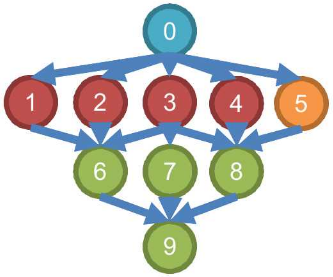
SAND2019-0384PE



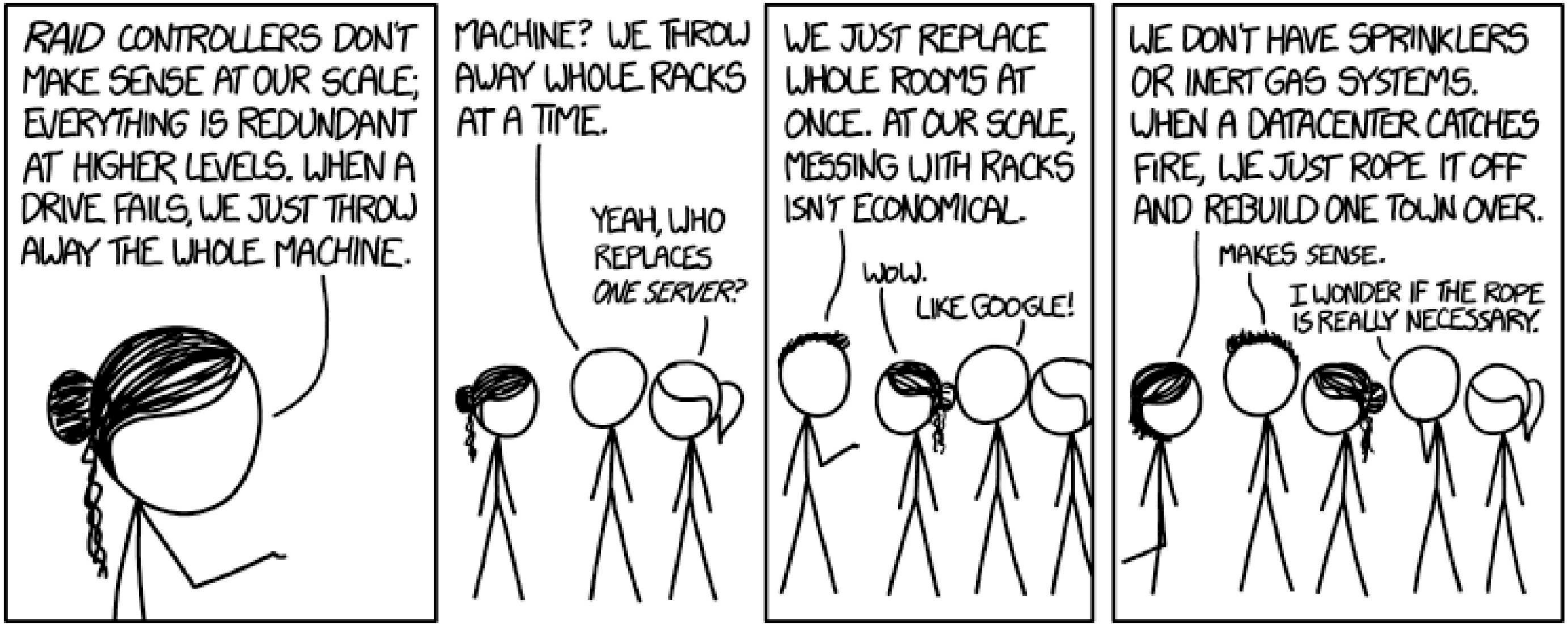
Local Failure Local Recovery: Toward Scalable Resilient Parallel Programing Model

Keita Teranishi, Sandia National Laboratories, California, USA

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

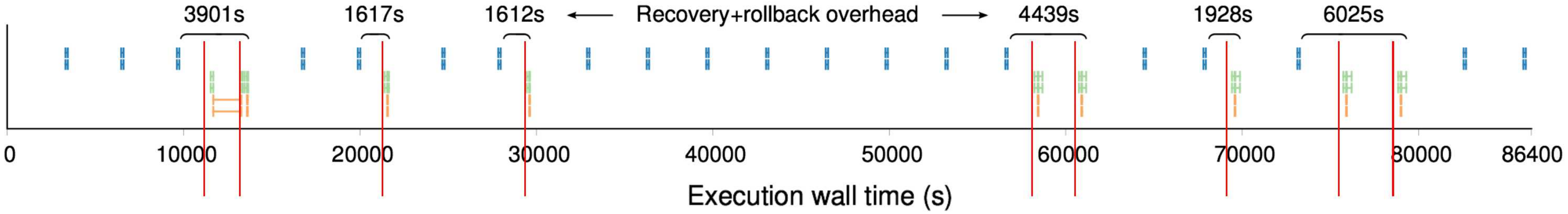
Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

MOTIVATIONS AND BACKGROUND


RAID CONTROLLERS DON'T MAKE SENSE AT OUR SCALE; EVERYTHING IS REDUNDANT AT HIGHER LEVELS. WHEN A DRIVE FAILS, WE JUST THROW AWAY THE WHOLE MACHINE.
MACHINE? WE THROW AWAY WHOLE RACKS AT A TIME.
YEAH, WHO REPLACES ONE SERVER?
WE JUST REPLACE WHOLE ROOMS AT ONCE. AT OUR SCALE, MESSING WITH RACKS ISN'T ECONOMICAL.
WOW.
LIKE GOOGLE!
WE DON'T HAVE SPRINKLERS OR INERT GAS SYSTEMS. WHEN A DATACENTER CATCHES FIRE, WE JUST ROPE IT OFF AND REBUILD ONE TOWN OVER.
MAKES SENSE.
I WONDER IF THE ROPE IS REALLY NECESSARY.

Courtesy: https://xkcd.com/1737

More frequent failures than advertised



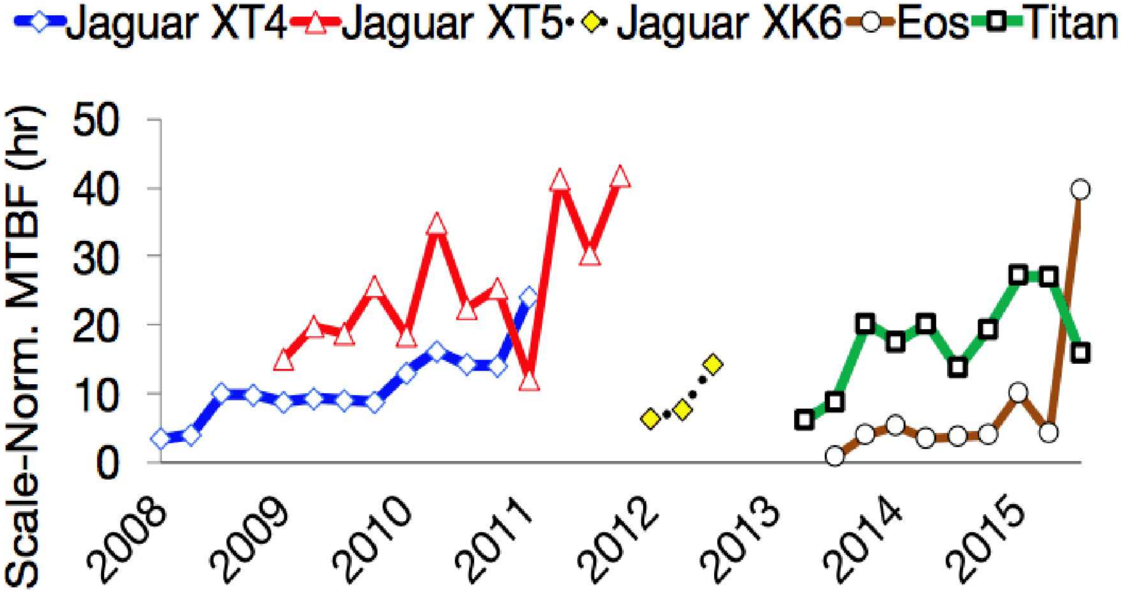
- 24-hour tests using Titan (125k cores)
- Expected MTBF: 9-12 hours
- 9 process/node failures over 24 hours
- Failures are promoted to **job failures**, causing all 125k processes to exit
- Checkpoint (5.2 MB/core) is done to the PFS
- Burst buffer provides more BW than the traditional file IO, but the major bottleneck is the connection to the burst buffer nodes.

		Total cost
Checkpoint (per timestep)	**55 s**	1.72 %
Restarting processes	**470 s**	5.67 %
L		
R		
T		

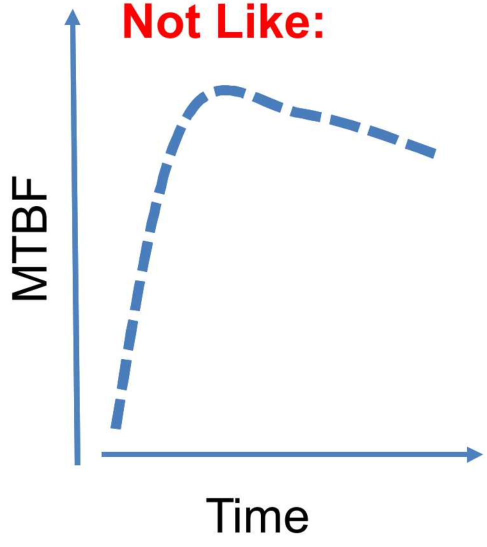
"Exploring Automatic, Online Failure Recovery for Scientific Applications at Extreme Scales", SC14 Marc Gamell, Daniel S. Katz, Hemanth Kolla, Jacqueline Chen, Scott Klasky, Manish Parashar

System reliability is hard to predict

That means many failures could happen all of sudden.



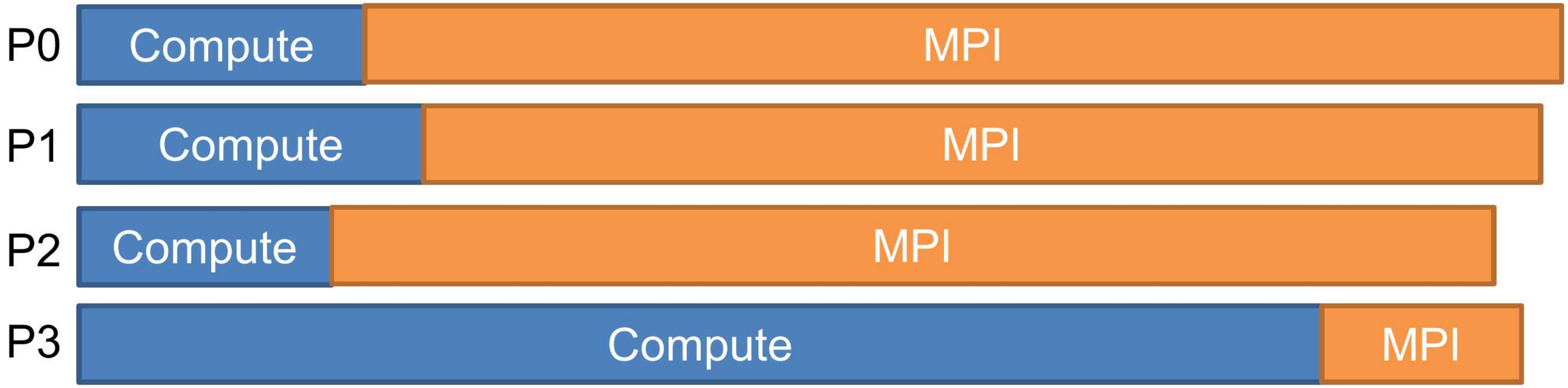
Not Like:



Courtesy to **Gupta et al**, "*Failures in large scale systems: long-term measurement, analysis, and implications*," SC17

- **Reliability of large scale HPC systems has been the major concern**
  - **Exascale Goal: 1 Week MTBF with C/R**
- **No clear reliability improvement (in scaled MTBF)**
- **No effective predictable model of reliability derived from observations (Gupta et al and Ferreira et al.)**

Resilience is essential for performance variability

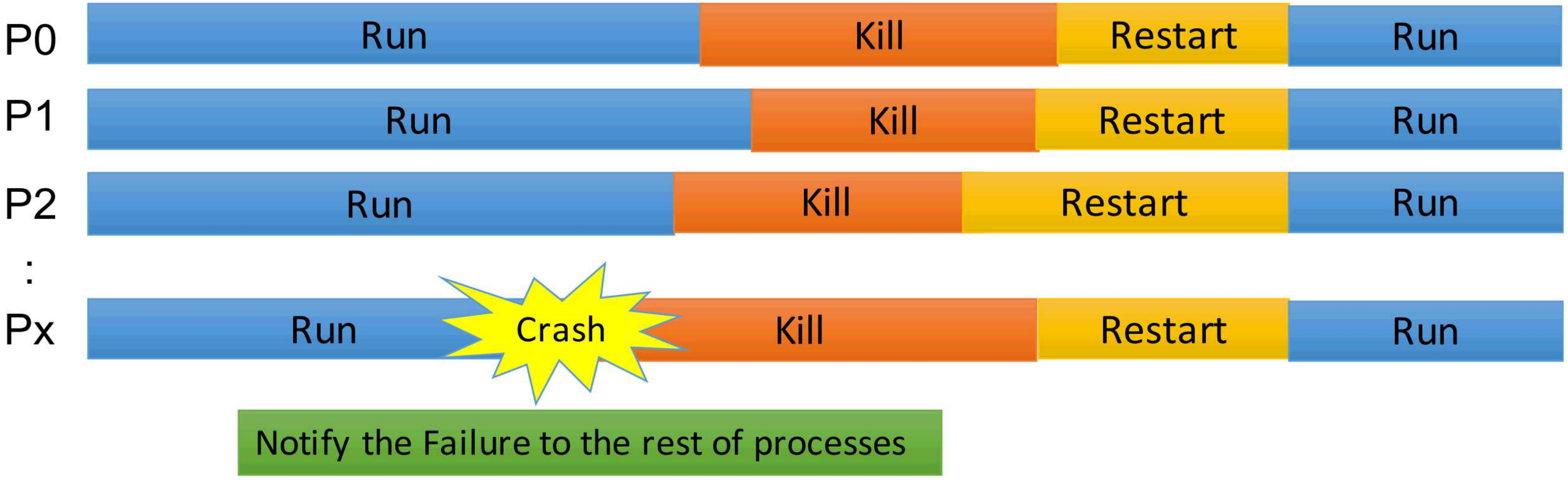


- **Performance variability** is a new type of system failure.
  - Trinity at LANL experienced a 25x slowdown of a single compute node
    - Static load balancing based on data size won't work
    - Errors in a single DRAM module
    - MPI did not report any errors
  - Resulted in 25x application delays

Resilience is essential for System Co-Design

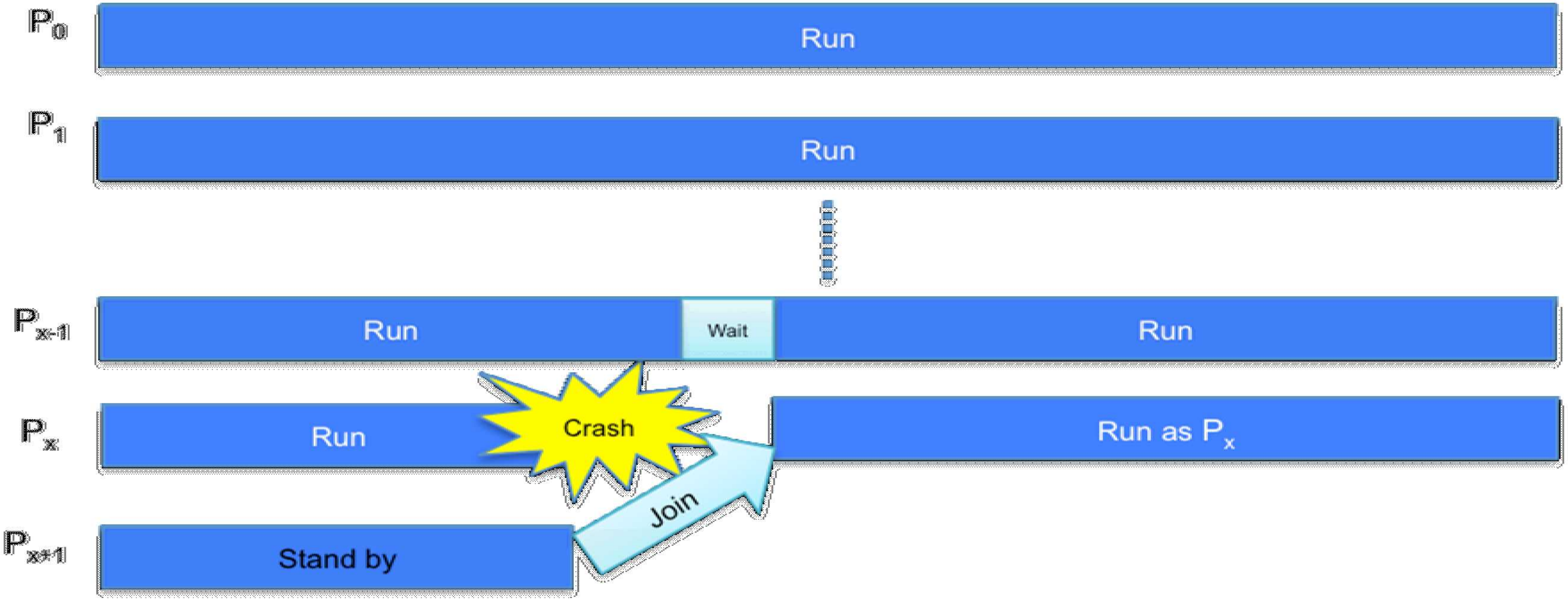
- **Programming model (APIs)** that embraces/controls failures and unconventional errors permits a greater flexibility in system co-design.
  - Probabilistic CMOS (PCMOS) for efficiency and low power
    - Palem at RICE U. (Performance and accuracy modeling)
    - Rinard at MIT (Programming language for unreliable computing, and ideas for fault oblivious computing)
  - Memory subsystems with selective reliability
    - HBM (high bandwidth, less reliable SEC-DED) and DDR (low bandwidth, reliable Chipkill)
      - – Gupta, UCSD and AMD

Checkpoint/Restart evolves toward Exascale Computing, but….



- VeloC (ECP: https://veloc.readthedocs.io/en/latest/) accommodates efficient checkpointing/restart
  - Multi-level checkpointing
  - Leverage the latest I/O technology.
- Disproportionate use of computing resources is inevitable
  - Majority (50-85%) of failures happen at single node/process.
  - Cost of global tear-down and global restart (redo).
- Is it designed to handle soft-errors and online recovery?

Local Failure and Local Recovery Enables Scalable Recovery



- Software framework to augment existing apps with resilience capability
  - The remaining processes stay alive with **isolated** process/node failure
  - Multiple implementation options for recovery
    - Roll-back, roll-forward, asynchronous, algorithm specific, etc.
  - **Hot Spare Process for recovery**

RESILIENT PROGRAMMING MODEL FOR MPI PROGRAMMING

MPI-ULFM (User Level Fault Mitigation)

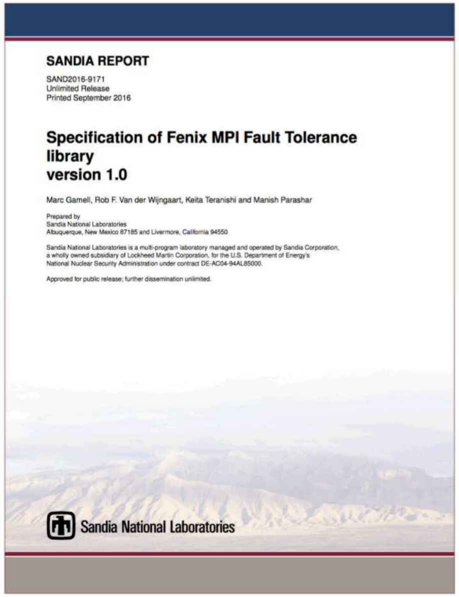
- Proposed for future MPI standard
- MPI calls (recv, irecv, wait, collectives) notify errors when the peer process(es) dies
- Survived processes continue to run
- New MPI functions for fixing MPI communicator
  - MPI_Comm_agree --- Sanity check (resilient collective)
  - MPI_Comm_revoke --- Invalidate MPI Communicator
  - MPI_Comm_shrink --- Fix MPI Communicator removing dead process
- User is responsible for the recovery after MPI_Comm_shrink Prototype code is available at http://fault-tolerance.org
  - Developed by U of Tennessee

MPI-ULFM does not prescribe how to recovery

- MPI-ULFM only provides "minimum" set of low-level APIs for application recovery
  - **Users are responsible for fixing MPI communicator**
    - Shrunk Communicator is no longer the same as the original MPI Communicator
    - Rank-Process mapping changes after comm_shrink
    - Typical MPI applications are not designed for the shrinking recovery
  - **Users are responsible for recovering the application state**
    - Writing an error handler is cumbersome
    - No data recovery
    - No rollback
- **Our Solution: Fenix**

Fenix 1.0 Specification (SAND2016-9171)

- 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

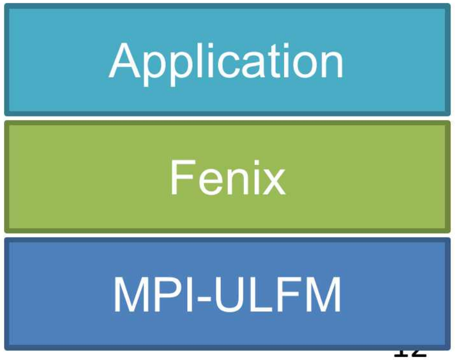
Marc Gamell, Rob F. Van der Wijngaart, Keita Teranishi and Manish Parashar

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



Application

Fenix

MPI-ULFM

S3D Modifications

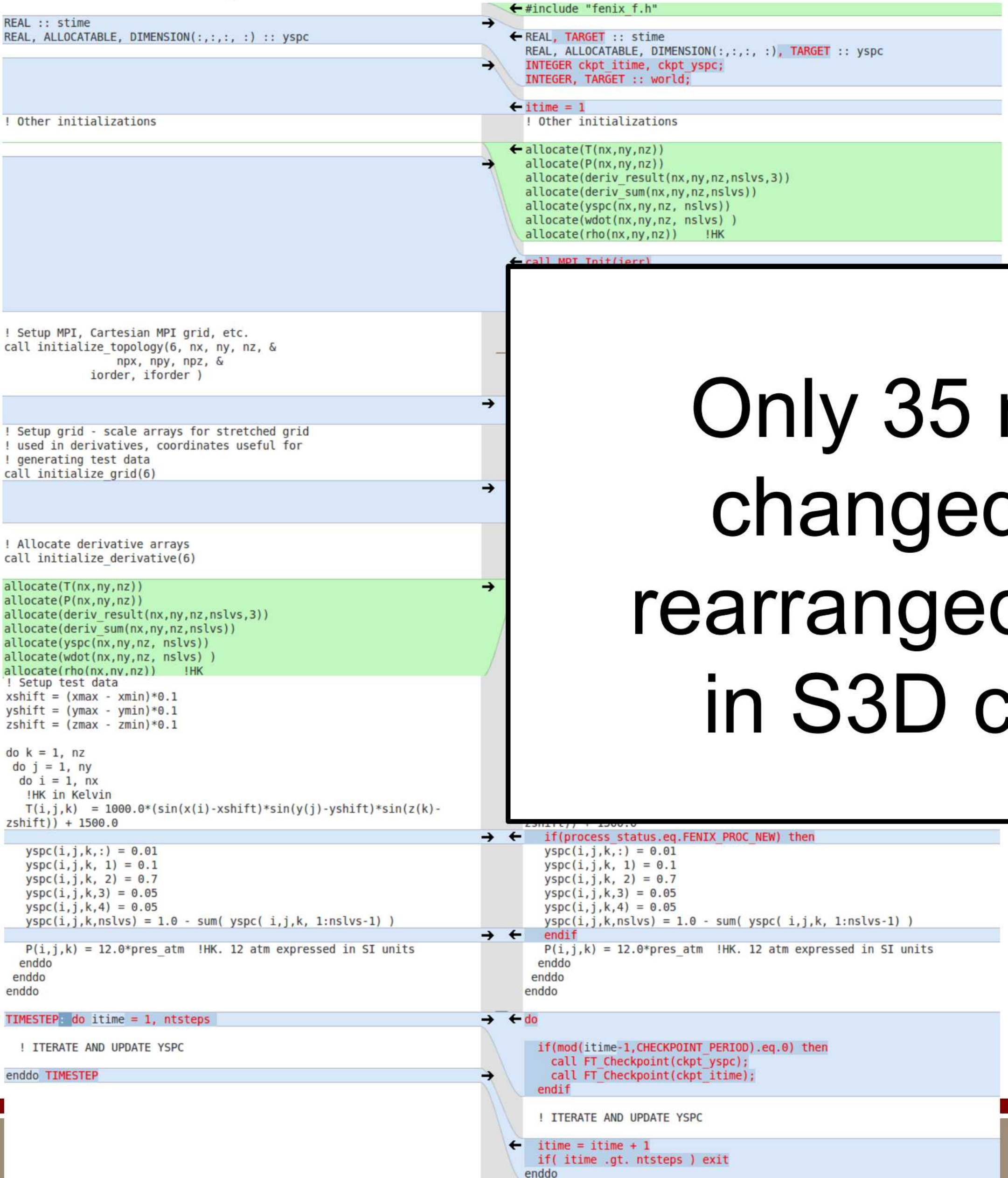
- Only 35 new, changed, or rearranged lines in S3D

unction

gy module

Only 35 new, changed, or rearranged lines in S3D code

Original vs Fenix-enabled



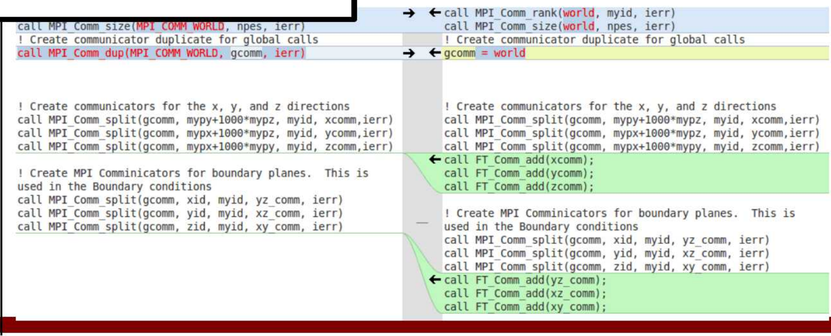
Original:

```
REAL :: stime
REAL, ALLOCATABLE, DIMENSION(:,:,:, :) :: yspc

! Other initializations

! Setup MPI, Cartesian MPI grid, etc.
call initialize_topology(6, nx, ny, nz, &
                 npx, npy, npz, &
             iorder, iforder )

! Setup grid - scale arrays for stretched grid
! used in derivatives, coordinates useful for
! generating test data
call initialize_grid(6)

! Allocate derivative arrays
call initialize_derivative(6)

allocate(T(nx,ny,nz))
allocate(P(nx,ny,nz))
allocate(deriv_result(nx,ny,nz,nslvs,3))
allocate(deriv_sum(nx,ny,nz,nslvs))
allocate(yspc(nx,ny,nz, nslvs))
allocate(wdot(nx,ny,nz, nslvs) )
allocate(rho(nx,ny,nz))     !HK
 ! Setup test data
xshift = (xmax - xmin)*0.1
yshift = (ymax - ymin)*0.1
zshift = (zmax - zmin)*0.1

do k = 1, nz
 do j = 1, ny
  do i = 1, nx
   !HK in Kelvin
   T(i,j,k)  = 1000.0*(sin(x(i)-xshift)*sin(y(j)-yshift)*sin(z(k)-
zshift)) + 1500.0

   yspc(i,j,k,:) = 0.01
   yspc(i,j,k, 1) = 0.1
   yspc(i,j,k, 2) = 0.7
   yspc(i,j,k,3) = 0.05
   yspc(i,j,k,4) = 0.05
   yspc(i,j,k,nslvs) = 1.0 - sum( yspc( i,j,k, 1:nslvs-1) )

   P(i,j,k) = 12.0*pres_atm  !HK. 12 atm expressed in SI units
  enddo
 enddo
enddo

TIMESTEP: do itime = 1, ntsteps

  ! ITERATE AND UPDATE YSPC

enddo TIMESTEP
```

Fenix-enabled:

```
#include "fenix_f.h"

REAL, TARGET :: stime
REAL, ALLOCATABLE, DIMENSION(:,:,:, :), TARGET :: yspc
INTEGER ckpt_itime, ckpt_yspc;
INTEGER, TARGET :: world;

itime = 1
! Other initializations

allocate(T(nx,ny,nz))
allocate(P(nx,ny,nz))
allocate(deriv_result(nx,ny,nz,nslvs,3))
allocate(deriv_sum(nx,ny,nz,nslvs))
allocate(yspc(nx,ny,nz, nslvs))
allocate(wdot(nx,ny,nz, nslvs) )
allocate(rho(nx,ny,nz))     !HK

call MPI_Init(ierr)

zshift)) + 1500.0
   if(process_status.eq.FENIX_PROC_NEW) then
   yspc(i,j,k,:) = 0.01
   yspc(i,j,k, 1) = 0.1
   yspc(i,j,k, 2) = 0.7
   yspc(i,j,k,3) = 0.05
   yspc(i,j,k,4) = 0.05
   yspc(i,j,k,nslvs) = 1.0 - sum( yspc( i,j,k, 1:nslvs-1) )
   endif
   P(i,j,k) = 12.0*pres_atm  !HK. 12 atm expressed in SI units
  enddo
 enddo
enddo

do

  if(mod(itime-1,CHECKPOINT_PERIOD).eq.0) then
    call FT_Checkpoint(ckpt_yspc);
    call FT_Checkpoint(ckpt_itime);
  endif

  ! ITERATE AND UPDATE YSPC

  itime = itime + 1
  if( itime .gt. ntsteps ) exit
enddo
```

vs Fenix-enabled



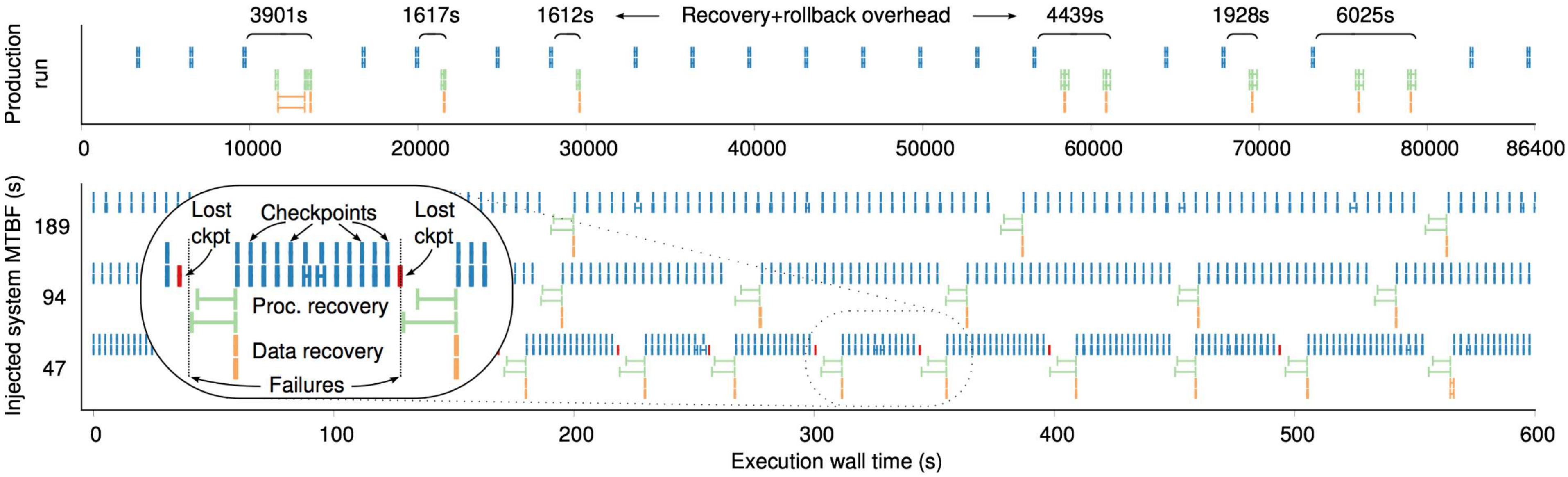
Original:

```
call MPI_Comm_size(MPI_COMM_WORLD, npes, ierr)
! Create communicator duplicate for global calls
call MPI_Comm_dup(MPI_COMM_WORLD, gcomm, ierr)


! Create communicators for the x, y, and z directions
call MPI_Comm_split(gcomm, mypy+1000*mypz, myid, xcomm,ierr)
call MPI_Comm_split(gcomm, mypx+1000*mypz, myid, ycomm,ierr)
call MPI_Comm_split(gcomm, mypx+1000*mypy, myid, zcomm,ierr)

! Create MPI Comminicators for boundary planes.  This is
used in the Boundary conditions
call MPI_Comm_split(gcomm, xid, myid, yz_comm, ierr)
call MPI_Comm_split(gcomm, yid, myid, xz_comm, ierr)
call MPI_Comm_split(gcomm, zid, myid, xy_comm, ierr)
```

Fenix-enabled:

```
call MPI_Comm_rank(world, myid, ierr)
call MPI_Comm_size(world, npes, ierr)
! Create communicator duplicate for global calls
gcomm = world


! Create communicators for the x, y, and z directions
call MPI_Comm_split(gcomm, mypy+1000*mypz, myid, xcomm,ierr)
call MPI_Comm_split(gcomm, mypx+1000*mypz, myid, ycomm,ierr)
call MPI_Comm_split(gcomm, mypx+1000*mypy, myid, zcomm,ierr)
call FT_Comm_add(xcomm);
call FT_Comm_add(ycomm);
call FT_Comm_add(zcomm);

! Create MPI Comminicators for boundary planes.  This is
used in the Boundary conditions
call MPI_Comm_split(gcomm, xid, myid, yz_comm, ierr)
call MPI_Comm_split(gcomm, yid, myid, xz_comm, ierr)
call MPI_Comm_split(gcomm, zid, myid, xy_comm, ierr)
call FT_Comm_add(yz_comm);
call FT_Comm_add(xz_comm);
call FT_Comm_add(xy_comm);
```

Global Online Recovery – Results



	MTBF	*Total overhead*
Production	**2.6 h**	**31 %**
Global recovery	189 s	10 %
Global recovery	94 s	15 %
Global recovery	**47 s**	**31 %**

- Uses S3D (scientific application)
- Titan Cray XK7 (#3 on top500.org)
- Injecting node failures (16-core failures)

Asynchronous Localized Online Recovery

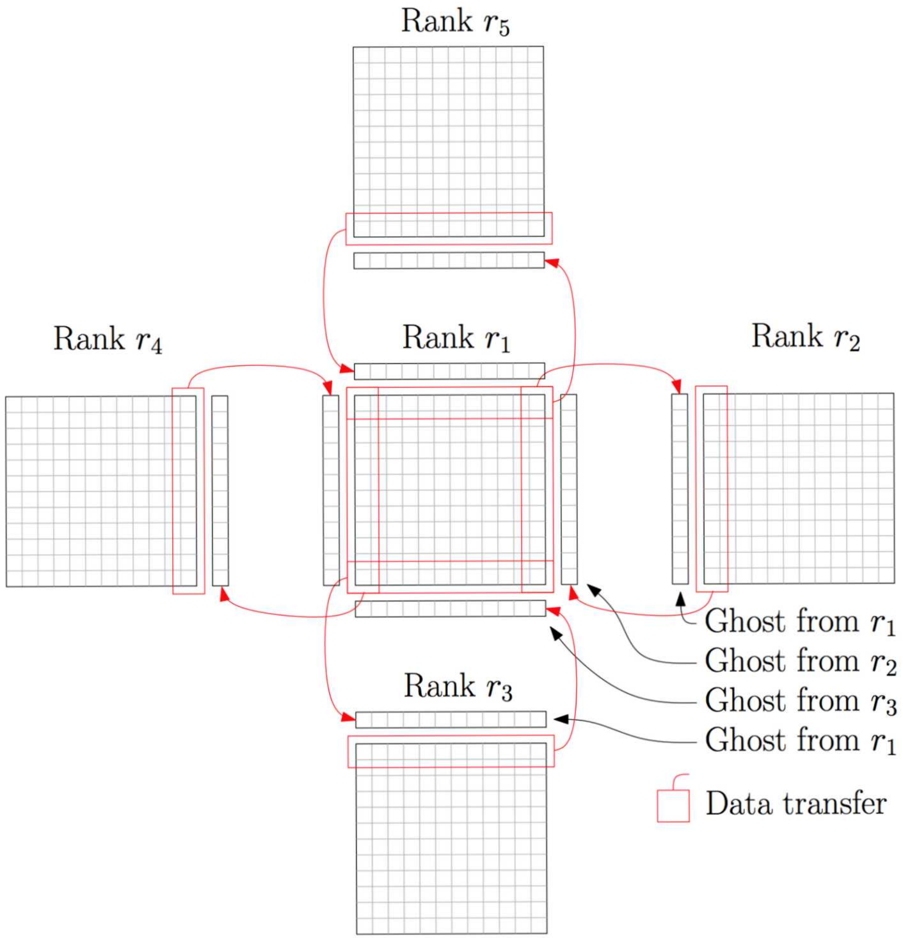
- Fenix-1.0 is the first step toward local recovery
  - Avoid global termination and restart
  - All processes rollback to the Fenix_Init() call
  - Natural for algorithms and applications that makes collective calls frequently
- Some applications fit more scalable recovery model
  - Stencil Computation
  - Master-Worker execution model
- Solution: Local Online Recovery

Local Recovery Methodology

1. Replace failed processes
2. Rollback to the last checkpoint (only replaced processes)
3. Other processes continue with the simulation

- How do we guarantee consistency?
  - Implicitly coordinated checkpoint
  - Log messages since last checkpoint in local sender memory
  - Message logging has been studied in MPI fault tolerance and Actor Execution Model (Charm++)
    - Performance may not be optimal for many parallel applications
    - **Stencil computation provides built-in message logging == Ghost Points**
- Implemented in new framework: **FenixLR**

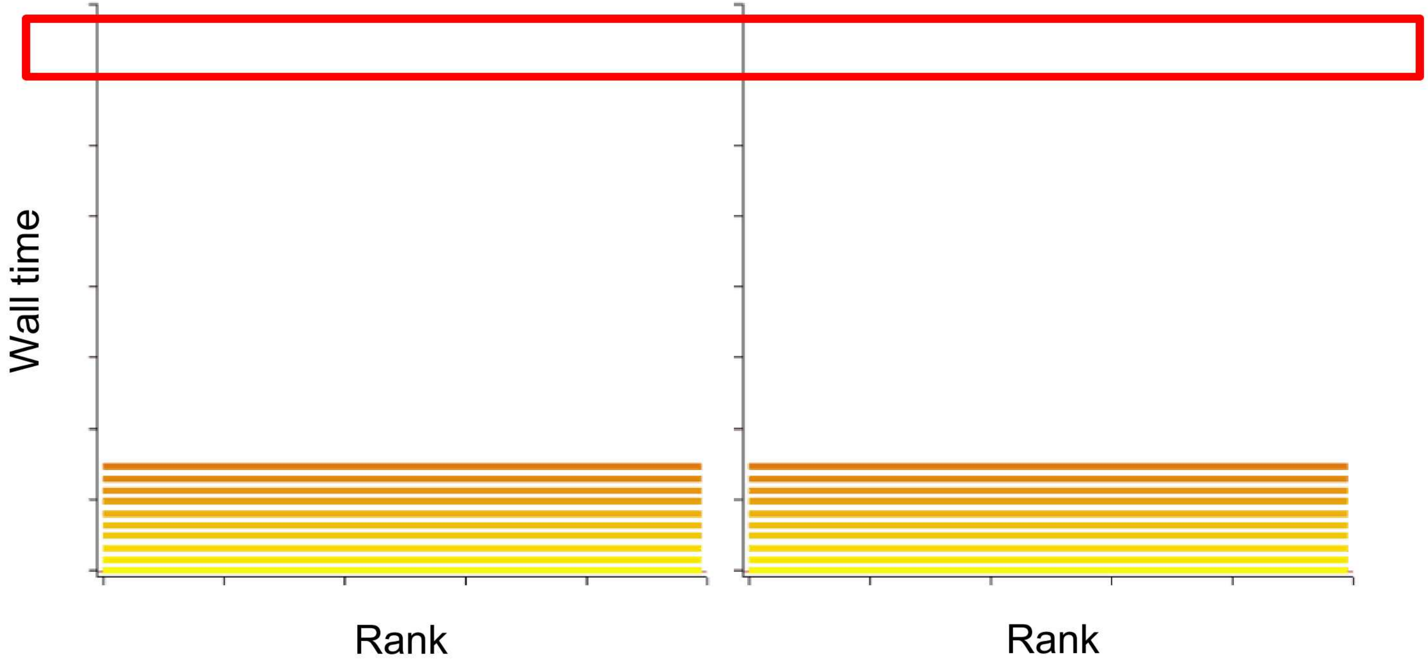
Target: Stencil-based Scientific Applications



- Application domain is partitioned using a block decomposition across processes
- Typically, divided into iterations (*timesteps*), which include:
  - Computation to advance the local simulated data
  - Communication with immediate neighbors
- Example: PDEs using finite-difference methods, S3D

Performance Model of Local Recovery

Simulated execution of a 1D PDE

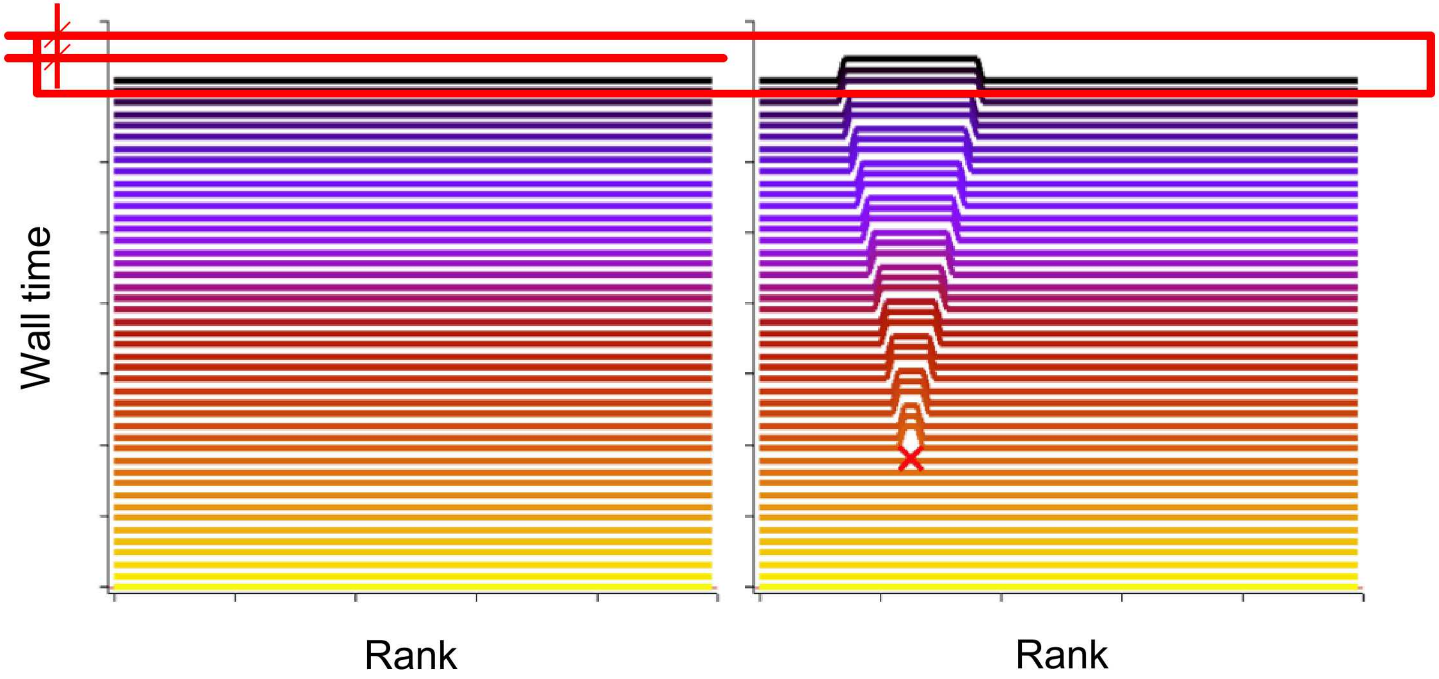


No failures

One failure

Effect of Multiple Failures with Local Recovery

Simulated execution of a 1D PDE

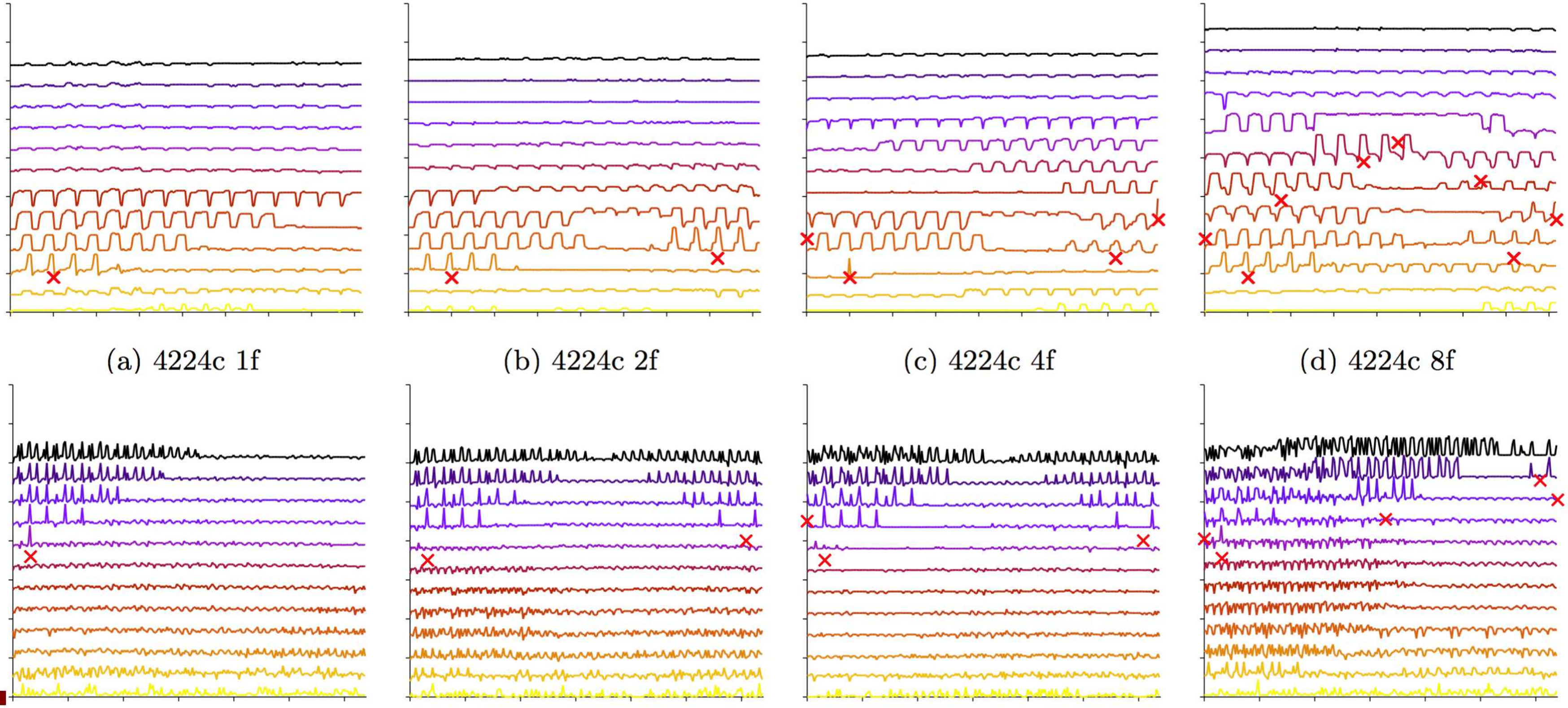


No failures

One failure

Experimental Evaluation with S3D

- Same experiment executed injecting different number of failures
- X axis is rank number, but more complex to see than 1D, because 3D domain is mapped to core ranking in a linear fashion
- Note that total overhead is as if only one failure occurred (except in 4224c 8f)



(a) 4224c 1f

(b) 4224c 2f

(c) 4224c 4f

(d) 4224c 8f

(q) 64128c 1f

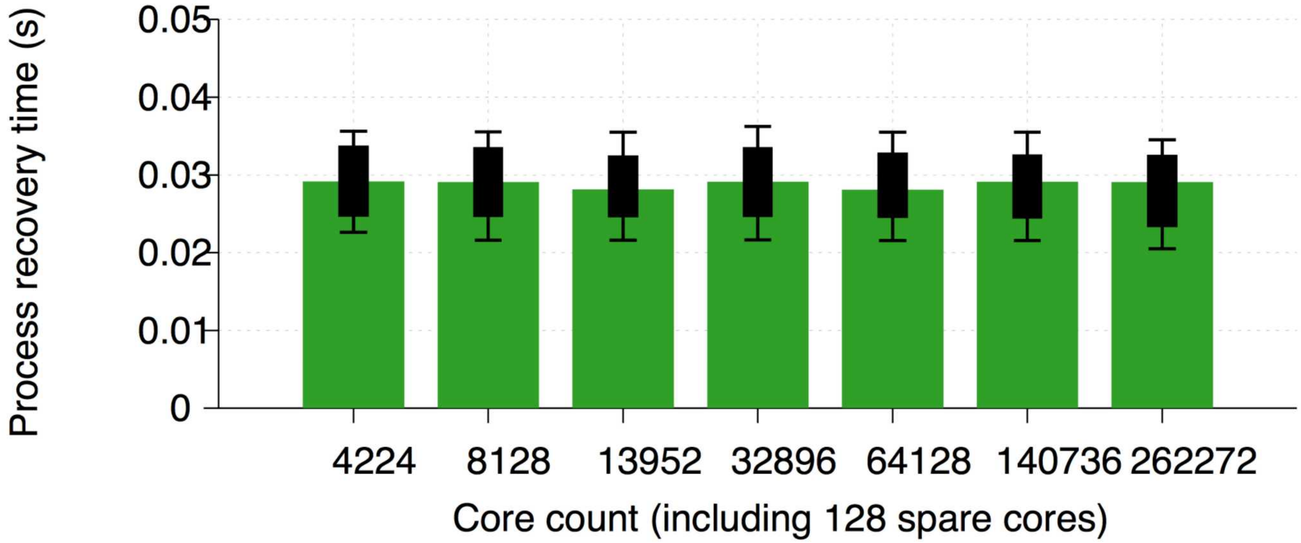
(r) 64128c 2f

(s) 64128c 3f

(t) 64128c 5f

Performance of Fenix-LR

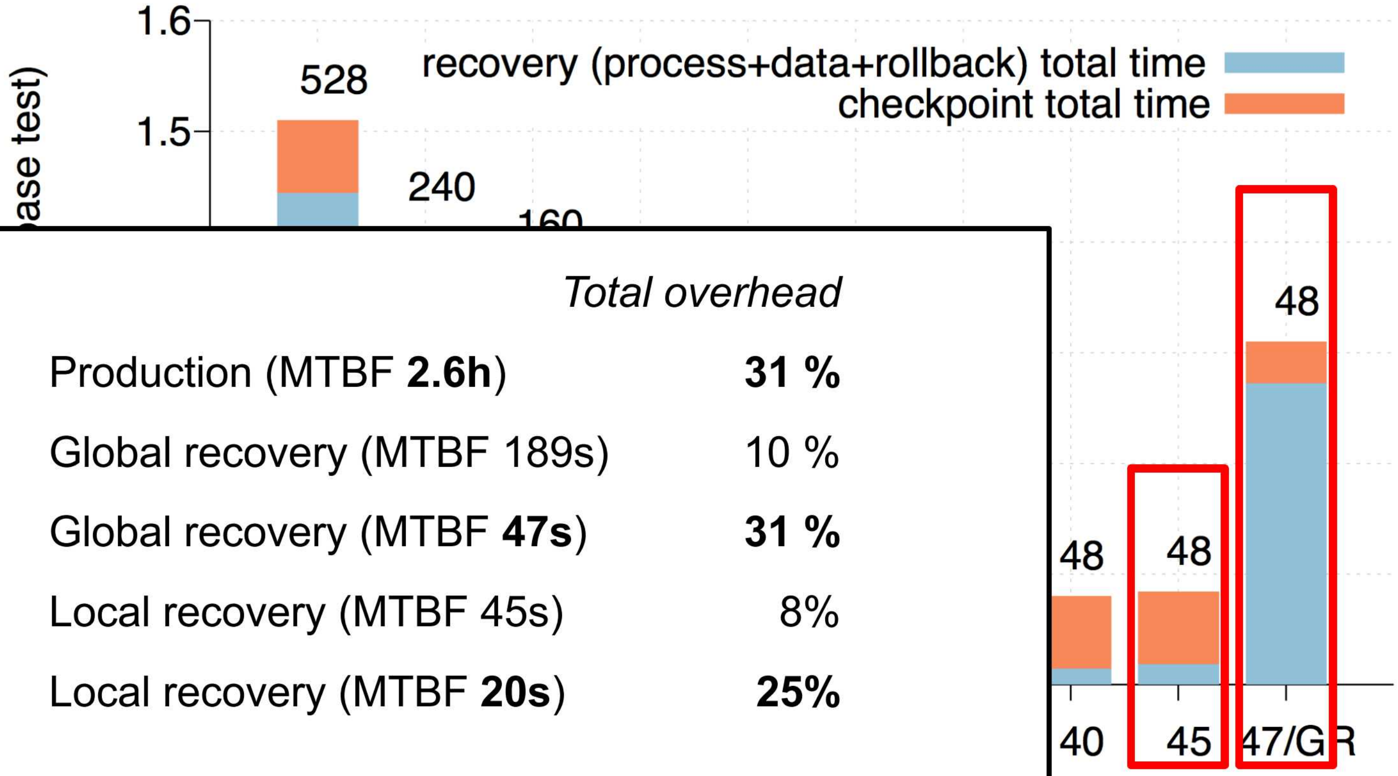
- Using MTBF of 10s
- Core count from 4224 to 262272 (including 128 spare cores)
- Result shows the average recovery time for all failures injected.



- Conclusion:
  - Process recovery time is independent of system size
  - Good scalability

Total Overhead of Fault Tolerance

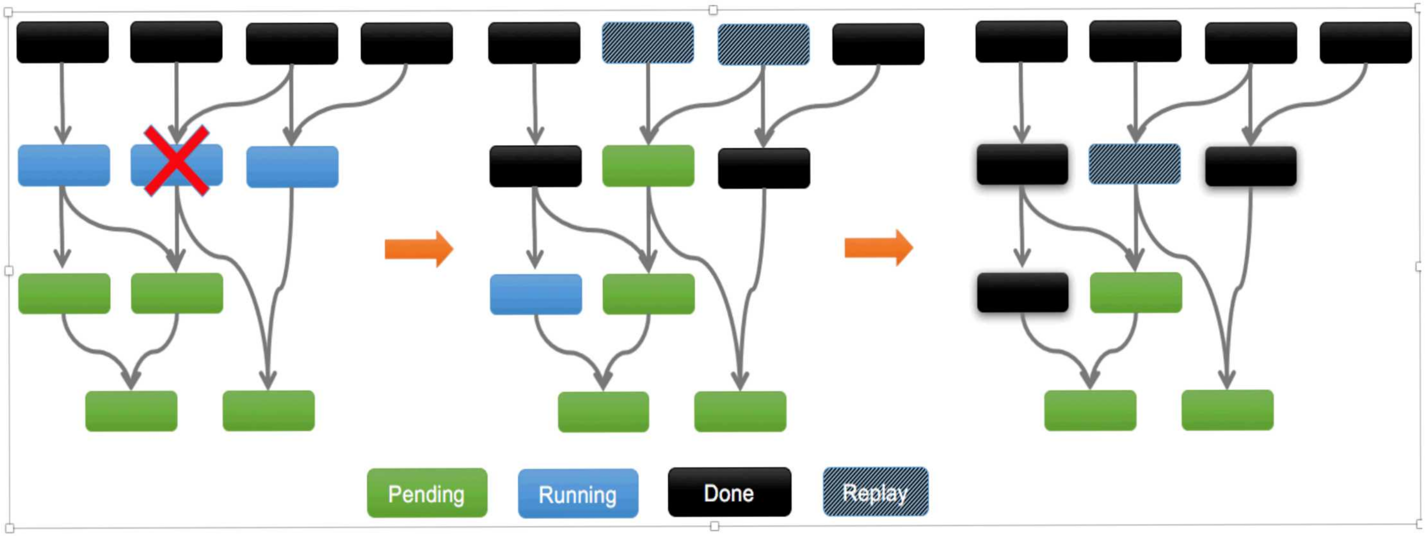
- End-to-end time vs failure-free, checkpoint-free time
- Overall overhead:
  - Checkpoint
  - Process/da recovery
  - Rollback
- 4096 cores + spare cores
- Right-most bar global recovery with MTBF of 4
- Local recovery has scalability advantages over global recovery



	Total overhead
Production (MTBF **2.6h**)	**31 %**
Global recovery (MTBF 189s)	10 %
Global recovery (MTBF **47s**)	**31 %**
Local recovery (MTBF 45s)	8%
Local recovery (MTBF **20s**)	**25%**

- **Local recovery is superior to global recovery in this scenario:**
  - **compare MTBF 45s (8%)**
  - **with MTBF 47/GR (31%)**

Resilient Asynchronous Many Task (AMT) Parallel Execution Model



- AMT allows
  - Concurrent task execution
  - Overlap of communication and computation
  - Over-decomposition of Data
  - Abstraction of data objects and tasks allows failure containment and transparent application recovery with ease.
- Node/Process Failure is manifested as loss of task and data
  - Generic model for online local recovery
  - Recovery is done through task replay, replication and ABFT task (special task for recovery)

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);
```

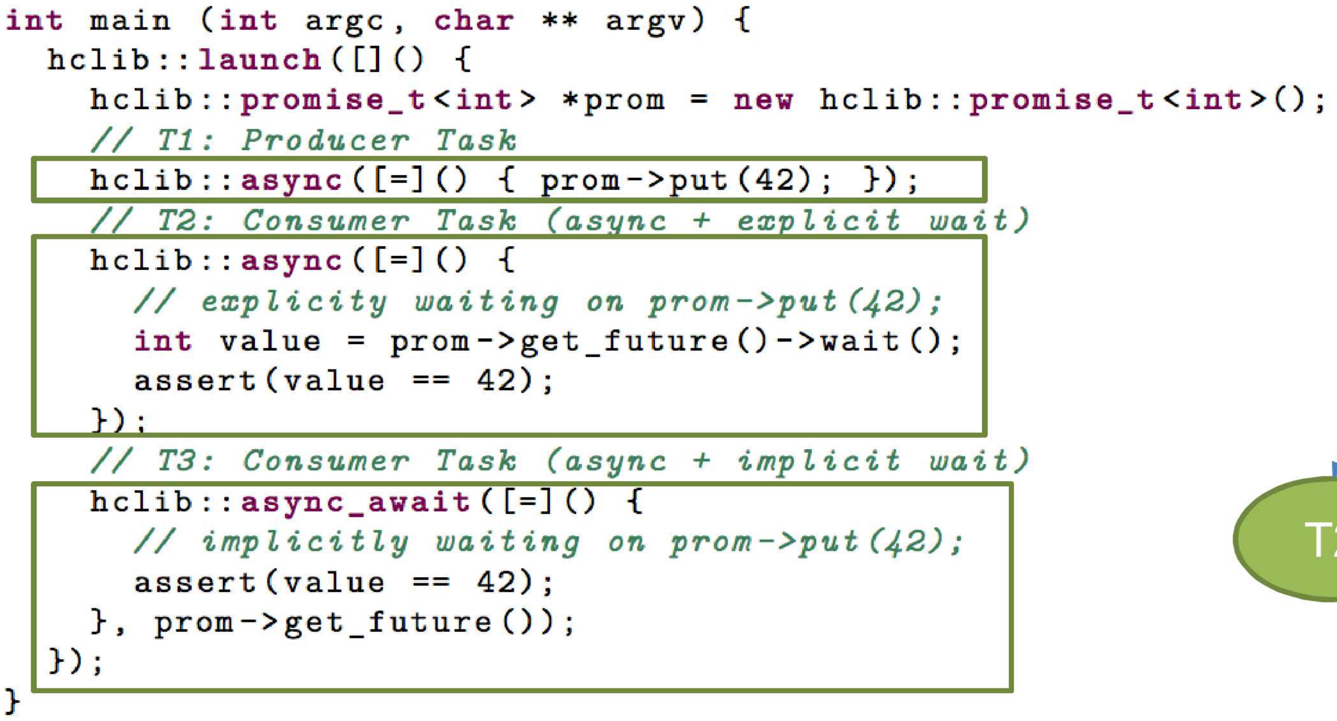
Habanero-C++ Library (HClib)

- Project led by Vivek Sarkar (GaTech/Rice U)
- Library-based tasking runtime and API
  - Semantically derived from X10
- Focused on: lightweight, minimal overheads; flexible synchronization; locality control; composability with other libraries;
- Simplified deployment: no custom compiler, entirely library-based, only requires C++11 compliant compiler
- Uses runtime-managed call stacks to avoid blocking
- https://github.com/habanero-rice/hclib

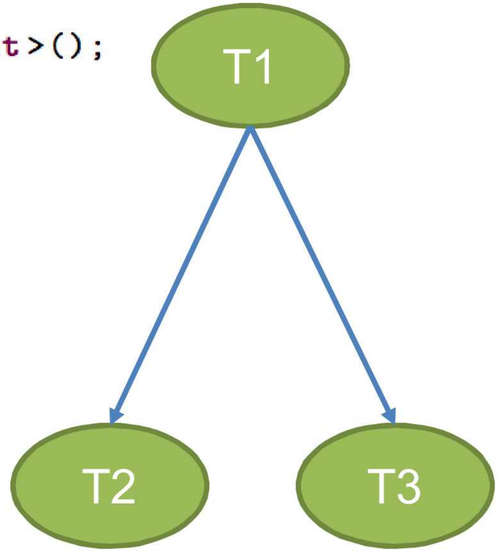
Three major constructs for AMT programming with HClib

- Use Lambda to describe **tasks. (async and async_await)**
- Express data dependencies using **promises** and **futures**.
  - `hclib::promise`
    - Store a value using single assignment semantics : **`promise.put(value)`**
  - `hclib::future`
    - Retrieve the value stored in a promise : **`value = future.get()`**
    - Can be used as dependency for tasks
- Relation between **promise** and **future**
  - **`future = promise.get_future()`**
  - If accessed from different threads **`put()`** and **`get()`** are synchronized thus enabling a way for synchronization.

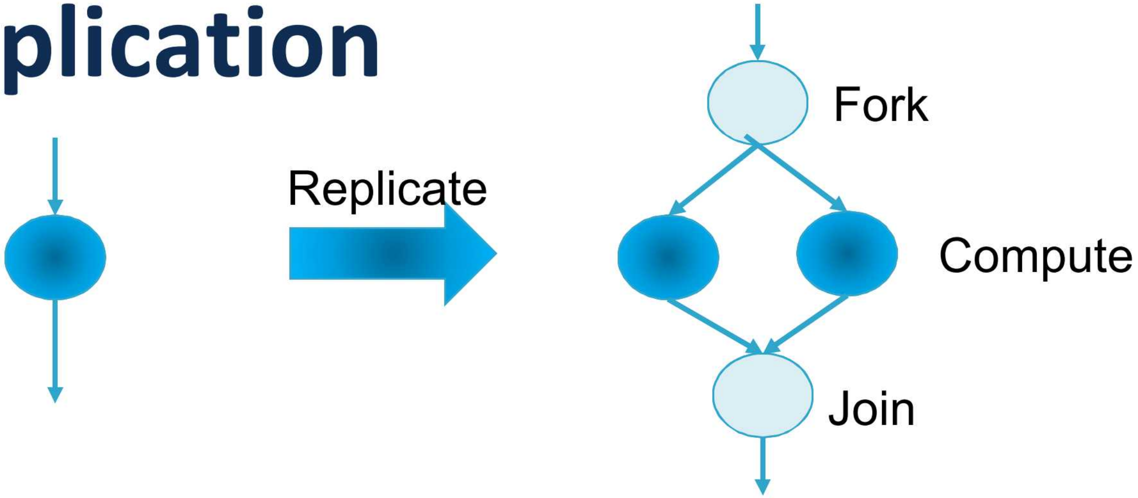
HClib Code Example



```
int main (int argc, char ** argv) {
  hclib::launch([]() {
    hclib::promise_t<int> *prom = new hclib::promise_t<int>();
    // T1: Producer Task
    hclib::async([=]() { prom->put(42); });
    // T2: Consumer Task (async + explicit wait)
    hclib::async([=]() {
      // explicity waiting on prom->put(42);
      int value = prom->get_future()->wait();
      assert(value == 42);
    });
    // T3: Consumer Task (async + implicit wait)
    hclib::async_await([=]() {
      // implicitly waiting on prom->put(42);
      assert(value == 42);
    }, prom->get_future());
  });
}
```



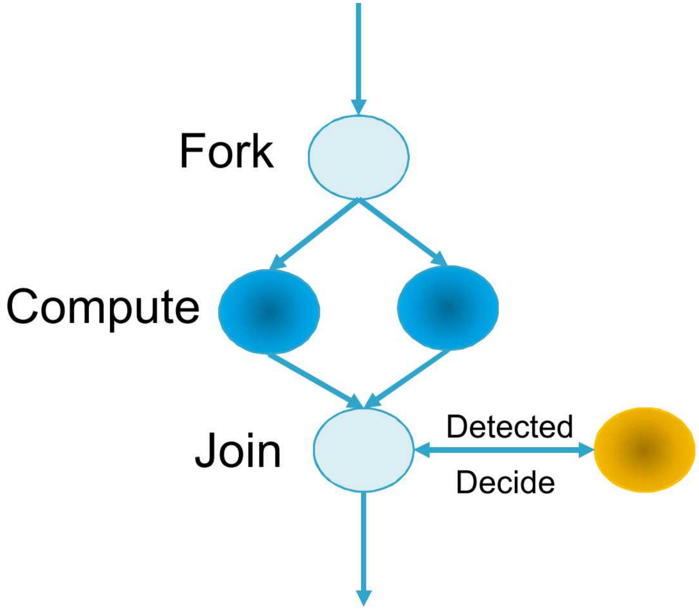
Task Replication



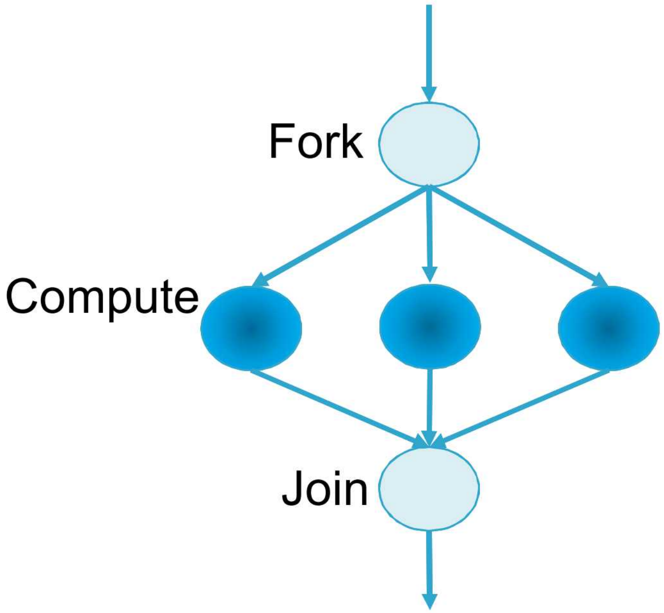
- `diamond::async_await_check<N> ( lambda, hclib::promise<int> out, hclib_future_t *f1, .., hclib_future_t *f4);`
    - Preventive failure mitigation
    - N-plicates the task and checks for equality of put operations at the end of the task
    - If error checking succeeds, actual puts are done
    - If error checking fails, puts are ignored and the error is reported using an output promise

Replication (Continued)

```
diamond::async_await_check<2>(…
```

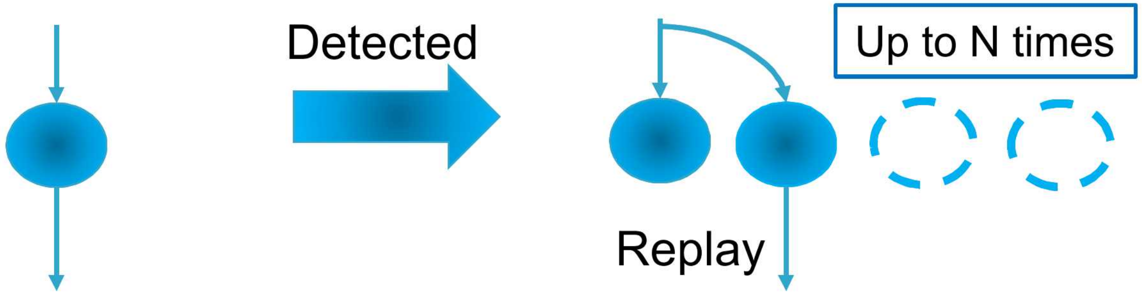


```
diamond::async_await_check<3>( …
```



- Duplicate (N=2) – Create two tasks and check for error in puts
  - If error checking fails, a third task is created
- Triplicate and more (N=3 ore more) – Create three tasks and check for error in puts
  - Two out of three outputs should match for success

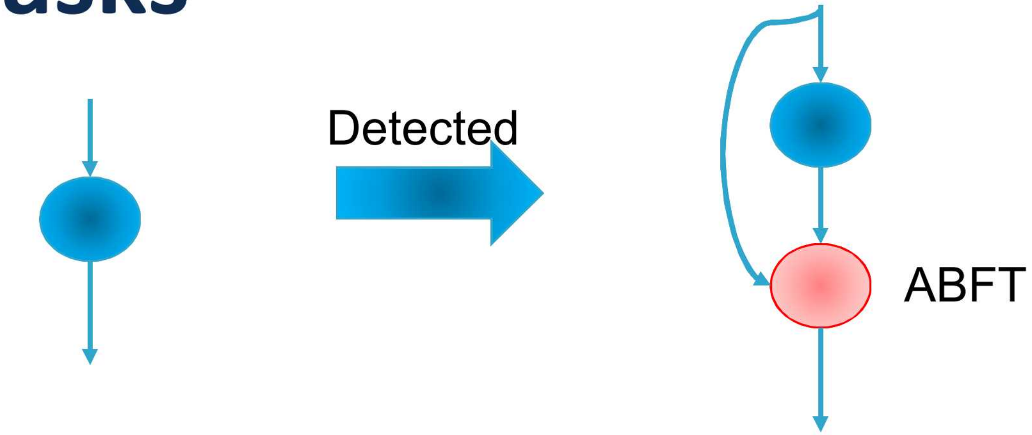
Task Replay



```
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);
```

- Dynamic response to failure
- Executes the task and checks for error using the error checking function
- error_check_fn(params) returns true if there is no error
- The task is executed **N** times at most if there is any error
  - If error checking fails, puts are ignored and the error is reported using an output promise

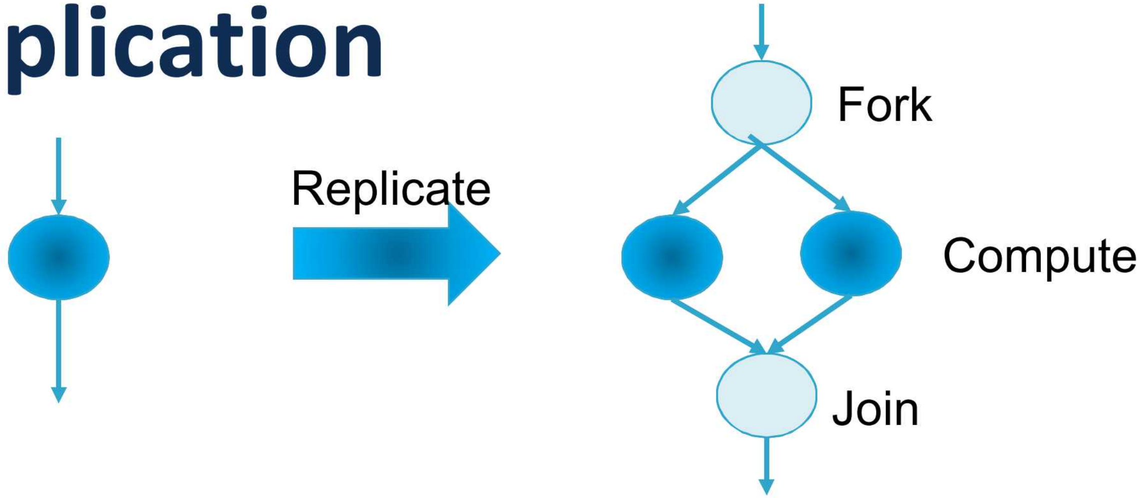
ABFT Tasks



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

- Executes the task and checks for error using the error checking function
- error_check_fn(params) returns true if there is no error
- If there is error then **ABFT_lambda** is executed and checked for error again at its end
    - If error checking fails, puts are ignored and the error is reported using an output promise

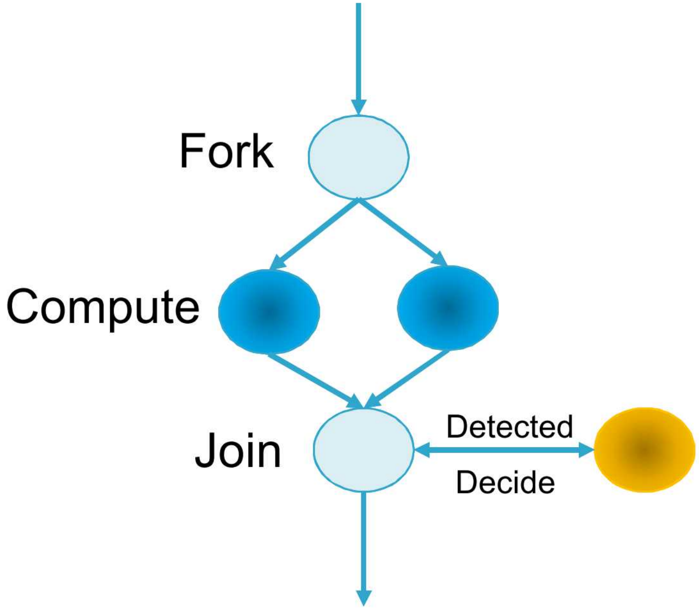
Task Replication



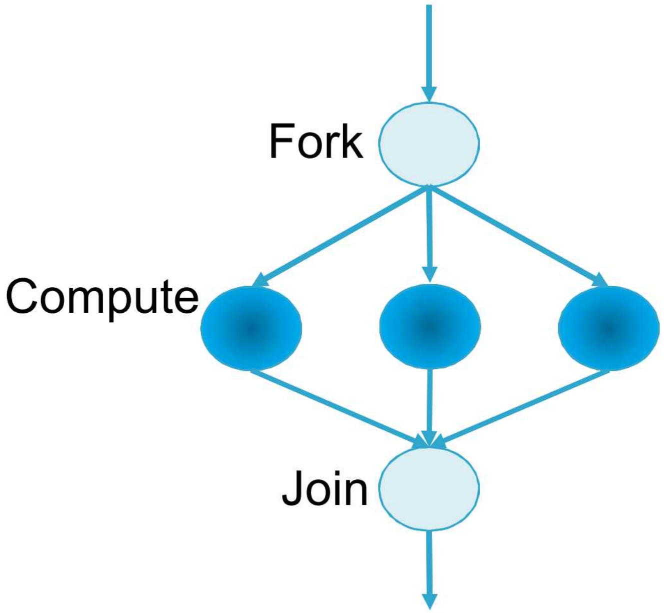
- **`diamond::async_await_check<N> ( lambda, hclib::promise<int> out, hclib_future_t *f1, .., hclib_future_t *f4);`**
  - Preventive failure mitigation
  - N-plicates the task and checks for equality of put operations at the end of the task
  - If error checking succeeds, actual puts are done
  - If error checking fails, puts are ignored and the error is reported using an output promise

Replication (Continued)

```
diamond::async_await_check<2>(…
```

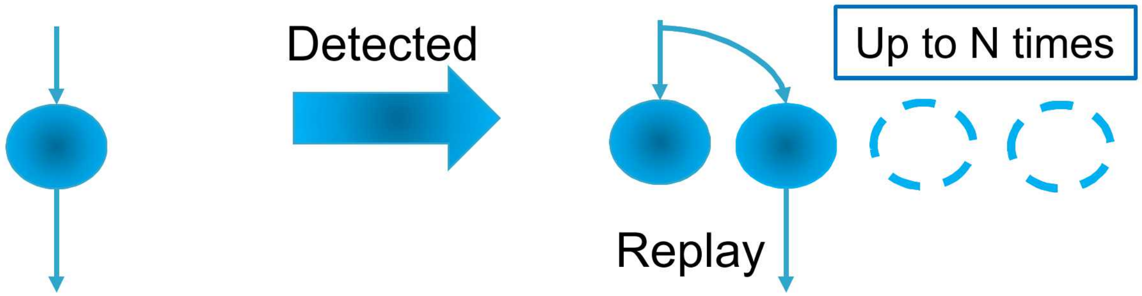


```
diamond::async_await_check<3>( …
```



- Duplicate (N=2) – Create two tasks and check for error in puts
  - If error checking fails, a third task is created
- Triplicate and more (N=3 ore more) – Create three tasks and check for error in puts
  - Two out of three outputs should match for success

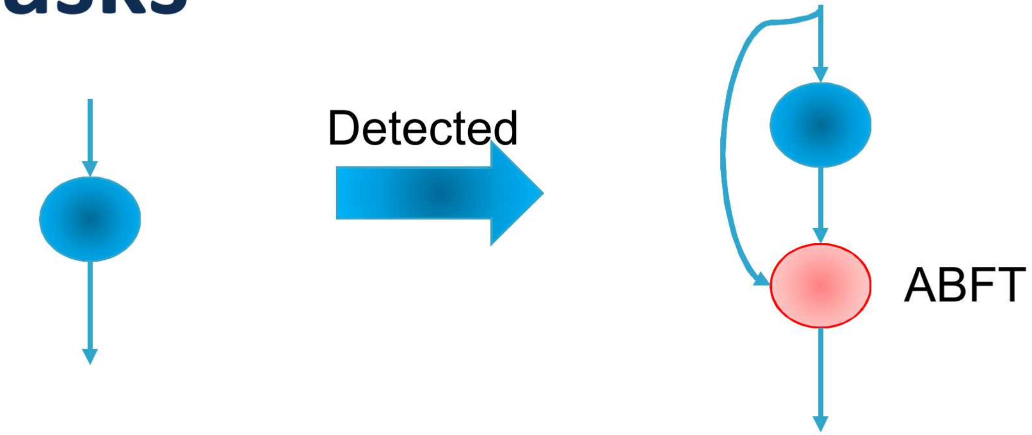
Task Replay



```
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);
```

- Dynamic response to failure
- Executes the task and checks for error using the error checking function
- error_check_fn(params) returns true if there is no error
- The task is executed **N** times at most if there is any error
  - If error checking fails, puts are ignored and the error is reported using an output promise

ABFT Tasks

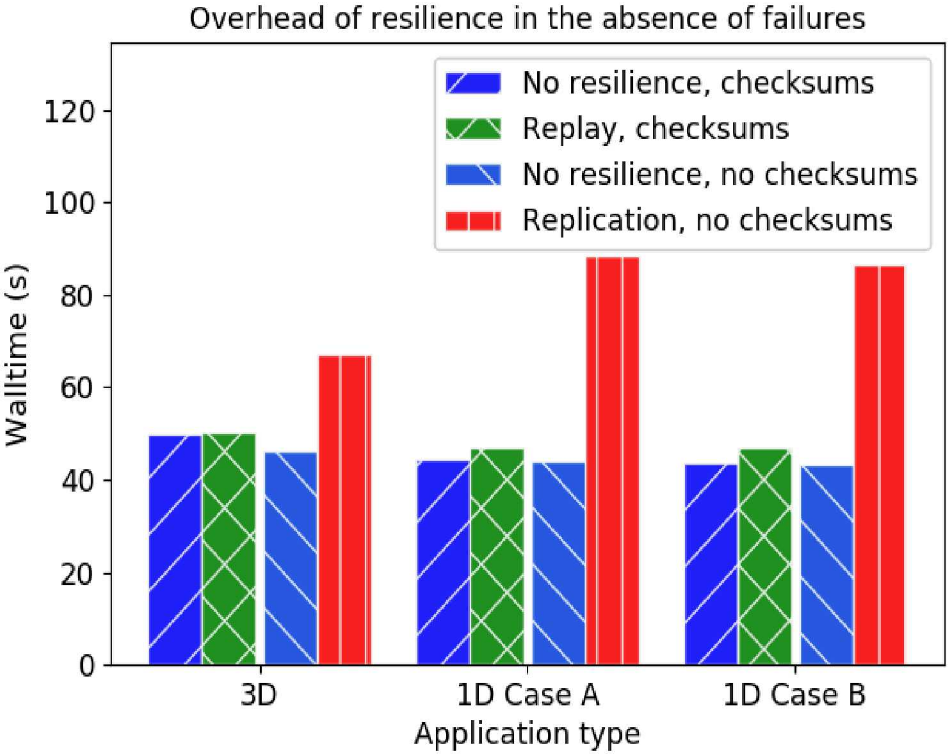


```
abft::async_await_check ( lambda, hclib::promise<int>
out, std::function<int(void*)> error_check_fn, void *
params, hclib_future_t *f1, .. , hclib_future_t *f4,
ABFT_lambda);
```

- Executes the task and checks for error using the error checking function
- error_check_fn(params) returns true if there is no error
- If there is error then **ABFT_lambda** is executed and checked for error again at its end
  - If error checking fails, puts are ignored and the error is reported using an output promise

Resilience Overhead in the absence of failure

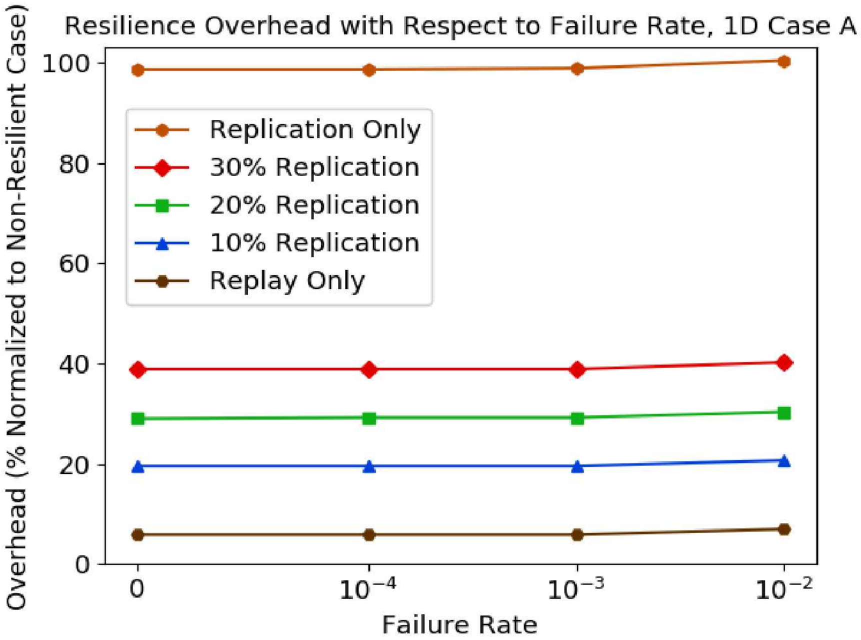
- Replay is less expensive
  - 7%-9%
- In the 1D cases, replication doubles the execution time. (+101%)
- In the 3D cases, the replication penalty is about 45%.
  - More L3 cache hits are observed



Resilience with synthetic failure injection

- Test a range of task failure rate (0.01%-1%)
- Failure is detected as checksum error (replay) or different results from the first two tasks (replication)
- We applied mixed mode so that the last X% of iterations are replicated, and replay is applied to the first (100-X)% of iterations.
- The performance numbers **from replay-enabled code with no-failure** are fed to our resilient-AMT simulator to predict the execution time with different task failure rate.
  - Overhead of replay and replication are based on the cost task.

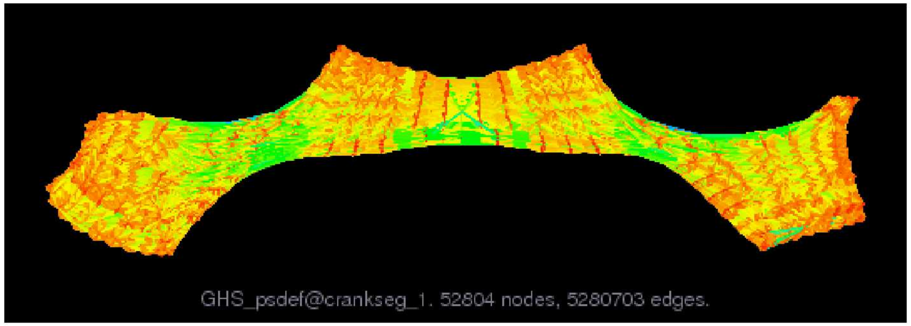
Resilience with synthetic failure injection (1D Stencil, 128 tiles of 16000 doubles)

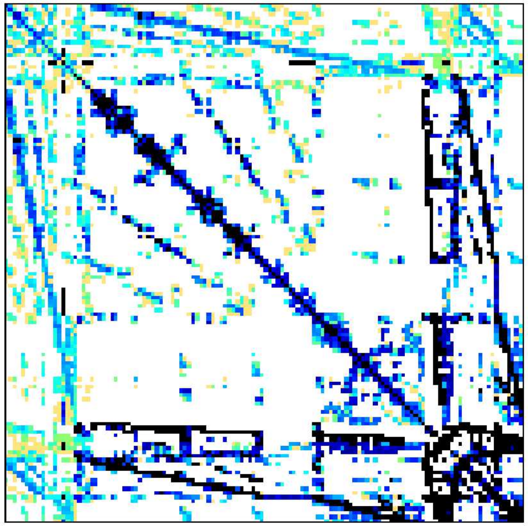


- Slight increase in the wall time with the increase of task-failure rate.

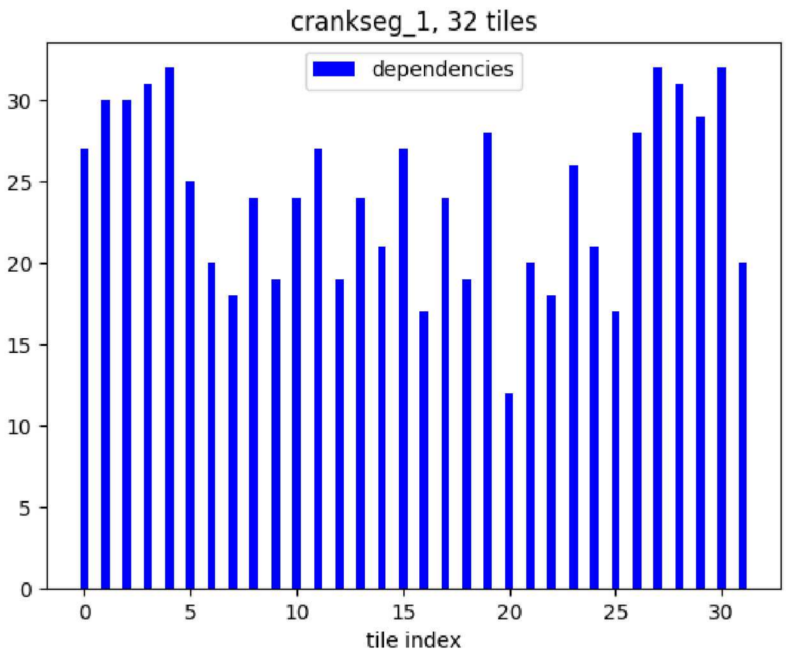
Explicit PDE Solver for Unstructured Mesh

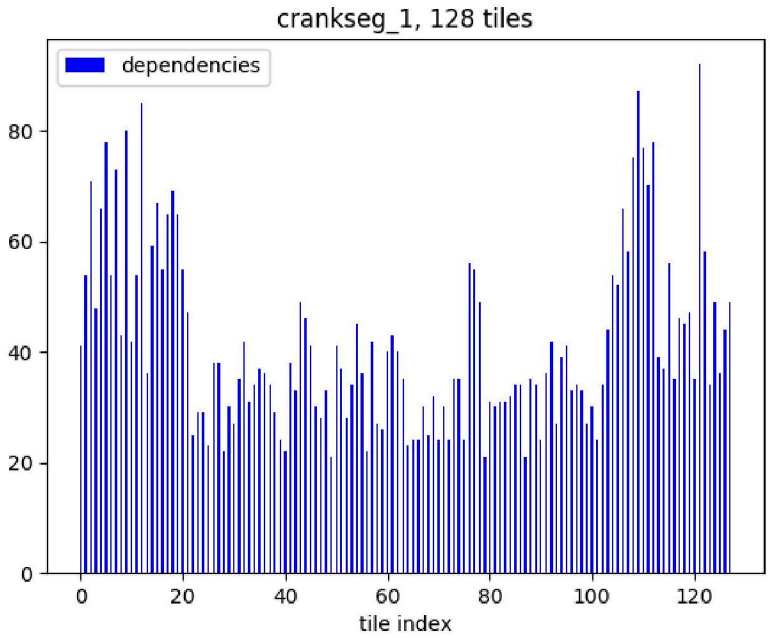
- Repetition of Task based SPMV
- Evaluated crankseg_1 matrix from SuiteSparse web site at Texas A&M.
- Tried 32 and 128 tile cases
    - No overdecomposition
    - Overdecomposition by the factor of 8
- 500 hundred iterations



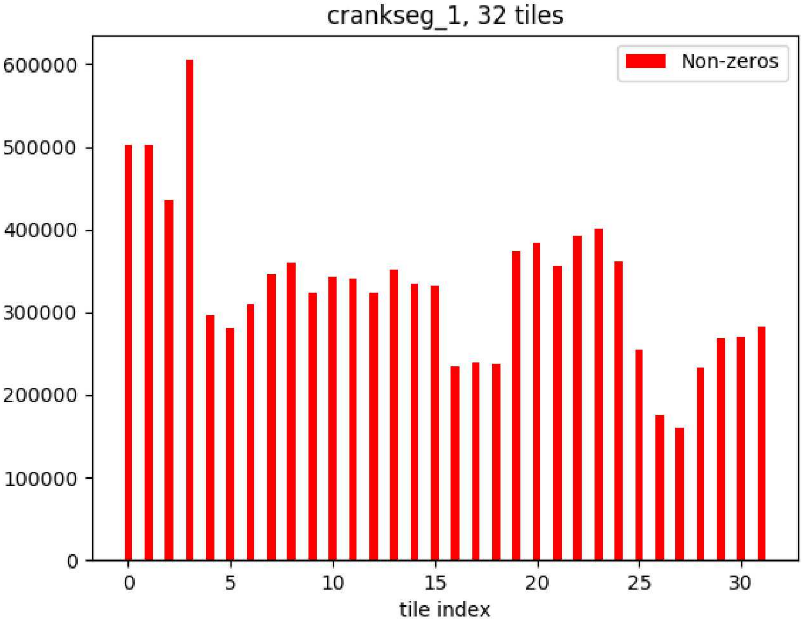


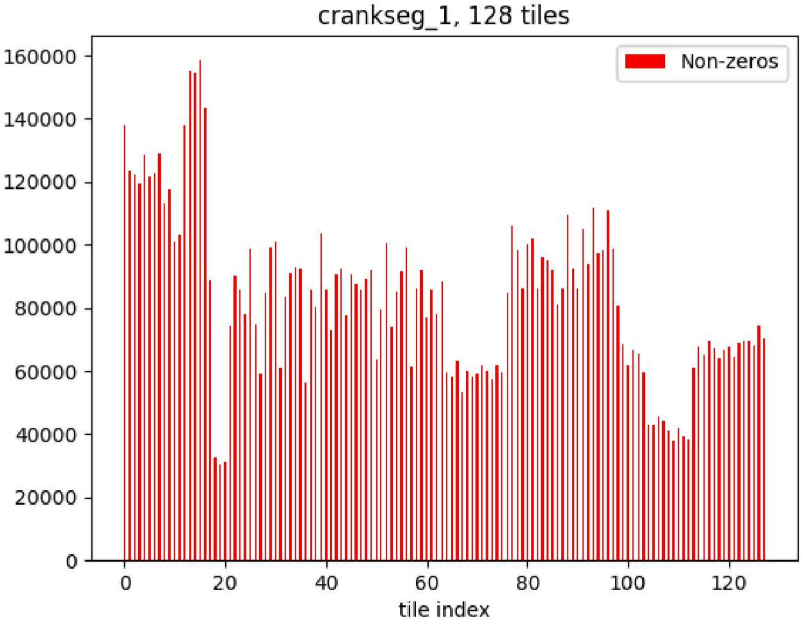
Irregular distribution of task dependencies



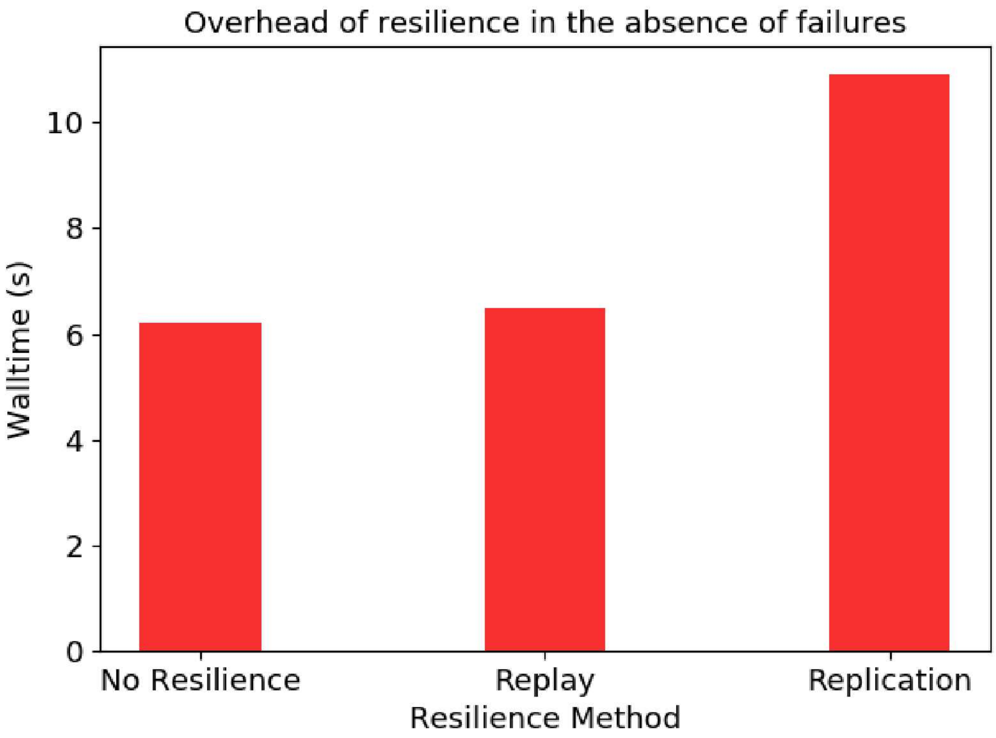


Irregular distribution of nonzero entries per task



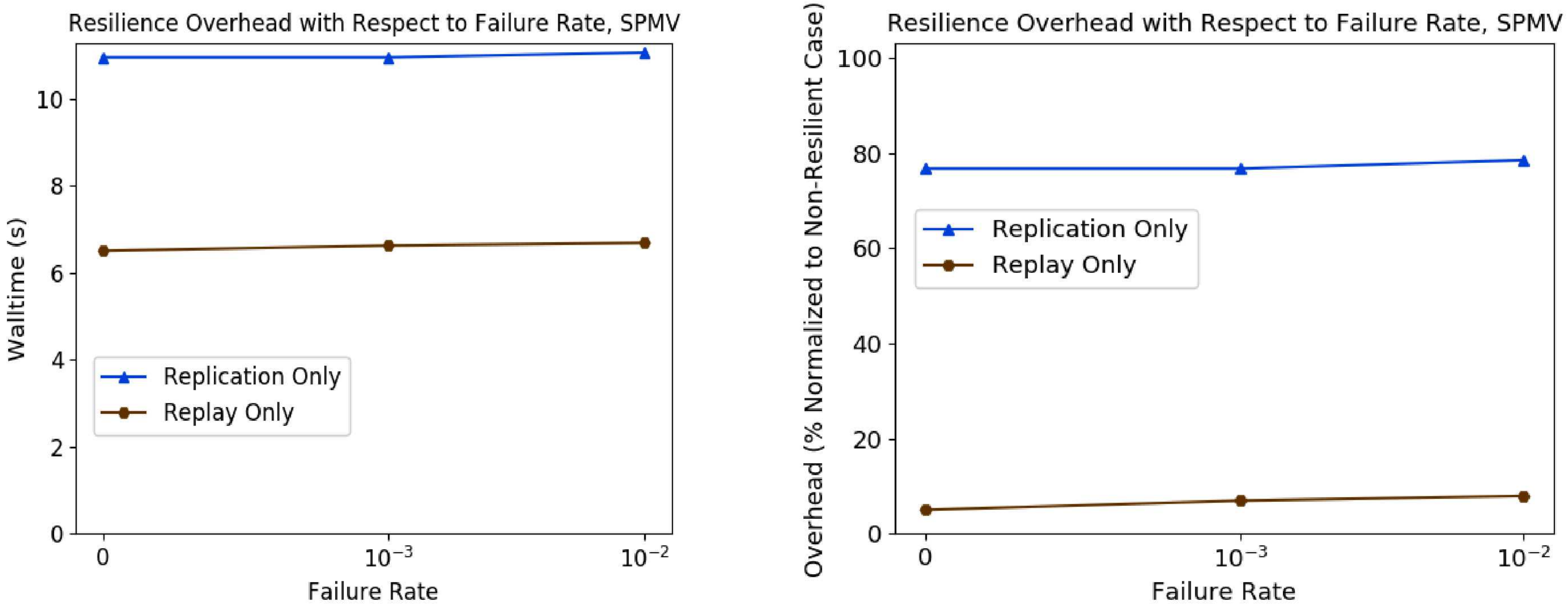


Overhead of Resilience Techniques in the absence of Failures



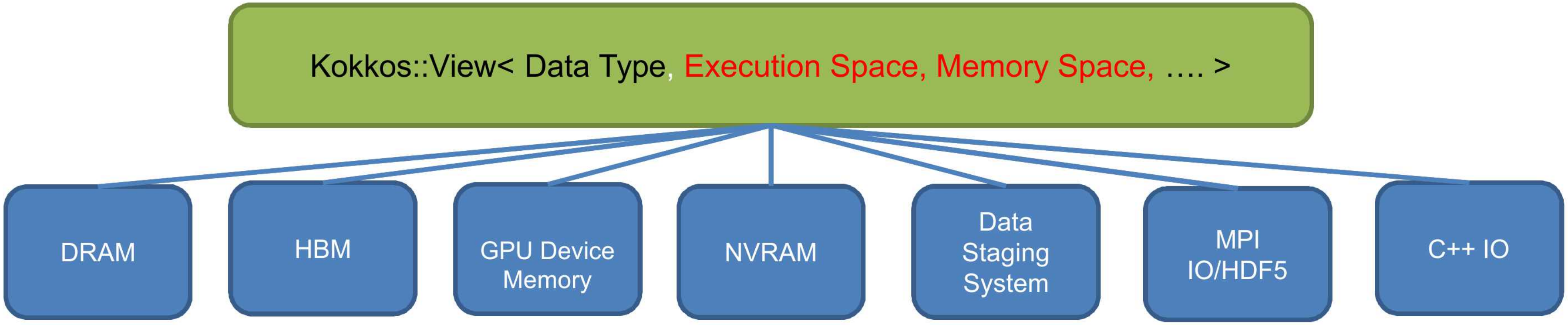
- Approximately 5% of overhead to enable replay.
- Replication doubles the execution time.

Execution Time under synthetic failures



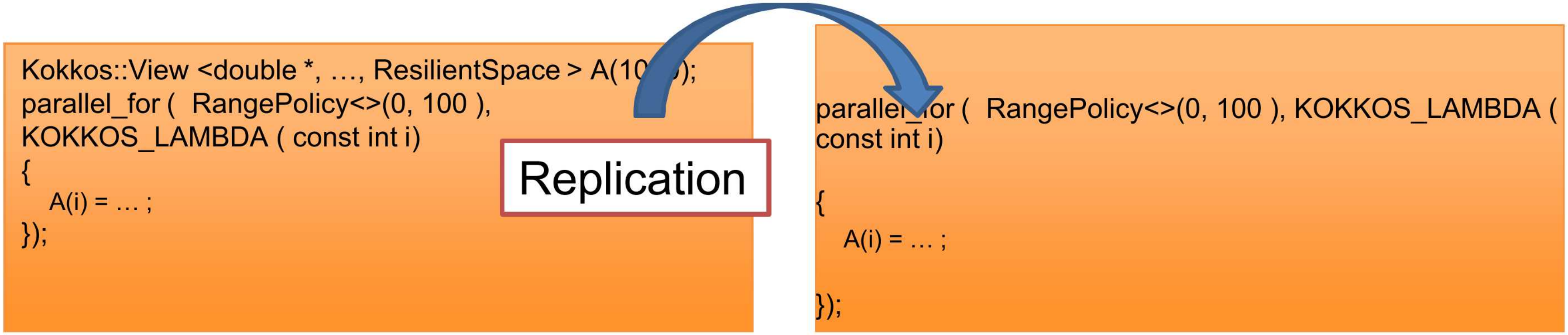
- Slight increase in the execution time.
- Tasking can hide the delay due to failures.

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 and Memory Spaces** to provide performance portability over multiple node architecture
  - Exploit C++ Lambda to support parallel program execution
- Resilient Kokkos provides “resilient” data and execution spaces to enable resilience/fault tolerance without major modification in application program source.

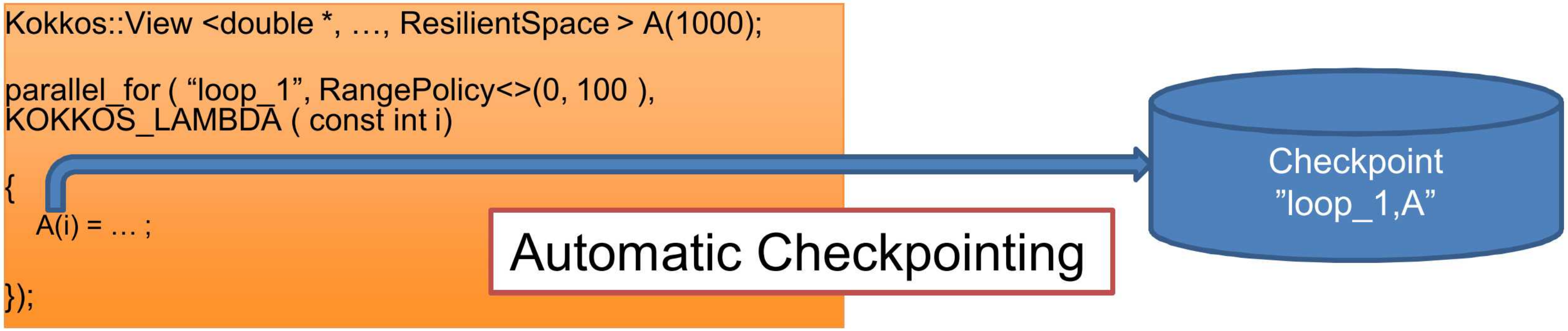
Resilient Kokkos enables resilient data parallel computation



```
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 <double *, …, ResilientSpace > A(1000);

parallel_for ( “loop_1”, RangePolicy<>(0, 100 ),
KOKKOS_LAMBDA ( const int i)

{
   A(i) = … ;

});
```

Automatic Checkpointing

Checkpoint
”loop_1,A”

CONCLUSION

Conclusion

- Discussed Resilient Programming Models for:
  - SPMD (MPI) Model
    - Online recovery
    - Fenix accommodates generalization of recovery using MPI-ULFM capability
  - Localized Recovery (Fenix-LR)
    - Exploit application’s (stencil) communication pattern to enable redundancy
    - Failure-Masking to hide the major recovery overhead
  - Asynchronous Many Task Programming Model
    - Resilience is embedded to the programming model itself.
    - Simple extension of tasking API to enable resilient computation patterns
  - Kokkos
    - Extend **Memory and Execution Space** concept to enable reslience in application data and computation

Acknowledgement

- Robert Clay, Hemanth Kolla, Michael Heroux, David Hollman, Jackson Mayo, Jeff Miles, Nicole Slattengren, Christian Trott, Matthew Whitlock (Sandia National Labs)
- Shaohua Duan, Mark Gamell (Ab-Initio LLC), Pradeep Subedi and Manish Parashar (Rutgers U.)
- George Bosilca, Aurélien Bouteiller and Thomas Herault (U of Tennessee)
- Seonmyeon Bak, Sri Raj Paul, Akihiro Hayashi, and Vivek Sarkar (Georgia Tech and Rice U.)
- Hartmut Kaiser and Adrian Serio (Louisiana State U.)

Q&A