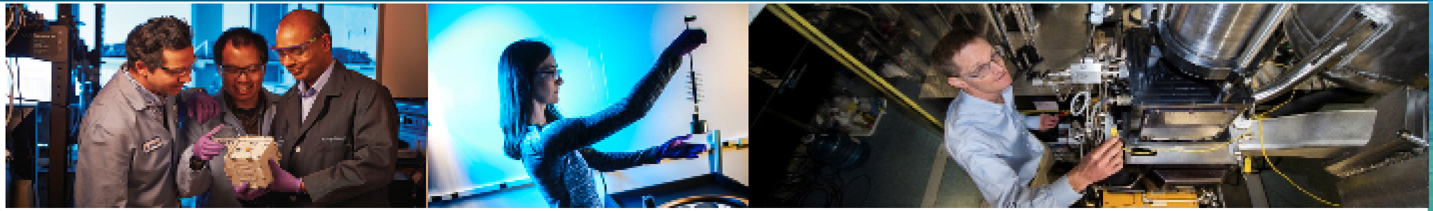
SAND2022-2116C

Online Fault Tolerance with Kokkos Resilient Execution Spaces

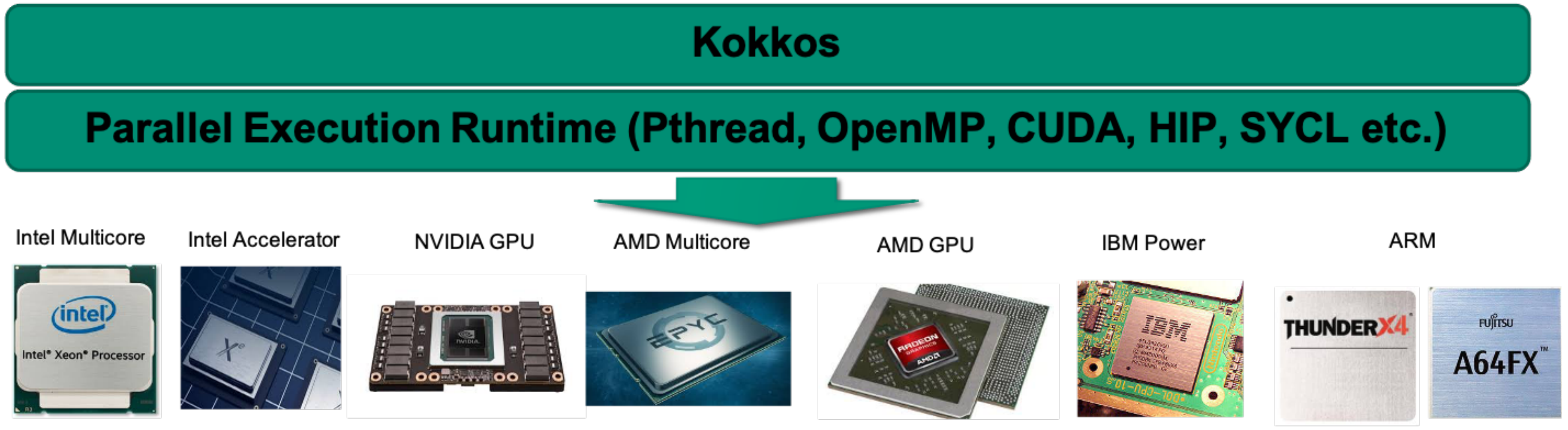


Elisabeth Giem, Nicolas Morales, Keita Teranishi, Matthew Whitlock

PRESENTED BY

Elisabeth Giem

Increasing Heterogeneity at Extreme Scale



- Increasing heterogeneity at extreme scale: macro- and micro-architectures
- Different parallel runtime expertise required for maximum performance
- *Performance portability* as opposed to portability

Kokkos

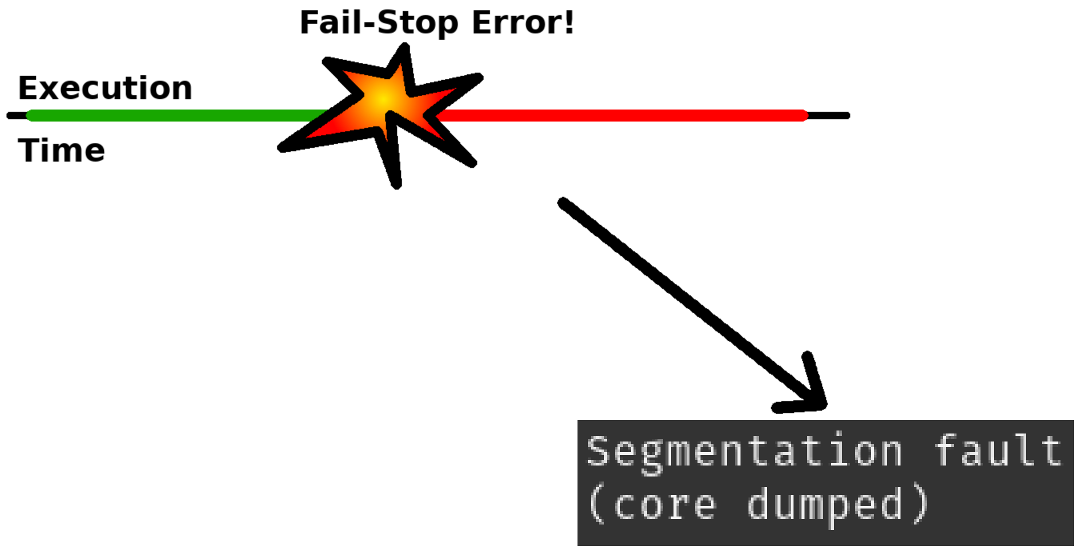
Kokkos is a performance portable parallel programming model

- C ++library, modern C++ compliant (requires C++ 14, soon will require C++ 17)
- No new language extensions
- Scalable
- Write algorithm using Kokkos and C++, run on many architectures
  - CPU, GPU
  - plus more, with other backends in development

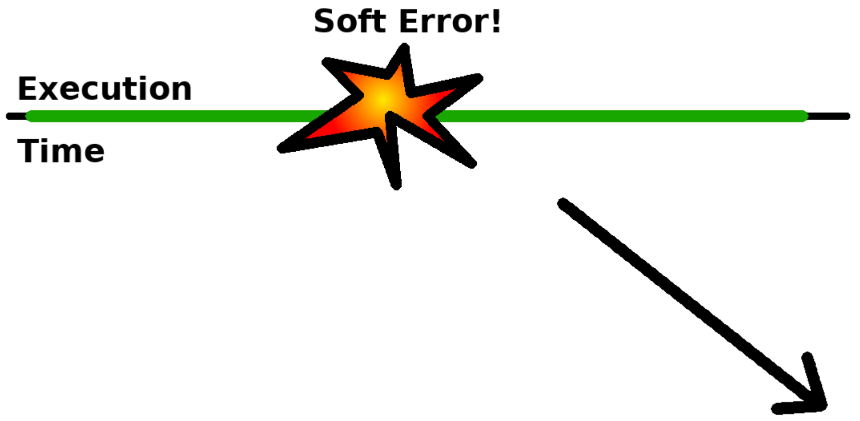
C++ is a popular scientific programming language, and Kokkos is highly versatile and widely used, a good choice for a resilient performance portable parallel programming model

Transient Faults

There are two types of faults:



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



$1+1 = 3$

- Fail-stop errors are easy to detect, the program crashes
- Often handled by checkpointing and restart
- Out of scope for this talk, interested in fail-continue (soft) errors
- Observed fail-continue errors: lock semantics, encryption/decryption, database index corruption, and more

Resiliency

Resiliency is the ability to tolerate errors in computing or memory.
Identified by the DOE as a top-10 challenge in future exascale computing, and by industry:

- Blue Waters—NCSA: Cray machine with unexpected distribution of faults and inadequate failover mechanisms
- Titan—ORNL: 18k GPUs requiring 11k GPUs replaced because of faults due to aging
- Facebook—18-month fleet study, soft errors a systemic issue occurring orders of magnitude more frequently than FIT simulation estimates
- Google—Fleet study, “Mercurial Cores” generating soft errors, cannot be fixed with updates and require resiliency

Our work is to create a resilient execution space for Kokkos, so that algorithms written using Kokkos can handle soft errors during computing

Parallel Patterns

Parallel Patterns

- Parallel patterns execute in a given parallel execution space
- Our research is focused on making these patterns resilient
- Three main patterns:
  - `parallel_for`
  - `parallel_reduce`
  - `parallel_scan`
- Accompanied by a policy and body

Serial

```
std::vector<double> X(N);
for (size_t i = 0; i < N; ++i)
{
  X[i] = …
}
```

OpenMP

```
std::vector<double> X(N);
#pragma omp parallel for
for (size_t i = 0; i < N; ++i)
{
  X[i] = …
}
```

Kokkos

```
Kokkos::View<double *> X(N);
parallel_for (  N,  [=]  (const size_t i)
{
  X(i) = …
});
```

Parallel Policies

Parallel Policies

- Execution policy informs pattern how to iterate
- Many policy types:
  - RangePolicy (1D range)
  - MDRangePolicy (multidimensional tiling)
  - Teams (hierarchical parallelism)
  - Others

Bodies and Work Items

Bodies and Work Items

- Each iteration of a computational body is a work item
- A total amount of work items are given to Kokkos patterns as a functor
- For ease of use, can be represented as lambda capture
- If using a lambda capture, must be by value and not by reference

```
Kokkos::View<double *> X(N);
parallel_for ( N, [=] (const size_t i)
{
  X(i) = …
});
```

Kokkos Views

`Kokkos::View`

- Multi-dimensional array with architecture-dependent layouts
- Takes care of memory management
  - Caching on CPU
  - Coalescing on GPU
  - Etc.
- User only needs to know minimal specifics of architecture
- Same API regardless of hardware

```
View < double*, LayoutRight, HostSpace > X(N);
parallel_for ( RangePolicy < OpenMP > (0, N), [=] (int i) {
  X(i) = ...
});
```

Code example courtesy Carter Edward

Resilient Execution Spaces

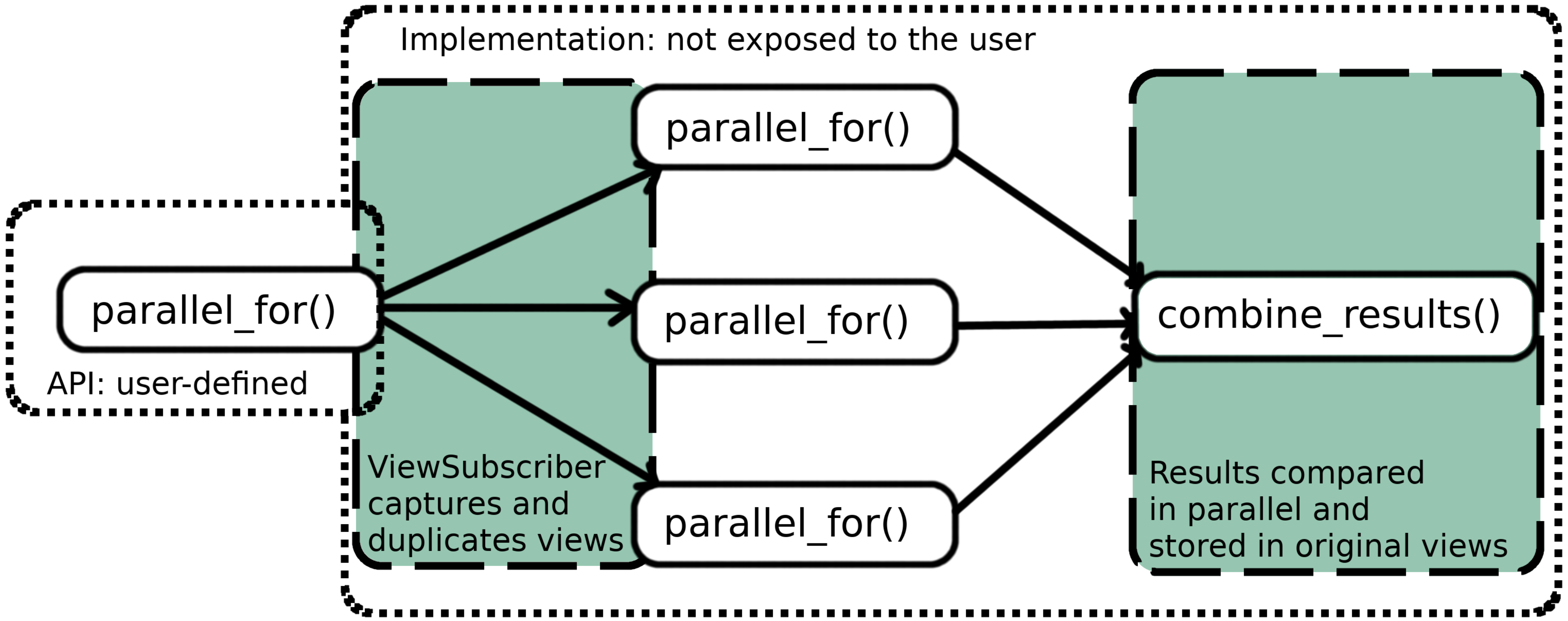
```
ResilientView < double*, LayoutRight, ResHostSpace > X(N);
parallel_for ( RangePolicy < ResOpenMP > (0, N), [=] (int i) {
  X(i) = ...
});
```

We introduce a natural extension to execution spaces: the resilient execution space.

- Views are replicated, and then patterns are executed concurrently on the replicated views
- Simple majority voting system: three executions, if two agree, the result is considered correct and the application progresses
- Voting step proceeds in parallel after triplicate execution completed

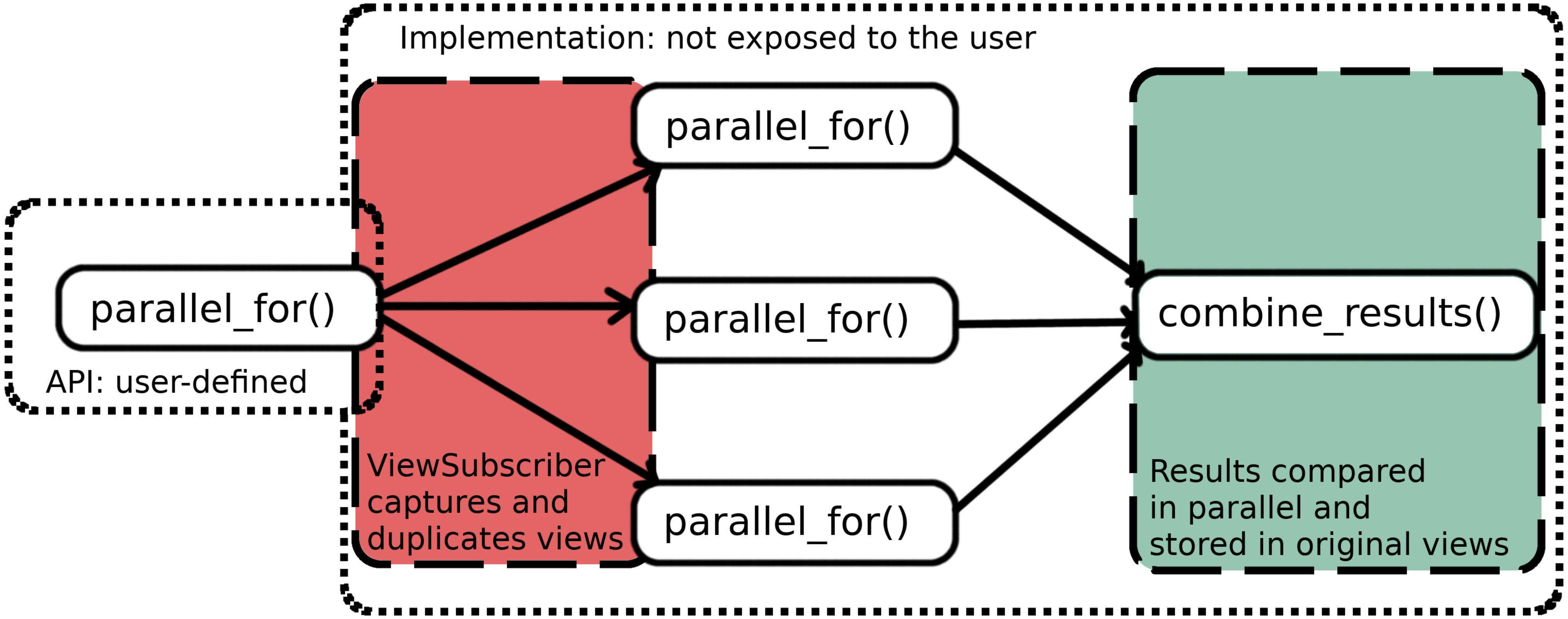
Control Flow

Resilient execution space with triple modular redundancy


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

Control Flow: View Duplication

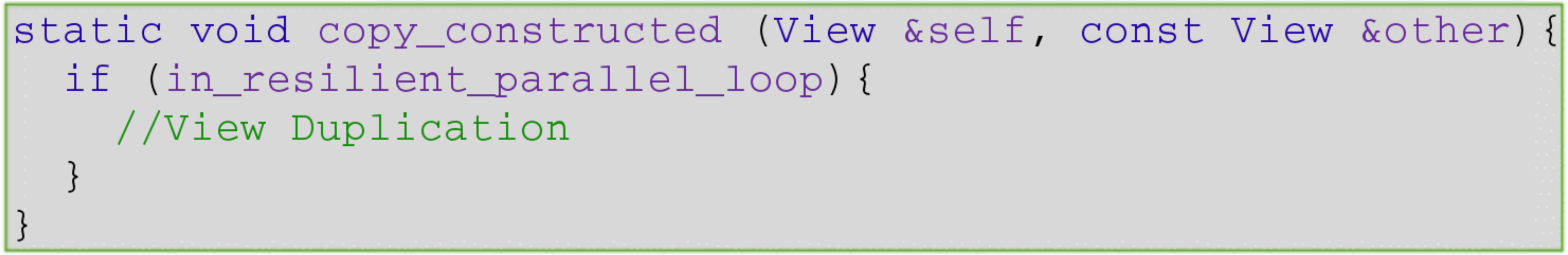
Resilient execution space: View duplication


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

View Duplication

How is view duplication actually achieved? The easy ResilientView API presented to the user covers a ViewSubscriber added to the original Kokkos View.

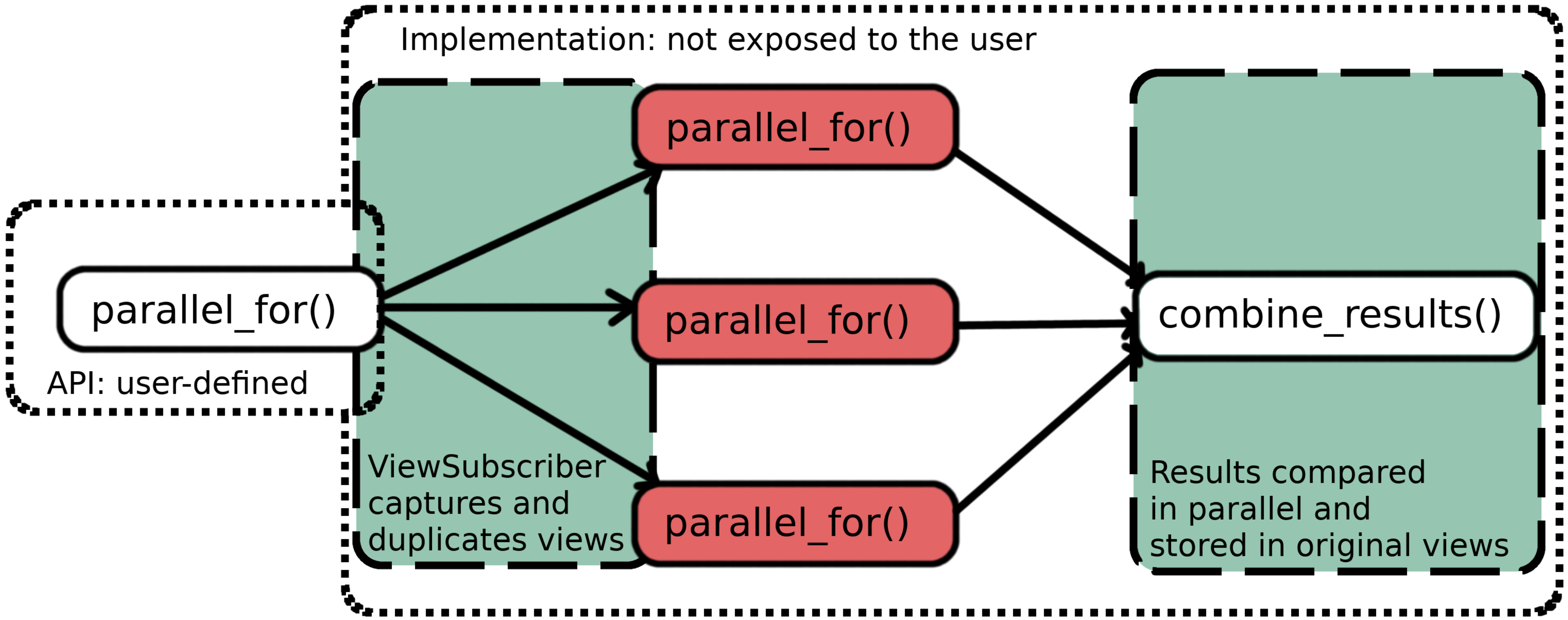
- Upcoming experimental feature in Kokkos
- Compile time callback
- Copy constructor invokes view duplicator
- Directly invokes view duplicator, not a dynamic dispatch
- Zero overhead in terms of actually invoking code



```
static void copy_constructed (View &self, const View &other){
  if (in_resilient_parallel_loop){
    //View Duplication
  }
}
```

Control Flow: Triple Execution

Resilient execution space: Triple execution


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

Concurrent Execution

Once the views are duplicated, three executions must then proceed. Two obvious desirable characteristics for these executions: Asynchronous operation and concurrent execution.

Asynchronous operation is easily achieved:

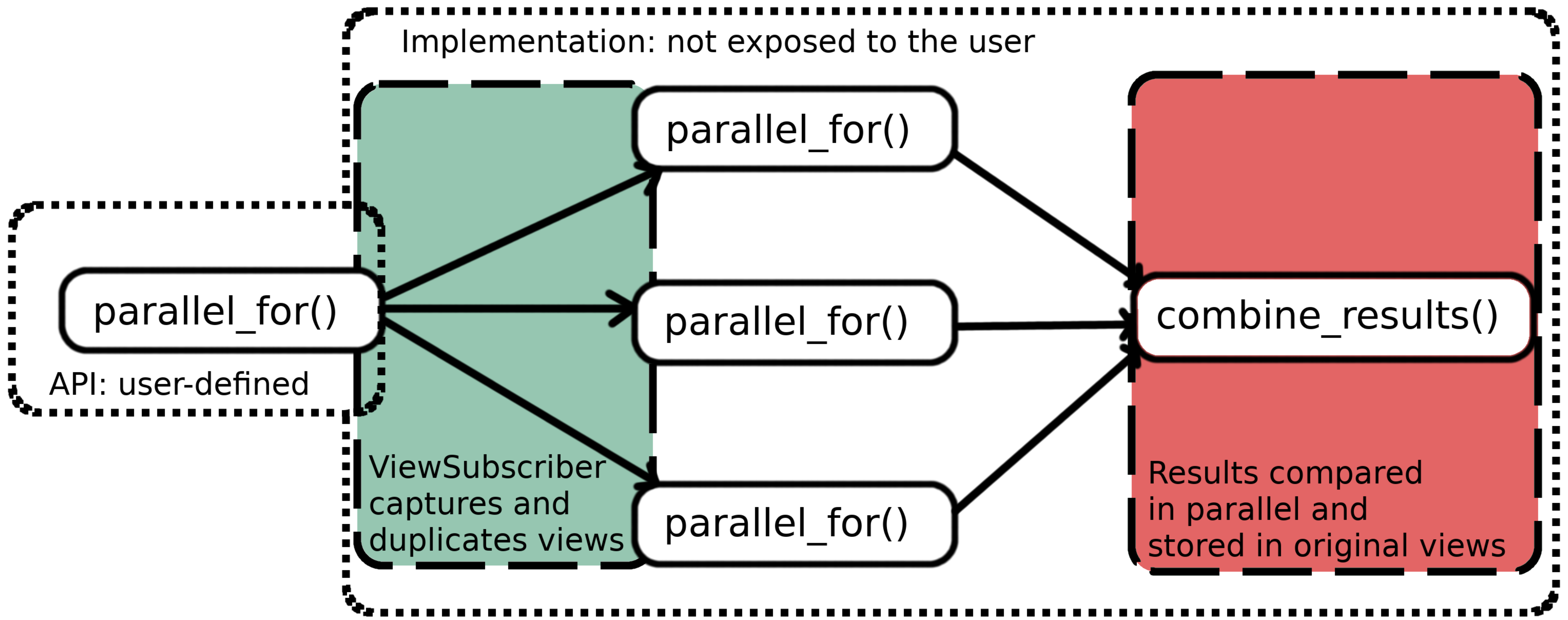
- Independent work items are required for the pattern in the first place to execute correctly
- The only synchronization which must be performed is at the end of execution: this is explicitly called

Concurrent execution requires more thought:

- Scheduling achieved by more evenly balancing work using smart distribution
- Lambda captured functor is triplicated
- Highly tailored: every pattern and every policy requires a different implementation

Control Flow: Duplicate Resolution

Resilient execution space: Duplicate resolution


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

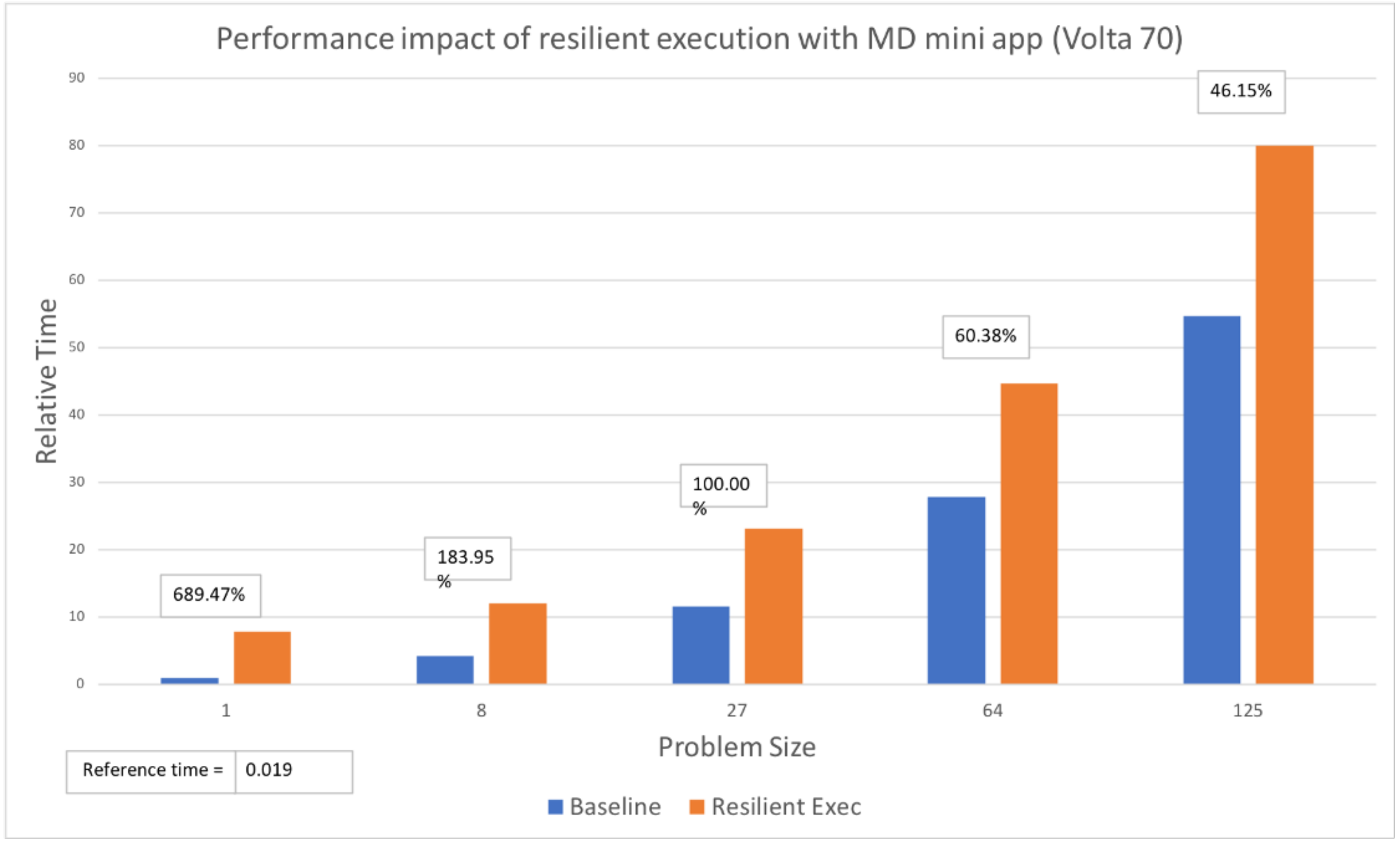
Detecting Soft Errors

Control is handed from the pattern to the duplicate combiner, which invokes the view subscriber for duplicate resolution.
View subscriber requires datatype at compile time to perform equality check between duplicated views:

- $a = b$ for integral types
- $|a - b| < \epsilon$ for floating point types
- Check can be easily customized

If no resolution is reached, the combiner hands control back to the pattern and re-execution is attempted a user-specified number of times.
If no resolution is reachable, an exception is thrown.
All checks are done in parallel.

CUDA Result



Performance impact of resilient execution with MD mini app (Volta 70)
Relative Time
0
10
20
30
40
50
60
70
80
90
689.47%
183.95 %
100.00 %
60.38%
46.15%
1
8
27
64
125
Problem Size
Reference time = 0.019
Baseline
Resilient Exec

Conclusions and Future Work

- We have successfully created a prototype resilient performance portable parallel programming model by adding triple modular redundancy to Kokkos execution spaces
- Results show that overhead is less than might be expected, but larger studies are needed
- Future steps include more applications and different types of execution spaces and further polices to adapt
- We also want to explore using execution spaces in concert with checkpointing

Thank you! Questions?

This work was sponsored in part by NICE: NRT for Integrated Computational Entomology, US NSF award 1631776.