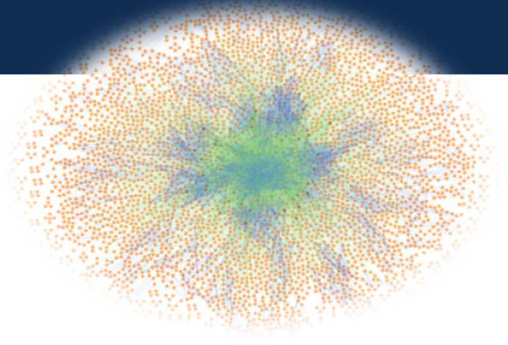
This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

Exceptional service in the national interest

Sandia National Laboratories

SAND2019-6329C



Programming Model Tradeoffs for Global vs Local Recovery: Algorithm Based Fault Tolerance

Hemanth Kolla, Keita Teranishi, Jackson Mayo, Maher Salloum, Rob Armstrong

Sandia National Laboratories, California, USA

U.S. DEPARTMENT OF ENERGY
NNSA National Nuclear Security Administration

Sandia National Laboratories is a multi-program 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-NA-0003525

Outline: Talk of Two Halves

1. Programming models for scalable resilience.

2. Algorithm Based Fault Tolerance: Global vs Local recovery.

Outline

1. Programming models for scalable resilience.
2. Algorithm Based Fault Tolerance: Global vs Local recovery.

Motivations and Background

Sandia National Laboratories

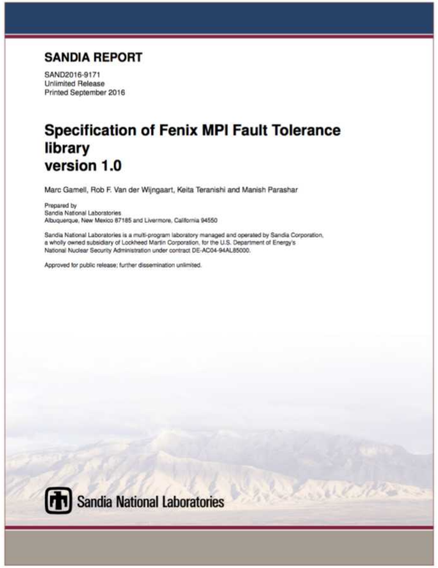
- Reliability has become a major concern of large scale computing systems
  - Complexity of hardware and software (number of components)
  - Overhead for reliability enhancement (20% penalty in power and performance)
  - Performance variability across cores, accelerators and nodes
- System level approach is expensive and C/R cannot resolve all resilience issues.
- Need better programming model support
  - Extension of Fault Tolerant MPI Proposal (Fenix)
  - On-node parallel computing
    - Asynchronous Many Task (AMT)
    - Resilience Extension of Kokkos
  - Distributed (AMT)

Fenix: Extending MPI-ULFM

- MPI-ULFM has been proposed for the fault tolerance APIs of the MPI standard (hard errors and failures)
- Survived processes continues after MPI rank failures
- New MPI functions for fixing MPI communicator
  - MPI_Comm_agree --- Sanity check (resilient collective)
  - MPI_Comm_revoke --- Invalidate MPI Communicator
  - MPI_Comm_shrink --- Fix MPI Communicator removing dead process
- User is responsible for the recovery after MPI_Comm_shrink

Fenix:

- Fault Tolerant Programming Framework for MPI Applications
  - Separation between process and data recovery
    - Allows third party software for data recovery
    - Multiple Execution Models
  - **Process recovery**
    - Extend **MPI-ULFM**
    - Process recovery through **hot spare process pool**
    - Process failure is checked at **PMPI layer** and recovery happens automatically under the cover
  - **Data recovery**
    - In-memory data redundancy
    - Multi-versioning (similar to GVR by U Chicago & ANL)


SANDIA REPORT

SAND2016-9171
Unlimited Release
Printed September 2016

Specification of Fenix MPI Fault Tolerance library version 1.0

Marc Gamell, Rob F. Van der Wijngaart, Keita Teranishi and Manish Parashar

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Fenix – Process Recovery Interface

```
void Fenix_Init (MPI_Comm comm,
                 MPI_Comm *newcomm,
                 int *role,
              int *argc, int ***argv,
              int num_spare_ranks,
              int spawn,
                MPI_Info,
              int *error);
```

If **newcomm** is NULL, Fenix tacitly replaces **comm** everywhere with resilient communicator

role:
- FENIX_ROLE_INITIAL_RANK
- FENIX_ROLE_RECOVERED_RANK
- FENIX_ROLE_SURVIVOR_RANK

spawn:
- 0:NO_SPAWN
- 1:SPAWN

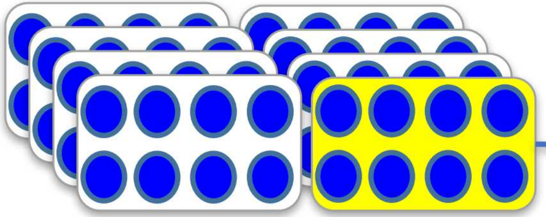
App should use **resilient communicator** (newcomm) instead of comm

Process failure triggers process recovery and long-jump to Fenix_init

```
void Fenix_Finalize ( );
```

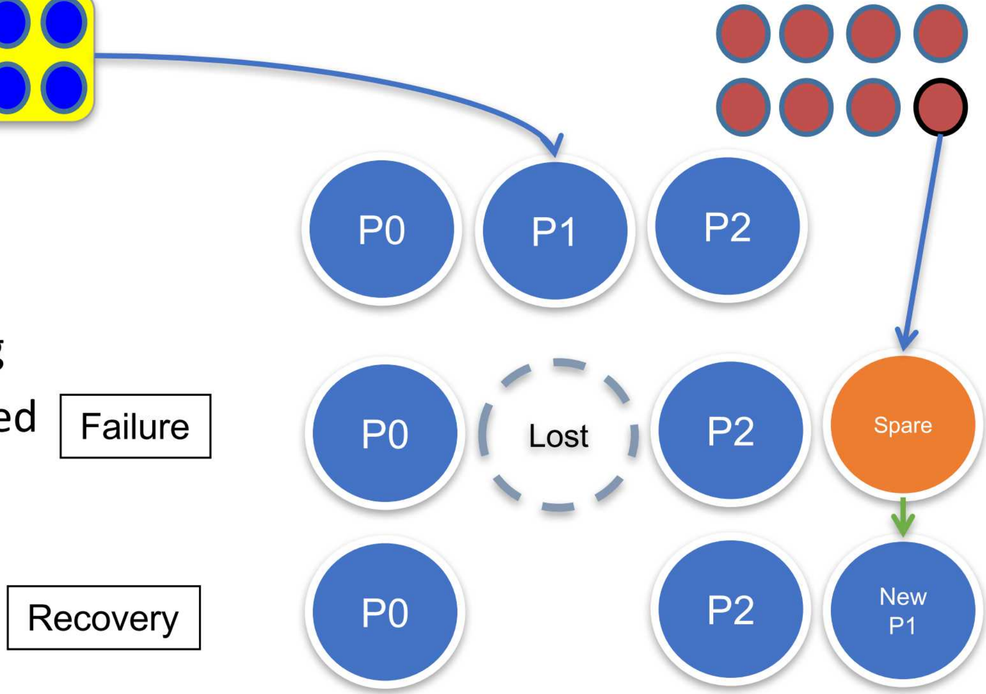
Non-Shrinking Model (with spare processes)

Compute Processes



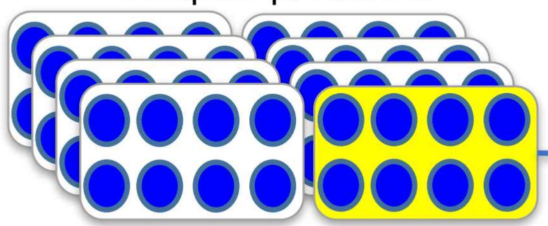
Spare processes
(allocated at job launch)

- Fenix automatically corrects rank-ordering
- Spare ranks are ordered to fill the lost ranks

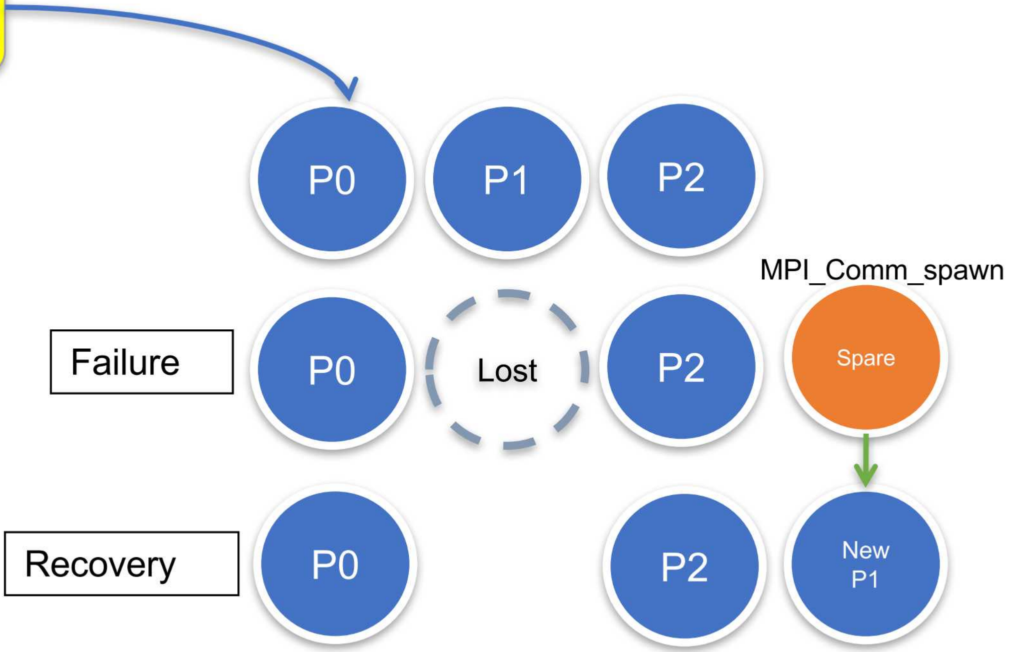


Non-Shrinking Model (Spawn)

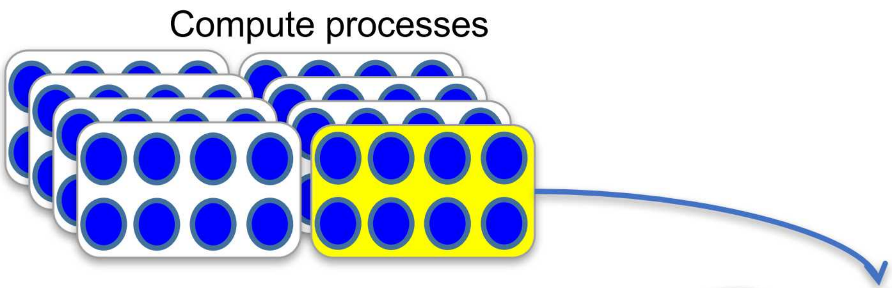
Sandia National Laboratories



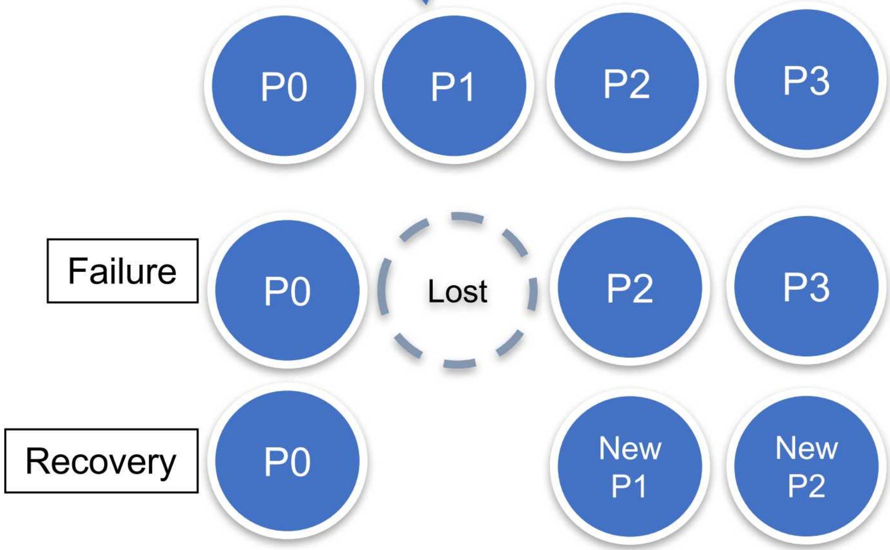
- Fenix automatically correct rank-ordering
- Spare ranks are order to fill the lost ranks
- Depends on the support of MPI_Comm_spawn



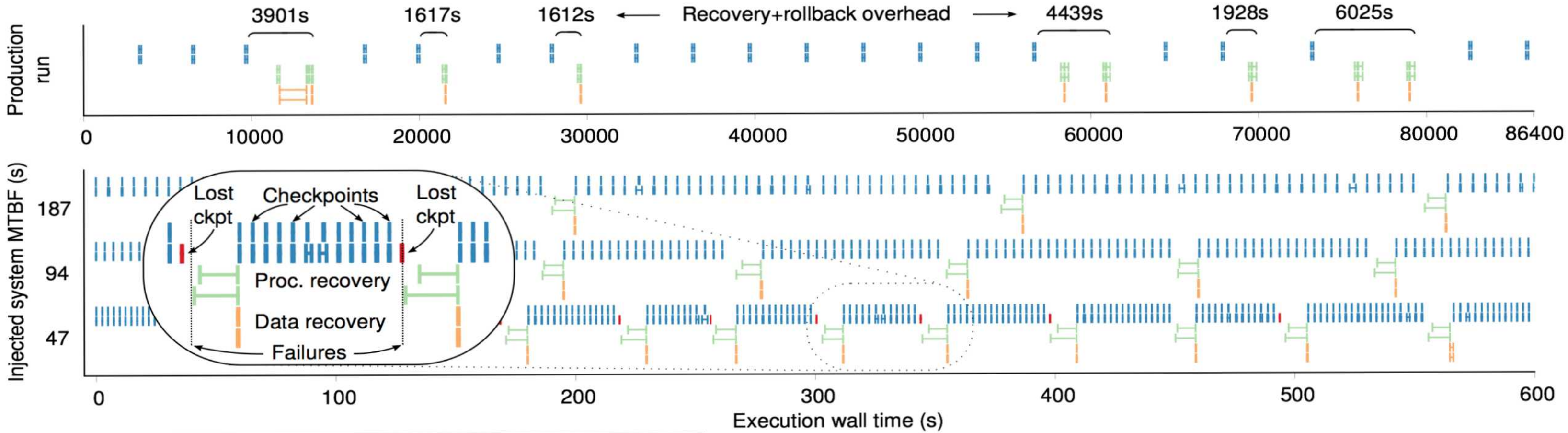
Shrink Model



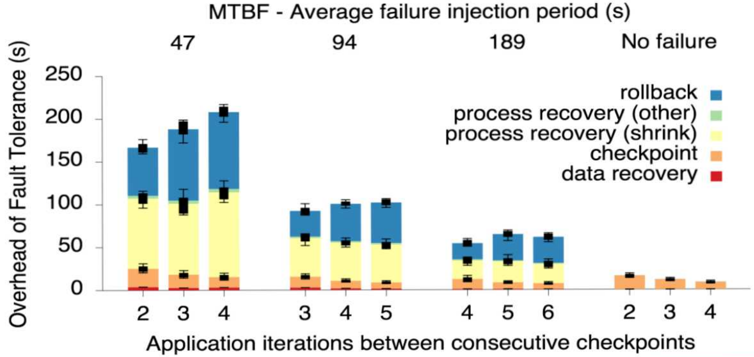
- Rank ordering after the failure is determined by MPI-ULFM.
- Fenix returns the program to the beginning.
  - User is responsible for reconstruct the state



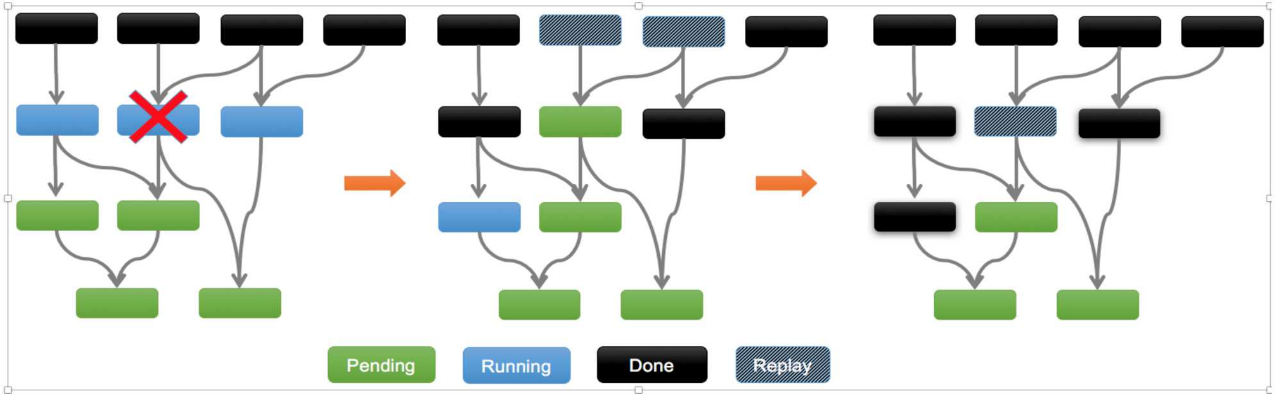
FENIX can recover from **frequent failures**



- Online recovery allows the usage of in-memory checkpointing, O(1s).
- Efficient recovery from high frequency node failures, as exascale compels.
- **With failures injected every 189, 94 and 47 seconds,** the total job run-time penalty is as low as **10%, 15% and 31%, respectively**.
- This can dramatically improve by optimizing ULFM shrink.



Node Level Parallel Programming Model



- Abstraction of computation and data objects allows automatic resilience support
  - Runtime scheduler orchestrates computations encapsulated by Task and parallel_for
  - Data abstractions to describe dependencies, data layout and access patterns (Read/Write/RW)
- Simple extension to the existing API provides knobs to the users to selectively apply resilience

Resilient AMT Prototype

- Resilience Extension of Habanero C++
  - AMT programming Interface by Vivek Sarkar
- Simple extension allows the user to introduce 3 major resilient program execution patterns
  - Task Replication Interface
  - Task Replay Interface
  - ABFT Interface

Original Task Launch

```
hclib::async_await ( lambda,
hclib_future_t *f1, ..,
hclib_future_t *f4);
```

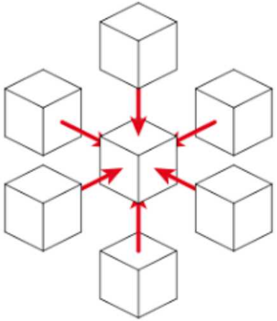
Task Launch with Replication

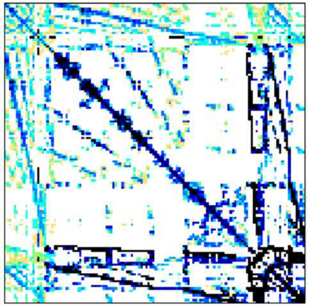
```
diamond::async_await_check<N> (
lambda, hclib::promise<int>
out, hclib_future_t *f1, ..,
hclib_future_t *f4);
```

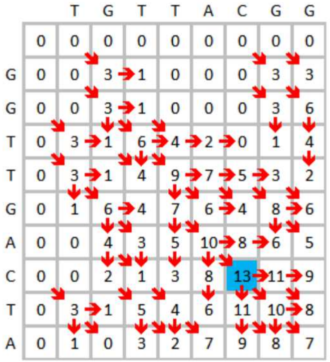
Task Launch with Replay

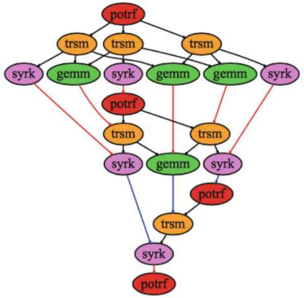
```
replay::async_await_check<N>(
lambda, hclib::promise<int>
out, std::function<int(void*)>
error_check_fn, void * params,
hclib_future_t *f1, .. ,
hclib_future_t *f4);
```

Performance









- On 2 Haswell CPU node (16x2 cores)
- 1D and 3D stencil code
- Conjugate Gradient with crank_1 sparse matrix
- Smith-Waterman (SW) algorithms
- Task-parallel Fault-Tolerant Cholesky Factorization
  - Based on the Cao and Bosilca (IPDPS2016)
- The application data is **over-decomposed**.
  - 4 way for stencil and CG
  - 64x64 for SW and Cholesky

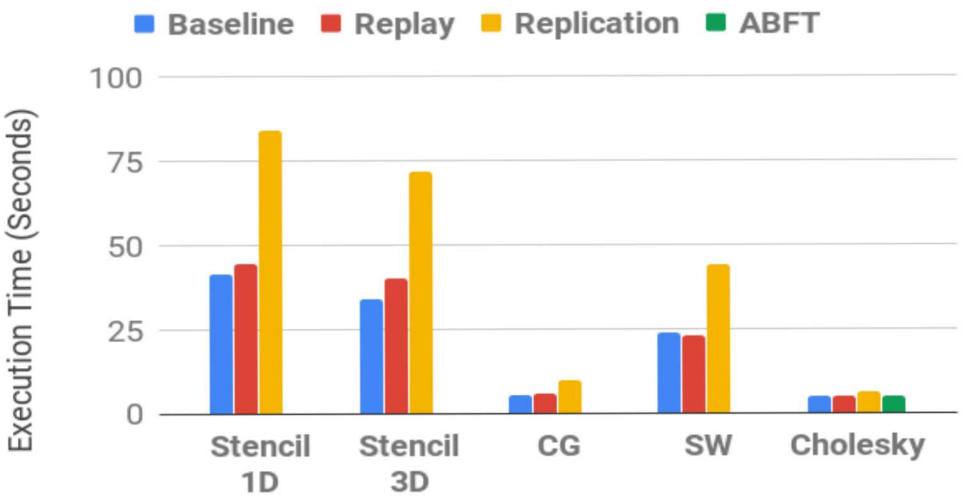
Replay and replication do not double the memory overhead

	Synthetic	Stencil 1D				
	vanilla	vanilla	Replay	Replication	Mix Replay	Mix Replication
1 worker	0.19 GB	0.67 GB	1.02 GB	0.98 GB	1.08 GB	1.05 GB
32 workers	6.19 GB	6.67 GB	7.02 GB	6.99 GB	7.08 GB	7.05 GB

- Synthetic benchmark just launch empty tasks iteratively
- Resilient 1D stencil code execute 128 tiles (16K points per tile) per iteration (**4 tasks per worker**)
- Executed 1M iterations
- Tested on NERSC’s Cori (2 Haswell CPUs, 32 cores total, 2.3GHZ) system

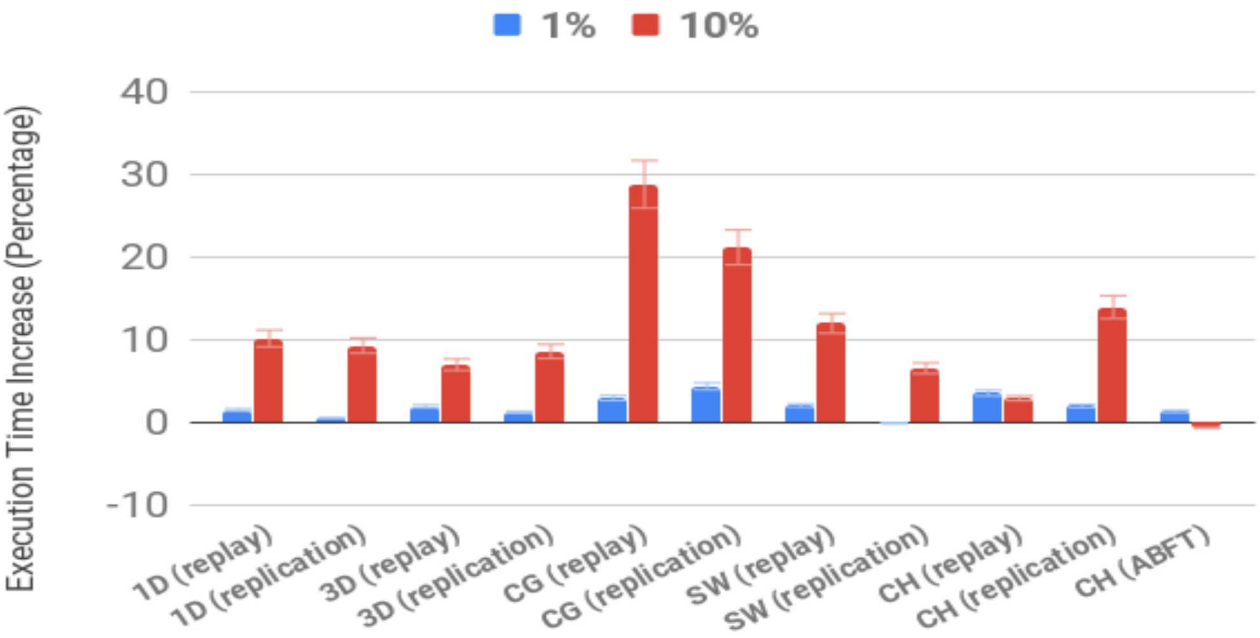
Performance without faults

Sandia National Laboratories

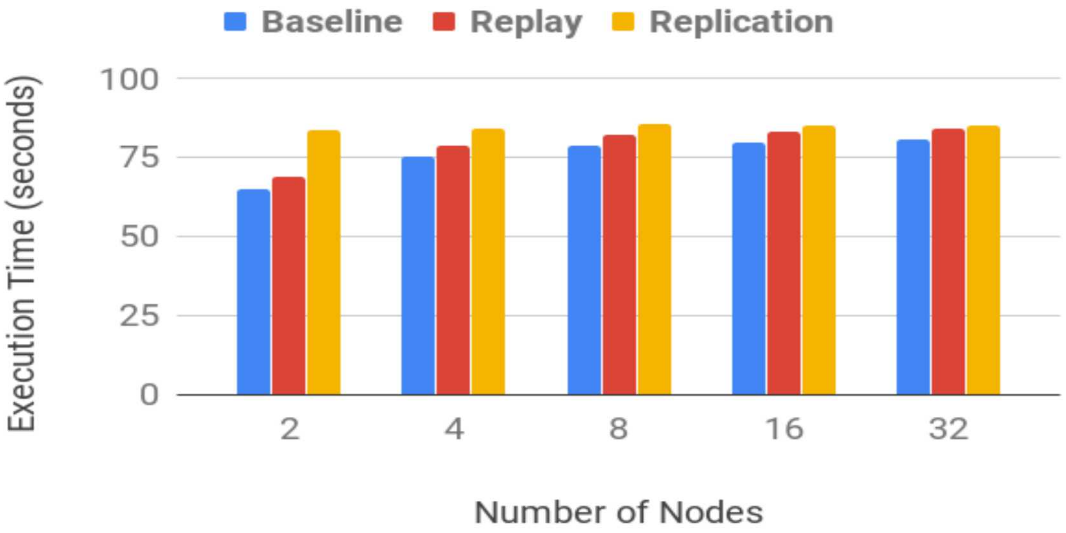


- Replication is expensive for 1D stencil, CG and SW.
- Observed some cache hits with 3D stencil
- High cache hits and critical path in task-base Cholesky suffers less replication overhead

Application delay is proportional to the # of failures

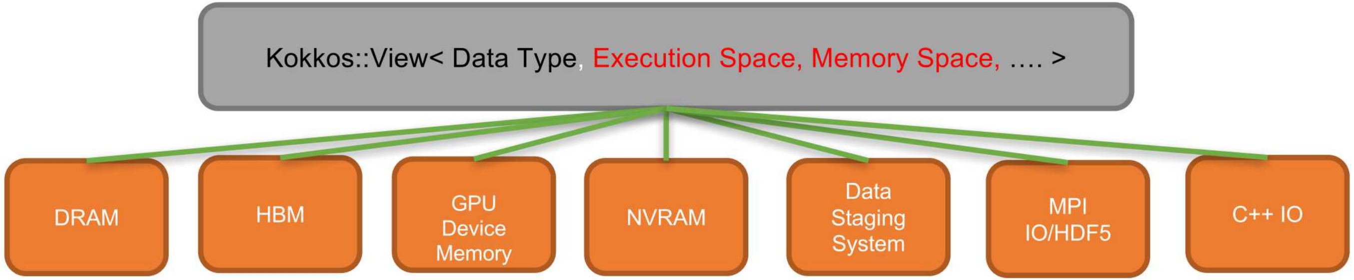


Scalability of 3D stencil code (MPI+Reslinet HCLIB)



- MPI-HCLIB implementation (1D, weak scaling, over-decomposed)
  - MPI (2-sided) calls are running on special worker (thread-funnel).
  - Preliminary results indicate replication overhead are masked by MPI overhead

Ongoing Work: Resilient Kokkos



- Kokkos provides abstraction of data and (on-node) parallel program execution
  - Kokkos::View provides an array with a variety of tunable parameters through template
  - **Execution Space** and **Memory Space** to provide performance portability over multiple node architecture
  - Exploit C++ Lambda to support parallel program execution
- Kokkos' abstraction to enable resilient parallel computation!
  - **Resilient Execution Space** for redundant program execution
  - **Resilient Memory Space** for checkpointing and data redundancy

Parallel Programing with Kokkos

Sandia National Laboratories

Serial

```
for (size_t i = 0; i < N; ++i)
{
  /* loop body */
}
```

OpenMP

```
#pragma omp parallel for
for (size_t i = 0; i < N; ++i)
{
  /* loop body */
}
```

Kokkos

```
parallel_for (( N, [=], (const size_t i)
{
  /* loop body */
});
```

- Provide parallel loop operations using C++ language features
- Conceptually, the usage is no more difficult than OpenMP. The annotations just go in different places.

Resilient Kokkos enables resilient data parallel computation with ease

Sandia National Laboratories

```
Kokkos::View<double *, ..., ResilientSpace> A(1000);
parallel_for( RangePolicy<>(0, 100), KOKKOS_LAMBDA(
const int i)
{
   A(i) = ... ;
});
```

Replication

```
parallel_for( RangePolicy<>(0, 100), KOKKOS_LAMBDA(
const int i)

{
  A(i) = ... ;

});
```

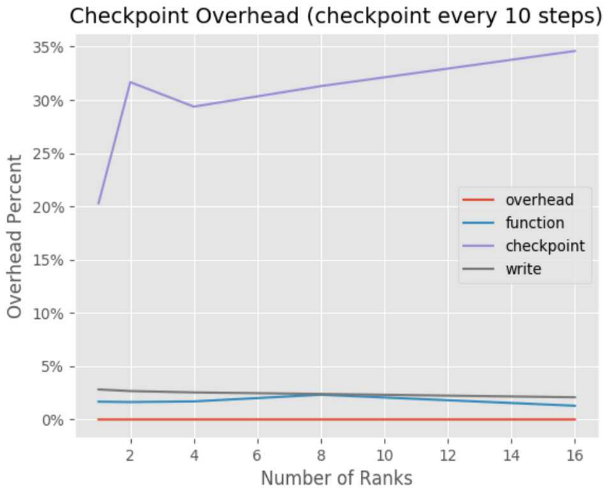
```
Kokkos::View< ... > a( "a", ... );
Kokkos::View< ... > b( "b", ... );
Kokkos::View< ... > c( "c", ... );

for ( int iter = 0; iter < 100; ++iter )
{
  // Will generate "compute_stuff/<view>.<iter>.bin" for all captured views
  Kokkos::checkpoint( "compute_stuff", iter, true, KOKKOS_LAMBDA {
    Kokkos::parallel_for( N, KOKKOS_LAMBDA( int i ) {
      // Some computation with a and b
    } );

    Kokkos::parallel_for( N, KOKKOS_LAMBDA( int i ) {
      // Some other computation with a and c
    } );
  } );
}
```

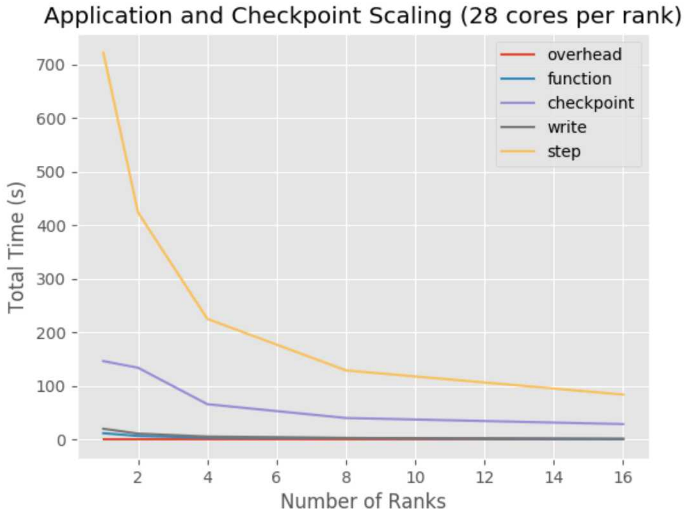
Lambda captures all Kokkos::View instances

Automatic Checkpointing

Checkpoint
"loop_1_A_B_C"

Performance of MiniMD





- Molecular Dynamics App: 32M atoms in 200x200x200 cells
- Strong scalability on 2CPUs/Node Haswell Cluster with FDR IB
- Checkpoint every 10 time steps
- Resilient Memory Space interfaced to VeloC (using the file-based checkpointing)
- **Negligible overhead for Kokkos runtime**

Outline

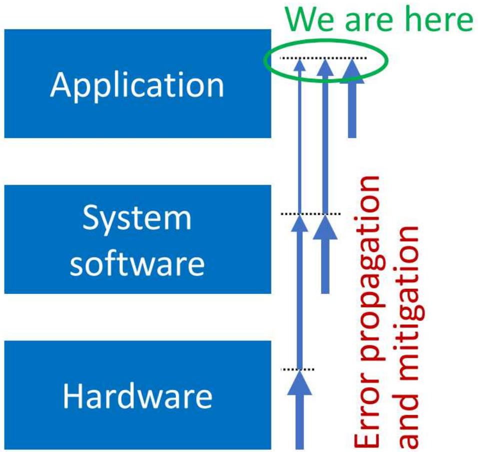
1. Programming models for scalable resilience.

2. Algorithm Based Fault Tolerance: Global vs Local recovery.

Algorithm Based Fault Tolerance (ABFT)

- Handling hard failures is not enough for resilience.
  - An error does not always cause a crash, but leads to a “wrong” answer.
  - Physics of the problem could be leveraged to detect an error, before wide spread failure.
- Application-level detection can be a powerful complement to resilient programming models

Levels:



Example uncertainties:

How solver apps will be (re)written (programming models); prevalence of unreproducible bugs

Ability of bugs here to corrupt the app state

Rates of (initially) undetected errors; costs of mitigating in hardware (higher voltage, redundancy, shielding)

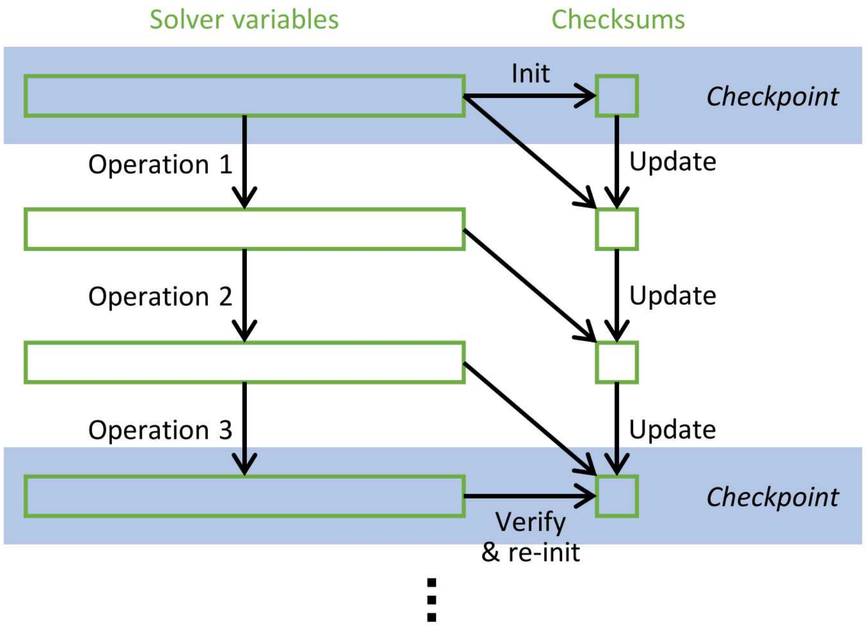
Generalizing ABFT: Physics Based Checksums

Sandia National Laboratories

- Leverage existing work on ABFT for linear algebra, but incorporate physics.
- **Key idea:** Focus only on detection of silent errors, treat them same as hard failures.
- Enabling assumptions:
  - Failures are locally rare (component MTBFs are long, if not system MTBFs).
  - Checkpoint/restart in some form will be used/available.
  - Keep resilience overhead small.
  - Applications are solving physics equations that satisfy conservation laws.
  - Silent hardware errors and software anomalies will typically violate conservation.

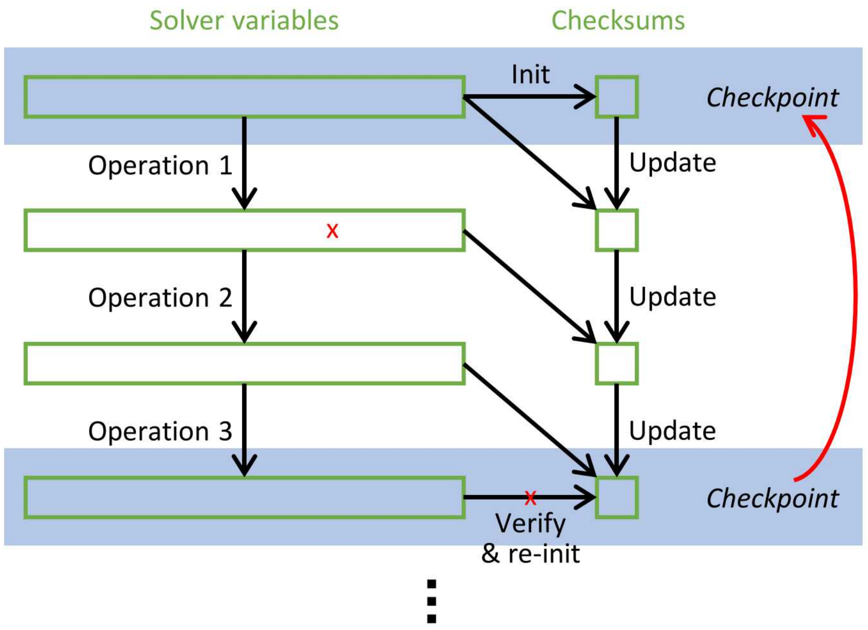
Checksums for Efficient Redundancy

- Introduce a smaller side computation that remains consistent with solver state if no errors.
- Verify consistency intermittently, just before each checkpoint



Checksums for Efficient Redundancy

- Introduce a smaller side computation that remains consistent with solver state if no errors.
- Verify consistency intermittently, just before each checkpoint



Checksums from Conservation Laws

- General local conservation law (density ρ, flux $\mathbf{J}$): $\frac{\partial \rho}{\partial t} = -\nabla \cdot \mathbf{J}$
- Conserved quantity in a region *R* (e.g., a computational subdomain): $Q(R) = \int_R dV\, \rho$
- Integrated conservation law: $\frac{dQ(R)}{dt} = -\oint_{\partial R} d\mathbf{S} \cdot \mathbf{J}$
- *Q(R)* changes only by flux through boundary, which is much faster to compute than *Q(R)* itself:
  - When discretized form of this conservation law holds, *Q(R)*, is a local physics-based checksum that can be updated efficiently and verified intermittently.
  - No global communication beyond what solver already performs; flux in each subdomain can be computed from data already being communicated between processes.

Application: 1D Linear Advection Equation

- 1D linear advection equation $\frac{\partial \phi}{\partial t} + \nu \frac{\partial \phi}{\partial x} = 0$
- Consider the Lax-Wendroff stencil (with $c = \nu \Delta t / \Delta x$):

$$\phi_j^{n+1} = \tfrac{1}{2} c(c+1)\phi_{j-1}^n + (1-c^2)\phi_j^n + \tfrac{1}{2} c(c-1)\phi_{j+1}^n$$

- The discretized conserved quantity on each local subdomain is $Q(\phi) = \sum_i \phi_i$
- The conserved quantity, checksum, can be updated independently of local state:

$$Q(\phi^{n+1}) = Q(\phi^n) + \frac{c(c+1)}{2}(\phi_{-1}^n - \phi_{N-1}^n) + \frac{c(c-1)}{2}(\phi_N^n - \phi_0^n)$$

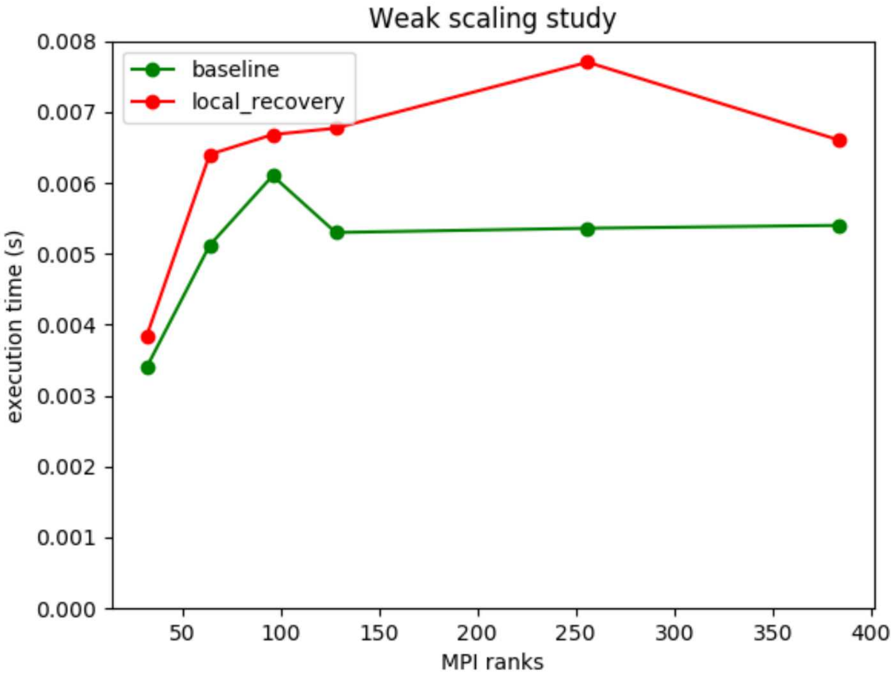
  - Checksum update requires only neighbouring (ghost) points.

Local vs Global Recovery

- The physics based checksum allow an efficient, purely local error detection mechanism.
- Upon detection, suitable resilience mechanisms can be deployed.
- Checkpoint/Rollback is most common; but global rollback recovery is a disproportionate response for a purely local detection mechanism.
- We examined “local” and “global” recovery using Fenix:
  - An MPI-based fault tolerance library for distributed resilience.
  - Primary design is for hard failure (process loss); built on top of MPI-ULFM.
  - Provides APIs for data “store”/ “restore” operations to recover from process loss.
  - Extended to provide purely local recovery (no MPI process loss, no communication for store/restore).

Results: Weak scaling study

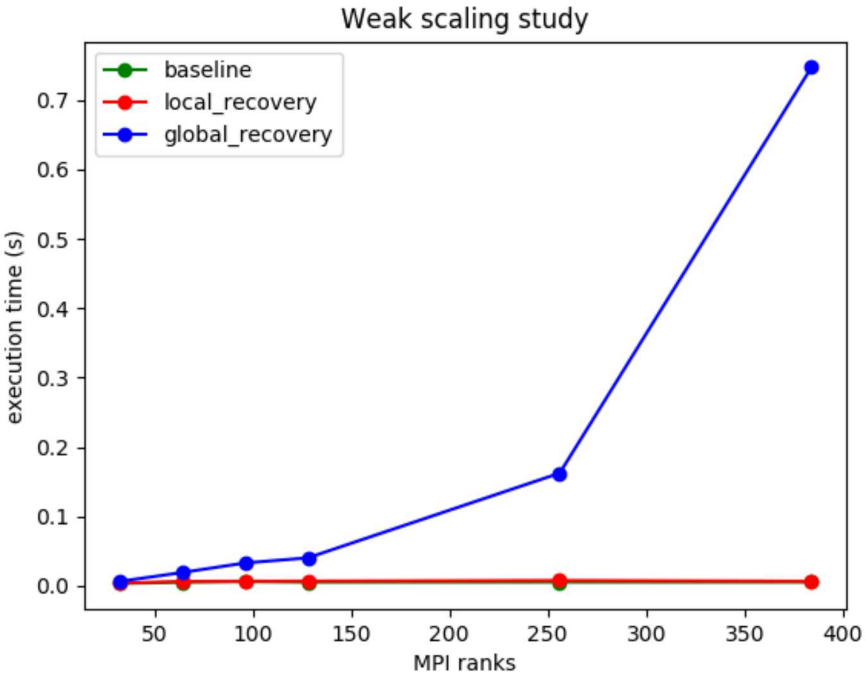
- Weak scaling study of baseline, local_recovery and global_recovery versions of 1D stencil.
- Emulated silent error rate of 0.01, 1000 grid points per rank, 1000 iterations, runs on Cori.



- Local checkpoint/restart adds marginal overhead.
- Does not disrupt weak scalability of the baseline code.
- Problem size is cache friendly.

Results: Weak scaling study

- Weak scaling study of baseline, local_recovery and global_recovery versions of 1D stencil.
- Emulated silent error rate of 0.01, 1000 grid points per rank, 1000 iterations, runs on Cori.



- Global recovery involves an global agreement (anyone fails, everyone rolls back).
- Not scalable, recovery cost scales with number of ranks.
- Cascading cost of recovery (some ranks stuck in an endless loop of restarts).