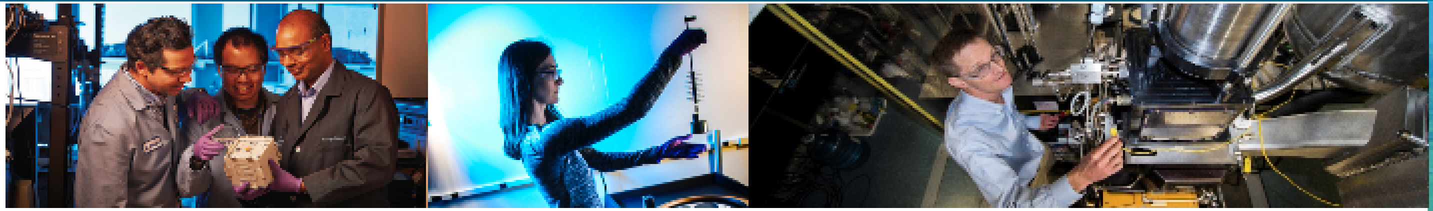
Resilient Kokkos:
Productive and Performance Portable User-Level Checkpointing for High Performance Computing



Nicolas Morales *Jeffery Miles* *Carson Mould*
Bogdan Nicolae *Hartmut Kaiser* *Keita Teranishi*

PRESENTED BY

Nicolas Morales

Resilience and Performance Portability

- Increase in performance correlates with increase in hardware heterogeneity
  - 6 of the top 10 supercomputers are using NVIDIA GPUs
  - Future exascale systems are using a greater variety of hardware with new ISAs
- Porting software across machines with a variety of hardware, ISAs, and software stacks is challenging. Motivates:
  - Kokkos (`https://github.com/kokkos/kokkos/`)
  - RAJA (`https://github.com/LLNL/RAJA`)
  - HPX (`https://github.com/STEllAR-GROUP/hpx`)
- However, mean time between failure (MTBF) is decreasing as complexity increases
  - How does resilience fit in with performance portability?

Data Portability

- Data movement between host and device required for checkpointing
- Explicit data movement can be avoided through schemes such as NVIDIA UVM or managed memory but has performance costs
  - e.g. page faults pulling from CPU memory hierarchy
  - GPUDirect Storage also a possibility, but may be limited in availability
- Data should typically be structured differently depending on memory space: e.g. for caching on CPU, coalescing on GPU
- Programming models such as Kokkos provide abstractions for portable data
  - `Kokkos::View` has compile-time traits for memory space and efficient access
    - Same API regardless of hardware
  - `Kokkos::deep_copy` for movement between memory spaces

Performance Portability and Resilience

- As MTBF decreases, we face challenges for making performance portable software resilient
  - Data movement for checkpointing complicates code, reduces maintainability
  - Minimizing the size of checkpoints for large data sets
  - Resiliency of GPU kernels requires more than just enabling ECC
- Take advantage of ubiquity of `Kokkos::View`s for data in Kokkos code

Performance Portability and Resilience

```c
VELOC_Mem_protect(0, &i, 1, sizeof(int));
VELOC_Mem_protect(1, h, M * N, sizeof(double));
VELOC_Mem_protect(2, g, M * N, sizeof(double));
if (VELOC_Restart_test("heatdis", 0) > 0) {
  assert(VELOC_Restart("heatdis", v) == VELOC_SUCCESS);
} else {
  i = 0;
}

while(i < ITER_TIMES) {
  le = doWork(np, rank, M, nbLines, g, h);
  if ((i % REDUCED) == 0)
    MPI_Allreduce(&le, &ge, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
  if (ge < PRECISION)
    break;
  i++;
  if (i % CKPT_FREQ == 0 && i != ITER_TIMES) {
    assert(VELOC_Checkpoint("heatdis", i) == VELOC_SUCCESS);
  }
}
```

Listing: Example checkpointing of a heat distribution code using VeloC.

Performance Portability and Resilience

```c
VELOC_Mem_protect(0, &i, 1, sizeof(int));
VELOC_Mem_protect(1, h, M * N, sizeof(double));
VELOC_Mem_protect(2, g, M * N, sizeof(double));
if (VELOC_Restart_test("heatdis", 0) > 0) {
  assert(VELOC_Restart("heatdis", v) == VELOC_SUCCESS);
} else {
  i = 0;
}

while(i < ITER_TIMES) {
  le = doWork(np, rank, M, nbLines, g, h);
  if ((i % REDUCED) == 0)
    MPI_Allreduce(&le, &ge, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
  if (ge < PRECISION)
    break;
  i++;
  if (i % CKPT_FREQ == 0 && i != ITER_TIMES) {
    assert(VELOC_Checkpoint("heatdis", i) == VELOC_SUCCESS);
  }
}
```

Listing: Example checkpointing of a heat distribution code using VeloC.

Performance Portability and Resilience

```
VELOC_Mem_protect(0, &i, 1, sizeof(int));
VELOC_Mem_protect(1, h, M * N, sizeof(double));
VELOC_Mem_protect(2, g, M * N, sizeof(double));
if (VELOC_Restart_test("heatdis", 0) > 0) {
  assert(VELOC_Restart("heatdis", v) == VELOC_SUCCESS);
} else {
  i = 0;
}

while(i < ITER_TIMES) {
  le = doWork(np, rank, M, nbLines, g, h);
  if ((i % REDUCED) == 0)
    MPI_Allreduce(&le, &ge, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
  if (ge < PRECISION)
    break;
  i++;
  if (i % CKPT_FREQ == 0 && i != ITER_TIMES) {
    assert(VELOC_Checkpoint("heatdis", i) == VELOC_SUCCESS);
  }
}
```

Listing: Example checkpointing of a heat distribution code using VeloC.

Performance Portability and Resilience

```
VELOC_Mem_protect(0, &i, 1, sizeof(int));
VELOC_Mem_protect(1, h, M * N, sizeof(double));
VELOC_Mem_protect(2, g, M * N, sizeof(double));
if (VELOC_Restart_test("heatdis", 0) > 0) {
  assert(VELOC_Restart("heatdis", v) == VELOC_SUCCESS);
} else {
  i = 0;
}

while(i < ITER_TIMES) {
  le = doWork(np, rank, M, nbLines, g, h);
  if ((i % REDUCED) == 0)
    MPI_Allreduce(&le, &ge, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
  if (ge < PRECISION)
    break;
  i++;
  if (i % CKPT_FREQ == 0 && i != ITER_TIMES) {
    assert(VELOC_Checkpoint("heatdis", i) == VELOC_SUCCESS);
  }
}
```

Listing: Example checkpointing of a heat distribution code using VeloC.

Existing Resilient Patterns

- Challenges
  - Explicit checking for checkpoint/restart
  - User determination of what data is checkpointed
  - Extra boilerplate (not shown) for device data

Kokkos Resilience

- Kokkos Resilience is a framework for checkpointing `Kokkos::View` and performing resilient Kokkos execution
- Features:
  - Avoid boilerplate for device-local data
    - Provide mechanism for manually deep copying a view to checkpoint storage
  - Efficient checkpoints without manual user tracking
    - Automatic tracking and checkpointing of `Kokkos::View`
  - Resilient parallel execution
    - Support both host and device execution space

Manual Checkpointing

- User specifies a checkpoint directory
  - Support for formats including HDF5
  - More format types can be added as memory spaces
- Create a filesystem mirror for views
  - No boilerplate code for copy from device
  - Copy implemented in API avoids extra data copy
- Checkpoint and restore with a single API call

Manual Checkpointing

```
KokkosResilience::HDF5Space space;

auto x_cp = Kokkos::create_chkpt_mirror(space, x);
auto v_cp = Kokkos::create_chkpt_mirror(space, v);
auto f_cp = Kokkos::create_chkpt_mirror(space, f);

// Write all mirror views to disk
KokkosResilience::HDF5Space::checkpoint_views();
```

Listing: Manual checkpointing using filesystem mirrors.

- Mirror spaces to be checkpointed in the filesystem space (here HDF5)
- `HDF5Space::checkpoint_views()` checkpoints all mirrored views with a `Kokkos::deep_copy`

Automatic Checkpointing

- Manually tracking which views need to be checkpointed is time consuming and error-prone
- This information is actually known conservatively by the compiler
    - e.g. It can't know the result of a conditional, but if a `Kokkos::View` can potentially be used, the compiler knows it
    - C++ lambdas with default capture will only actually capture what is used inside the lambda
- Checkpoint regions specified using a C++ lambda
- Mimicks control flow similarly to a conditional, is natural to add to existing code
- Uses the VeloC backend for efficient asynchronous checkpointing (https://github.com/ECP-VeloC/VELOC)

View Tracking

```
int i = 0;

while(i < ITER_TIMES) {
  le = doWork(np, rank, M, nbLines, g, h);
  if ((i % REDUCED) == 0)
    MPI_Allreduce(&le, &ge, 1, MPI_DOUBLE, MPI_MAX,
    ↪  MPI_COMM_WORLD);
  if (ge < PRECISION)
    break;
  i++;
}
```

Listing: Heat Distribution code without checkpointing.

```
auto i = 1 + KokkosResilience::latest_version( *ctx,
↪  "heatdis" );

while(i < ITER_TIMES) {
  KokkosResilience::checkpoint(ctx, "heatdis", i,
    [=, &le, &ge]() {
      le = doWork(np, rank, M, nbLines, g, h);
      if ((i % REDUCED) == 0)
        MPI_Allreduce(&le, &ge, 1, MPI_DOUBLE, MPI_MAX,
        ↪  MPI_COMM_WORLD);
  } );
  if (ge < PRECISION)
    break;
  i++;
}
```

Listing: Heat Distribution code with automatic checkpointing.

View Tracking

```
int i = 0;

while(i < ITER_TIMES) {
  le = doWork(np, rank, M, nbLines, g, h);
  if ((i % REDUCED) == 0)
    MPI_Allreduce(&le, &ge, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
  if (ge < PRECISION)
    break;
  i++;
}
```

Listing: Heat Distribution code without checkpointing.

```
auto i = 1 + KokkosResilience::latest_version( *ctx, "heatdis" );

while(i < ITER_TIMES) {
  KokkosResilience::checkpoint(ctx, "heatdis", i,
    [=, &le, &ge]() {
      le = doWork(np, rank, M, nbLines, g, h);
      if ((i % REDUCED) == 0)
        MPI_Allreduce(&le, &ge, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
  } );
  if (ge < PRECISION)
    break;
  i++;
}
```

Listing: Heat Distribution code with automatic checkpointing.

View Tracking

```
int i = 0;

while(i < ITER_TIMES) {
  le = doWork(np, rank, M, nbLines, g, h);
  if ((i % REDUCED) == 0)
    MPI_Allreduce(&le, &ge, 1, MPI_DOUBLE, MPI_MAX,
    ↪  MPI_COMM_WORLD);
  if (ge < PRECISION)
    break;
  i++;
}
```

Listing: Heat Distribution code without checkpointing.

```
auto i = 1 + KokkosResilience::latest_version( *ctx,
↪  "heatdis" );

while(i < ITER_TIMES) {
  KokkosResilience::checkpoint(ctx, "heatdis", i,
    [=, &le, &ge]() {
      le = doWork(np, rank, M, nbLines, g, h);
      if ((i % REDUCED) == 0)
        MPI_Allreduce(&le, &ge, 1, MPI_DOUBLE, MPI_MAX,
        ↪  MPI_COMM_WORLD);
  } );
  if (ge < PRECISION)
    break;
  i++;
}
```

Listing: Heat Distribution code with automatic checkpointing.

View Tracking

```
int i = 0;

while(i < ITER_TIMES) {
  le = doWork(np, rank, M, nbLines, g, h);
  if ((i % REDUCED) == 0)
    MPI_Allreduce(&le, &ge, 1, MPI_DOUBLE, MPI_MAX,
    ↪   MPI_COMM_WORLD);
  if (ge < PRECISION)
    break;
  i++;
}
```

Listing: Heat Distribution code without checkpointing.

```
auto i = 1 + KokkosResilience::latest_version( *ctx,
↪   "heatdis" );

while(i < ITER_TIMES) {
  KokkosResilience::checkpoint(ctx, "heatdis", i,
    [=, &le, &ge]() {
      le = doWork(np, rank, M, nbLines, g, h);
      if ((i % REDUCED) == 0)
        MPI_Allreduce(&le, &ge, 1, MPI_DOUBLE, MPI_MAX,
        ↪   MPI_COMM_WORLD);
  } );
  if (ge < PRECISION)
    break;
  i++;
}
```

Listing: Heat Distribution code with automatic checkpointing.

View Tracking Implementation

- 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 being copied, it is added to a list of checkpointable views
  - Minimal performance cost, but can be disabled if resilience is not desired
  - Limitations – references to views. Must avoid default capture of `this` (illegal in Kokkos code already)!

Non-View Checkpointing

```
serializable_object x;

KokkosResilience::checkpoint( *ctx, "chk", 0, [=]() {
  Kokkos::parallel_for( dim0, KOKKOS_LAMBDA( int i ) {
    /*...*/
  } );
}, x );
```

Listing: Appending HPX serializable objects to the checkpoint.

- Often require non-view elements to be checkpointed.
- Serialization method needs to be explicitly defined
- In these cases, we support serialization through HPX's serialization framework
- Items must be explicitly specified

Resilient Execution

```
using view_type = Kokkos::View<double*, KokkosResilience::ResCudaSpace>;

view_type data("data", N);
Kokkos::RangePolicy<KokkosResilience::ResCuda> rp(0, N);

Kokkos::parallel_for(rp, KOKKOS_LAMBDA(const int i) {
    data(i) = i;
});
```

Listing: Example of using the resilient execution API.

- Redundant execution using multiple streams
- Three step execution
  - Replicate referenced data
  - Concurrent execution using parallel execution spaces
  - Recombine via voting

Resilient Execution Implementation

- Replicated data captured through `Kokkos::View`
- Concurrent kernel execution
  - CUDA streams
  - OpenMP tasks
- Voting step dependent on datatype at compile time
  - $a = b$ for integral types
  - $|a - b| < \epsilon$ for floating point types

Resilient Execution Implementation

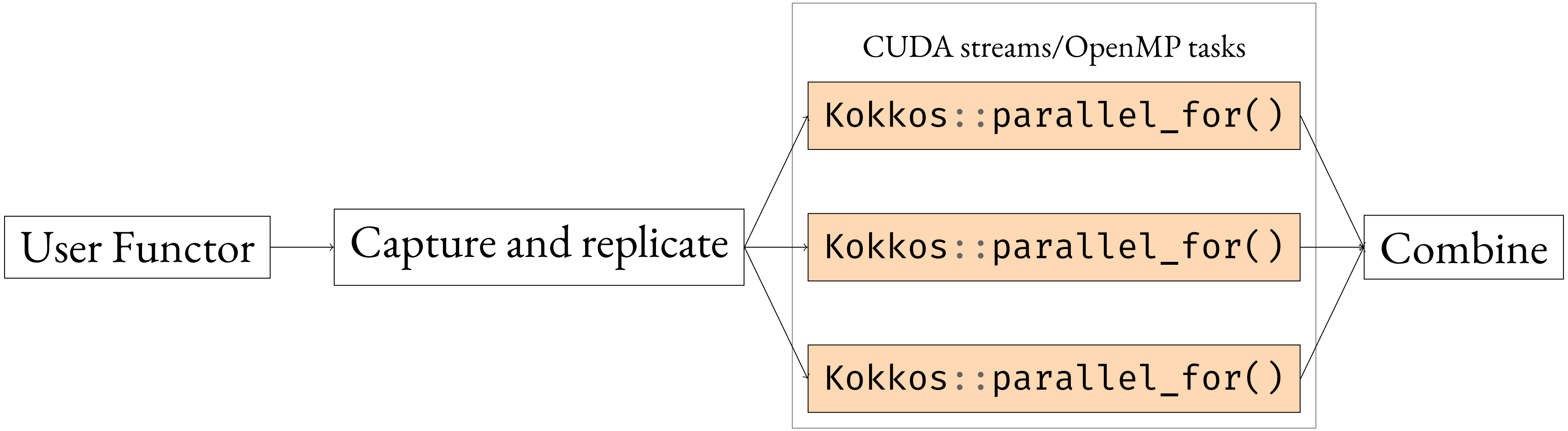


Figure: Overview of resilient Kokkos execution

MiniMD Manual Checkpoint Experiment

- Application: MiniMD molecular dynamics
- Combination Kokkos (OpenMP) + MPI
- 128GB Broadwell Nodes
- 4 OpenMP threads per rank

MiniMD Manual Checkpoint Results

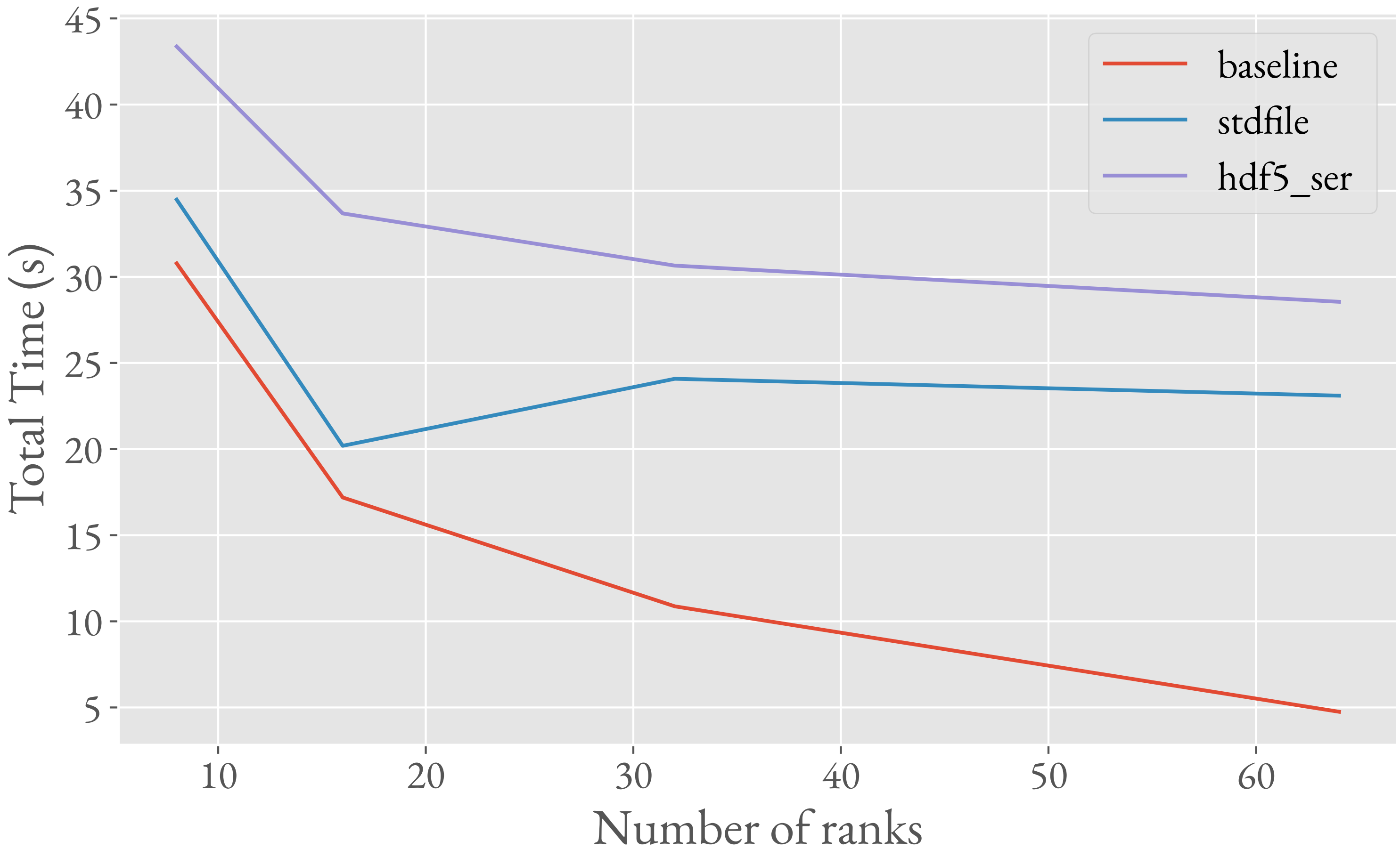


Figure: Strong scaling with manual checkpointing in MiniMD.

MiniMD Automatic Checkpoint Experiment

- Application: MiniMD molecular dynamics
- Combination Kokkos (OpenMP) + MPI
- Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz
- 2 sockets, 32 threads per socket
- 62 threads for minimd, 1 thread for VeloC backend
- 1.3GB/node data
- 100 total time steps

MiniMD Automatic Checkpointing Results

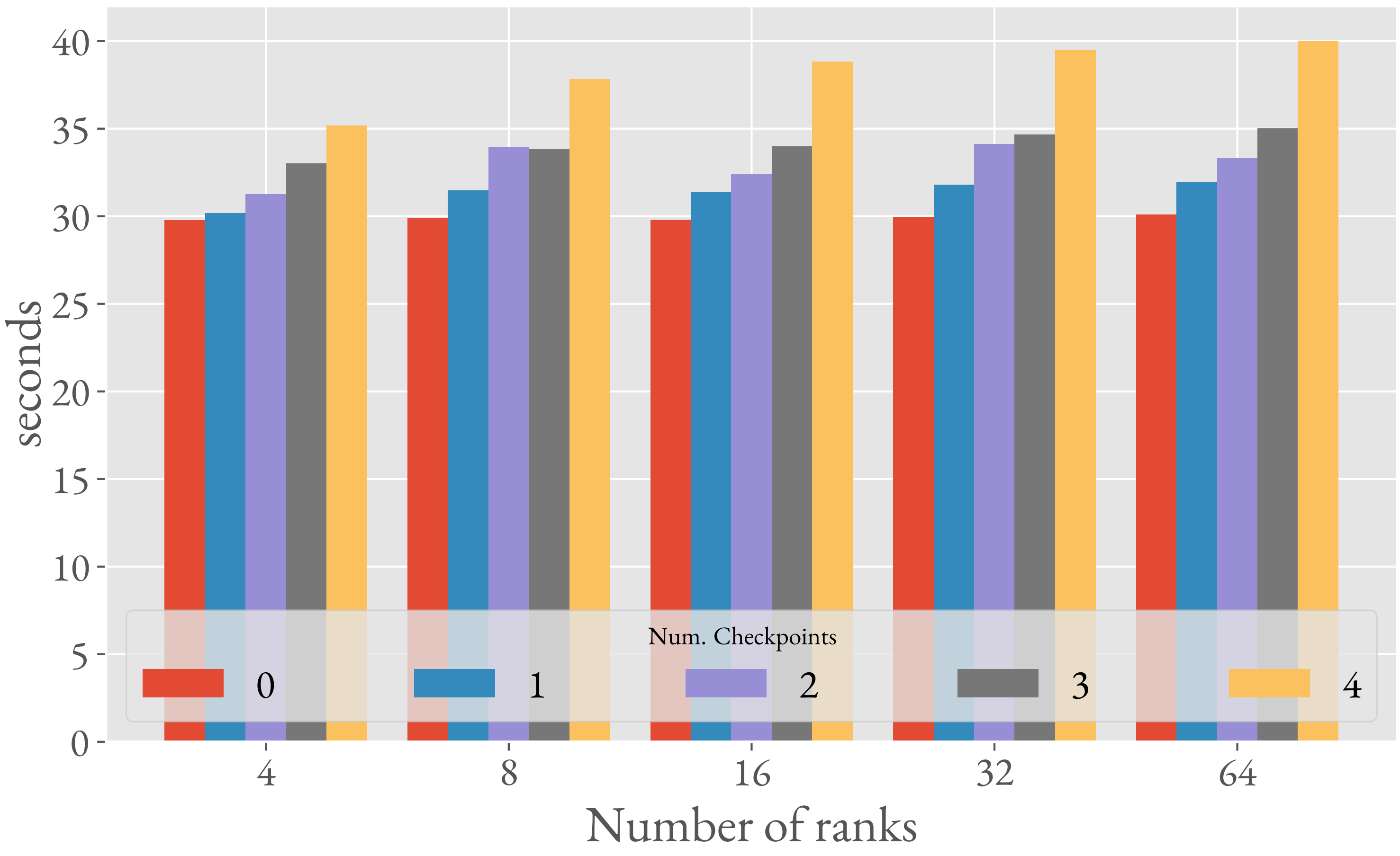


Figure: Weak scaling with automatic checkpointing in the MiniMD app.

HeatDis Automatic Checkpoint Experiment

- Application: Simple Heat Distribution solver
- Combination Kokkos (OpenMP) + MPI
- Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz
- 2 sockets, 32 threads per socket
- 62 threads for HeatDis, 1 thread for VeloC backend
- Two experiments: 4GB/node and 2GB/node
- 600 total time steps

HeatDis1 Results

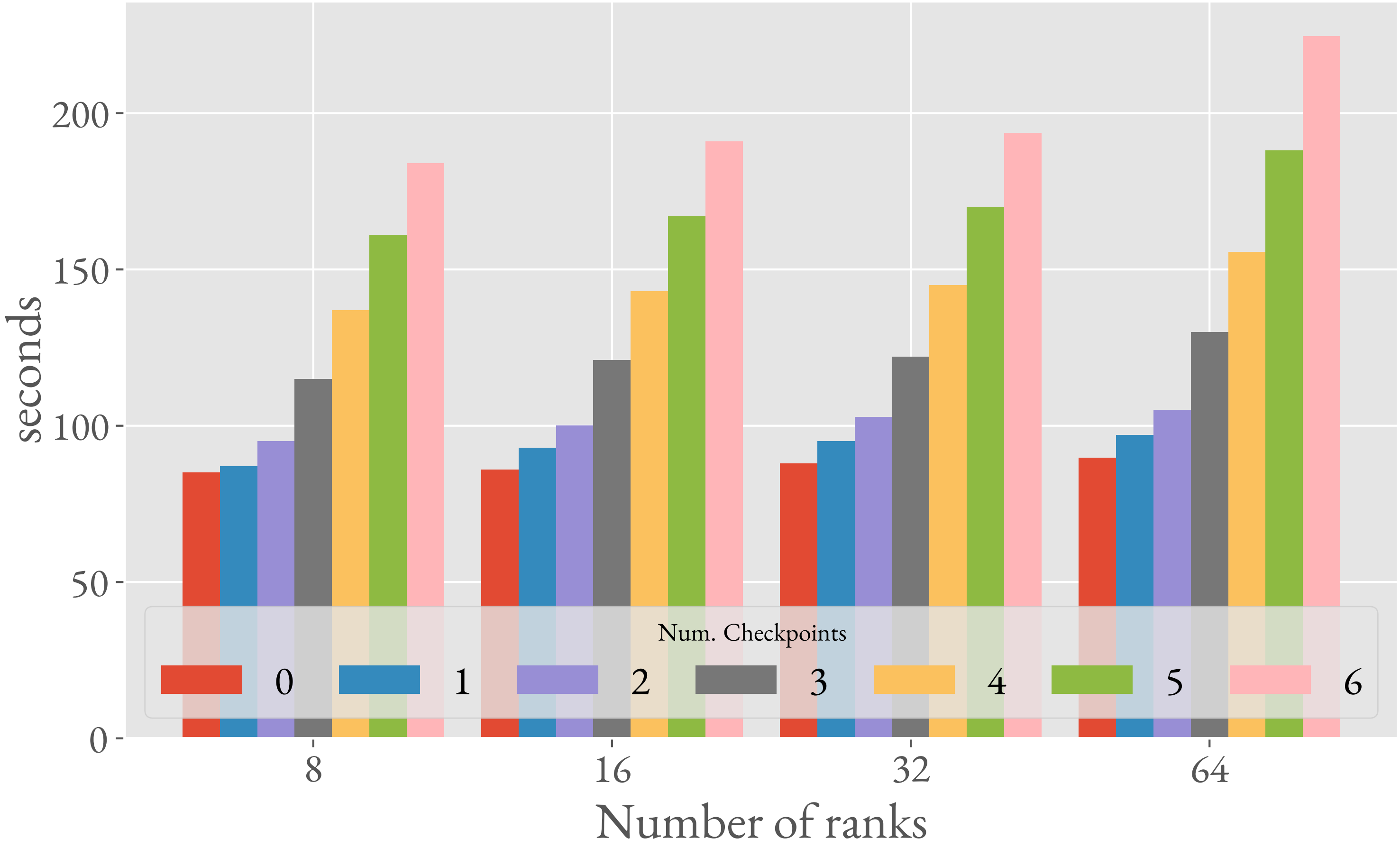


Figure: Weak scaling with automatic checkpointing in the heatdis app at 4GB/node.

HeatDis2 Results

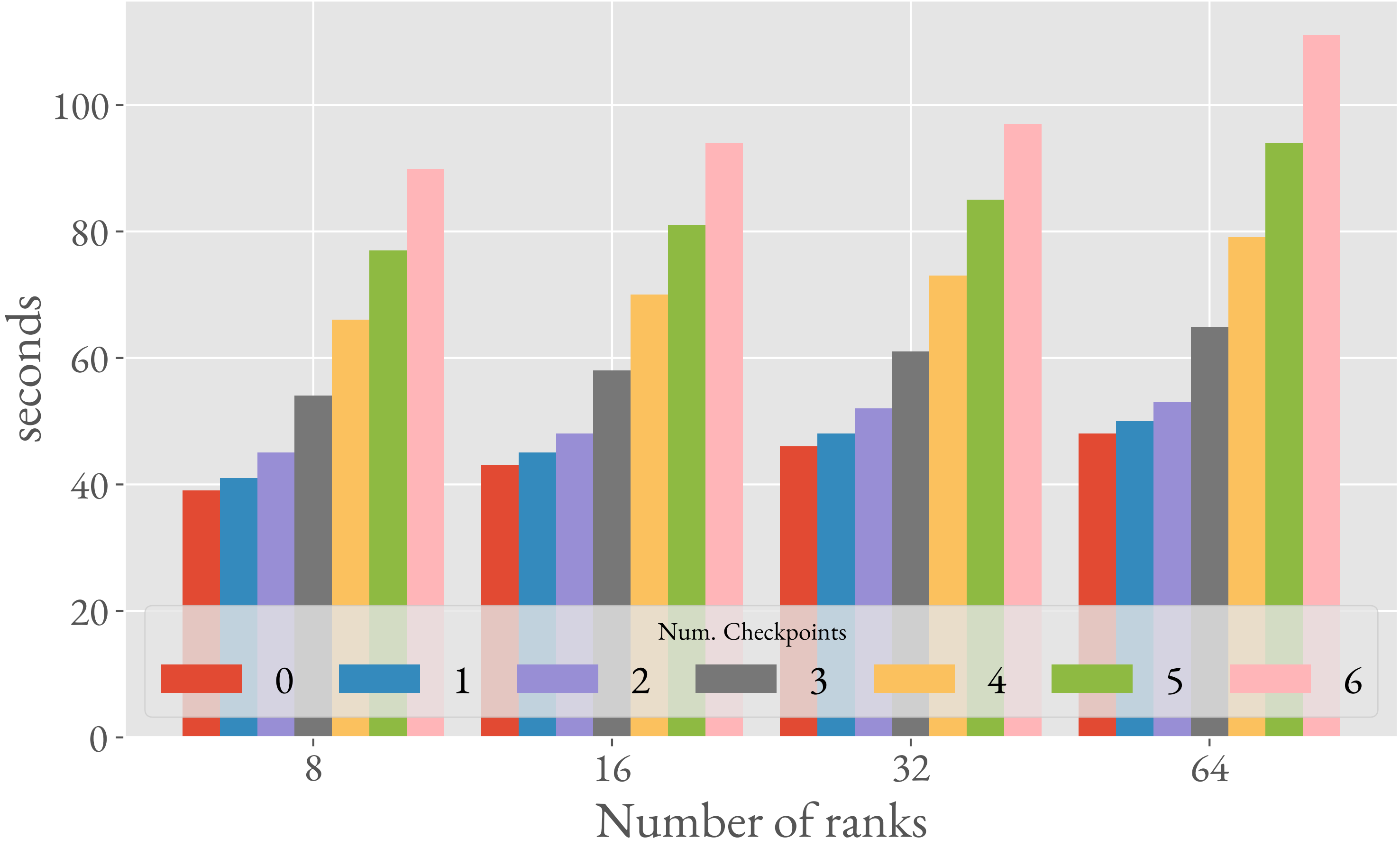


Figure: Weak scaling with automatic checkpointing in the heatdis app at 2GB/node.

HeatDis Overhead

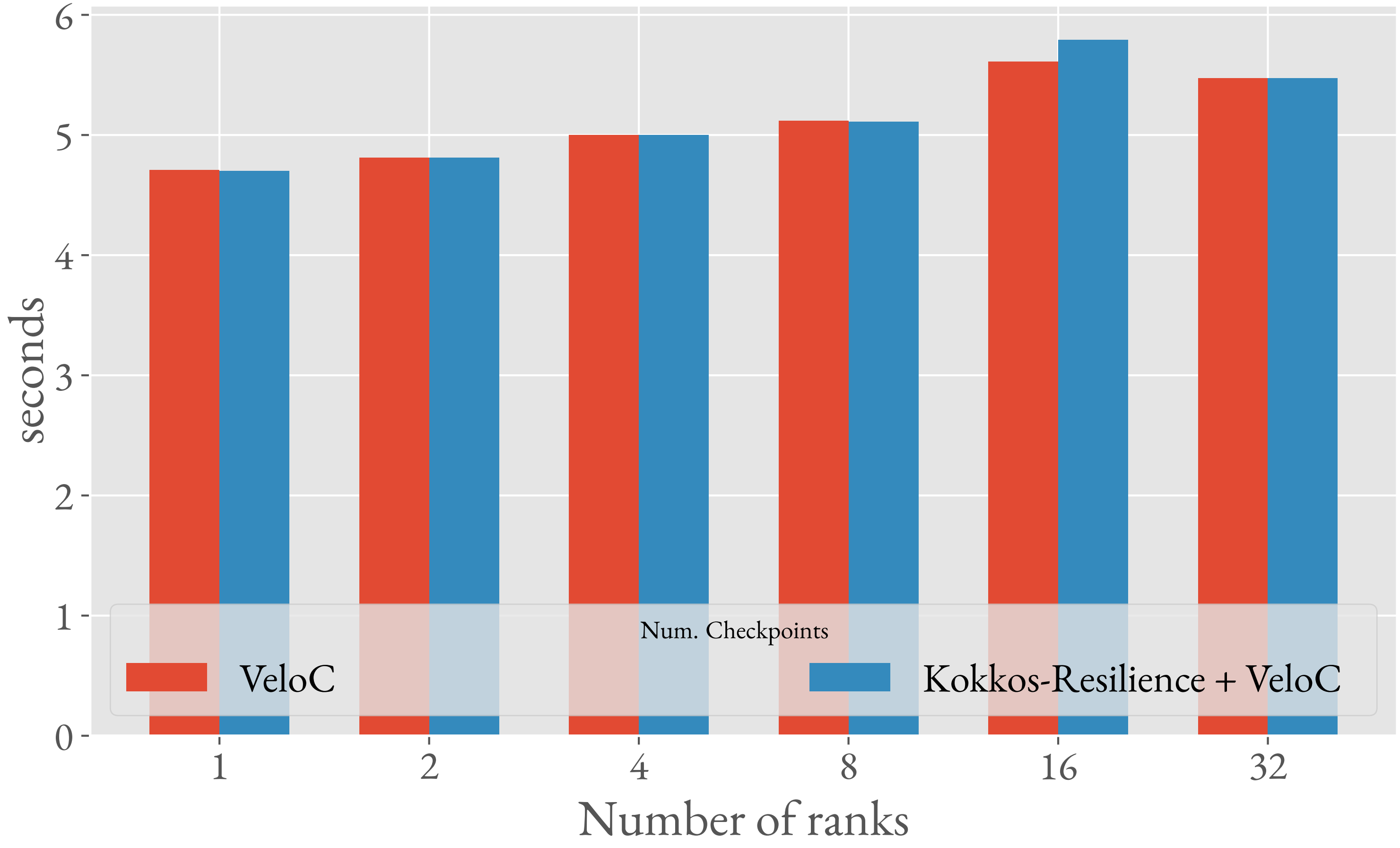


Figure: Performance overhead of Kokkos Resilience is negligible. The total time is equivalent to VeloC on its own.

Resilient Execution Spaces Experiment

- Application: MiniMD molecular dynamics
- Kokkos (CUDA) single node
- Multiple GPU architectures: Kepler, Pascal, Volta
- Dual socket 28 core Intel Xeon E5-2683v3 with 256 GB RAM

Resilient Execution Spaces Result

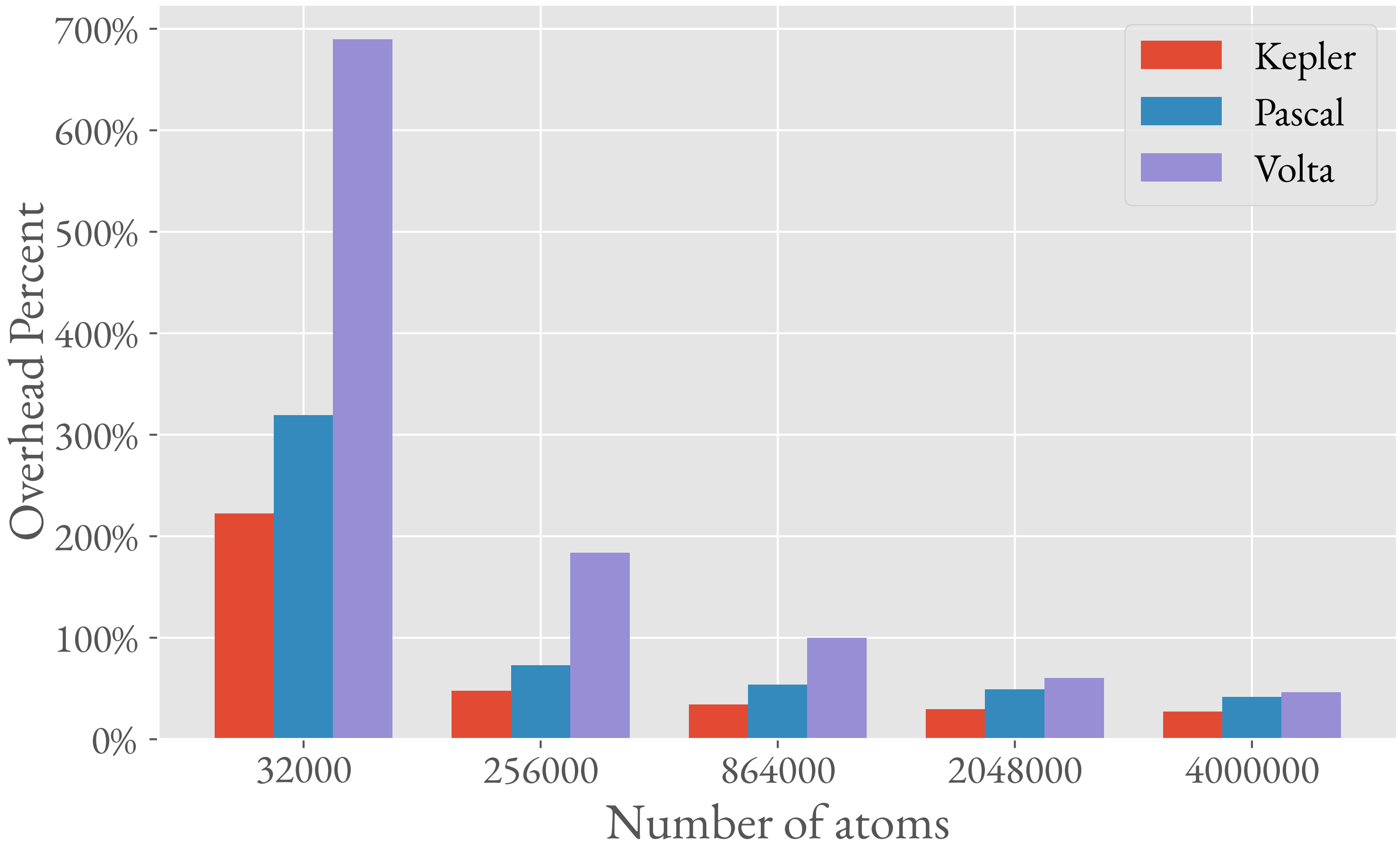


Figure: Increasing the problem size for MiniMD and resilient execution. At small problem sizes, launch time dominates. Concurrency is implemented using CUDA streams.

Conclusion

- As we start using exascale clusters
    - More and more applications will use performance portable abstractions
    - MTBF decreases
- Resilient Kokkos provides an extension to Kokkos that enables easy, efficient, and reliable resilience for applications
    - Manual checkpoint allows for user control while abstracting data movement through Kokkos memory spaces
    - Automatic checkpointing allows applications to use resilient backends like VeloC with minimal effort
    - Resilient execution spaces allow applications to easily make kernels resilient

Future Work

- The lambda capture trick for automatic checkpointing has limitations
  - References involve some manual tracking
  - Compilers already have data flow analysis – could this be used for a compiler plugin that can be used to determine what data is referenced?
- HPX and HCLib backends for resilience under development
- MPI ULFM support for failure detection and recovery (via Fenix library)
- Resilient execution spaces for OpenMP

Bonus Slide: ResilientRef

```
struct mixed_data
{
  mixed_data()
    : x( "test", 5 ), y( false )
  {}

  Kokkos::View< double * > x;
  bool y;

  KOKKOS_INLINE_FUNCTION void work() { /* ... */ };
};
```

Listing: Mixed data structure that can't be captured under reference semantics.

- How does Automatic checkpointing work with references to mixed data structures or pointers to `this`?
- Capture by ref wouldn't trigger copy constructor for the View
- Capture by value would incorrectly copy non-view fields

Bonus Slide: ResilientRef

```
auto dat = mixed_data( /*...*/ );
auto ref = KokkosResilience::Ref< mixed_data >( dat );

KokkosResilience::checkpoint( *ctx, "chk", 0, [ref]() {
  ref.y = true;
  Kokkos::parallel_for( dim0, KOKKOS_LAMBDA( int i ) {
    ref.work();
  } );
} );
```

Listing: Example of using the ResilientRef API

- Class `Ref` acts as a checkpoint-aware wrapper for references to mixed data
- Capture by value triggers copy constructors of held data, but keeps original references.