Kokkos Resilience

Nicolas Morales Matthew Whitlock Elisabeth Giem

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and 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.
SAND2023-14589C

- Cutting-edge hardware can suffer from severe reliability issues
  - Losing days off a job because of a fault is frustrating and costly
- There are a wide variety of faults
  - Hard failures (i.e. loss of device, node going down, crash, etc)
  - Soft faults, including silent data corruption
    - These can be challenging to even detect!
- Cause of errors
  - Hardware wear-and-tear
  - Cosmic rays causing random bitflips
  - Software bugs

- Mitigating hard failures
  - Checkpoint/restart
  - Algorithm-Based Fault Tolerance (ABFT)
  - Process recovery (i.e. MPI-ULFM + Fenix)
    - Can include localized recovery
- Mitigating and detecting soft errors
  - Hashing
  - ABFT
  - Double/Triple Modular Redundancy (DMR/TMR)
    - Run your kernel multiple times

- Kokkos Resilience is an experimental extension to Kokkos for providing built-in and automatic resilience to Kokkos Applications
  - Automatic tracking of `Kokkos::Views` in the application for Checkpoint/Restart
  - (WIP): Resilient Execution spaces for detecting soft errors
  - Multiple backends, including VeloC (for efficient async checkpointing) and `std::filesystem` backend

Automatic Checkpoint/Restart

- Checkpoint/Restart can often times be a very manual process
- Developers typically have to track critical data structures that must be checkpointed in order to successfully recover
- Errors can cause corruption in checkpoints or checkpointing unnecessary data

- Kokkos Resilience has functionality designed to automate this process
- Three key assumptions:
  - Critical data is stored in `Kokkos::View`
  - Kernels do not have side effects (e.g. modifying global state)
  - Views are conservatively annotated with `const`
    - Good practice anyway
    - Helps us optimize checkpoints

```
1  auto view = Kokkos::View< double ** >( /* ... */ );
2
3  for ( int iter = 0; iter < max_iter; ++iter ) {
4    Kokkos::parallel_for( /* ... */, KOKKOS_LAMBDA( int i ) {
5      do_calculation( view );
6      // More operations on view...
7    } );
8  }
```

```
1   auto view = Kokkos::View< double ** >( /* ... */ );
2
3   for ( int iter = 0; iter < max_iter; ++iter ) {
4     KokkosResilience::checkpoint(plugin, "test_checkpoint", iter, [=]() {
5       Kokkos::parallel_for( /* ... */, KOKKOS_LAMBDA( int i ) {
6         do_calculation( view );
7         // More operations on view...
8       } );
9     } );
10  }
```

- `KokkosResilience::checkpoint` introduces a *resilient region*
  - A resilient region defines the potential checkpoint/resume points of the application
  - Defined by (usually) a lambda but can be any functor that is copyable
  - Any `Kokkos::Views` captured inside the region will be checkpointed
  - The results of any operations (including kernels) are stored in the checkpoint
  - A region can contain multiple kernels

```
1  template< typename Context, typename F, typename FilterFunc >
2  void checkpoint( Context &ctx, // checkpointing config
3                   const std::string &label, // checkpoint label
4                   int iteration, // checkpoint versioning
5                   F &&fun, // lambda defining the resilient region
6                   FilterFunc &&filter ) // how frequent to checkpoint
```

- 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
- Lambdas **must** be *pure* and avoid writing to global entities or any other global state (output should be done by writing to views). No side effects are allowed because we cannot capture them.

- When using default lambda capture, we have a conservative superset of views that are required in a computational region
- Lambda introspection – how do we know what is captured by a lambda?
  - The `Kokkos::View` copy constructor implements reference counting – views are a bit like shared pointers
  - We provide hooks inside the view copy constructor that implement extra bookkeeping
    - When a view is copied through the lambda copy, it is added to a list of checkpointable views
  - Need a special "hook" to enable. We provide a type `KokkosResilience::View`
  - Limitations – references to views. Must avoid default capture of `this`

Consider:

```
1   Kokkos::View< double * > ping( /*...*/ ), pong( /*...*/ );
2   for ( int i = 0; i < max_ts; ++i ) {
3     Kokkos::View< const double * > read;
4     Kokkos::View< double * > write;
5     if ( i % 2 )
6       read = pong; write = ping;
7     else
8       read = ping; write = pong;
9   }
10  Kokkos::parallel_for( /*...*/, KOKKOS_LAMBDA( int j ) {
11    write( j ) = do_calculation( read );
```

Now made resilient:

```
1   KokkosResilience::View< double * > ping( /*...*/ ), pong( /*...*/ );
2   for ( int i = 0; i < max_ts; ++i ) {
3     KokkosResilience::View< const double * > read;
4     KokkosResilience::View< double * > write;
5     if ( i % 2 )
6       read = pong; write = ping;
7     else
8       read = ping; write = pong;
9     KokkosResilience::checkpoint( ctx, "iterate", i, [=]() {
10      Kokkos::parallel_for( /*...*/, KOKKOS_LAMBDA( int j ) {
11        write( j ) = do_calculation( read );
12      } );
13    } );
14  }
```

- More complex applications or poorly served by naively checkpoint every view
- The example shows a simple ping-pong buffer sample where the read and write views alternate
- Considering the logic of the example, we are checkpointing 2x what we should be

- 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 could be applied to views that are "transient", i.e. only used as scratch space inside an iteration (but would need manual tagging)

- Kokkos Resilience provides an abstraction layer for user applications for tracking execution regions that should bec checkpointed or restarted
- Actual checkpointing deferred to the resilience backend
- Backend is responsible for format, location, compression, etc. of checkpoints and restarts
- This makes our approach flexible and adaptable to a wide variety of heterogeneous I/O systems
- Current backends:
  - *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

- System specs:
  - Machine has about 1500 compute nodes
  - 2.1 GHz Intel Broadwell CPUs (36 cores, 72 hardware threads)
  - 128 GB of RAM
  - Omni-Path interconnect and Lustre filesystem
- Software:
  - Modified version of *Kokkos* 3.0
  - *VELOC* version 1.4, with `/dev/shm` staging and asynchronous mode enabled

- We performed two types of experiments: microbenchmarks to evaluate specific performance aspects and two mini-apps, to test real-world use-cases
- Microbenchmarks
  - Registration overhead with large number of views
  - Ping-pong example (shown previously) to test view optimization
- Mini-apps
  - *Heatdis* – a heat distribution solver (HeatDis) that is used as an example by *VELOC*
  - *MiniMD* – a parallel molecular dynamics application written using Kokkos abstractions for the Mantevo project

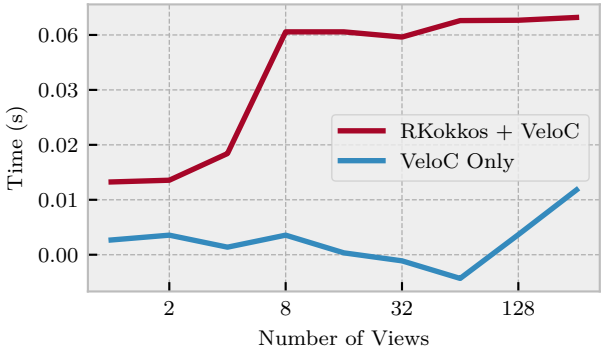


Figure: Bookkeeping overhead related to tracking the memory views using our approach vs. manual registration using *VELOC*. Checkpointing is deactivated, but tracking is still active. This timing data includes compute time. The number of views is normally related to the complexity of the program rather than the data size.

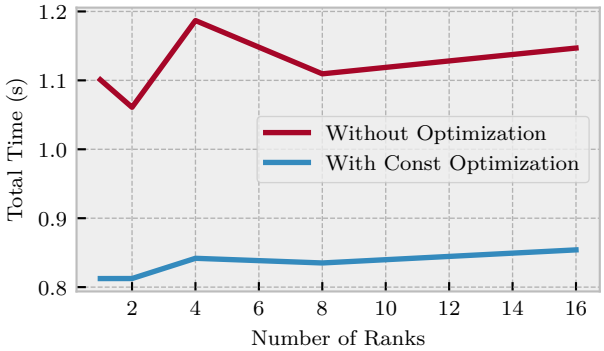


Figure: Weak scaling of the *ping-pong* microbenchmark with and without the *const* memory view tracking optimization. The checkpoint size is 50% smaller when the const-tracking is activated, which significantly improves the checkpointing performance. The microbenchmark uses 1 kB views.

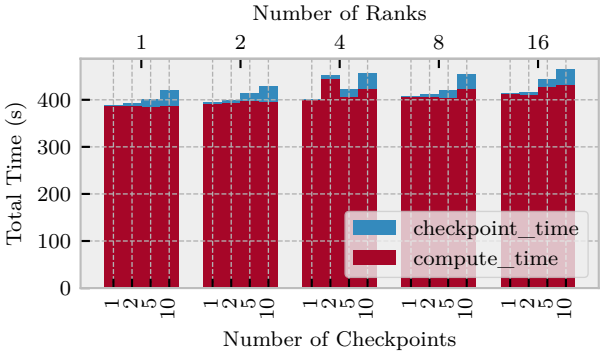


Figure: *MiniMD*: Breakdown of checkpointing time vs. compute time for a increasing number of nodes and checkpoints in a weak scalability scenario over 1000 time steps.

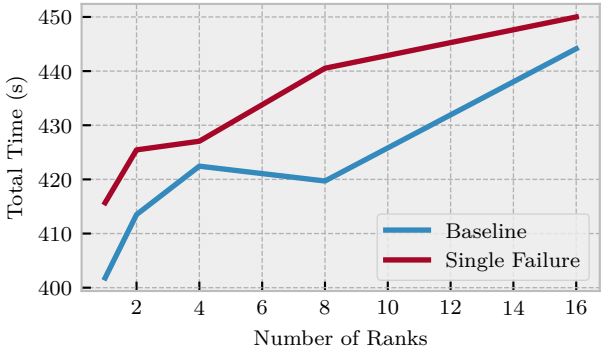


Figure: *MiniMD*: Weak scalability of MiniMD with a checkpoint every 200 iterations. The red plot shows the performance when there is a single failure.

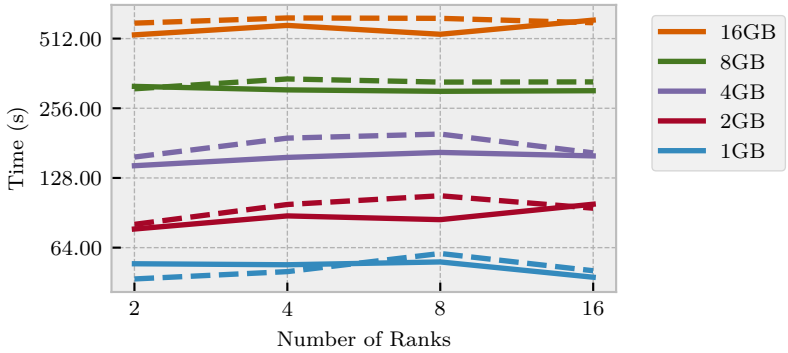


Figure: *HeatDis*: Weak scaling of heatdis with and without restart (1000 iterations, checkpoint every 100 iterations). Dashed lines are restart timings.

- Probably the biggest one: critical data structures should be in `Kokkos::View`s
  - This includes control flow data structures
  - We provide some tools to "wrap" non-Kokkos data structures to enable tracking for them
  - How do we serialize these wrapped data structures? We need to look at production applications
- Resilient regions shouldn't have side-effects
  - I would argue well written code should avoid side-effects anyway, but real-world code is not always well-written

- Pre C++ 20 lambda capture of `this`

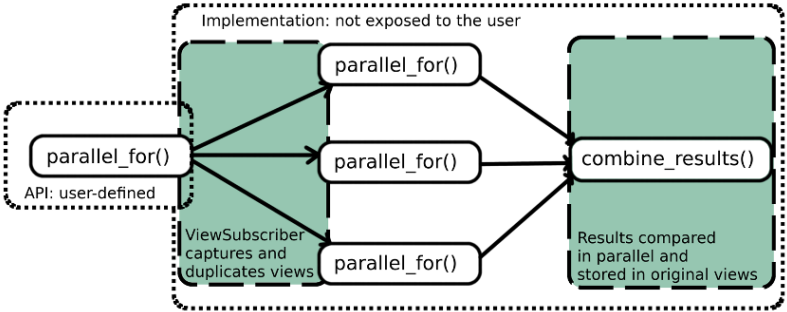
```
1  struct foot {
2    auto view = Kokkos::View< double ** >( /* ... */ );
3    void iterate() {
4      for ( int iter = 0; iter < max_iter; ++iter ) {
5        KokkosResilience::checkpoint(plugin, "test_checkpoint", iter, [=]() {
6          Kokkos::parallel_for( /* ... */, KOKKOS_LAMBDA( int i ) {
7            do_calculation( view ); // Ouch! Not captured!
8          } );
9        } );
10     }
11   }
12 };
```

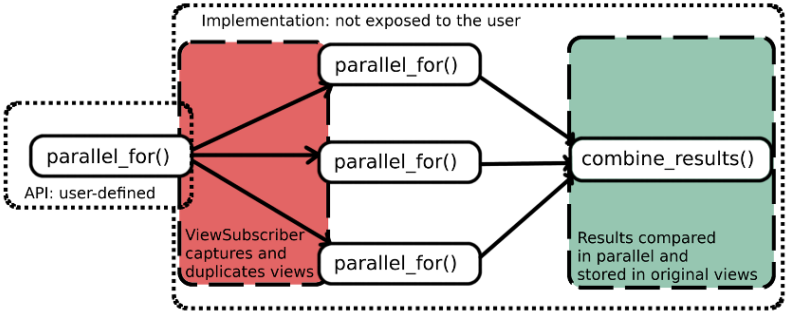
(WIP) Resilient Execution Spaces

- Fail-stop errors are easy to detect
  - The program just crashes or aborts
  - Checkpoint/Restart is sufficient to mitigate
- Fail-continue (soft) errors can cause subtle and difficult to detect problems
  - Race conditions from improperly released locks
  - Problems with encryption/decryption
  - Database index corruption
  - Numerical instability
  - etc...

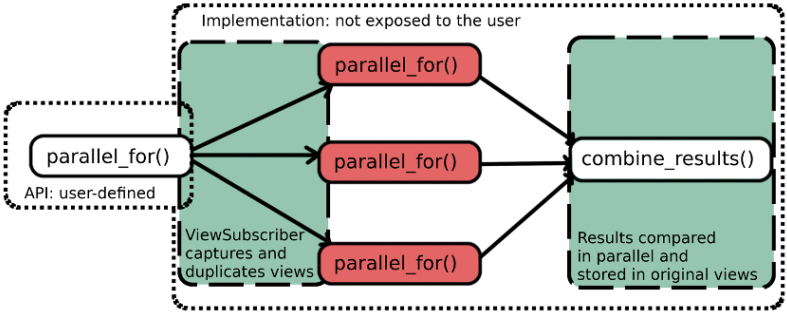
```
1 using namespace Kokkos;
2 using namespace KR = KokkosResilience;
3 KR::View<double*, LayoutRight, ResHostSpace> V = /*...*/;
4 parallel_for( RangePolicy< KR::ResOpenMP >(0, N), KOKKOS_LAMBDA( int i ) {
5   // Kernel that will now be run three times
6 } );
```

- Work-in progress feature for mitigating some categories of soft errors
- A natural extension to execution spaces
  - Caveat: currently only available for OpenMP, other spaces are in development
- Implements automatic double/triple moduler redundancy (DMR/TMR)

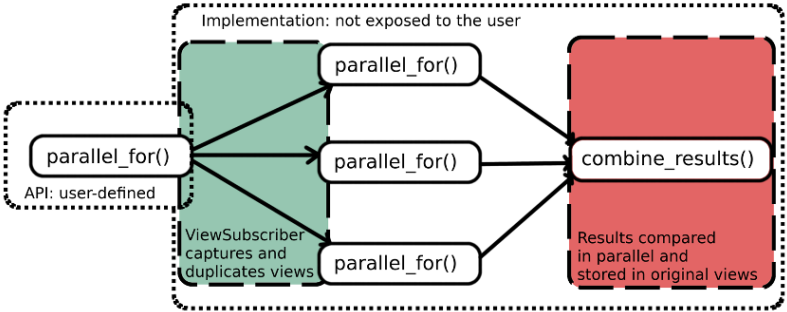

Implementation: not exposed to the user
parallel_for()
parallel_for()
parallel_for()
parallel_for()
API: user-defined
ViewSubscriber
captures and
duplicates views
combine_results()
Results compared
in parallel and
stored in original views


Implementation: not exposed to the user
parallel_for()
parallel_for()
parallel_for()
parallel_for()
API: user-defined
ViewSubscriber
captures and
duplicates views
combine_results()
Results compared
in parallel and
stored in original views

- Exploits a similar lambda trick as Automatic Checkpointing but with Kernel lambdas
- We copy the kernel functor in order to capture views
- The “hook” actually happens at compile time, which makes it very low overhead.
  - Still have to copy the view data


Implementation: not exposed to the user
parallel_for()
parallel_for()
parallel_for()
parallel_for()
API: user-defined
ViewSubscriber
captures and
duplicates views
combine_results()
Results compared
in parallel and
stored in original views

- Triplicate parallel kernels are not actually run as separate kernels
  - Re-index original range policy
  - Don't have to pay the launch cost of a kernel multiple times
  - Helps avoid potentially rescheduling the same index on the same core
    - Only a mitigation, we are looking at maybe perturbing indices more but requires more thought


Implementation: not exposed to the user
parallel_for()
parallel_for()
parallel_for()
parallel_for()
combine_results()
API: user-defined
ViewSubscriber
captures and
duplicates views
Results compared
in parallel and
stored in original views

- Combiner is dependent on the datatype in the view
  - $a = b$ for integral types
  - $|a - b| < \epsilon$ for floating point types
- 2/3 votes on a result to resolve
- If there is no resolution we retry the kernels until a specified limit, then abort if there is still no resolution

- Kokkos Resilience provides tools for enabling resilience and fault tolerance in Kokkos-based codes
- We would be interested in working with applications who are interested in our approach
- Things we've done but I didn't talk about:
  - Process resilience integration with Fenix + MPI-ULFM
- Future/ongoing work:
  - Compiler plugin for resilience
    - More robust View tracking and don't need weird lambda tricks
  - Integration with distributed tasking + process resilience of AMT runtimes

Questions?