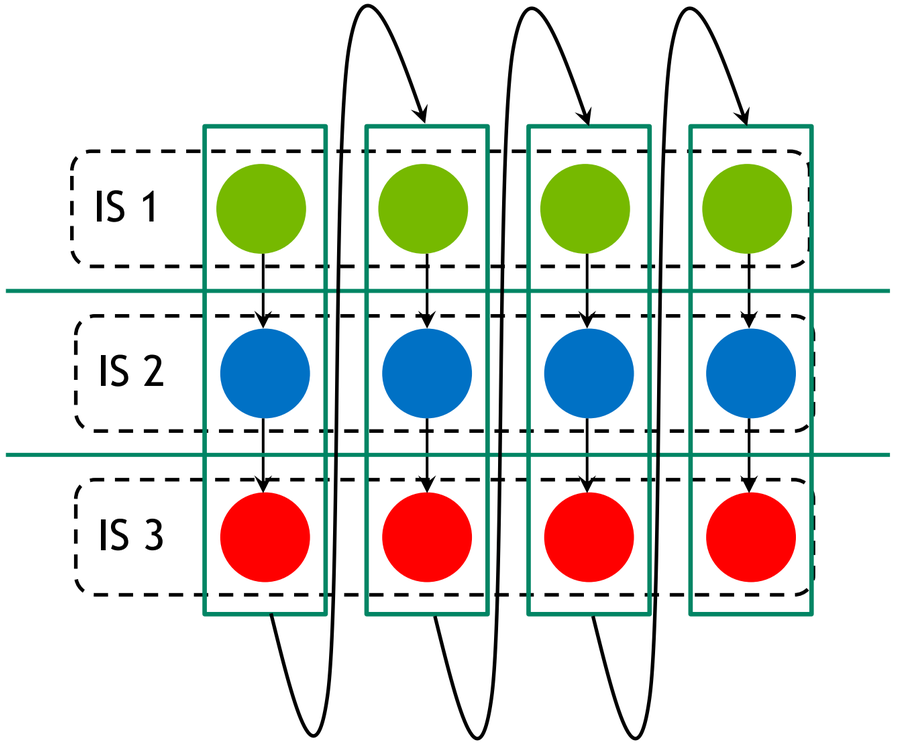
How do you “strip-mine” across index space task launches?

Today index space task launches imply mapping dependences from all points in one launch to all points in the next launch

Does not allow for memory-constrained scheduling

Idea: detect when there are name-based point-wise mapping dependences between index space task launches (does not need to be strictly data parallel)

Hook up point-wise mapping dependences so mappers can achieve schedules that they wouldn’t be able to otherwise



Note: related to intra-index-space point-wise mapping dependences, but works across index space tasks

INDEX SPACE TASK ENHANCEMENTS

Making Index Space Tasks Easier to Use

Index space task launches are the “secret sauce” of Legion: they make everything scalable

Index tasks only work if they are representing computations that require “significant” fractions of the machine

I’ve noticed an increasing propensity for users to avoid using index task launches when they could be used

It’s not clear to me why this is, but it suggests that there is some friction with the interface that needs investigation

One exception: Regent users that rely on the auto-parallelizer

GREAT REFACTORING

Cleaning up the repo

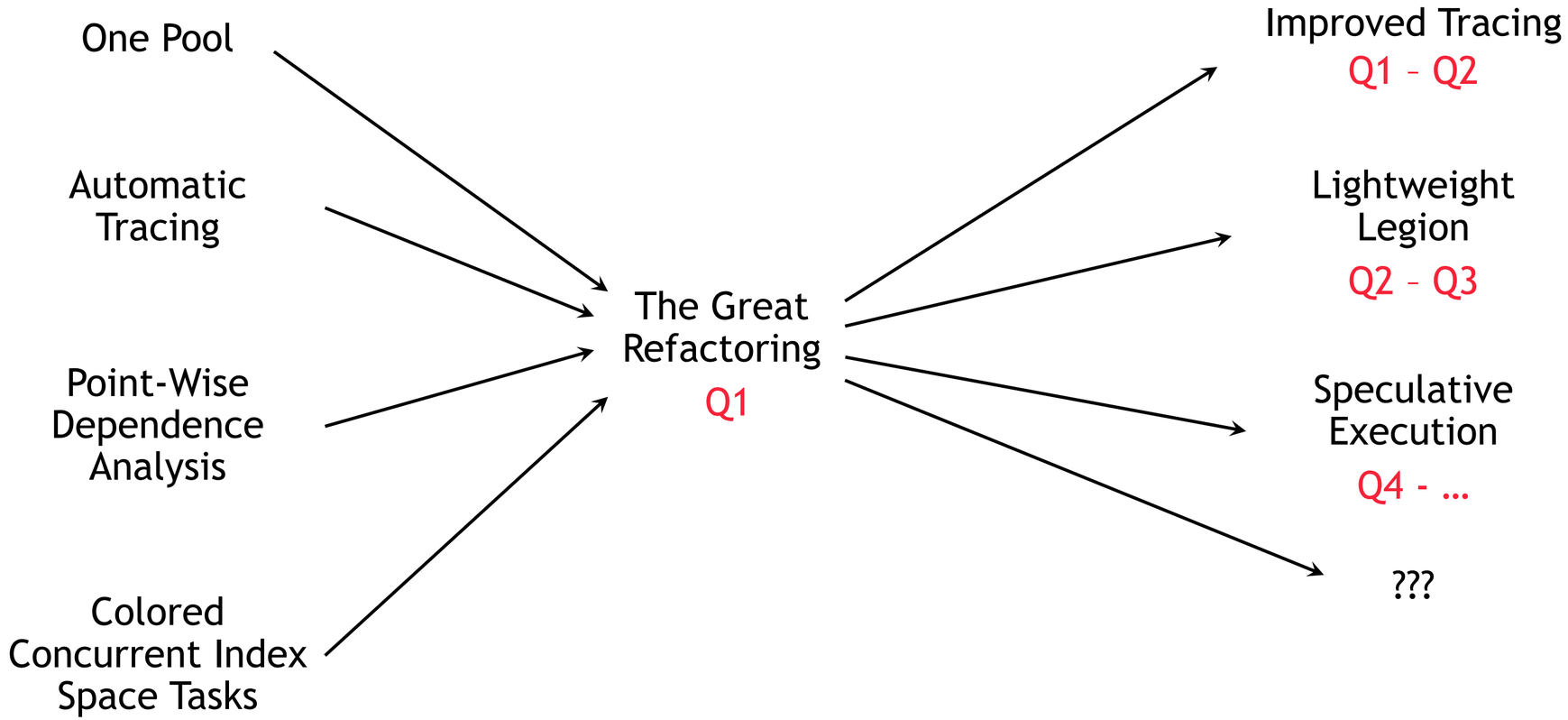
Many improvements to runtime:

- Better error reporting (task tree, stack trace, provenance, semantic info names)
- Rewritten about 700 of the 1200 error messages
- Framework for clean exits from crashes and errors
- Dynamic error checking decoupled from build mode
- Track all memory allocations to easily identify memory usage and leaks by the Legion runtime itself
- Impose a common style guide using clang-format

Needs all outstanding Legion branches merged to avoid creating lots of conflicts

ROADMAP

What's next



LIGHTWEIGHT LEGION

Optimizations for Single-Node Execution

Not all applications need full multi-node Legion and cannot benefit from tracing, need to be interpreted

No need for mapper-level parallelism to hide cross-node communication for dependence analysis

Can skip logical dependence stage of the pipeline and do program-order mapping

If running in “shared-memory mode” only one instance for each logical region (either in CPU or GPU)

Can skip parts of mapping and coherence analysis too

SPECULATIVE EXECUTION

Avoiding Stalls in Execution

Today we have predicated execution

Starting to see need for speculative execution
(LANL, Legate, output regions)

Explicit API calls to “branch” in Legion

Mapper chooses speculative branch value

Will be safe for control replication

Relight-style recovery? (must have good branch prediction)

Runtime can recover back if misspeculate/fault

```
// Branch on a Boolean future
if (runtime->branch(future, mapper, tag)){
    // Do something
    // …
} else {
    // Do something else
    //…
}
```

CONCLUSION

Lots still to do

We’ve spent lots of time making complex features correct

Need to spend more time and effort making common cases fast and easy to use

What are we missing?

Legion