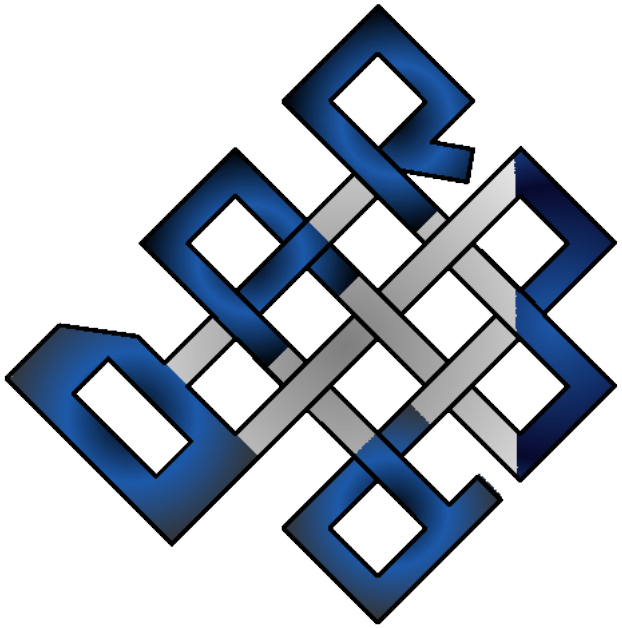
SAND2020-11597C



Sandia National Laboratories

Exceptional service in the national interest

Design and Implementation Techniques for an MPI-Oriented AMT Runtime

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

U.S. DEPARTMENT OF ENERGY
National Nuclear Security Administration

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?

Sandia National Laboratories

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*

Outline

Sandia National Laboratories

1. Motivation for developing our AMT runtime
2. Execution model and implementation ideas
   - Handler registration
   - Lightweight, composable termination detection
   - Safe MPI collectives
3. Serialization
   - ‘Serialization Sanitizer’ Analysis
   - Polymorphic classes
4. Application demonstration
5. Conclusion

Outline

Sandia National Laboratories

1. **Motivation for developing our AMT runtime**
2. Execution model and implementation ideas
   - Handler registration
   - Lightweight, composable termination detection
   - Safe MPI collectives
3. Serialization
   - ‘Serialization Sanitizer’ Analysis
   - Polymorphic classes
4. Application demonstration
5. Conclusion

Motivation

➤ Context of AMT development

- MPI has dominated as a distributed-memory programming model (SPMD-style)
  - Deep technical and intellectual ecosystem
    - Developers and training materials, courses, experiences
    - Ubiquitous implementations across a variety of platforms
    - Application code & Libraries
    - Integration with execution environments
    - Development tools for debugging and performance analysis
    - Extensive research literature
- Production Sandia applications are developed atop large MPI libraries/toolkits
  - e.g., Trilinos (linear solvers, etc.); STK (Sierra ToolKit) for meshing
  - There's little chance that the litany of MPI libraries used by production apps at Sandia will be rewritten to target an AMT runtime
- Conclusion
  - We must coexist and provide transitional AMT runtimes to **demonstrate incremental value**

Motivation

Sandia National Laboratories

➤ Philosophy

- Thus, our philosophy:
  - AMT runtimes must be highly interoperable allowing parts of applications to be incrementally overdecomposed
    - This provides an **incremental value model** for adoption
  - Transition between MPI/AMT must be inexpensive; expect frequent context switches from MPI to AMT runtime (many times, every timestep!)

- For domain developers:
  - Provide SPMD constructs in AMT runtimes for a natural transition while retaining asynchrony
  - Coexist with existing diversity of on-node techniques
    - CUDA, OpenMP, Kokkos, etc.
  - Allow MPI operations to be safely interwoven with AMT execution

- Side note:
  - We've found that serialization and checkpointing is a backdoor into introducing AMT libraries

Outline

Sandia National Laboratories

1. Motivation for developing our AMT runtime
2. **Execution model and implementation ideas**
   - Handler registration
   - Lightweight, composable termination detection
   - Safe MPI collectives
3. Serialization
   - ‘Serialization Sanitizer’ Analysis
   - Polymorphic classes
4. Application demonstration
5. Conclusion

Execution Model

➤ Handler Registration

- Handler registration across nodes
  - Many lower-level runtimes (e.g., GASNet, Converse) rely on manual registration of function pointers/methods for correctness
  - Manual registration is error prone and is not cleanly composable across modules of an application
  - Any potential solution must be valid with ASLR (memory addresses can vary across nodes)
- Example of manual registration:

```
void activeFunc(MyMsg* m) { /* handler code */ }

int main() {
  int handle_id = registerFuncCollective(activeFunc);
  if (rank == 0) {
    send(1, handle_id, new MyMsg);
  }
}
```

Execution Model

➤ Handler Registration

- Potential solutions
  - Code generation to generate registrations at startup
    - Charm++ does this with the CI file
    - Disadvantage: requires an extra step/interpreter
  - Try to match the name of the function/method at runtime?
    - Not C++ standard compliant/fragile
  - In the future: maybe C++ proposals on reflection could aid?
- VT's solution:
  - We initially started with manual, collective registration; then, we had a breakthrough
  - Build a static template registration pattern that consistently maps types (encoded as "non-type" templates) to contiguous integers across ranks
  - Across a broad range of compilers, linkers, loaders, and system configurations we find this method to be effective!
    - i.e., GNU (4.9.3, 5, 6, 7, 8, 9, 10), Clang (3.9, 4, 5, 6, 7, 8, 9, 10), Intel (18, 19), Nvidia (10.1, 11)

Execution Model

➤ Handler Registration

- C++11 compatible technique
- User code in VT with automatic registration
  - The highlighted handler automatically registers the function pointer across all ranks at the send callsite through a non-type template instantiation
  - Registration occurs at load time during dynamic initialization
  - This technique is highly composable, coupling the use of a handler with its registration across all ranks

```
struct MyMsg : vt::Message {};

void handler(MyMsg* msg) {
  /* handler code to execute */
}

int main() {
  if (rank == 0) {
    auto msg = vt::makeMessage<MyMsg>();
     // send a message to node 1
    vt::theMsg()->send<MyMsg, handler>(1, msg);
  }
}
```

Execution Model

➤ Handler Registration

- C++11 compatible technique
- User code in VT with automatic registration
  - The highlighted handler automatically registers the function pointer across all ranks at the send callsite through a non-type template instantiation
  - Registration occurs at load time during dynamic initialization
- For details on the C++ implementation and example code, read our paper at the SC'20 workshop ExaMPI [1]

```
struct MyMsg : vt::Message {};

void handler(MyMsg* msg) {
  /* handler code to execute */
}

int main() {
  if (rank == 0) {
    auto msg = vt::makeMessage<MyMsg>();
     // send a message to node 1
    vt::theMsg()->send<MyMsg, handler>(1, msg);
  }
}
```

[1] *J. Lifflander, P. Miller, N. L. Slattengren, N. Morales, P. Stickney, P. P. Pébaÿ Design and Implementation Techniques for an MPI-Oriented AMT Runtime, ExaMPI 2020*

Execution Model

➤ Lightweight, composable termination detection

- Granular, multi-algorithm distributed termination detection with **epochs**
  - Rooted epochs (starts on a single rank and uses a DS-style algorithm)
  - Collective epochs (starts on a set of ranks and uses a wave-based algorithm)
- Rooted and collective epochs can be nested arbitrarily
- Runtime manages a graph of epoch dependencies

```
using MyMsg = vt::Message;

void ring(MyMsg* msg) {
  if (rank != nranks - 1) {
    sendToNext();
  }
}

void sendToNext() {
  auto next = (rank + 1) % nranks;
  auto msg = vt::makeMessage<MyMsg>();
  vt::theMsg()->sendMsg<MyMsg, ring>(next, msg);
}
```

Rooted example:

```
void sendRing() {
  if (rank == 0) {
    vt::runInEpochRooted([]{
      sendToNext();
    });
  }
}
```

Collective example:

```
void sendRing() {
  vt::runInEpochCollective([]{
    if (rank == 0) {
      sendToNext();
    }
  });
}
```

**After this statement, all messages are received, including causally-related message chains*

Execution Model

➤ Lightweight, composable termination detection

- What does `vt::runInEpochCollective` actually do?

```
void runInEpochCollective(ActionType&& fn) {
  /*uint64_t*/ EpochType ep = theTerm()->makeEpochCollective();
  theMsg()->pushEpoch(ep);
  fn();
  theMsg()->popEpoch(ep);
  theTerm()->finishedEpoch(ep);
  theSched()->runSchedulerWhile([=]{
    return !theTerm()->isEpochTerminated(epoch);
  });
}
```

Execution Model

➤ Lightweight, composable termination detection

- Advantages
  - Asynchronous runtimes often induce a pattern where work must be synchronized with messages if there is a dependency or work relies on the completion
    - For example, broadcasts followed by a reduction
  - Epochs make ordering work (especially in a SPMD context) easier and enable lookahead

```
vt::PendingSend createFutureSend() {
  auto msg = vt::makeMessage<MyMsg>();
  // does not send the message until PendingSend
  // is released or goes out of scope
  vt::PendingSend p =
    vt::theMsg()->sendMsg<MyMsg, handler>(msg);
  return p;
}
vt::PendingSend createInEpoch(vt::EpochType ep) {
  theMsg()->pushEpoch(ep); // push on stack
  auto ps1 = createFutureSend();
  theMsg()->popEpoch(ep);  // pop off of stack
  return ps1;
}
```

Ordering two operations (*e1*, *e2*) with epochs

```
void sequencedSends() {
  vt::DependentSendChain chain;

  vt::EpochType e1 = vt::theTerm()->makeEpochRooted();
  chain.add(e1, createInEpoch(e1));

  vt::EpochType e2 = vt::theTerm()->makeEpochRooted();
  chain.add(e2, createInEpoch(e2));
}
```

Execution Model

➤ Lightweight, composable termination detection

- EMPIRE
  - Electromagnetic/electrostatic plasma physics application
  - Initial PIC particle distributions can be spatially concentrated, creating **heavy load imbalance**
  - Particles may move rapidly across the domain, inducing **dynamic workload variation** over time
- Our overdecomposition strategy
  - Develop VT implementation of PIC while retaining the existing pure MPI implementation to demonstrate the value of load balancing
  - Main application/PIC driver should be agnostic to backend implementation or asynchrony that is introduced
  - EMPIRE physics developers should not need to fully understand VT's asynchrony to add operations

Execution Model

➤ Lightweight, composable termination detection

- Example code of EMPIRE's VT code
- Calls into VT implementation without knowing about the asynchrony or overdecomposition

```
void vtAgnosticTimestepper() {
  for (int t = 0; t < nstep; t++) {
    frontend->setDT(myDT);
    frontend->injectParticles();
    frontend->weightFields();
    frontend->accelerateParticles();
    frontend->moveParticles();
    // ...
  }
}
```

```
struct ParticleFrontendVT {
  void setDT(double dt) {
    // Similar to injectParticles, weightFields,
    // accelerateParticles
    chains_->nextStep([=](vt::Index2D idx) {
      return proxy[idx].send<DTMsg, Backend::setDT>(dt);
    });
  }

  void moveParticles() {
    chains_->nextStepCollective([=](vt::Index2D idx) {
      return proxy[idx].send<MoveMsg, Backend::moveParts>();
    });
  }

private:
  // set of element chains managed by this rank
  vt::CollectionChainSet<vt::Index2D> chains_;
};
```

Execution Model

➤ Safe MPI Collectives

- Problem
  - A runtime, application, or library may want to embed MPI operations while the runtime scheduler is running
    - Multiple asynchronous operations dispatched to collective MPI calls might be ordered improperly (see example)
    - A rank might hold up progress on another rank
      - The runtime scheduler and progress function may stop turning when one rank starts executing a collective MPI invocation
      - That progress might be required to start the operation (e.g., broadcast along spanning tree) on another node
  - Any blocking call that uses MPI can cause this problem
    - MPI window creation for one-sided RDMA
    - MPI barriers, reduces, gathers, scatters, group creation, …
    - Zoltan hypergraph partitioning invocation
    - Libraries that rely on blocking MPI collectives

Example code snippet:

```
void foo() { MPI_Reduce(...); }

void bar() { MPI_Reduce(...); }

void main() {
  async_broadcast(&foo);
  async_broadcast(&bar);
}
```

- What order do these get scheduled?
- Is that order consistent across nodes?
- Program specification? What did the user intend here?
- How do we guarantee that all ranks are ready for an operation before we start it?

Execution Model

➤ Safe MPI Collectives

- Our solution
  - We use distinct collective scopes to create independently matched strands of collective operations
    - Each collective scope is identified with a tag: *t*
    - Each operation within a scope is identified with a sequence integer: *s*
    - Thus, every operation can be distinctly identified as the tuple *(t, s)*
  - Runtime employs a distributed consensus algorithm to agree on a collective operation *(t, s)* to execute across ranks

Example user code:

Optional tag, defaults to increment

```
void main() {
  auto scope = vt::theCollective()->makeCollectiveScope();
  vt::runInEpochCollective([&]{
    scope.mpiCollectiveAsync([]{ foo(); });
    scope.mpiCollectiveAsync([]{ bar(); });
  });
}
```

```
void foo() {
  MPI_Reduce(...);
}

void bar() {
  MPI_Reduce(...);
}
```

Execution Model

➤ Safe MPI Collectives

- Consensus Algorithm
  1. When a MPI collective is enqueued, start an asynchronous VT reduction tagged on *(t, s)*
  2. When the reduction completes, broadcast a message that puts *(t, s)* in a special queue
  3. Achieve consensus by picking an operation consistently
     - Could also be achieved with an async all-reduce min over *(t, s)*
  4. Use an Ibcast to inform all ranks of *(t, s)*
  5. Use an Ibarrier to ensure all ranks have the operation
  6. Stop running the scheduler and execute the action on every rank

```
1:  procedure ACHIEVECONSENSUS(rank, q)
2:      MPI_Request req;
3:      if rank is designated root then
4:          (t, s) ← select arbitrary from q
5:      else
6:          (t, s) ← ∅
7:      end if
8:      MPI_IBCAST((t, s), root, &req)
9:      while req not completed do
10:         vt::runScheduler();  ▷ Keep progressing scheduler
11:     end while
12:     MPI_IBARRIER(&req)          ▷ Ensure all ranks have
    (t, s)
13:     while req not completed do
14:         vt::runScheduler();  ▷ Keep progressing scheduler
15:     end while
16:     op ← GETACTION((t, s))
17:     op()                        ▷ Execute the collective operation
18: end procedure
```

Outline

Sandia National Laboratories

1. Motivation for developing our AMT runtime
2. Execution model and implementation ideas
   - Handler registration
   - Lightweight, composable termination detection
   - Safe MPI collectives
3. **Serialization**
   - ‘Serialization Sanitizer’ Analysis
   - Polymorphic classes
4. Application demonstration
5. Conclusion

Serialization

➤ ‘Serialization Sanitizer’ Analysis

- Serializers are difficult and error prone to maintain
- Difficult to test and maintain as application evolves
  - If serialization tests aren’t updated with changes to a class, a difficult-to-find bug can be easily introduced
- C++ static reflection proposals may address this problem (and even make explicit traversal unnecessary?)
  - For now, we are stuck with manually traversing classes
- Our prototyped solution:
  - A combination of static instrumentation and dynamic analysis to verify serializers
  - Relies on a new Clang frontend AST pass which can be hooked into the build process

Serialization

➤ ‘Serialization Sanitizer’ Analysis

- Static instrumentation using Clang frontend compiler toolkit
  - Traverse all members of all classes with intrusive or non-intrusive serialization methods
  - Generate an alternative partial specialization of the serialize method that traverses all members with a special serializer type
- Dynamic checks at runtime
  - Run the checker specialization alongside the serialize method; use sets to compare memory addresses of actually serialized and checked members; allow users to explicitly skip members

```
struct MyClass {
  template <typename SerializerT>
  void serialize(SerializerT& s) {
    s | a | c;
  }

  std::vector<int> a;
  int b;
  std::unordered_set<int> c;
};
```

```
/* begin generated code */
template <>
void MyClass::serialize<checkpoint::Checker>(checkpoint::Checker& s) {
  s.check(a, "MyClass::a");
  s.check(b, "MyClass::b");
  s.check(c, "MyClass::c");
}
/* end generated code */
```

Serialization

➤ Polymorphic classes

- Applications often use object polymorphism to express hierarchies of classes and behaviors
  - Polymorphic objects may need to be serialized
  - While messages and simple structures used for communication may avoid polymorphism, checkpoints of application data will often include these data structures
- Polymorphic objects are difficult to serialize correctly
  - The serializer needs to be invoked on all classes in the hierarchy recursively
  - The proper concrete type must be transmitted/reconstituted if the object is sent
- Existing solutions rely on C++ strings
  - PUP: uses `register_PUP_ID` with a string to register the type
  - Boost: uses `BOOST_CLASS_EXPORT_GUID` or `register_type` to rely on typeinfo or user-specified strings to register the classes
  - For templates this might be difficult and very expensive, especially when writing to disk or sending across the network

Serialization

➤ Polymorphic classes

- We exploit the static template registration pattern to generate integers for each type
  - This solves the problem of generating a unique, consistent identifier across ranks for each type

- We insert with a shim layer in the hierarchy or a macro virtual methods that automatically traverse the hierarchy

- Because serializers are templated, we explicitly instantiate them at compile time with all possible serializer types and register the serializer types

Serialization

➤ Polymorphic classes

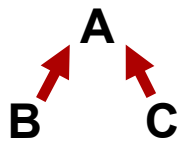
- Example user code:

Base class

```
struct A {
  checkpoint_virtual_serialize_root()

  template <typename SerializerT>
  void serialize(SerializerT& s) {
    s | a | b;
  }

  int a, b;
};

struct B : A {
  checkpoint_virtual_serialize_derived_from(A)

  template <typename SerializerT>
  void serialize(SerializerT& s) {
    s | c | d;
  }

  int c, d;
};
```

```
struct C : A {
  checkpoint_virtual_serialize_derived_from(A)

  template <typename SerializerT>
  void serialize(SerializerT& s) {
    s | e | f;
  }

  int e, f;
};
```

```
struct D {
  D() : ptr(std::make_unique<C>()) { }

  template <typename SerializerT>
  void serialize(SerializerT& s) {
    s | ptr;
  }

  std::unique_ptr<A> ptr;
};
```



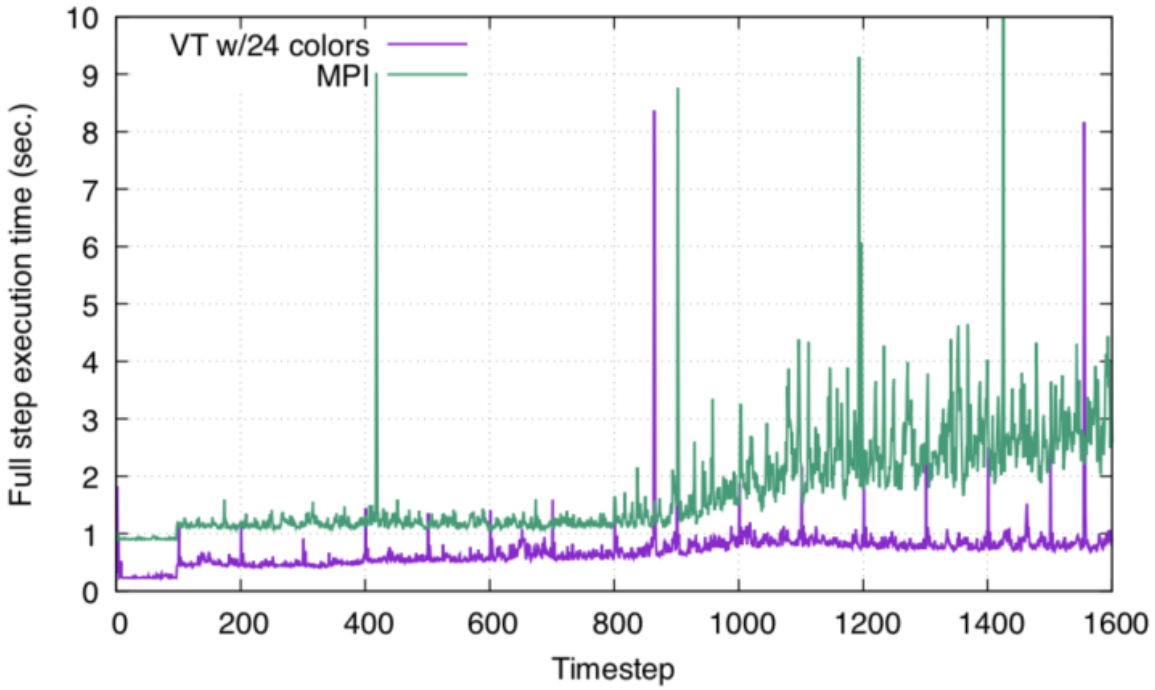
Automatically serializes and reconstructs the correct type

Outline

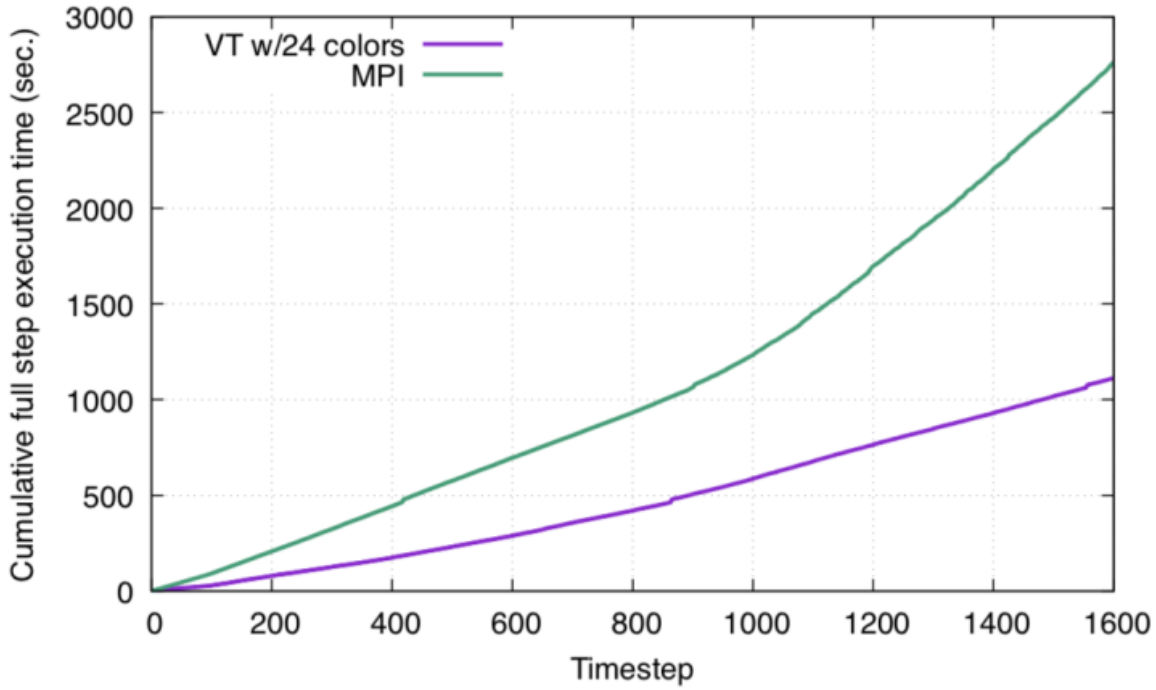
Sandia National Laboratories

1. Motivation for developing our AMT runtime
2. Execution model and implementation ideas
   - Handler registration
   - Lightweight, composable termination detection
   - Safe MPI collectives
3. Serialization
   - Polymorphic classes
   - ‘Serialization Sanitizer’ Analysis
4. **Application demonstration**
5. Conclusion

Application Demonstration: EMPIRE



(a) Per-timestep Execution Time



(b) Cumulative Execution Time

Fig. 2: Per-timestep performance comparison of EMPIRE "B-Dot" problem using the *VT* and MPI implementations of PIC. The run was performed on 900 nodes of a cluster with dual socket Intel Sandy Bridge Xeon CPUs, with two processes per node and an Infiniband network. The *VT* implementation attains a 2.5× speedup over the MPI implementation for the whole run using 24 colors per process and load balancing every 100 simulation time steps. The MPI implementation suffers substantially more from the severe noise on the system than the *VT* implementation.

Conclusion

Sandia National Laboratories

- We are working on an AMT runtime that provides incremental value as users adopt it for small segments of their applications
- We've found that serialization/checkpointing is a good way to introduce our toolkit to applications
  - Thus, we've build the DARMA/checkpoint library as a small, distinct piece that can be used standalone
- We can retain a SPMD style that is familiar to domain experts while still obtaining the benefits of asynchrony, overdecomposition, and load balancing
- Virtual Transport is moving into production use for the EMPIRE L1 milestone
  - Supports a wide range of problem configurations on the largest supercomputers

Execution Model

➤ Generalized Requests

- In our ideal vision, driving execution progress to termination of an epoch, would be expressed as an MPI_Wait or MPI_Test
  - Generalized MPI requests as standardized are insufficient due to our need to call the VT scheduler inline
  - The Extended Generalized Requests interface proposed by Latham, et al. would suffice

```
MPI_Request transferParticles() {
  MPI_Request req = vt::makeEpochCollective([]{
    /* recursively move particles */
  });
  return req;
}

void caller() {
  MPI_Request req = transferParticles();
  MPI_Wait(&req, MPI_STATUS_NULL);
}
```