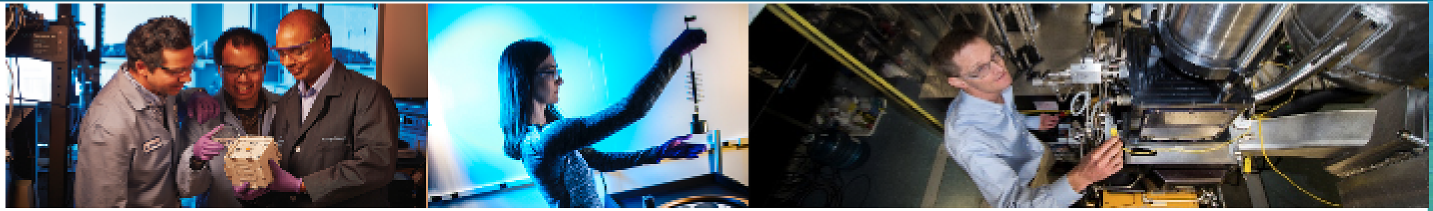
This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

SAND2022-9347C

Sandia National Laboratories

Kokkos Resilience



Nicolas Morales *Elisabeth Giem* *Matthew Whitlock* *Keita Teranishi*

PRESENTED BY

Nicolas Morales

Hardware Heterogeneity, Resilience, and the Exascale Era

- Recent years have shown a drive towards increased hardware heterogeneity in computing clusters
  - Computational heterogeneity (many-core, GPUs, other accelerators)
  - I/O heterogeneity (deep memory heirarchies, local persistent storage, external parallel file systems, key-value stores, etc)
- Mean time between failure (MTBF) is decreasing (errors occur more often!) as systems become more complex
- How can software leverage the diversity of hardware in both computation and I/O while remaining resilient and performant?

Performance Portability and Resilience

- *Performance portability* is a property of software where the software can be written once and re-used without performance degredation on heterogeneous hardware
  - Usually enabled by a programming model or library
  - Libraries include *Kokkos*, *RAJA*, *DPC*++, and others
- Main resilience strategy for a lot of hardware is checkpoint/restart
  - Can we exploit the ubiquity of programming models such as *Kokkos* to provide resilience for Exascale applications?

Kokkos-Resilience

- We introduce *Kokkos-Resilience*, an extension of *Kokkos* for adding resilience to performance-portable applications
  - Allow user to define checkpoint regions for resilient code where `Kokkos::Views` are captured
  - Introduces Resilient Execution spaces for detecting and rectifying soft errors
  - Interoperates with data checkpointing frameworks such as *VELOC* and process-level recovery frameworks
- `https://github.com/kokkos/kokkos-resilience`

Kokkos Background

- Programming model for performance portable C++
- `https://github.com/kokkos/kokkos`
- Data abstraction: `Kokkos::View<>`
- Execution abstraction: `Kokkos::parallel_for()`, `Kokkos::parallel_reduce()`, `Kokkos::parallel_scan()`
- Common use patterns:
  - View creation at program initialization
  - Iterations reading and writing to Views via parallel dispatch

Our Starting Code

```
auto view = Kokkos::View< double ** >( /* ... */ );

for ( int iter = 0; iter < max_iter; ++iter ) {
  Kokkos::parallel_for( /* ... */, KOKKOS_LAMBDA( int i ) {
    do_calculation( view );
    // More operations on view...
  } );
}
```

- How would an application author typically add checkpoint/restart?
  - Usually would want to checkpoint `view` every few iterations
  - Would need to write I/O code and check for restart
  - Custom logic for restart to ensure restart correctness

Resilient Abstractions - Scoped Resilient Execution Contexts

```cpp
auto view = KR::View< double ** >( /* ... */ );

for ( int iter = 0; iter < max_iter; ++iter ) {
  KR::checkpoint(plugin, "test_checkpoint", iter, [=]() {
    Kokkos::parallel_for( /* ... */, KOKKOS_LAMBDA( int i ) {
      do_calculation( view );
      // More operations on view...
    } );
  } );
}
```

- Define a scope in which any operations on Kokkos views are made resilient
- Resilience is enabled by checkpoint/restart
- Can include multiple parallel dispatch operations
- Implicit checkpointing of views used inside of the scope
- Checkpoint regions can be defined by lambdas (preferred) but any functor will do

Morales, Nicolas, Keita Teranishi, Bogdan Nicolae, Christian Trott, and Franck Cappello. ”Towards High Performance Resilience Using Performance Portable Abstractions.” In European Conference on Parallel Processing, pp. 451-465. Springer, Cham, 2021.

Resilient Abstractions - Scoped Resilient Execution Contexts

- Automatic recovery
  - Check if restart conditions are met (existence of a checkpoint, restart flag, etc.)
  - The lambda *is not* executed
  - Stored checkpoints written to the captured views
  - Execution proceeeds *as if* the lambda was executed
- Before someone asks – this does mean the lambdas generally should be *pure* and avoid writing to global entities (output should be done by writing to views)

Resilient Abstractions - Optimized View Tracking

```
KR::View< double * > ping( /*...*/ ), pong( /*...*/ );
for ( int i = 0; i < max_ts; ++i ) {
  KR::View< const double * > read;
  KR::View< double * > write;
  if ( i % 2 )
    read = pong; write = ping;
  else
    read = ping; write = pong;
  KokkosResilience::checkpoint( ctx, "iterate", i, [=]() {
    Kokkos::parallel_for( /*...*/, KOKKOS_LAMBDA( int j ) {
      write( j ) = do_calculation( read );
    } );
  } );
}
```

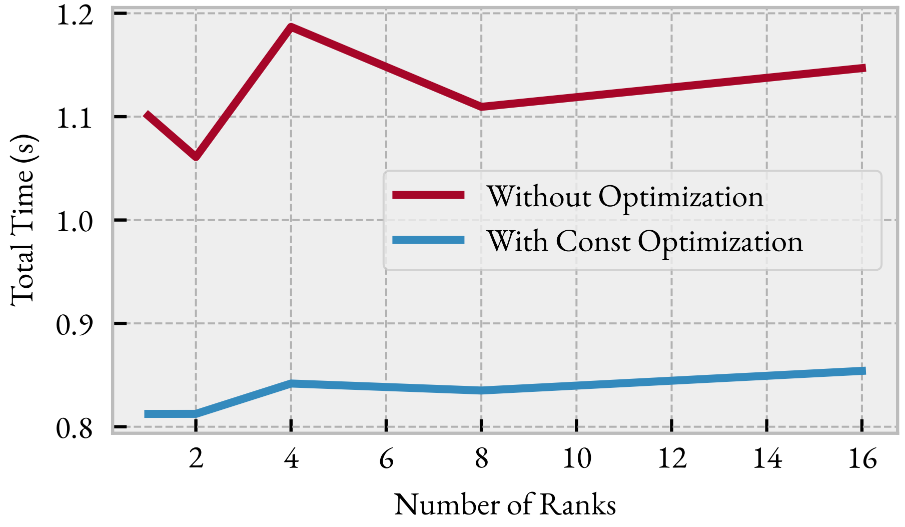


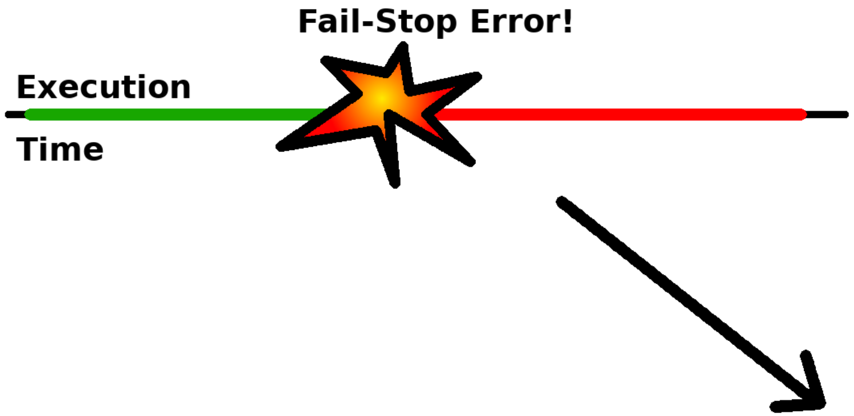
Figure: Weak scaling of the ping-pong microbenchmark.

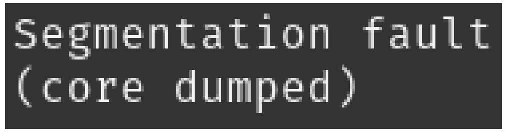
- More complex applications or poorly served by naively checkpoint every view
- The example above shows a simple ping-pong buffer sample where the read and write views alternate

Resilient Abstractions - Optimized View Tracking

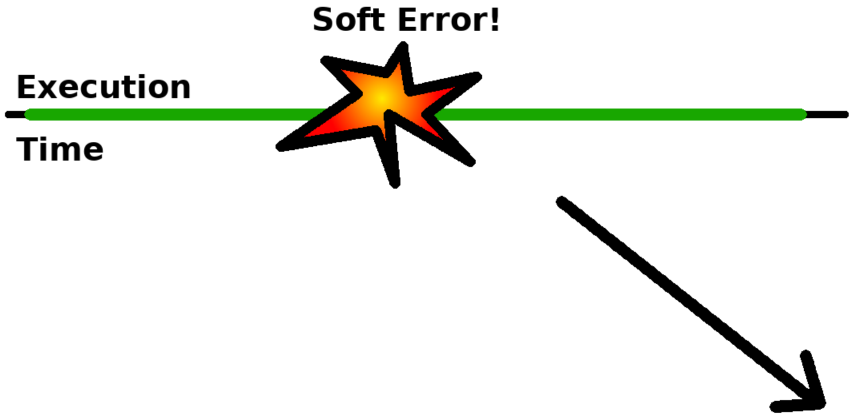
- Deduplication:
  - Hash views by label (to account for user intent in case a view is reallocated)
  - Provide functions to declare *aliases* in the rare circumstance that the same view has a different label
- Read-only views:
  - Views that are declared as **`const`** can only be read from
  - We can use this type information to avoid checkpointing these views
  - Similar techniques can be applied to views that are “transient”, i.e. are only used as scratch space inside an iteration (needs manual tagging)

Soft Errors





```
Segmentation fault
(core dumped)
```



1+1 = 3

- Fail-stop errors are easy to detect, the program crashes
- How can we detect fail-continue (soft) errors with Kokkos Resilience?
- Example errors: lock semantics, encryption/decryption, database index corruption, and more

Resilient Execution Spaces

```
auto view = Kokkos::View< double **, KR::ResHostSpace >( /* ... */ );
auto range_policy = Kokkos::RangePolicy< KR::ResOpenMP >( )/* ... */ );

for ( int iter = 0; iter < max_iter; ++iter ) {
Kokkos::parallel_for( range_policy, KOKKOS_LAMBDA( int i ) {
  do_calculation( view );
  // More operations on view...
} );
}
```

- Views are replicated, and then the kernel is executed concurrently on the replicated views
- Double and Triple Modular Redundancy supported
- Voting step proceeds in parallel after triplicate execution completed

Interopability

- We interop cleanly with non-resilient Kokkos code
- *Kokkos-Resilience* provides an abstraction layer for user applications for tracking execution regions that should be checkpointed or restarted
  - Actual checkpointing deferred to the resilience backend
    - *VELOC* – Multi-level asynchronous checkpoint/restart `https://github.com/ECP-VeloC/VELOC`
      - Used for low-overhead efficient asynchronous checkpointing
      - Designed specifically for high performance HPC machines
    - *stdfile* – Standard IO binary blob
      - Synchronous, Primarily used for testing purposes

Process Recovery

- There are many aspects of recovery – data recovery, control-flow recovery, and process recovery
  - *Kokkos-Resilience* provides control flow recovery
  - Also uses, backends like *VELOC* for data recovery
  - Integrating this with process recovery is challenging particularly since we depend on MPI communicators when interfacing with MPI+Kokkos programs
    - We have done recent work on interopability and integration with Fenix on top of ULFM for process recovery
    - Adapts to changes in the communicator, recovery status, failure states

Ongoing Work

The framework is still experimental, we need your feedback to improve it.

- We are working on adding more resilient execution spaces, checkpointing backends, and parallel constructs.
- There are also several collaboration efforts:

GaTech Supplement Kokkos resilience with a Clang-based tool and runtime checkpoint manager

  - More robust compile-time information for the runtime manager

UTK Pattern aware checkpoints for improving checkpoint performance of Kokkos applications

UofU Data staging interface for `Kokkos::View`

Kokkos Resilience

- We've introduced a new resilience capability to Kokkos applications: Kokkos Resilience
    - Automatic View checkpointing
    - Resilient execution spaces
    - Interopability with data level and process level recovery frameworks
- `https://github.com/kokkos/kokkos-resilience`
- Please feel free to contact us on the `#resilience` channel on the Kokkos Slack.

Additional Acknowledgements

- Bogdan Nicolae (ANL)
- Franck Cappello (ANL)
- Akihiro Hayashi (GaTech)
- Vivek Sarkar (GaTech)
- Christian Trott (SNL)
- Nigel Tan (UTK/ANL)
- George Bosilca (UTK)
- Aurelien Bouteiller (UTK)

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.