SAND2023-05432C

Exploiting the overlapping challenges of distributed AMT and Resilience

Supporting automated resilience

Matthew Whitlock, Nic Morales, Keita Teranishi

Platform for Advanced Scientific Computing, 2023

Presented for the Performance in I/O and Fault Tolerance for Scientific Applications Minisymposium

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.

ing 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.

HPC's Growing Demands

- Bidirectional Growth: Applications ←→ Clusters
- Both gain complexity
  - More time to develop and use
  - More domains of expertise required
  - More edge cases
  - More variance
- Developers require strong tools
  - Support dynamic, asynchronous, heterogeneous, load-balanced execution
    - Kokkos, Darma-VT (Virtual Transport)
  - Support automatic, asynchronous, data-aware resilience
    - Kokkos Resilience, Fenix

Co-designing Runtimes & Resilience

- Historical checkpoint patterns
  - System freezes execution, checkpoints everything
  - App developer finds good inflection points, checkpoints minimal data
- Developers already telling runtimes how to handle data/compute
- New pattern: resilience tools also interact with runtimes
  - Developer give high-level requests to tool
  - Tool works out safe checkpoint times, data placement, data usage, etc. from runtime
  - What information should runtimes expose?
  - What promises can resilience tools demand/fulfill?

Automating Task-based Resilience

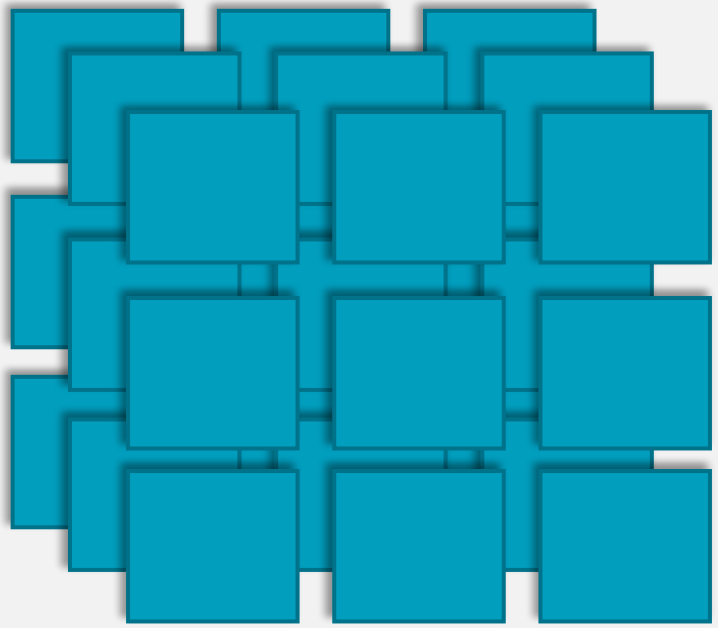
- Kokkos Resilience - https://github.com/kokkos/kokkos-resilience
  - Automated checkpoint/recovery for (MPI+)Kokkos
  - User defines resilient regions via lambda
  - Kokkos Resilience detects Kokkos::Views copied into resilient region
    - Can C/R on device memory spaces!
  - Ongoing work on soft failure recovery
- Darma Virtual Transport (VT) - https://github.com/DARMA-tasking/vt
  - Asynchronous, migratable actor pattern
    - Actors hold application data, which moves with them
    - Actors are elements of collections
  - Hierarchical, causal epochs manage control flow
    - Enforce order between generic units of work
    - “Wait for this message, but also any follow-up messages it sends, and so on”

Study: 3D Jacobi Iterative solver

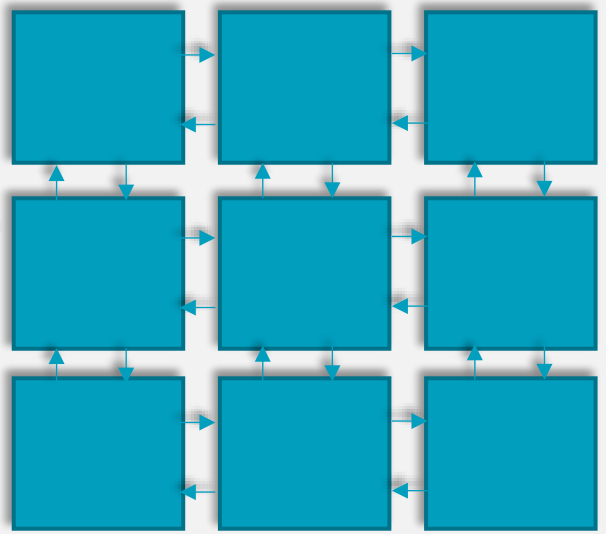
- 3D stencil application, decomposed into 3D VT Collection
- Main thread iterates a work loop to spawn tasks

```
while(!done){
```

Compute iteration (broadcast)



Send ghosts



```
}
```

- Placing loop contents in an epoch captures ALL asynchronous tasks related to that work!

What does Kokkos Resilience need from VT?

- How to serialize data?
  - Existing serialize functions for migrations
  - Extend serializer to support specializing for checkpointing
- Where is data located?
  - Messages addressed to elements, but handled by us
  - Track migrations
- What are the control-flow boundaries for checkpointing?
  - Define epochs around each iteration, try to minimally block on them
- When is data modified?
  - Message tracking? Element dereferences?

Kokkos Resilience Workflow

Target

```
ctx = make_context(“config.json”);
For(i):
    ctx->checkpoint(“main”, i, [&]{
        jacobi.broadcast(doIteration);
    });
```

Target Workflow

- Keep it simple
- Config file for portability
  - Checkpoint frequency, directory, etc.
- C/R based on loop iterator
- Autodetect variables
  - Extend from Kokkos::View copy-hooks
  - Track modifications
- Change no code within checkpoint region

Target

```
ctx = make_context("config.json");
For(i):
    ctx->checkpoint("main", i, [&]{
        jacobi.broadcast(doIteration);
    });
```

Current Workflow

Current

```
ctx = make_context(“config.json”);
For(i):
    ctx->checkpoint(“main”, i, [&]{
        jacobi.broadcast(doIteration);
    });
```

Current Workflow

- Pre-register objects for typing during recovery

Current

```
ctx = make_context(“config.json”);
ctx->register(jacobi);
For(i):
    ctx->checkpoint(“main”, i, [&]{
        jacobi.broadcast(doIteration);
    });
```

Current Workflow

- Pre-register objects for typing during recovery
- Register again upon modification
  - At collection or element granularity
- Remarkably close to ideal!

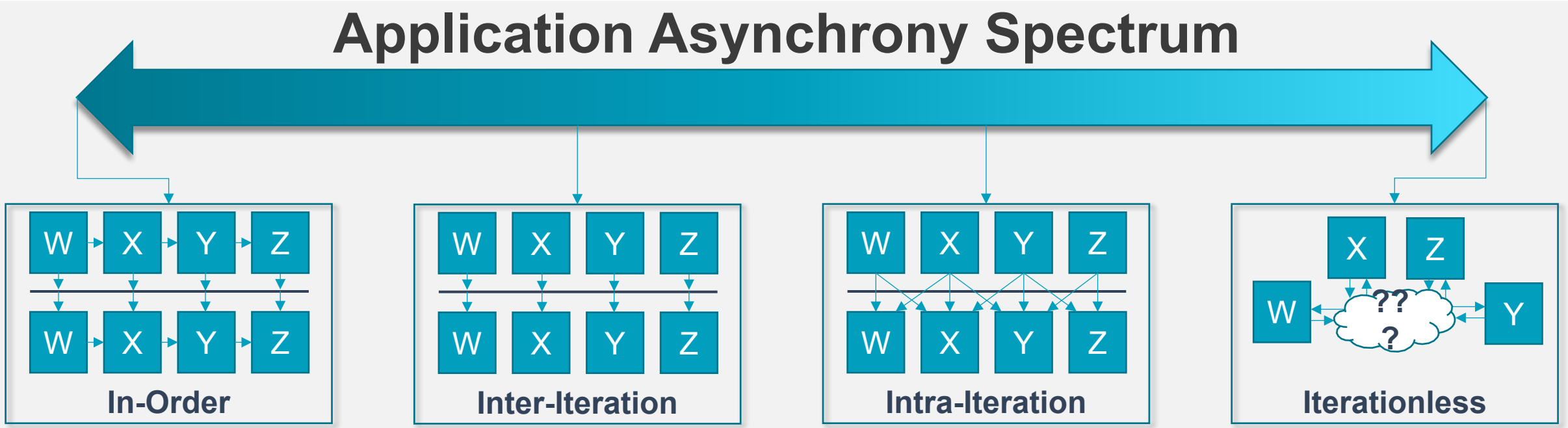
Current

```
ctx = make_context(“config.json”);
ctx->register(jacobi);
For(i):
    ctx->checkpoint(“main”, i, [&]{
        jacobi.broadcast(doIteration);
        ctx->register(jacobi);
    });
```

Lots of Features – less time

- Practice talk went way over time!
- Automatically track elements as they migrate
  - Recover elements to different distributions than checkpointed to
- Automatically track dynamically inserted elements
- Register non-local elements
  - Register from sending side
  - Avoid touching another library's code
- Register at element or collection granularity
  - Optimize messaging for managing collection checkpoint consistency
- Deregister elements
- **Today's focus**: Asynchronous checkpointing

Defining a Checkpoint Region



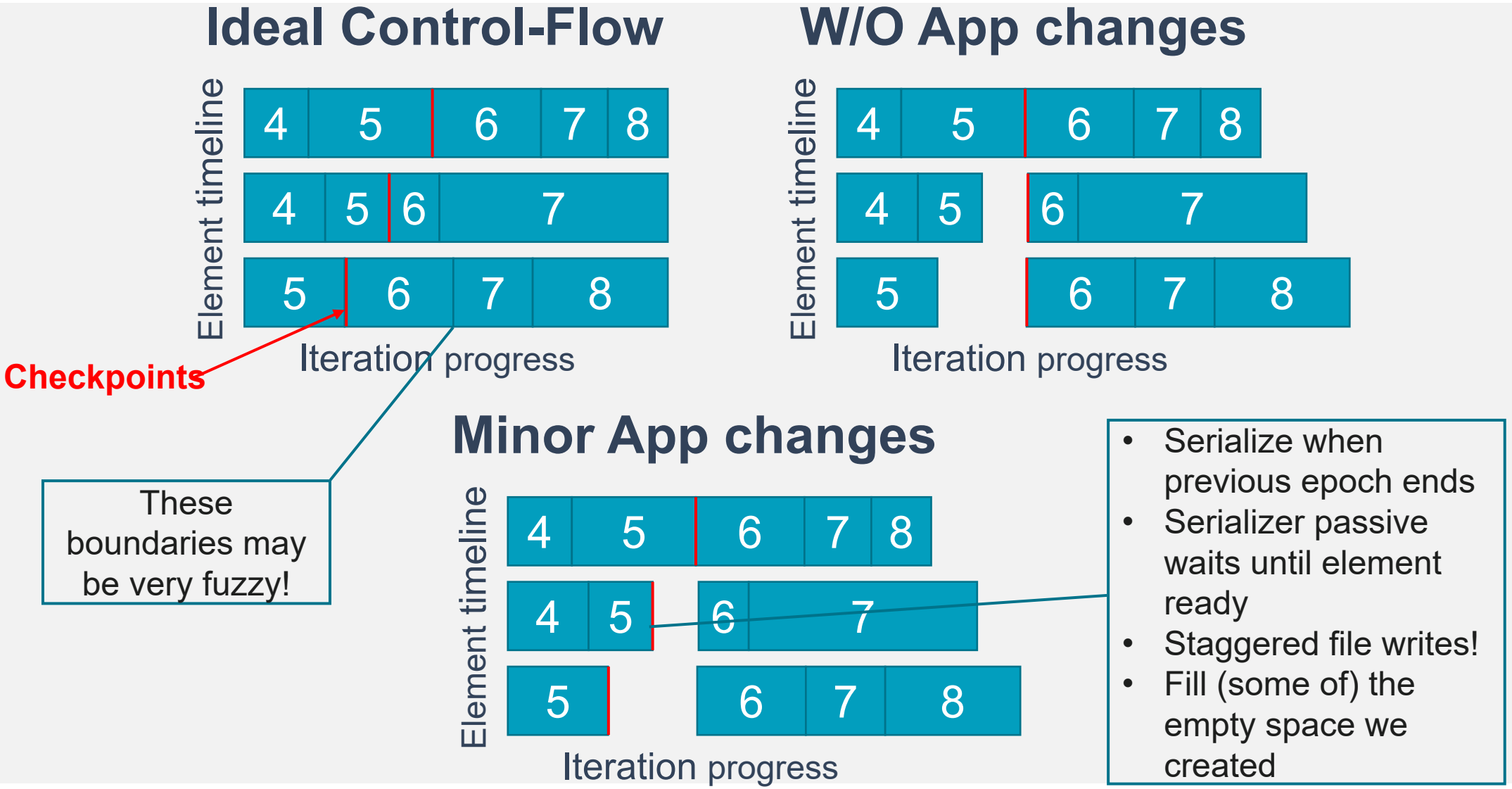
- Epochs enforce task order
- Checkpoint between any tasks

- Epochs enforce iteration order
- Checkpoint between iterations
- Individual elements may be ready sooner than others!

- Epochs are definable
- Checkpoint must prevent updates from future epochs
- Our focus

- Epochs only definable with application help
- Most apps define epochs for load balancing
- Or also checkpoint active messages

Resilience Control Flow



Checkpoint Entry Conditions

- Checkpointing iteration N
- **Required**: all context state updates initiated (register, etc.)
- Promised: initiated context updates globally completed
- Promised: iteration (N – x) completed
  - User configurable x maximum “offset” iterations
  - Ensure enough time has passed for updates to begin
  - Currently, must block at N-x until completed
    - Else scheduler runs our calls after all N have finished
    - Quirk of VT, or the app?

Checkpoint Exit Conditions

- Checkpointing iteration N
- Promised: All modified elements have finished serializing their data
- Promised: iteration N completed
  - Usually lines up with the above
  - Constraint means non-proxy data is consistent
- Can layer with VT's load-balancer constraints, if desired timing lines up

Initial tests

Hardware:

- 32 nodes
  - 2 sockets per node
  - 1 rank per socket
  - 28 threads (Kokkos OpenMP)
- Ceph filesystem
- Infiniband network

Measurement:

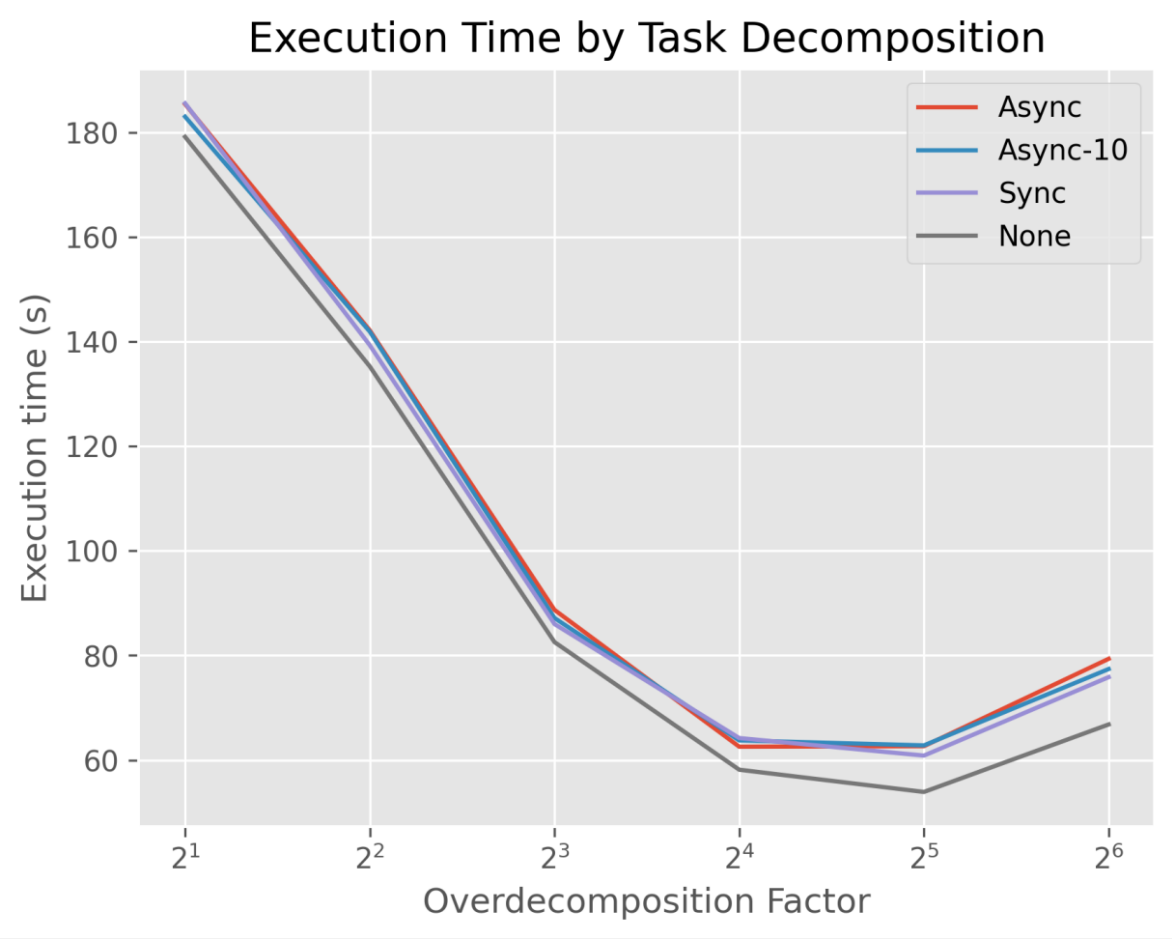
- Time application main loop only
- Reported times are average of 2

Application:

- Jacobi 3D iterative solver, run for 1000 iterations.
- 150 iterations per checkpoint
- 150 iterations per load-balancer phase
  - Disable actual load balancing, minimize variables, simulate workloads w/ inherent imbalance
- 1GB checkpoint data per node

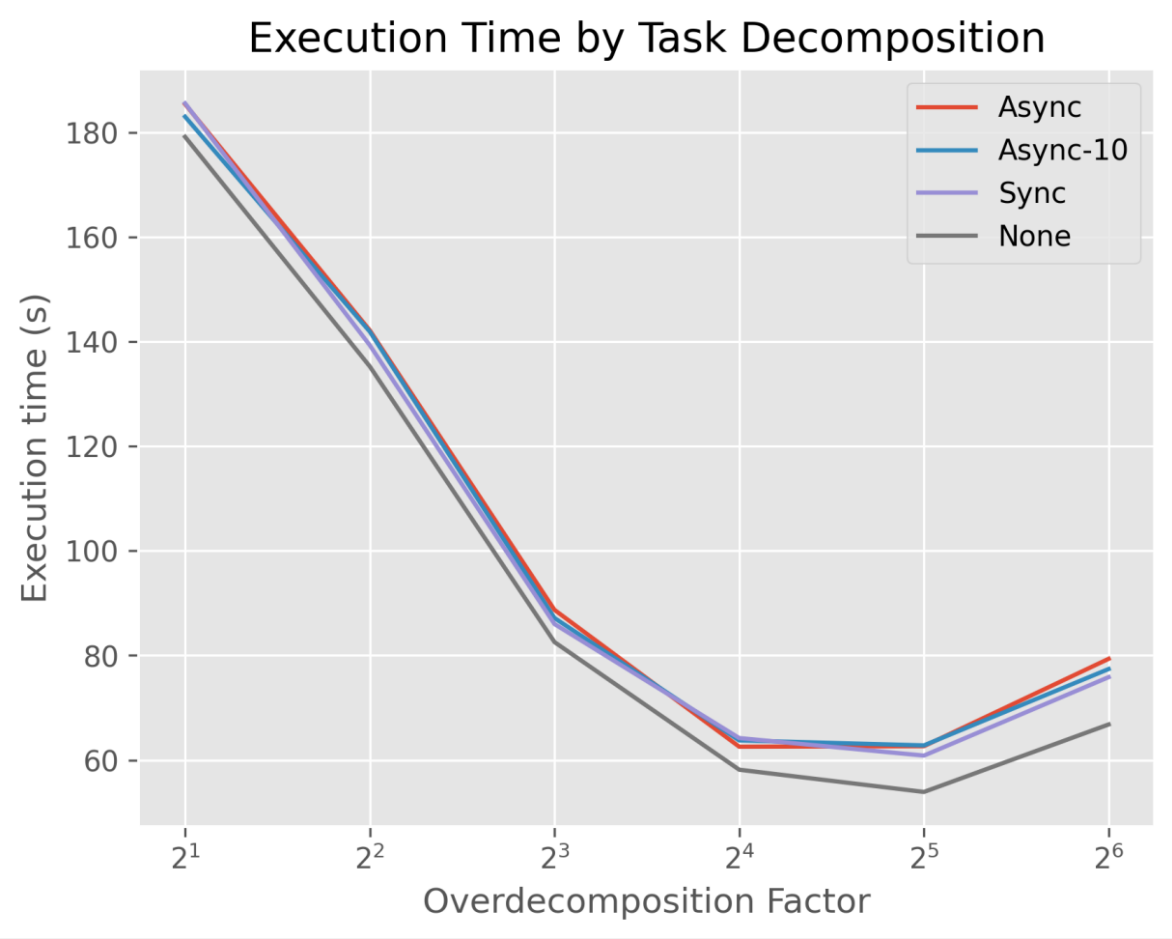
Initial results

- Three checkpoint modes:
  - **Async**: allow 149 iterations offset in starting vs ending serialization
  - **Async-10**: allow 10 iterations offset
  - **Sync**: no offset allowed
  - **None**: no checkpointing
- Testing against # elements per rank (i.e. available asynchrony)
  - Same data, broken into smaller chunks
  - VT depends on having enough elements per rank, hiding message latency



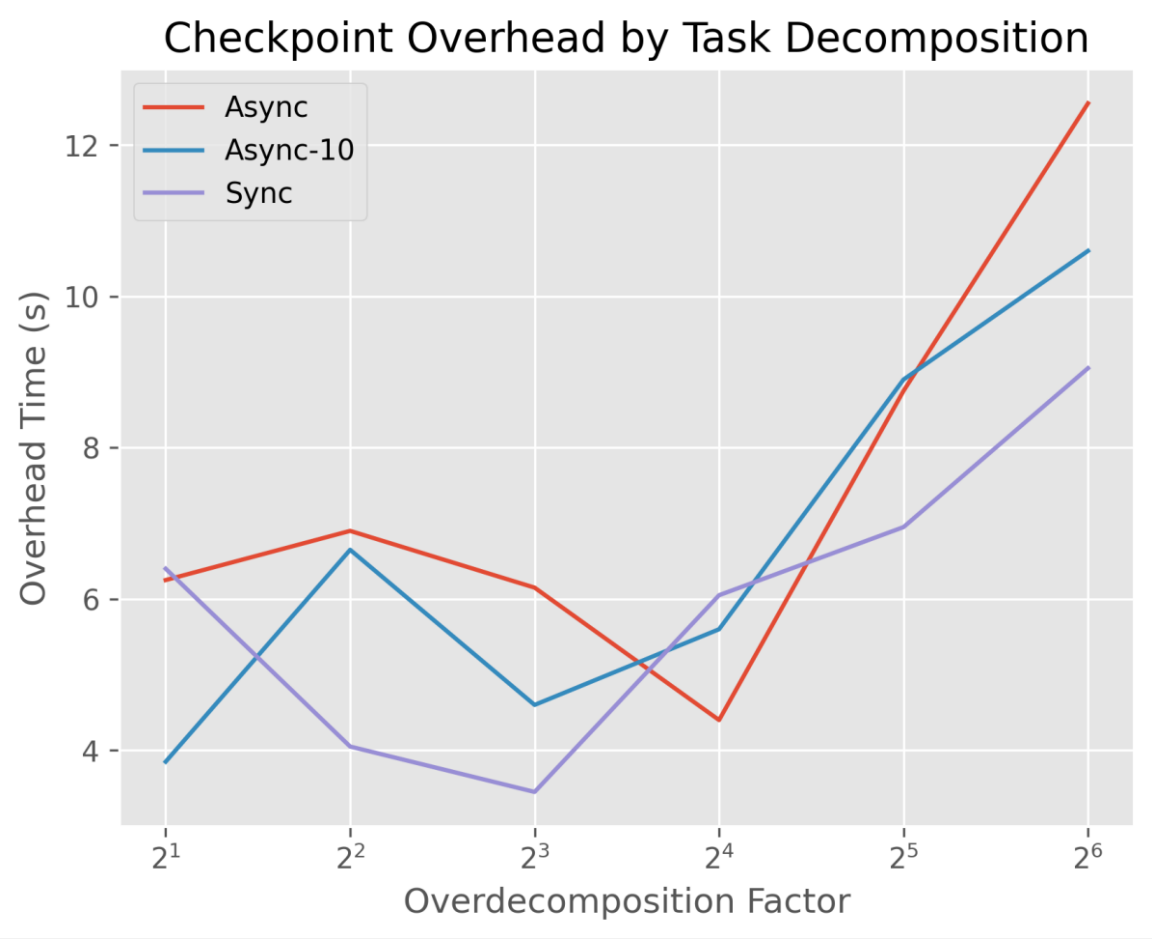
Initial results

- Resilience loses scalability vs # elements slightly faster than base app
  - Synchronization points are more expensive
  - More communication
  - Neither is an inherent cost, we can optimize!
- Checkpoint costs **can be** very low!



Initial results – Overhead only

- Large range of costs for checkpointing the same data
  - Even in synchronous case
  - Granularity of writes?
- Async: competing factors
  - Availability of time for “hiding” checkpointing
  - Cost of synchronization
- When you synchronize can have big impacts
  - Not always very predictable



Future Work

- Remove extra wait from async version
- Allow asynchrony in exiting checkpoints
  - Must still orchestrate our required data state actually existing
  - Lock individual elements in/out of epochs?
- Asynchronous file writes + asynchronous checkpoints
  - Elements asynchronously checkpoint to memory when ready
  - VeloC server handles backing up to disk
- Utilize VT Registry to pull type info from checkpoint
  - Recover data without needing pre-registration!
- Track element updates (messages, user dereferences)
  - Determine modified data automatically
- Larger test apps
  - NimbleSM, distBVH

Future research directions

- Delay checkpoints until after load balancer when overloaded?
  - Distribute cost of file writes to element recipients
  - How do we decide it's worth it?
- Checkpointing message/scheduler state?
  - Arbitrary tasks makes this difficult
  - VT manages to write tasks to messages
- Online process recovery
  - Fenix/ULFM, recover without job teardown/restart
  - How well can we avoid restarting every element?

Backup slides

Kokkos Resilience

- Automated checkpoint/recovery for (MPI+)Kokkos
  - User defines resilient regions via lambda
  - Add ViewHooks to Kokkos (available now in Kokkos 4!)
    - Template views with constructor callback hooks
  - Kokkos Resilience detects data copied into resilient region
    - Ignores const views!
    - Can C/R on device memory spaces!
- Ongoing: Automated soft-error recovery
  - Replicated execution spaces N-way duplicate work, gathers output consensus
  - Manage CPU/GPU with same API!
- https://github.com/kokkos/kokkos-resilience

Darma Virtual Transport (VT)

- Active messaging runtime, layering over MPI
- Abstract the node/rank/worker concept into migratable virtual entities
  - Dynamically load balanced
- Actor-based pattern
  - Messages addressed to virtual entity, which owns some data
  - Messages define their handlers, potentially modifying the entity's data
- Nested, transitive, causal epochs manage control-flow
- https://github.com/DARMA-tasking/vt