Advances in VT's Load Balancing Infrastructure and Algorithms

Team (alphabetically):

Jakub Domagala (NGA)
Ulrich Hetmaniuk (NGA)
Jonathan Lifflander (SNL)
Braden Mailloux (NGA)
Phil B. Miller (IC)
Nicolas Morales (SNL)
Cezary Skrzynski (NGA)
Nicole Slattengren (SNL)
Paul Stickney (NGA)
Jakub Strzeboński (NGA)
Philippe P. Pébaÿ (NGA)

NGA = NexGen Analytics, Inc
SNL = Sandia National Labs
IC = Intense Computing

SAND2020-11823

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

What is DARMA?

A toolkit of libraries to support incremental AMT adoption in production scientific applications

Module	Name	Description
DARMA/**vt**	Virtual Transport	MPI-oriented AMT HPC runtime
DARMA/**checkpoint**	Checkpoint	Serialization & checkpointing library
DARMA/**detector**	C++ trait detection	Optional C++14 trait detection library
DARMA/**LBAF**	Load Balancing Analysis Framework	Python framework for simulating LBs and experimenting with load balancing strategies
DARMA/**checkpoint-analyzer**	Serialization Sanitizer	Clang AST frontend pass that generates serialization sanitization at runtime

DARMA Documentation: *https://darma-tasking.github.io/docs/html/index.html*

Load Balancing R&D Lifecycle

Sandia National Laboratories

- Application runs with VT runtime with designated *phases* and *subphases*
- VT exports LB statistics files containing object loads, communication, and mapping
- LBAF loads the statistics files, and simulates possible strategies
  - LBAF analyzes the mapping and can produce a new mapping with an experimental LB implemented in Python
  - LBAF exports a new set of mapping files
- The application can be re-run with `StatsMapLB` to follow the LBAF-generated mapping and measure the actual impact
- Process can be iterated, shortening LB development and tuning cycle

Phase Management

- A *phase* is a collective interval of time over all ranks that is typically synchronized
  - In an application, a phase may be a timestep
  - In VT parlance, a phase will often be a “collective epoch” under termination detection
  - Load balancing in VT fundamentally operates over phases
- A phase can be broken down into *subphases*
  - A subphase is typically a substructure within a phase of an application’s work that has further synchronization
  - Creates **vector** representation of workload
- We have explored the idea of further ontological structuring for the purpose enriching LB knowledge, but so far have only implemented phases and subphases

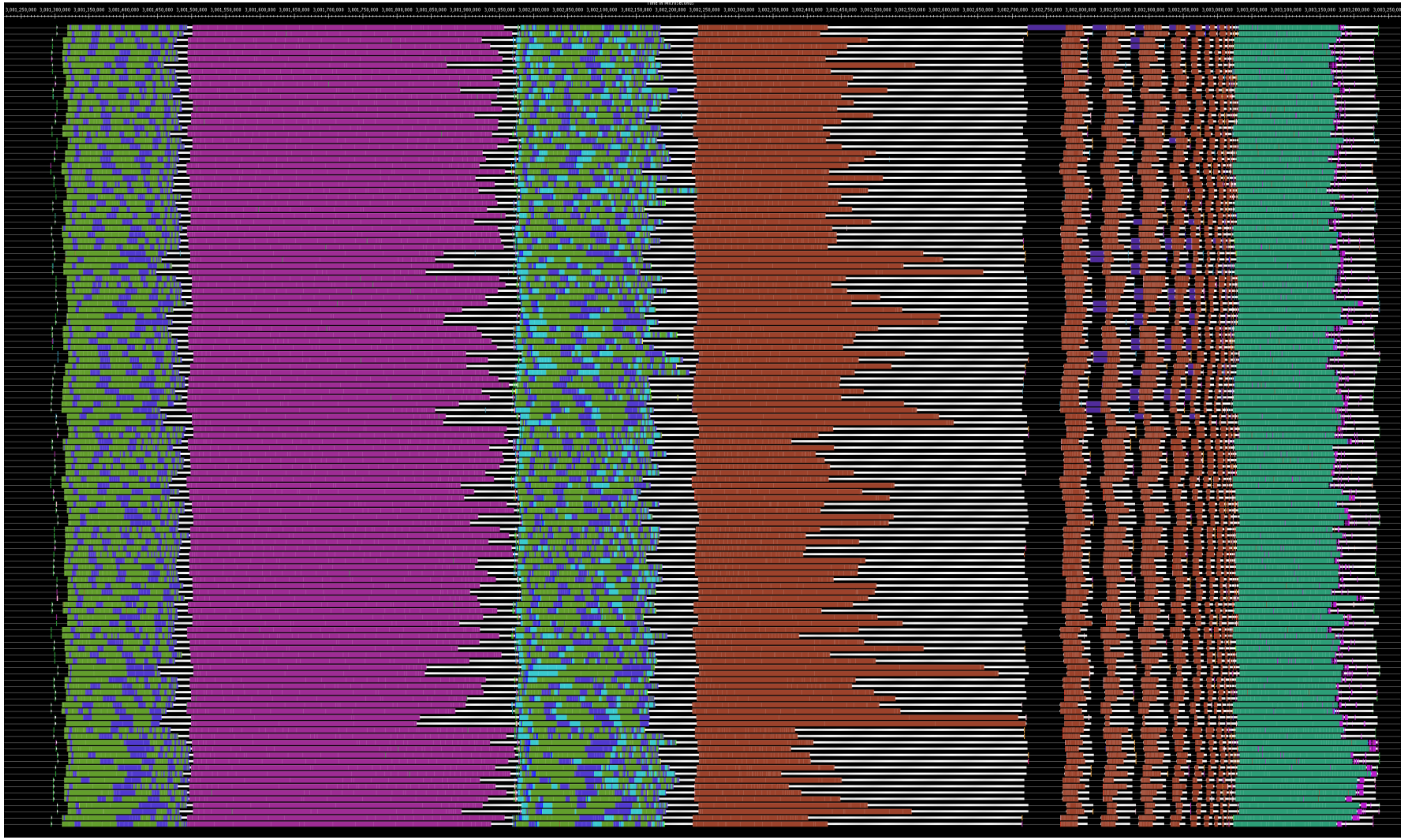
Phase Management

- Building general interface for general phase management
- Many components can naturally do things at phase boundaries
  - LB
    - Running a strategy (or several) and migrating objects accordingly
    - Outputting statistic files
  - Tracing
    - Specifying which phases traces should be enabled for which ranks
    - Specifying phase intervals for flushing traces to disk
  - Memory levels/high-water watermark for runtime/application usage
  - Diagnostics
    - Just finished developing a general diagnostic framework for performance counters/gauges of runtime behavior (e.g., messages sent/node, bytes sent/node, avg/max/min handler duration)
  - Checkpointing of system/application state
  - Termination
    - Recording state of epochs for debugging purposes

Phase Management

- A *phase* is a collective interval of time over all ranks that is typically synchronized
  - In an application, a phase may be a timestep
  - In VT parlance, a phase will often be a “collective epoch” under termination detection
  - Load balancing in VT fundamentally operates over phases
- A phase can be broken down into *subphases*
  - A subphase is typically a substructure within a phase of an application’s work that has further synchronization
  - Creates **vector** representation of workload
- We have explored the idea of further ontological structuring for the purpose enriching LB knowledge, but so far have only implemented phases and subphases

EMPIRE Load Structure – Phases, Subphases, Iterations



Subphase Vector Loads

$$t = \sum_s t_s$$

Total Time

$$t_s = \max_p w_{ps}$$

Subphase Times

$$\boldsymbol{W} = \boldsymbol{AL}$$

$$P \times S$$

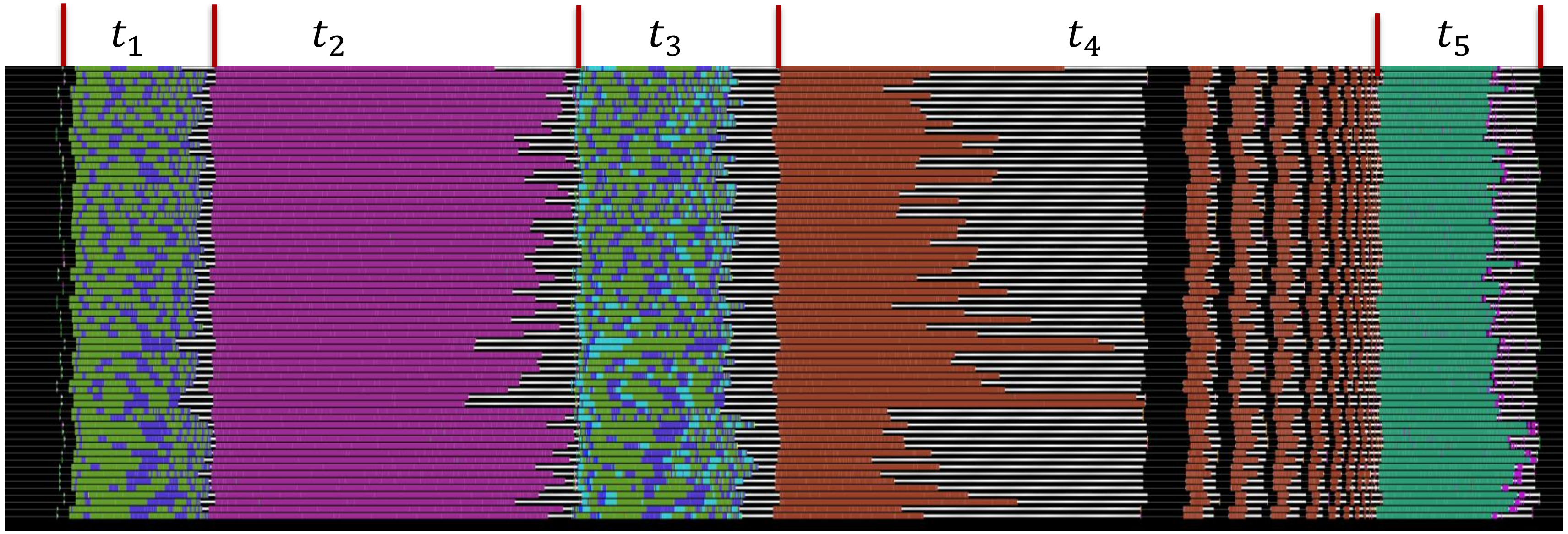
$$\boldsymbol{L}\colon\ \mathbb{R}^{N \times S}$$

Object **L**oads

$$\boldsymbol{A}\colon\ \mathbb{B}^{P \times N}$$

Object **A**ssignments

$$\forall_n \left[\sum_p a_{pn} = 1 \right]$$



Objective Function:

$$\min_A t$$

Subphase Vector Loads

- From 0-1 optimization to smaller Integer Program optimization

$\boldsymbol{A}\colon\ \mathbb{B}^{P\times N}$ $\qquad$ $a_{pn} = 1 \iff m_n = p$ $\qquad$ $\vec{M}\colon\ \mathbb{N}^{N}$

Object **A**ssignments $\qquad$ Object **M**appings

- Replace $t_s = \max_p w_{ps}$ with $\forall_p \left[t_s \geq w_{ps}\right]$ to (partially) linearize

- Plug this in to standard solvers
  - Possibly MPI-based for live use!

Load Modeling

Sandia National Laboratories

- When a selected strategy runs after a phase completes, it has access to data from the application's execution
- *Load models* provide a novel mechanism for manipulating how the load balancer observes instrumented data from phases and subphases, past and future
  - The most basic, naïve model would read raw instrumented data and assume it persists to the next phase/subphase to perform task assignment calculations for the subsequent phase
  - Explicit embodiment of "principle of persistence"
  - Offers configuration, alternatives
  - Composable functions, easy extension
- Can also map vector of per-subphase data to scalars for current strategies

Load Modeling

```
struct PhaseOffset {
  int phases;
  static constexpr unsigned int NEXT_PHASE = 0;
  unsigned int subphase;
  static constexpr unsigned int WHOLE_PHASE = ~0u;
};

class LoadModel {
  virtual TimeType getWork(
    ElementIDType object,
    PhaseOffset when
  ) = 0;
  // ...
};
```

Default:

NaivePersistence . Norm(1) . RawData

Load Model	Description	Reference
Utilities		
LoadModel	Pure virtual interface class, which the following implement	`vt::vrt::collection::balance::LoadModel`
ComposedModel	A convenience class for most implementations to inherit from, that passes unmodified calls through to an underlying model instance	`vt::vrt::collection::balance::ComposedModel`
RawData	Returns historical data only, from the measured times	`vt::vrt::collection::balance::RawData`
Transformers	Transforms the values computed by the composed model(s), agnostic to whether a query refers to a past or future phase	
Norm	When asked for a `WHOLE_PHASE` value, computes a specified l-norm over all subphases	`vt::vrt::collection::balance::Norm`
SelectSubphases	Filters and remaps the subphases with data present in the underlying model	`vt::vrt::collection::balance::SelectSubphases`
CommOverhead	Adds a specified amount of imputed 'system overhead' time to each object's work based on the number of messages received	`vt::vrt::collection::balance::CommOverhead`
PerCollection	Maintains a set of load models associated with different collection instances, and passes queries for an object through to the model corresponding to its collection	`vt::vrt::collection::balance::PerCollection`
Predictors	Computes values for future phase queries, and passes through past phase queries	
NaivePersistence	Passes through historical queries, and maps all future queries to the most recent past phase	`vt::vrt::collection::balance::NaivePersistence`
PersistenceMedianLastN	Similar to NaivePersistence, except that it predicts based on a median over the N most recent phases	`vt::vrt::collection::balance::PersistenceMedianLastN`
LinearModel	Computes a linear regression over on object's loads from a number of recent phases	`vt::vrt::collection::balance::LinearModel`
MultiplePhases	Computes values for future phases based on sums of the underlying model's predictions for N corresponding future phases	`vt::vrt::collection::balance::MultiplePhases`

Load Balancing Strategies

Sandia National Laboratories

Load Balancer	Type	Description	Reference
RotateLB	Testing	Rotate objects in a ring	`vt::vrt::collection::lb::RotateLB`
RandomLB	Testing	Randomly migrate object with seed	`vt::vrt::collection::lb::RandomLB`
GreedyLB	Centralized	Gather to central node apply min/max heap	`vt::vrt::collection::lb::GreedyLB`
GossipLB	Distributed	Gossip-based protocol for fully distributed LB	`vt::vrt::collection::lb::GossipLB`
HierarchicalLB	Hierarchical	Build tree to move objects nodes	`vt::vrt::collection::lb::HierarchicalLB`
ZoltanLB	Hyper-graph Partitioner	Run Zoltan in hyper-graph mode to LB	`vt::vrt::collection::lb::ZoltanLB`
StatsMapLB	User-specified	Read file to determine mapping	`vt::vrt::collection::lb::StatsMapLB`

Conclusions and Future Work

- Increase expressiveness of load data
- Shorten LB development and tuning cycles
- Improve abstractions in real implementations
- Formalize time-vector balancing challenge
  - Can actually try out dedicated solvers and general heuristics