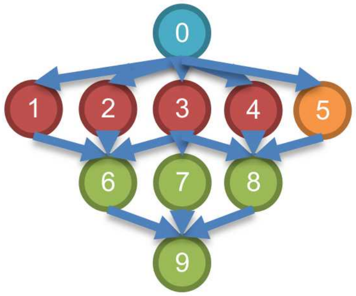
This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

SAND2019-2242C

Exceptional service in the national interest

Sandia National Laboratories



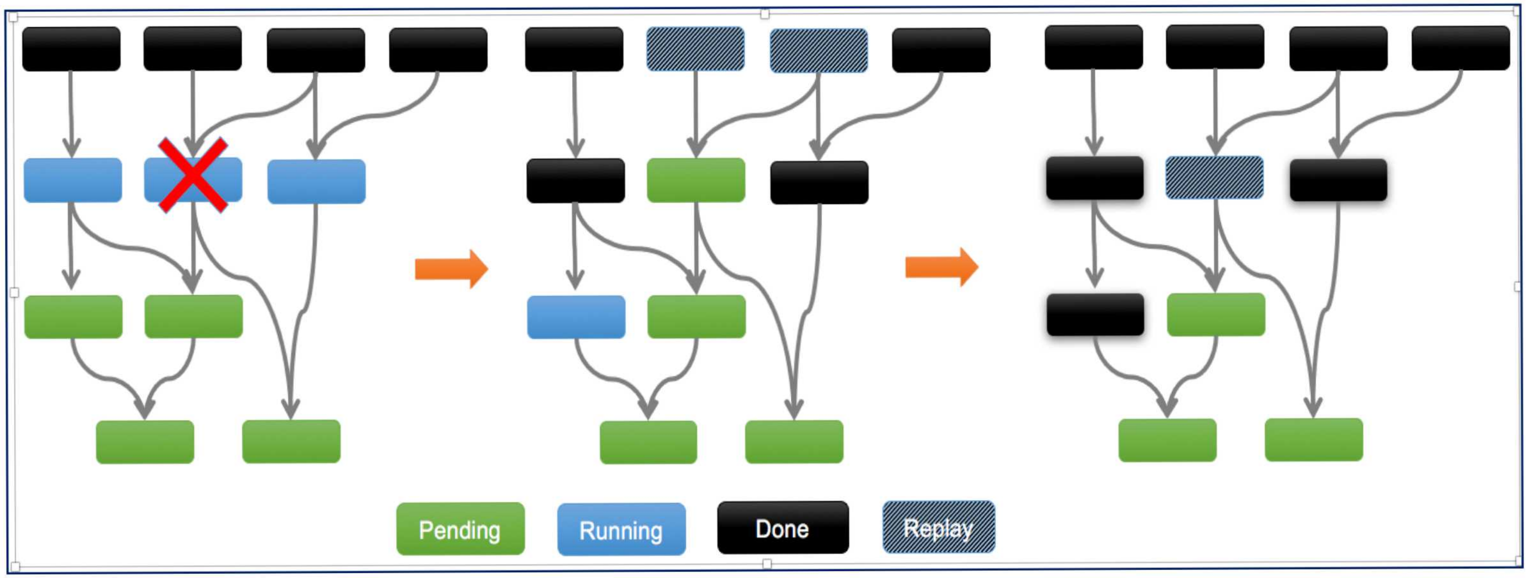
Scalable, Efficient Fault Tolerance in Asynchronous Many Task (AMT) Programming Models

SriRaj Paul, Akihiro Hayashi, Seonmyeong Bak, and Vivek Sarker
Georgia Institute of Technology
Keita Teranishi, **Hemanth Kolla**, Nicole Slattengren, Matthew Whitlock, and Jackson Mayo
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



- Substantial progress in resilience and asynchronous many-task (AMT) programming models, separately.
- AMT offer:
  - More flexible and efficient failure mitigation compared to conventional (e.g. checkpoint) strategies.
  - Ability to quantify the effects of failures and benefits of various resilience strategies.
- ***Complex tradeoffs of multiple AMT resilience techniques with dynamic failure behavior*** need to be understood/documented**.**
- Need ability to extrapolate tradeoffs to extreme(exa)-scale.

Objectives

Sandia National Laboratories

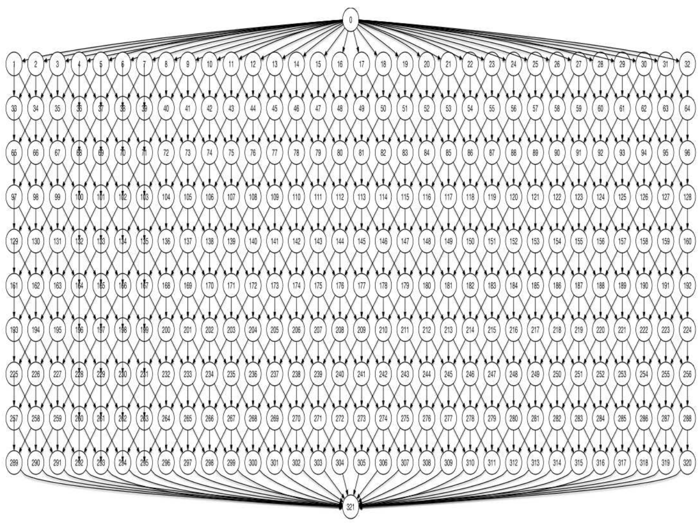
- An analysis of the scalability, performance and costs for multiple AMT resilience options.
- Prototype implementation of resilience schemes in actual asynchronous many-task programming model:
  - task replication.
  - task replay.
  - algorithm-based fault tolerance.
  - task-level checkpointing.
- An analysis of accuracy-cost tradeoffs of application-specific failure detection and mitigation schemes.
- Use representative mini-apps as basis for study.

Current Scope: On-Node AMT

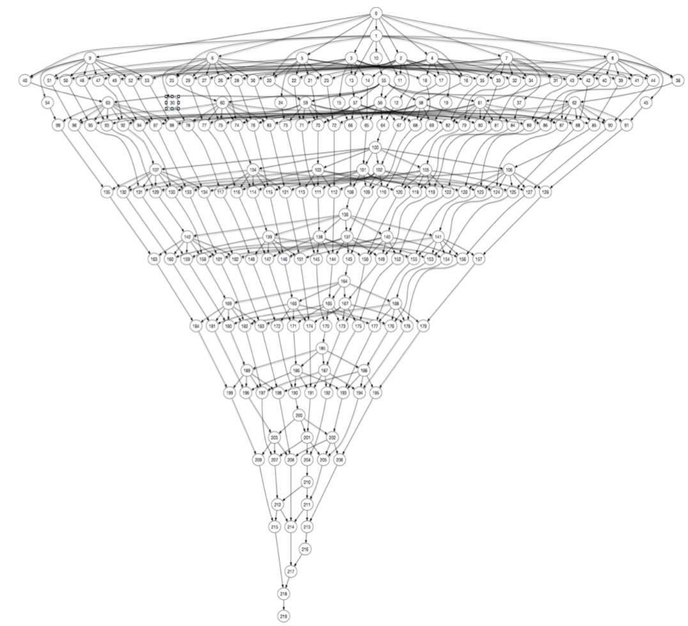
- MPI+(on-node) AMT an anticipated programming model for future complex node architecture.
- First comprehensive study with on-node AMT. Extend concepts to distributed AMT in future.
- Analyzing Failure/Error mitigation by On Node AMT is essential:
  - Hard failure of cores and accelerators, silent errors, performance degradation.
  - Failure can be manifested as task failure: non-finishing tasks, data corruption or very slow execution.
- We still need better understanding of failure-free AMT as a baseline, production-ready distributed AMT is scant.

Abstract Model of AMT Program

Graph representing 1D stencil Computation



Graph representing dense Cholesky Factorization



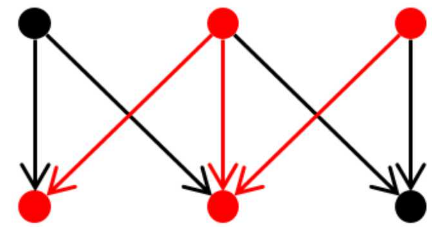
- AMT program execution can be graph and traverse it from the root.
- We started to investigate any analytical model to derive the performance and reliability.

Survey of DAG Analytical modelling

Analytical modelling intractable even for simple scientific task graphs like stencil 1D

- Conducted a survey on analytical modelling of directed acyclic graphs.
- Several papers addressed **series parallel graphs (SPG)** or their variants to derive the execution cost and reliability analytically
  - Requirements for being SPG specifically forbids an **N**-shaped subgraph. Unfortunately, even for **the simplest 1D-stencil task-DAG**, this is violated, and the N-subgraph occurs repeatedly.
  - If a graph is not SPG, the model has high complexity (#P) to compute.

Task-DAG for 1D-stencil Program



[1] R.A. Sahner, K.S. Trivedi, 1987, **IEEE Transactions on Software Engineering, vol. SE-13, no. 10, pp: 1105-1114.**
[2] J. Valdes, R.E. Tarjan, E.L. Lawler, 1982, **SIAM J. Comput., vol. 11, no. 2, pp: 298-313.**

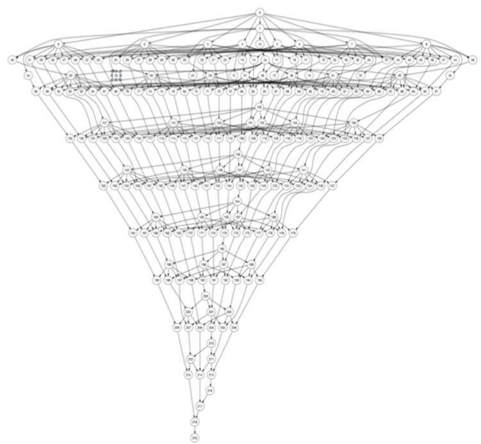
Alternate Solution: Task-DAG Simulator

Sandia National Laboratories

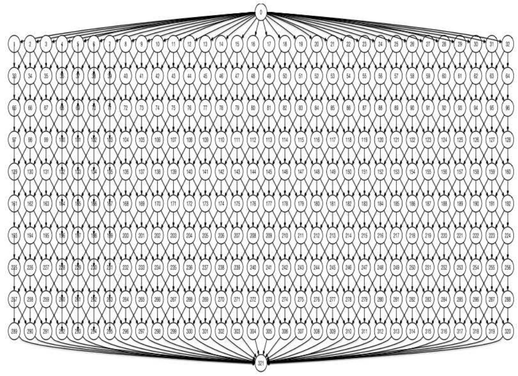
Task-DAG simulator hypothesizes the behavior of resilient AMT under numerous system and runtime situations

- Developed a tool to traverse task dags on multicore/multithreaded environment
  - 30+ Simulation Parameters including
    - # of threads
    - Scheduling
    - Task replay
    - Task replication
    - Checkpoint tasks (extra tasks inserted to take global state of data blocks)
    - Overhead for replay/replication
  - Emulate the scheduler of Habanero C++

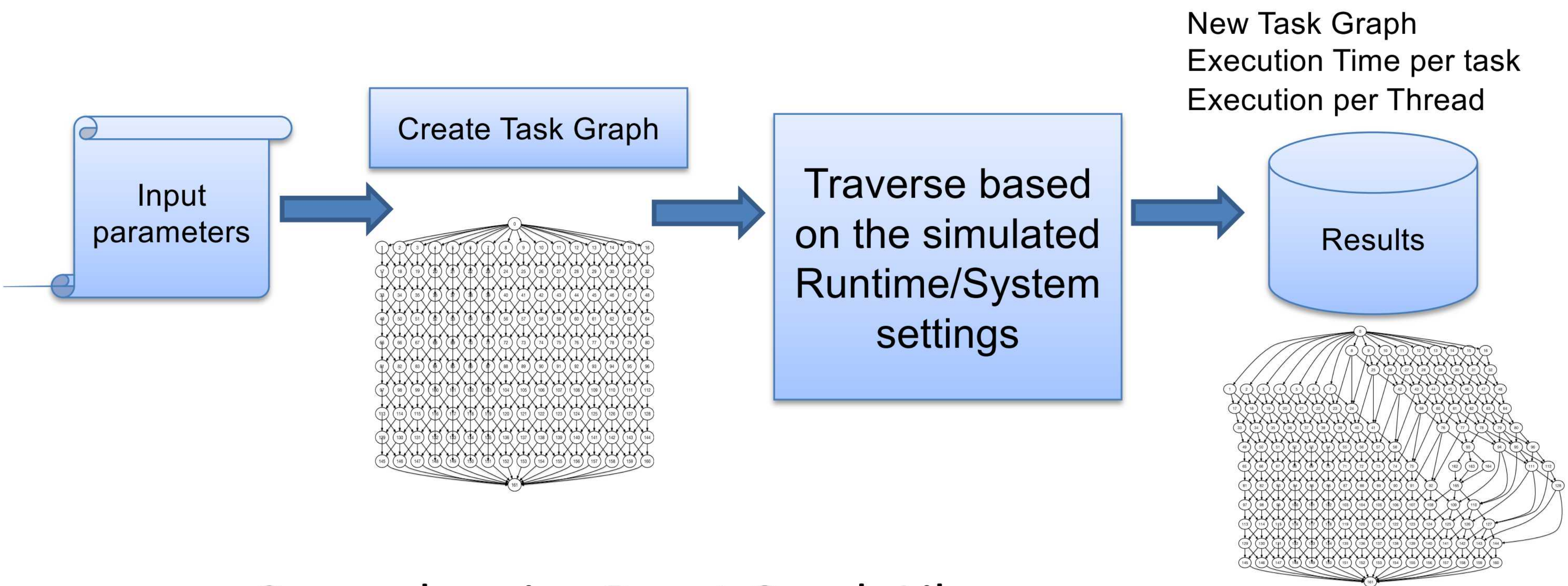
Graph representing dense Cholesky



Graph representing 1D stencil



Workflow of the Resilient-AMT Simulator

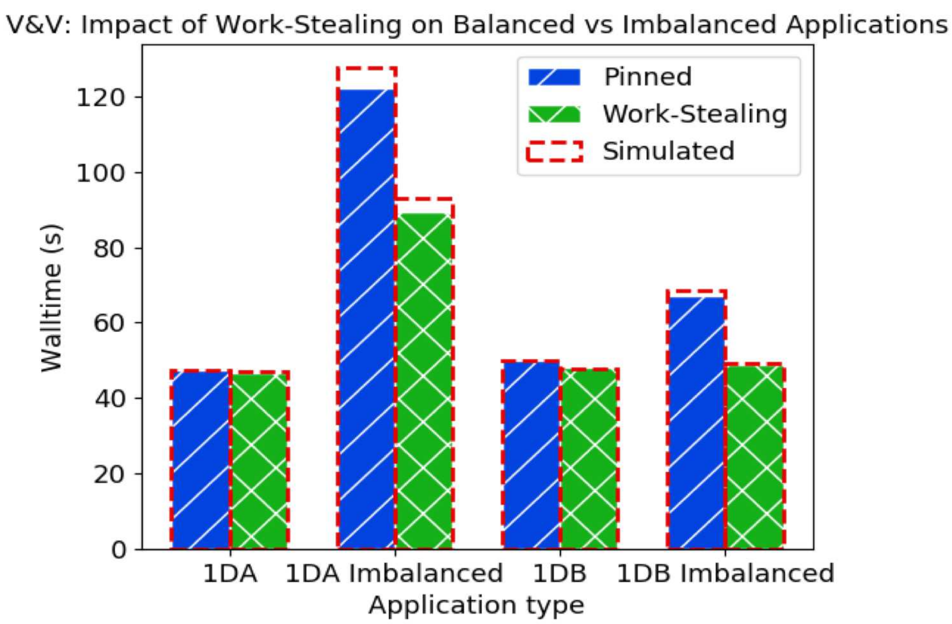


- C++ code using Boost Graph Library
- Python for visualization and organizing data

The Graph Generation Capability of the Resilient-AMT Simulator

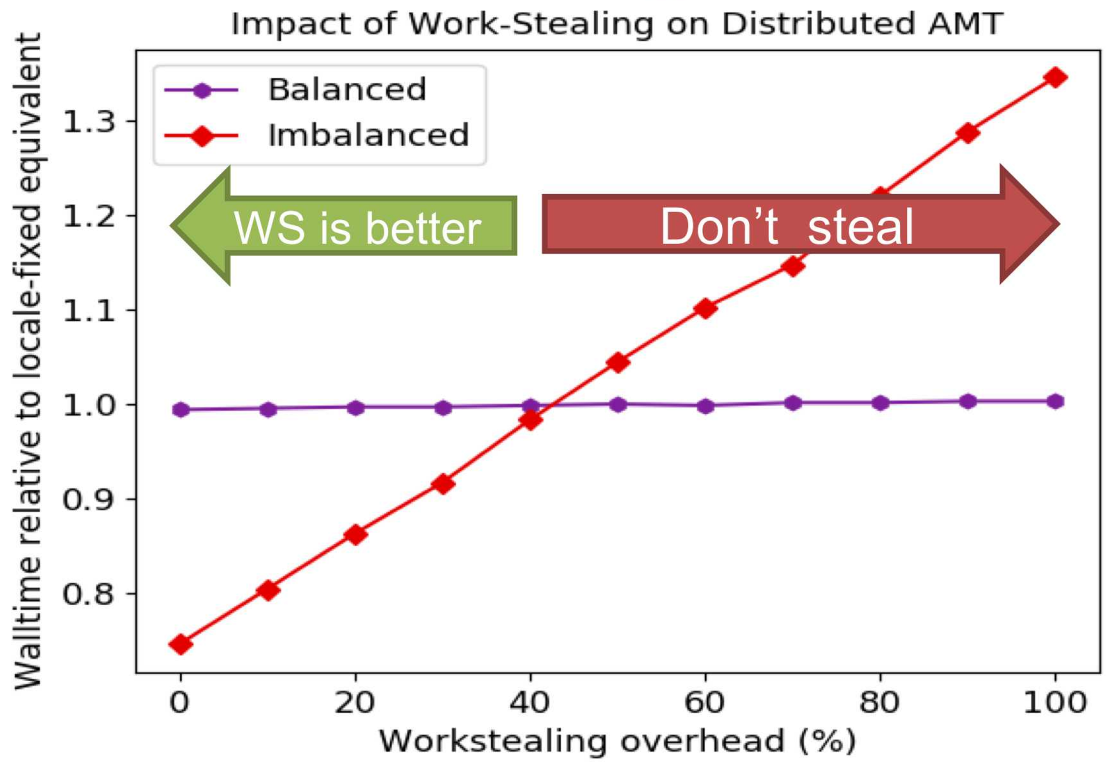
- Support generation of task graphs for:
  - 1D, 2D and 3D stencil code
  - Explicit PDE solver with unstructured mesh/arbitrary graph
  - Dense Cholesky Factorization
  - User can provide any task graphs as input files.

Simulator can predict the performance of the code in faulty situations.



- Obtain simulation parameters **just from non-resilient (no WS) executions.**
- Simulator runs the same task graph of the original program with specified resilient-AMT options.
- Accurately predicts the performance of task-replay resilience.
- Needs more rigorous performance model to simulate replications.

Simulator can explore hypothetical distributed AMT settings



- Distributed AMT settings on 8 nodes. 32 core per node.
- Overdecomposed 1D Stencil Problem
- Imbalanced Case: 10x single slow task in a single time step
- X-axis indicates the work stealing overhead relative to task execution time.

Resilient AMT Prototype

- Resilience Extension of Habanero C++
  - AMT programming Interface by Vivek Sarkar
- Simple extension allows the user to introduce 3 major resilient proguram 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++ Overview

- 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

Habanero-C++ Overview

Sandia National Laboratories

HClib constructs

Description	Example
Asynchronous task creation	`async(() -> { S1; });`

Bulk task synchronization	`finish(() -> {`
	`    async(() -> { S1; async(() -> S2;); });`
	`});`

Futures and promises	`async(() -> { prom->put(42); });`
	`async(() -> { prom->get_future()->wait(); });`
	`async_await(() -> {…}, prom->get_future());`

Bulk task creation	`forall(loop, (i, j, k) -> { S3; });`
Places for locality control	`async_at(pl, () -> { S4; });`

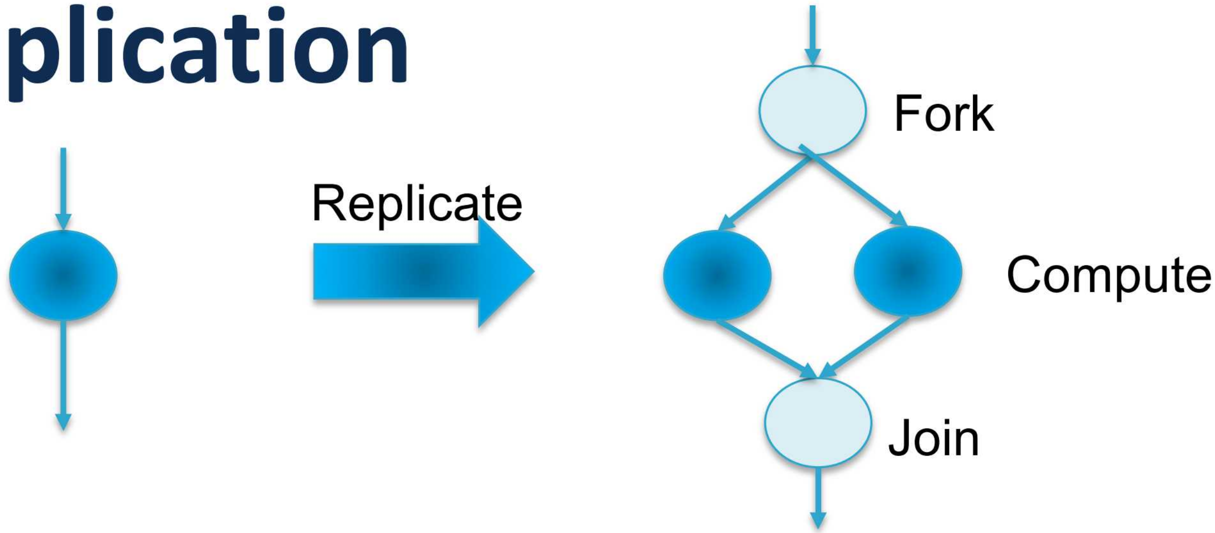
Habanero-C++ Overview

- 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 future and promise
  - **`future = promise.get_future()`**
  - If accessed from different threads **`put()`** and **`get()`** are synchronized thus enabling a way for synchronization.

HClib extension: (1) Reference Counting

- Current implementation leaves it to user to manage dynamic allocated memory (no automatic garbage collection).
- Reference counting semantics:
  - Provide a way to perform garbage collection based on the use of future as task dependency
  - Allows transparent handling of data access by replay/replicated tasks.
- Implementation extends **`promise`** to have a reference count
  - Count set during object construction
  - Count decreased using **`release()`** method
- Extend **`async_await`** to perform automatic reference counting
  - Reference count is decreased each time a future associated with the promise is used as dependency

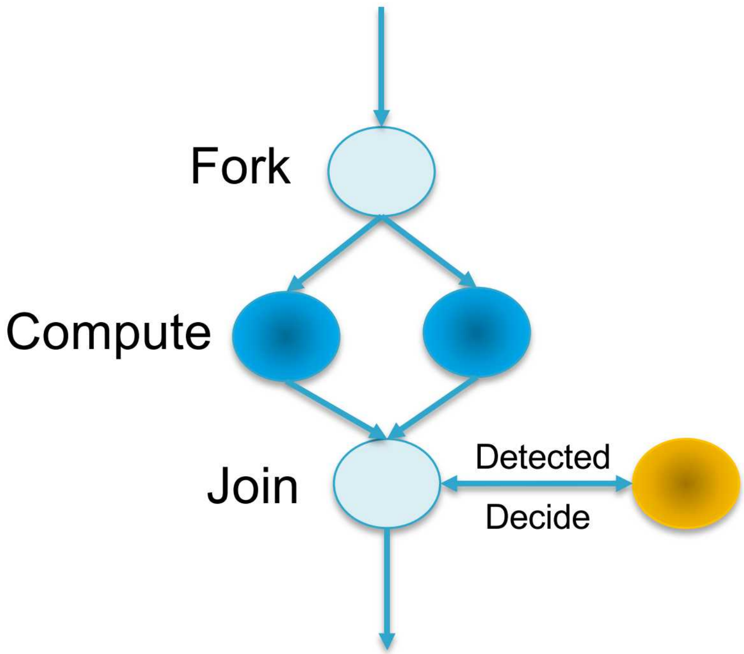
Task Replication



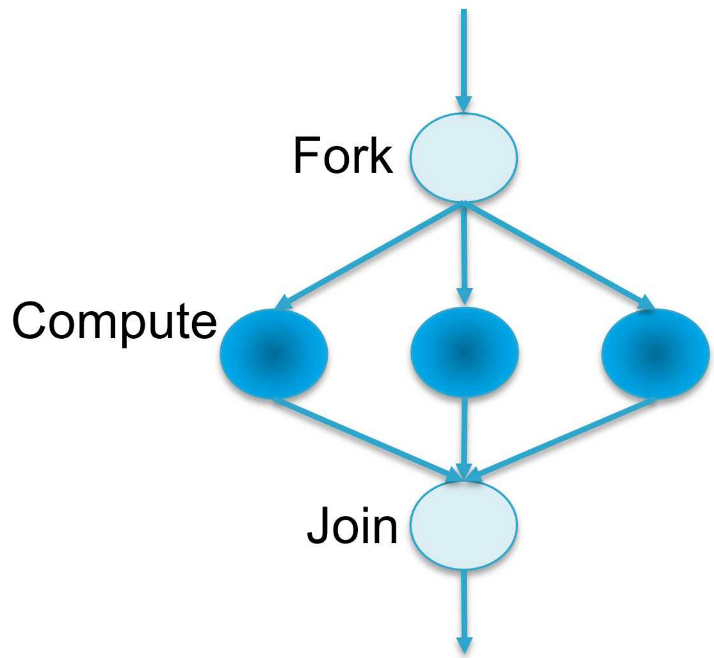
- `diamond::async_await_check<N> ( lambda, hclib::promise<int> out, hclib_future_t *f1, .., hclib_future_t *f4);`
  - Preventive failure mitigation
  - N-plicates the task and checks for equality of put operations at the end of the task
  - If error checking succeeds, actual puts are done
  - If error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

Replication (Continued)

`diamond::async_await_check<2>(…`

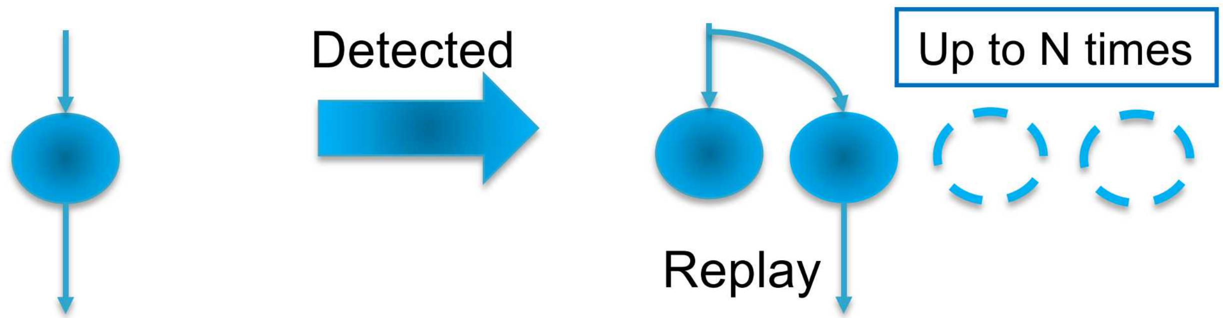


`diamond::async_await_check<3>( …`



- 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

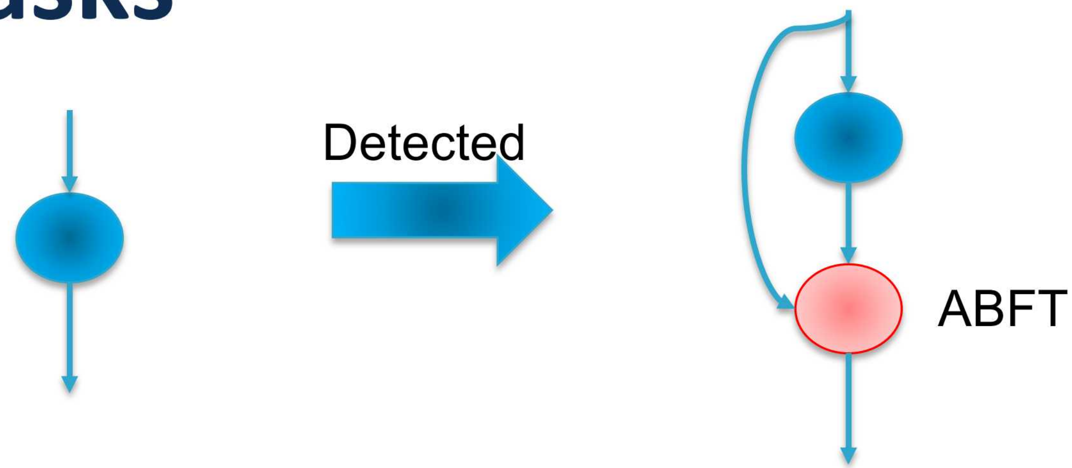
Task Replay




```
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

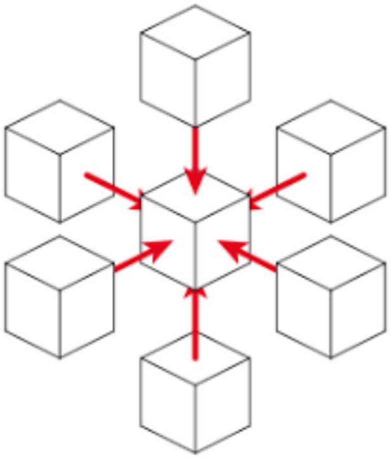
ABFT Tasks

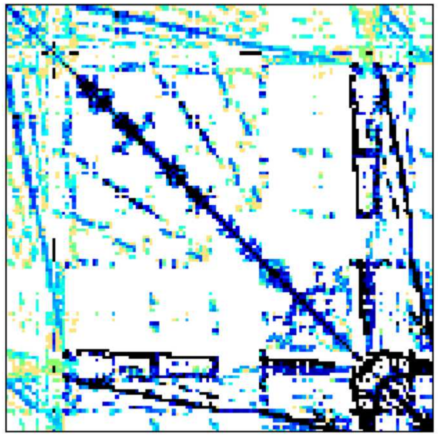


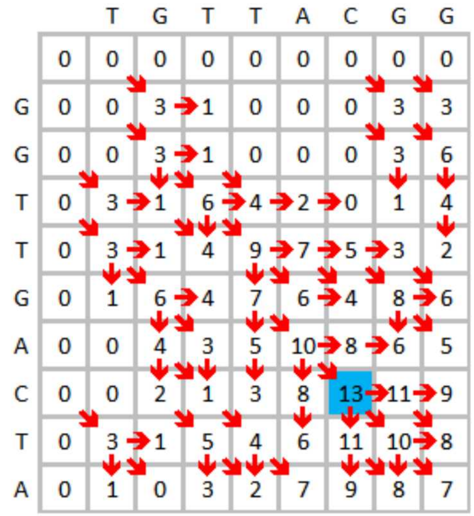

```
abft::async_await_check ( lambda, hclib::promise<int>
out, std::function<int(void*)> error_check_fn, void *
params, hclib_future_t *f1, .. , hclib_future_t *f4,
ABFT_lambda);
```

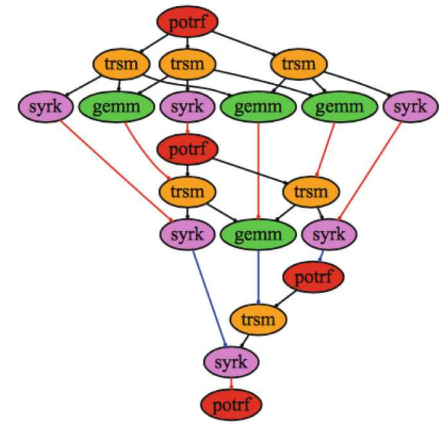
- Executes the task and checks for error using the error checking function
- error_check_fn(params) returns true if there is no error
- If there is error then **ABFT_lambda** is executed and checked for error again at its end
  - If error checking fails, puts are ignored and the error is reported using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

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

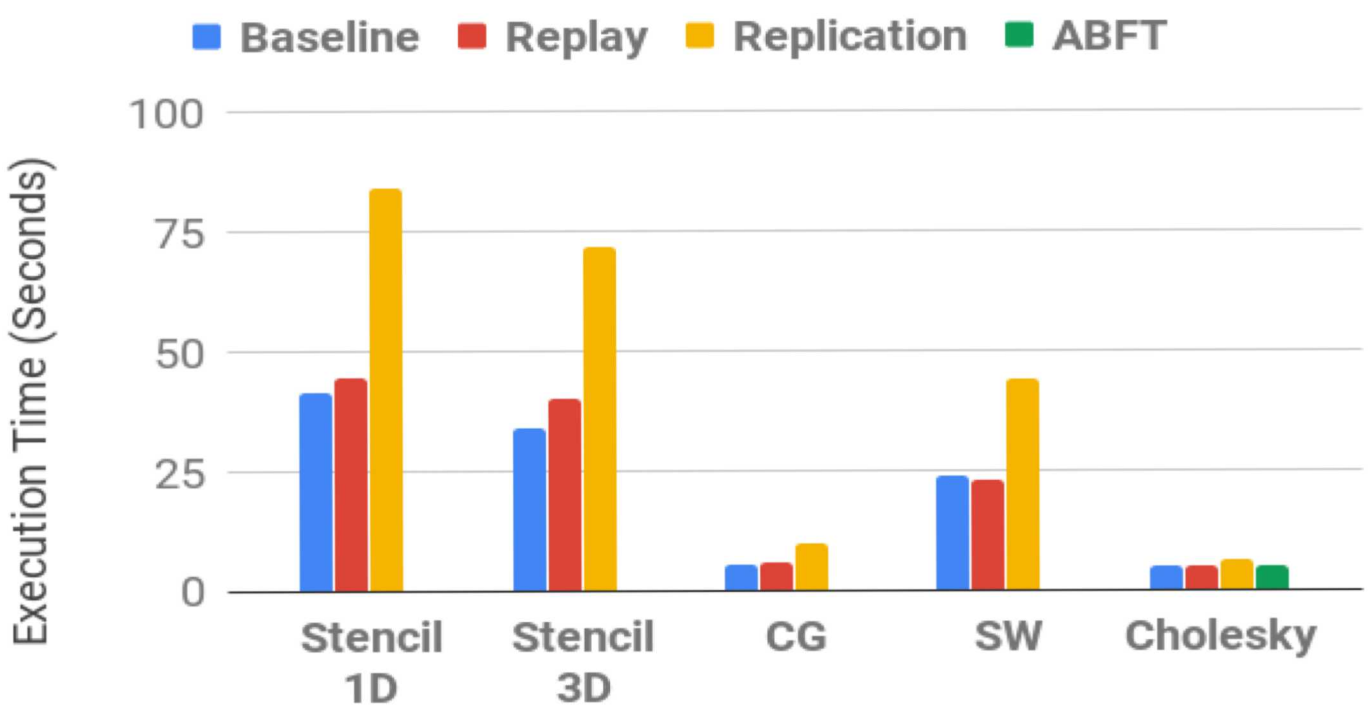
Replay and replication do not double the memory overhead

	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

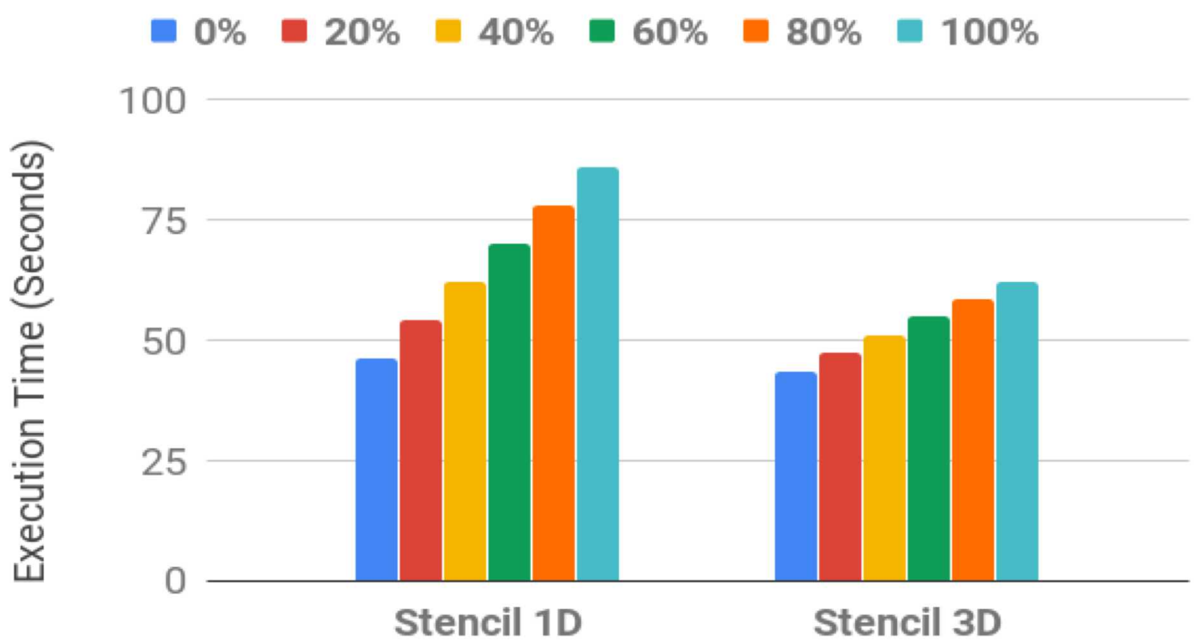
- Tested on NERSC's Cori (2 Haswell CPUs, 32 cores total, 2.3GHZ) system

Performance without faults



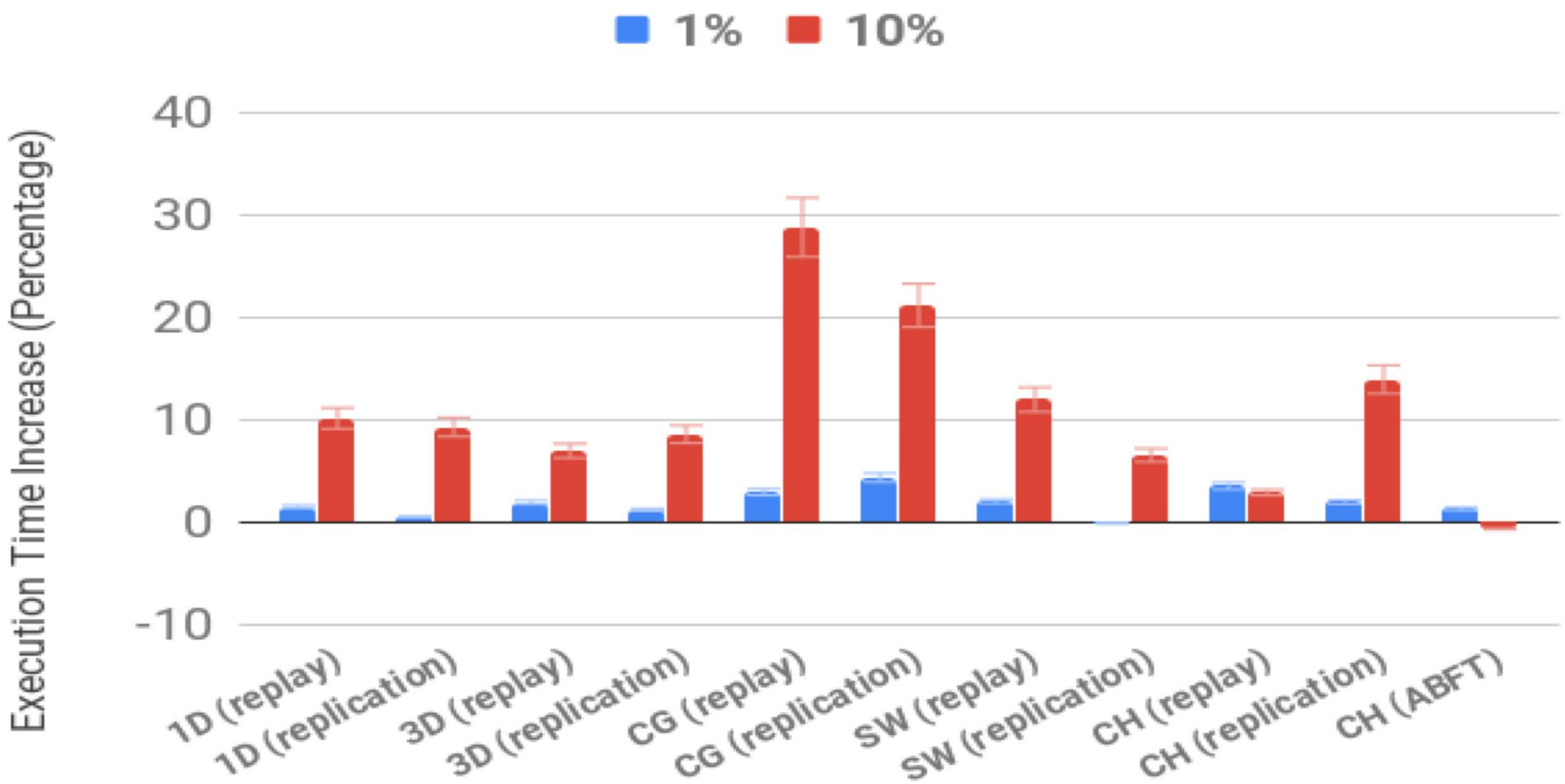
- Replication is expensive for 1D stencil, CG and SW.
- Observed some cache hits with 3D stencil
- High cache hits and critical path in task-base Cholesky suffers less replication overhead

Mixing replication and replay

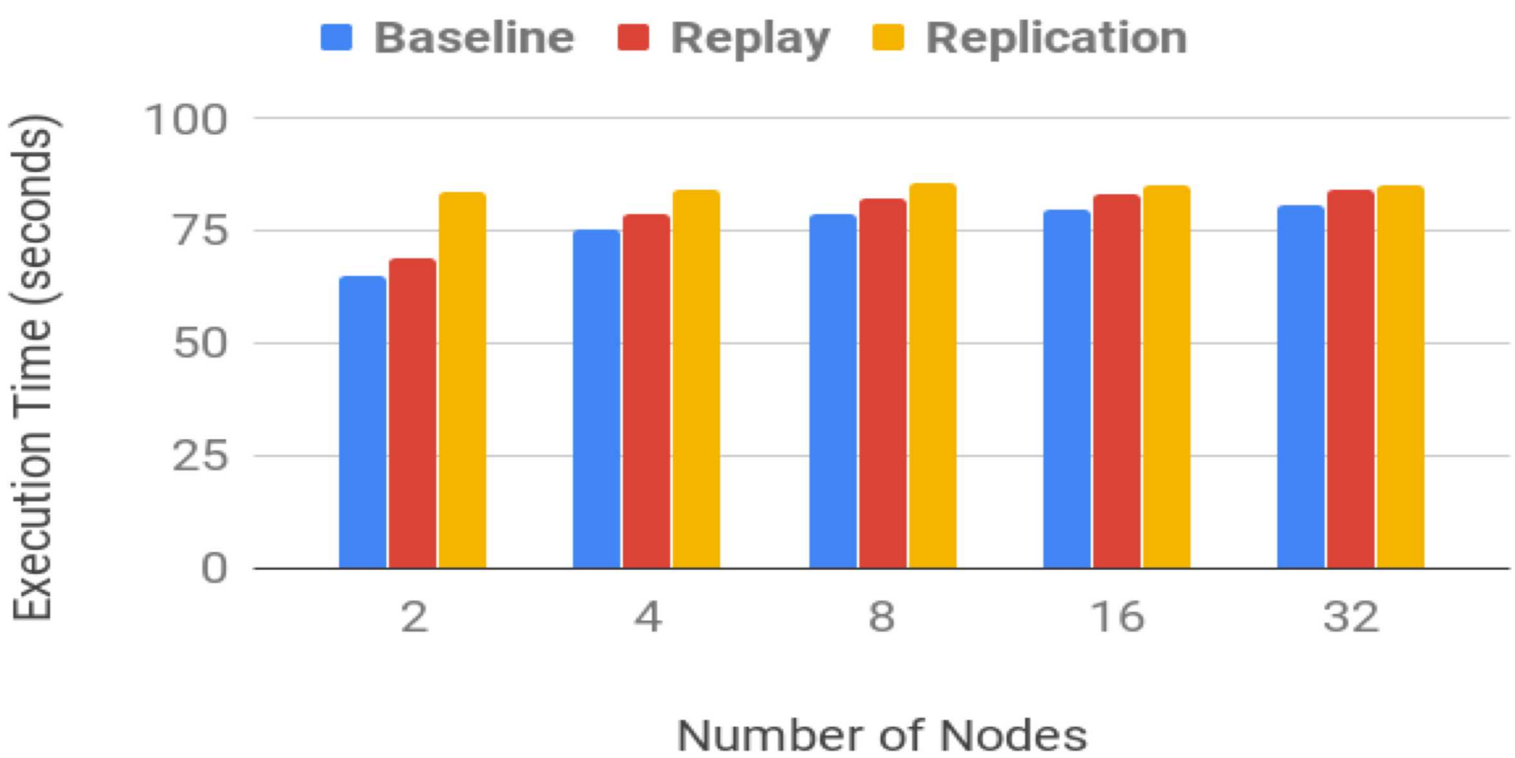


- Replication doubles the execution time of 1D case.
- We observed many L3 cache hits in the 3D case.
  - Less overhead for replication

Application delay is proportional to the # of failures

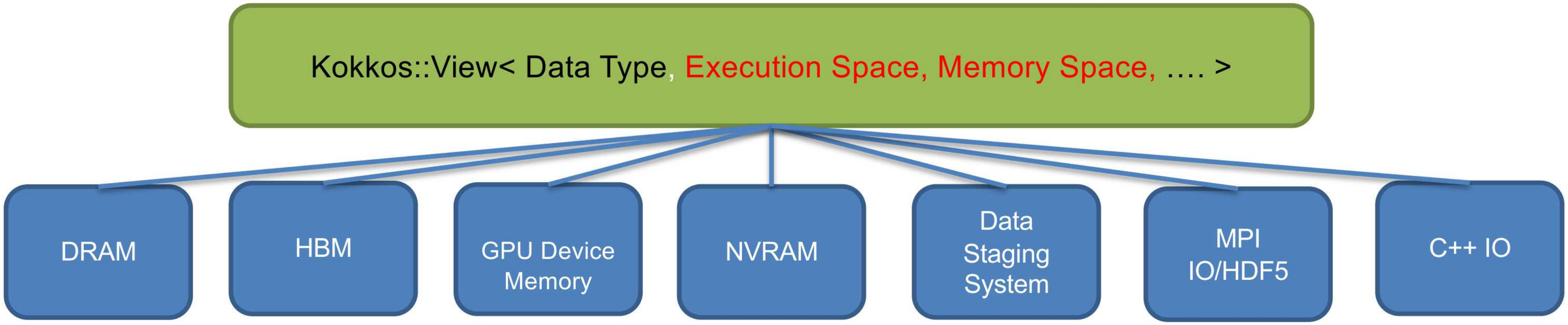


Scalability of 3D stencil code (MPI+Reslinet HCLIB)



- MPI-HCLIB implementation (1D, Weak scaling, over-decomposed)
  - No failure
  - 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 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 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

- Discussed Resilient Programming Models for:
  - Asynchronous Many Task Programming Model
    - Analytical model
    - Simulator based study
    - 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

Q&A