SAND2014-16454PE

Sandia National Laboratories

Exceptional service in the national interest

DHARMA: Distributed asyncHronous Adaptive Resilient Management of Applications

Janine Bennett, Robert Clay, John Floren, Ken Franko, Saurabh Hukerikar, Samuel Knight, Hemanth Kolla, Greg Sjaardema, Nicole Slattengren, Keita Teranishi, Jeremiah Wilke

Programming Models and Applications Workshop
August 5, 2014

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

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. SAND NO. 2011-XXXXP

Mission

- Assess & address fundamental challenges imposed by the need for performant, portable, scalable, fault-tolerant programming models at extreme-scale
- Two focus areas
  - Programming model analysis for next generation platforms
  - Demonstration of fault-tolerant programming model at extreme-scale

DHARMA is a fundamental Hindu concept referring to

- the order and custom which make life and a universe possible
- the behaviors appropriate to the maintenance of that order

The classical Sanskrit noun DHARMA derives from dhr

- meaning to hold, maintain, keep

Programming model analysis for next generation platforms

- Asynchronous many-task programming models are a leading new paradigm with many variants
- Goal: Address knowledge gaps
  - Comparative analysis of leading candidate solutions
  - Quantitative & qualitative tests using ASC-relevant codes
- Outcome: Guidance to code development road map for next generation platforms for ASC/ Integrated Codes

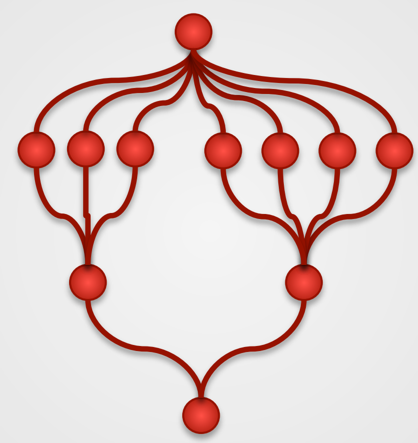
Runtimes
Uintah
Legion
HPX
Charm++
STAPL
StarPU
Swift/T

Assess
Scalability
Performance
Resilience
Interoperability with MPI

Demonstration of fault-tolerant programming model at extreme-scale

- Asynchronous many-task programming models
  - + Show promise at sustaining performance
  - + Work stealing enables load balancing
  - + Failed tasks can be re-executed
- Recovery (beyond checkpoint/restart) is challenging
  - – Distributed coherency problem
  - – Care is required to identify lost tasks due to work-stealing and asynchrony

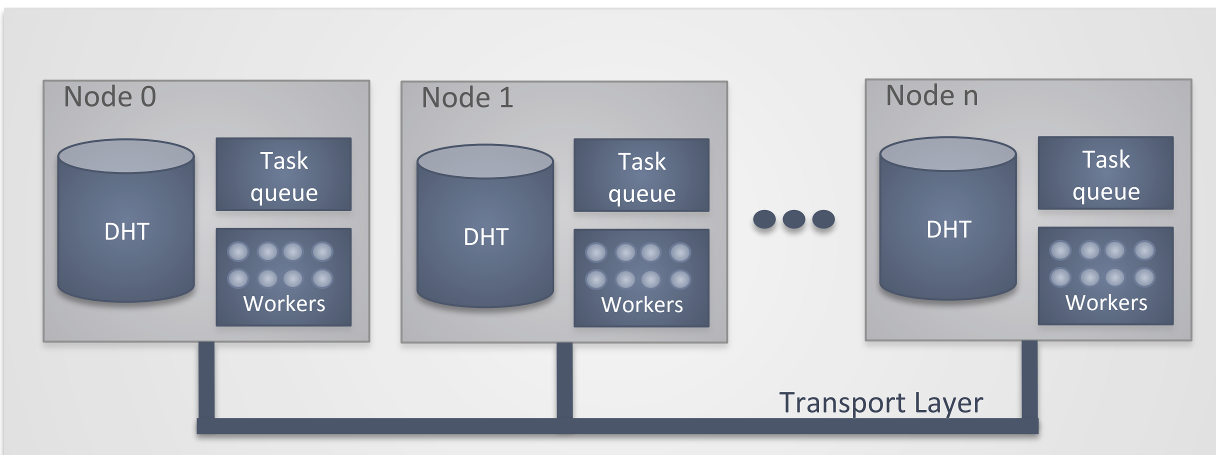
Task Graph



Nodes are tasks
Edges are data

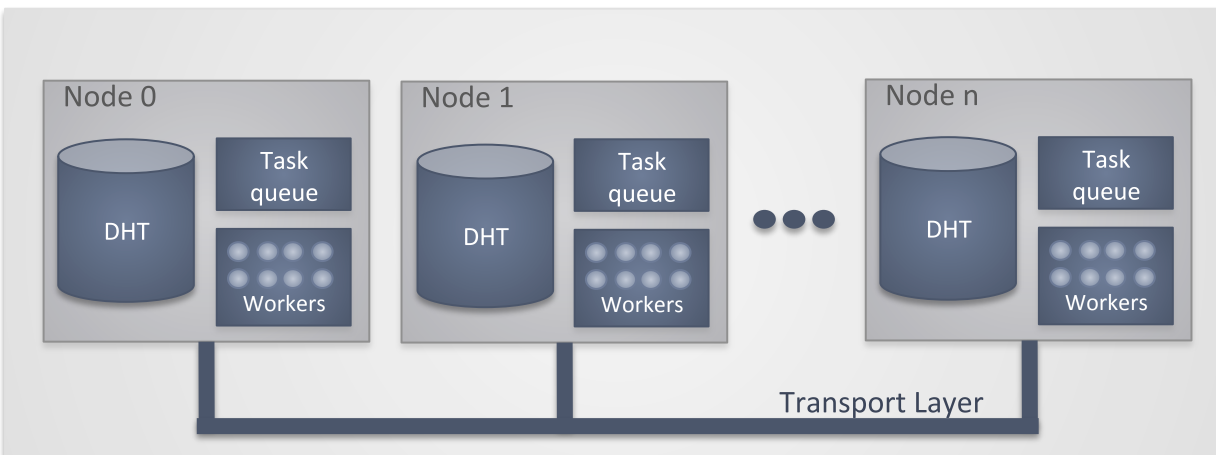
A holistic solution requires a number of fault-tolerant components

- Distributed Hash Table (DHT): Store task descriptors/data pointers
- Collection/task queue: Maintain state & work assignments
- Resilient Transport Layer
  - Fault-aware collectives: terminate cleanly with no result
  - Fault-tolerant collectives: heartbeat via overlay network to rigorously agree on which nodes are alive

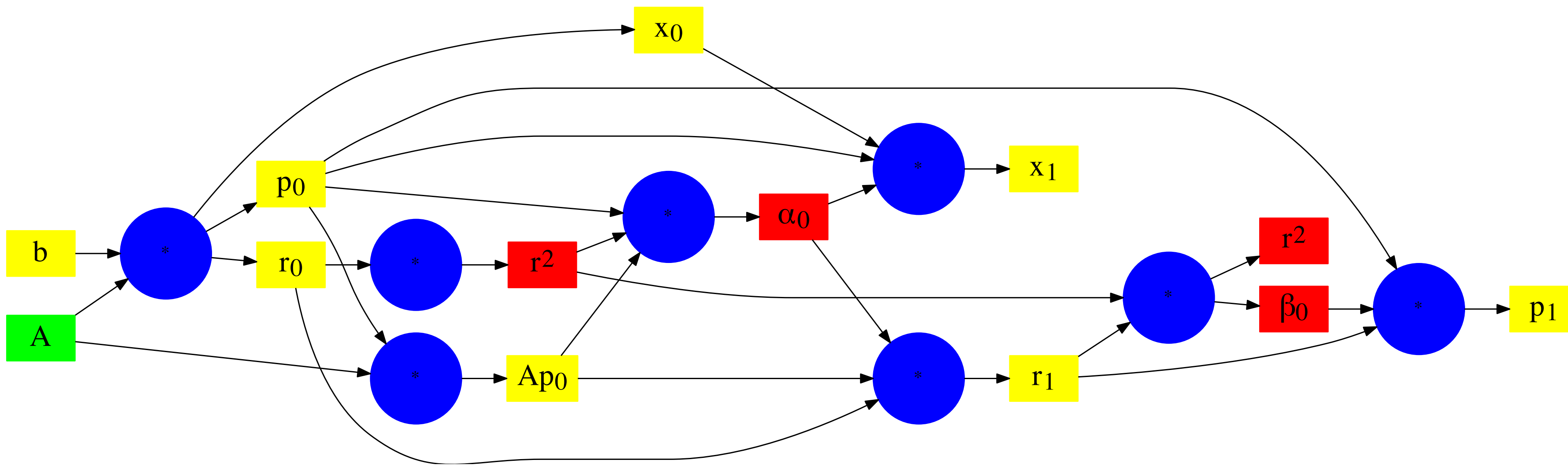


Related work

- Distributed Hash Table (DHT): Linda, Intel CnC, FOX, MATRIX
- Collection/task queue: Scioto, DAGuE, Legion, Uintah, Charm++
- Transport Layer: MPI-ULFM, FT-MPI, Hursey et. al “A log-scaling fault tolerant agreement algorithm for a fault tolerant MPI”



An example of a dense Conjugate Gradient (CG) task-graph



- Coarse-grained DAG + data parallelism
- Squares denote data (matrix/vector/scalar).
- Circles denote compute kernels.
- Data parallelism (matrix/vector) => large task parallelism.
- Each node (circle) in the coarse-grained DAG becomes a task collection.

Code example: Setting up runtime

```
void dharma_runtime::init()
{
 msg_api_    = new message_api(…);
 task_dht_   = new dht(…);
 mdata_dht_  = new metadata_dht(…);
 data_dht_   = new data_dht(…);
 backup_     = new nvram_backup(…);

 int max_steals  = 5;
 int eager_tasks = 100;
 queue_   = new task_queue(…, max_steals, eager_tasks);

 msg_api_->init();
}
```

Code example: Creating tasks

```
dharma_runtime* rt = new dharma_runtime;
rt->init(); //initialize the runtime
task_collection::ptr coll = new collection(rt,…, new generator(…));
rt->register_collection(coll);
.................................................
.................................................
void generator::generate_tasks()
{
 for (int i=0; i < overdecompose_; ++i){
    task::ptr t = new task(…);
    t->dependencies.push_back(new dependency(…)); //declare task deps
    append_task(t);  //adds the task to the collection
  }
}
```

Code example: Unrolling DAG

```
main
{
 dharma_runtime* rt = new dharma_runtime;
 rt->init(); //initialize the runtime
 cg_unroller starter(0,…); //start iteration 0
 starter.unroll(rt);
  ………………………………………………………………
}

void cg_unroller::unroll(dharma_runtime *rt)
{
  task_collection::ptr coll_Alpha_dp = new allreduce_collection(rt,…);
  task_collection::ptr coll_Alpha = new collection(rt,…);
  ………………………………………………………………
```

repeat

$$\alpha_k := \frac{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}{\mathbf{p}_k^{\mathrm{T}}\mathbf{A}\mathbf{p}_k}$$

$$\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k \mathbf{p}_k$$

$$\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k \mathbf{A}\mathbf{p}_k$$

if r_{k+1} is sufficiently small then exit loop

$$\beta_k := \frac{\mathbf{r}_{k+1}^{\mathrm{T}}\mathbf{r}_{k+1}}{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}$$

$$\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$$

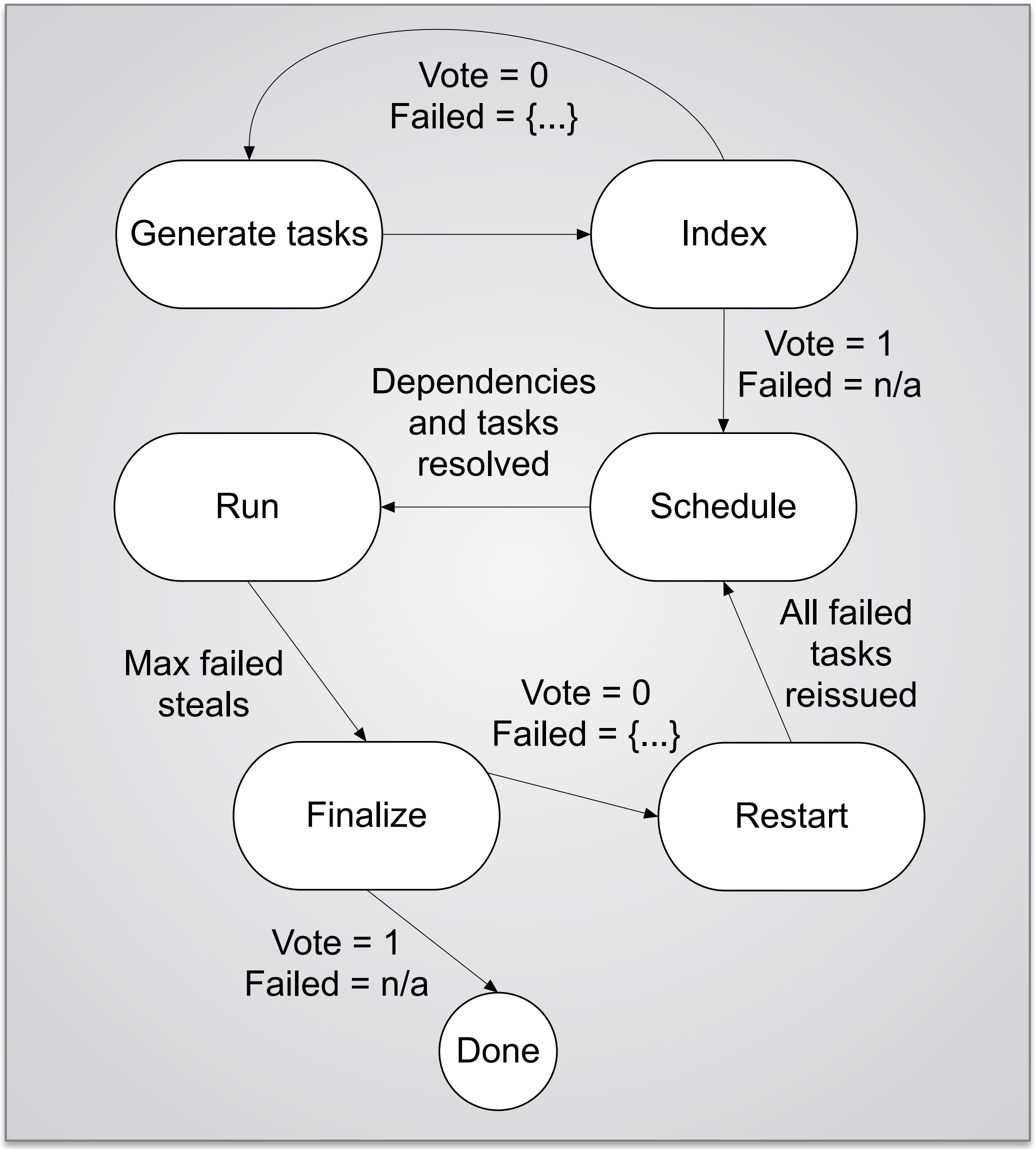
$$k := k + 1$$

end repeat

Code example: Unrolling DAG (contd…)

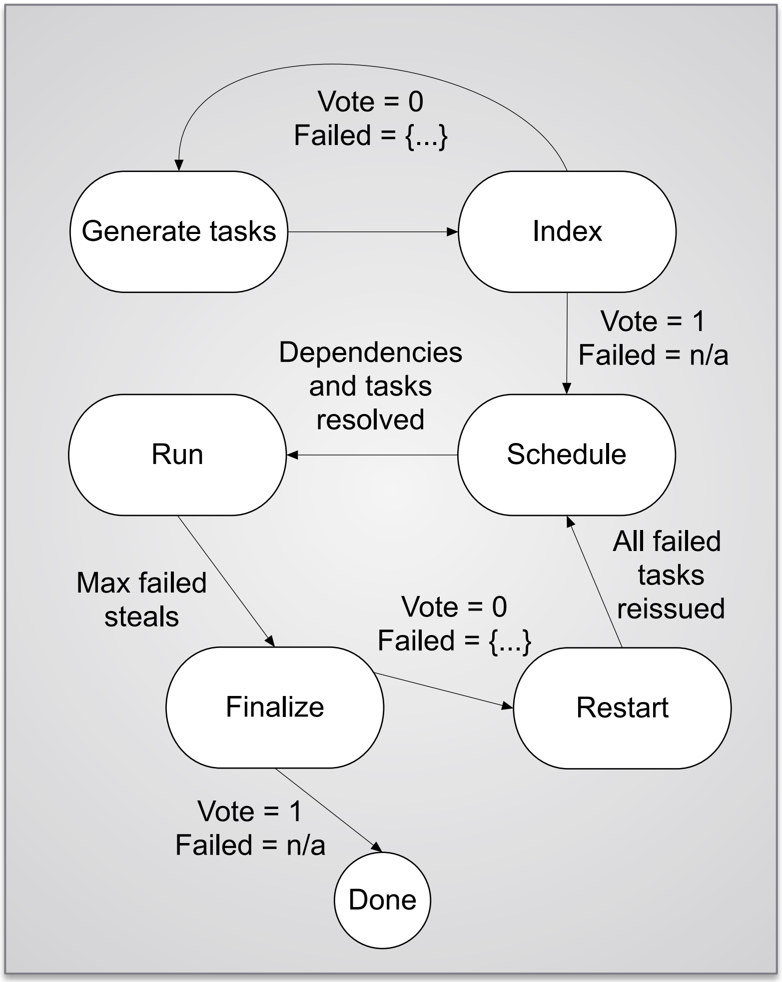
```
  ……………………………………………
  task_collection::ptr coll_p = new collection(rt,…);

  if (!end){
    coll_Beta->set_unroller(new cg_unroller(iter_+1,config_));
  }
  else {
    coll_p->set_final_collection();
  }

  rt->register_collection(coll_Alpha_dp);
  ……………………………………………
  rt->register_collection(coll_p);
}  //end cg_unroller
```

Work-flow diagram



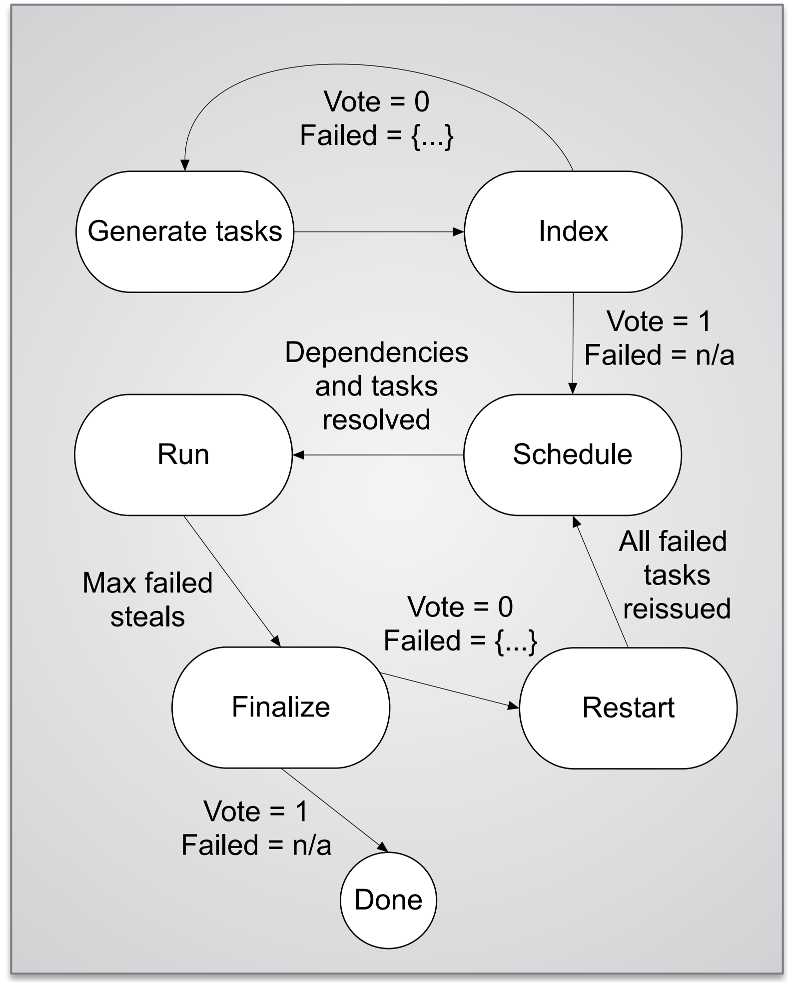
Why do you need an index phase?

- Tasks within a collection are generated locally.
- Every worker needs to *agree* on a unique label for each task i.e. tasks need to be *globally indexed*.
- The unique global index is required for:
  - scheduling a task remotely.
  - work stealing.
  - regenerating incomplete tasks due to a failure.
- This indexing is via an fault-aware **all_gather** collective.



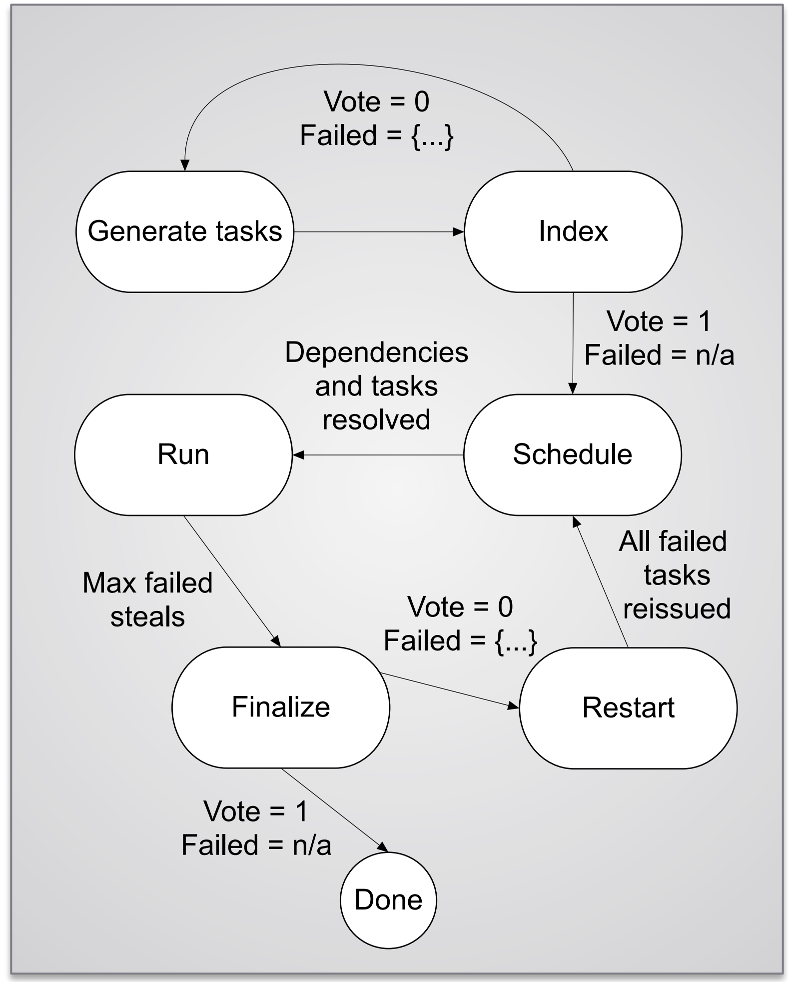
Why do you need a schedule phase?

- Each task needs to resolve its dependencies.
- The global dependency name is mapped to an actual physical location and address.
- The optimal location to run the task might be a remote node depending on:
  - data affinity (most input data resides on remote node).
  - load balancing (remote node has data backup copies and is idle).
- These decisions are made during the schedule phase.



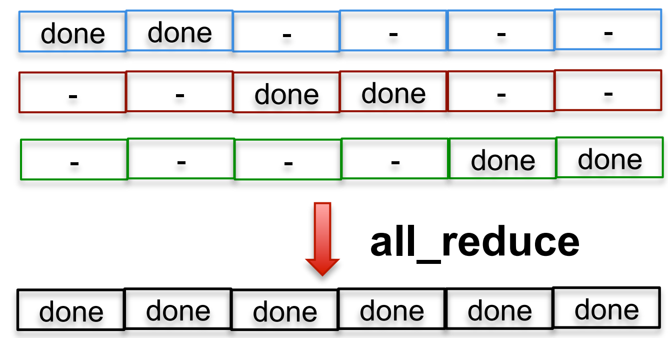
Why do you need a finalize phase?

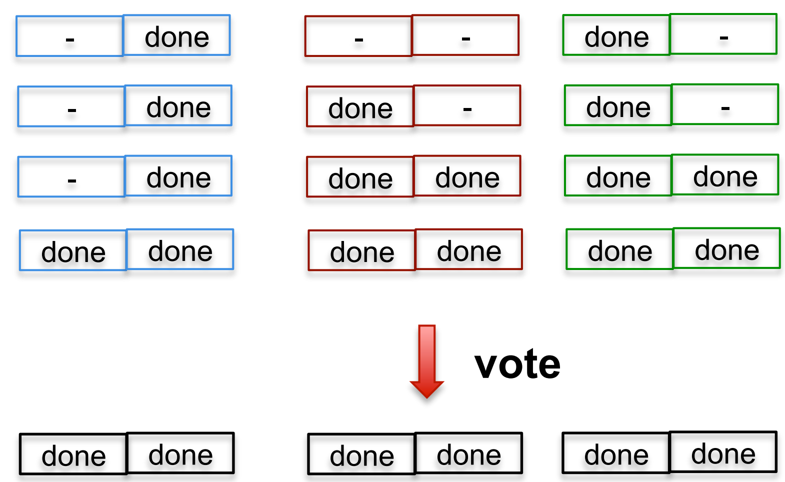
- When a worker exhausts local work:
  - it needs to determine if all work is depleted (e.g. successive steal attempts fail).
  - it needs to agree with everyone else if all work is depleted.
  - it needs to determine if any work was lost (due to failure).
- If any tasks remain incomplete due to failure, they can be detected and regenerated only in this phase.
- The finalize phase ensures that a collection is exhausted collectively by all participating workers.



Finalize phase – global agreement on task status array

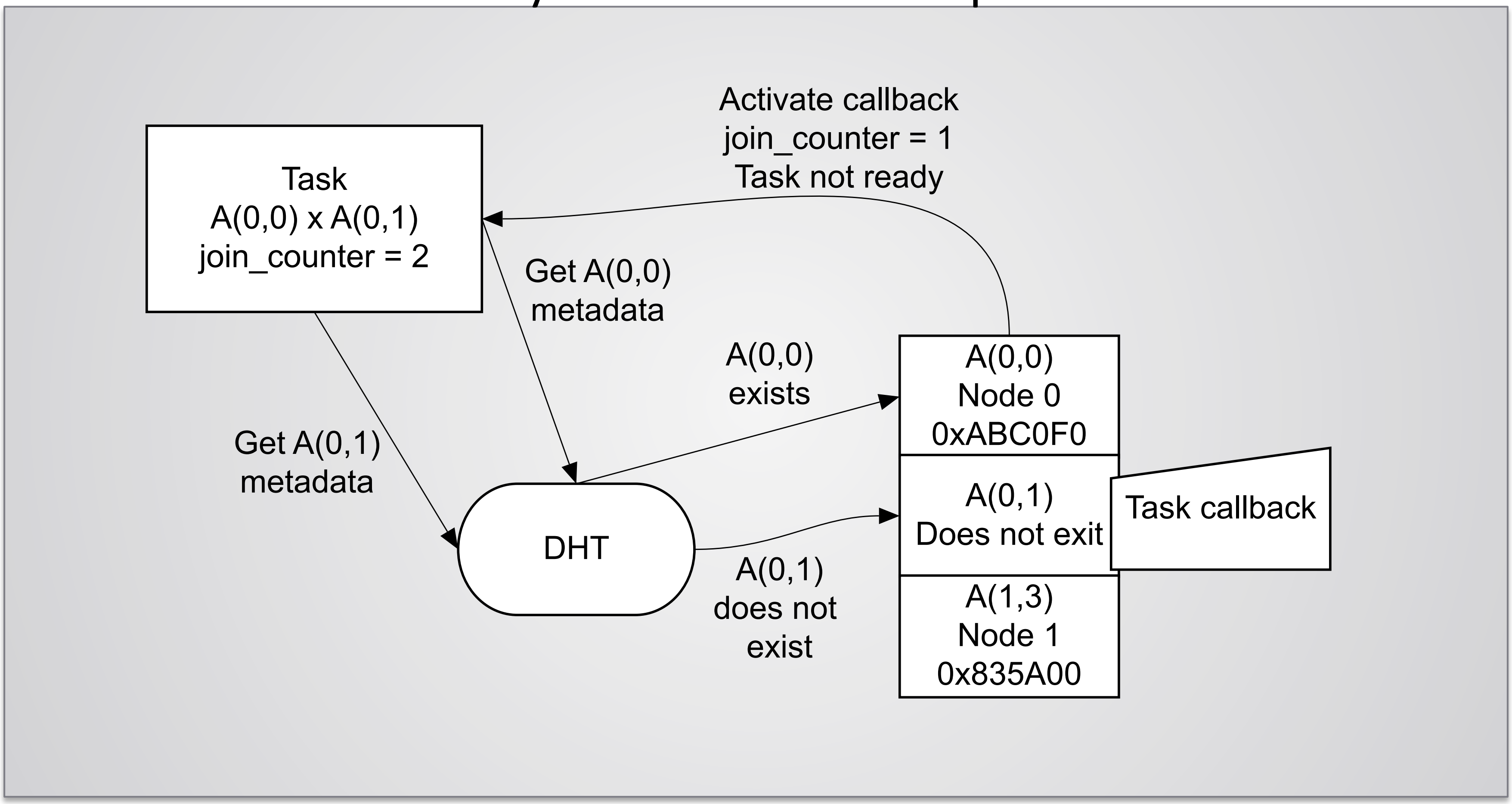
- Every node keeps task status array (bits) to confirm the global status of individual tasks.
- Naive approach:
  - each worker maintains/updates a copy of task array.
  - **all_reduce** the global (large) array.
  - not scalable (later results show).
- Alternative approach:
  - distribute the task status array.
  - completion of each task reported to designated node that is tracking its status.
  - multiple nodes can track status for a single task – redundancy.
  - when your portion of task array shows all “done” vote to finalize.

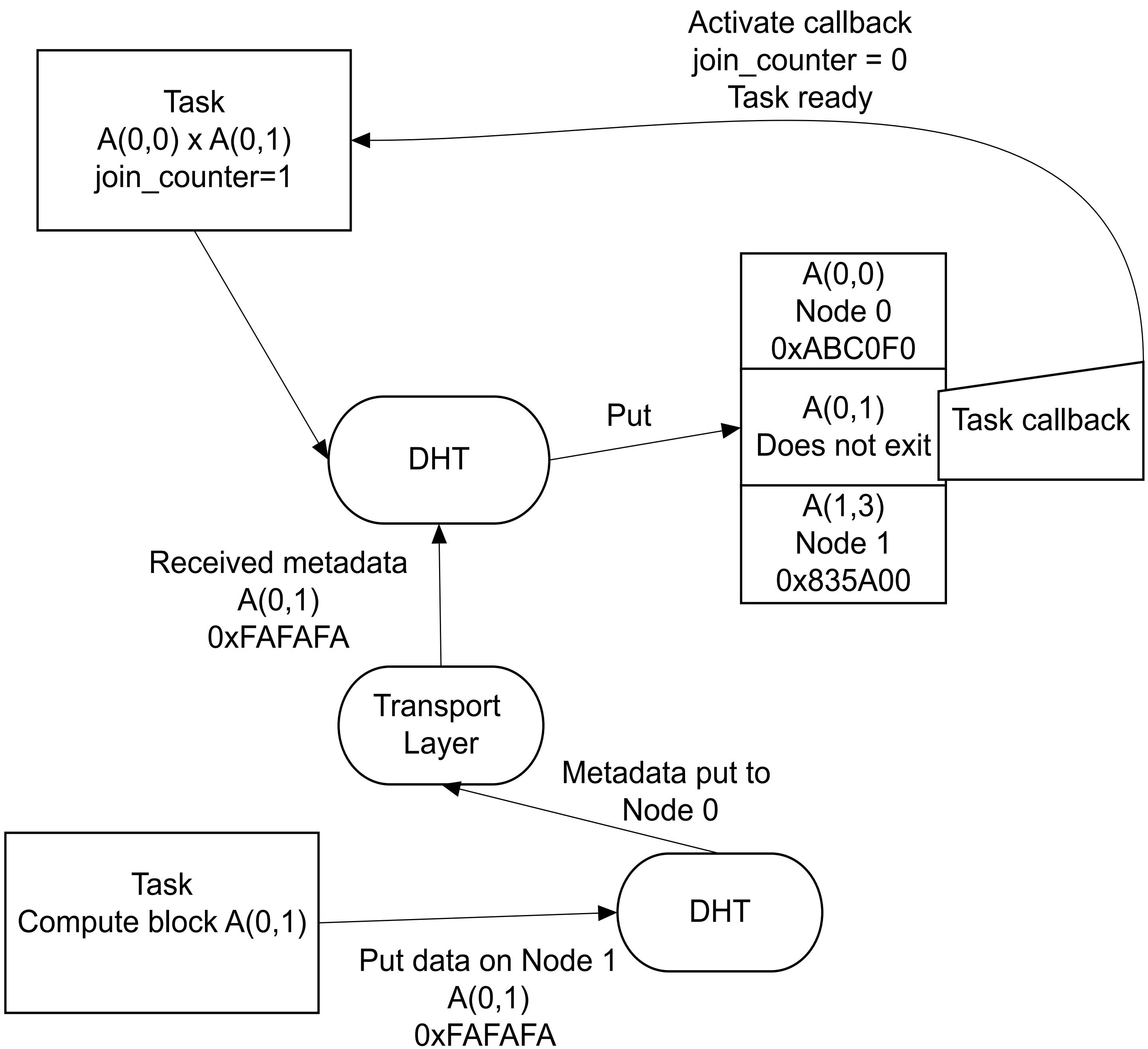




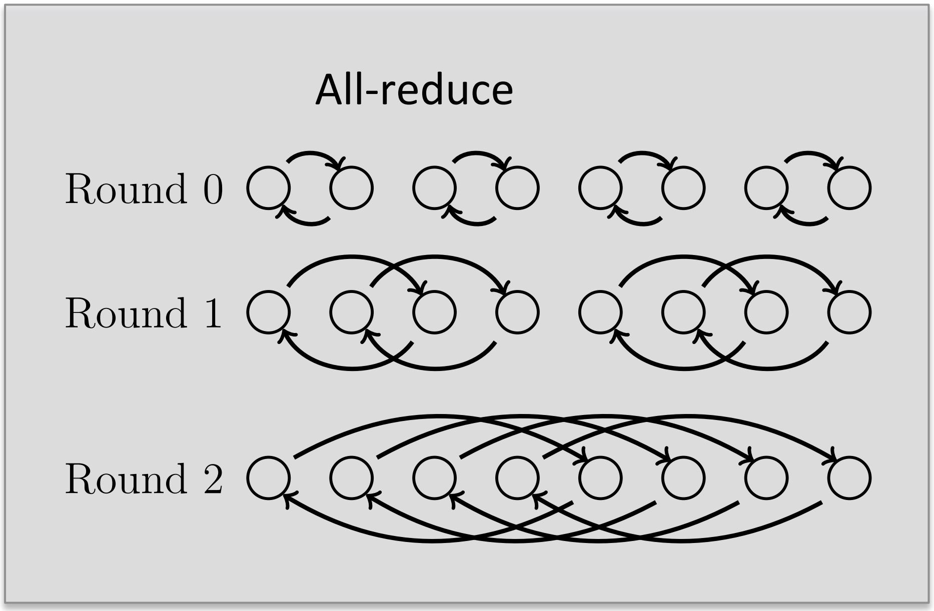
Dynamic data lookup with DHT

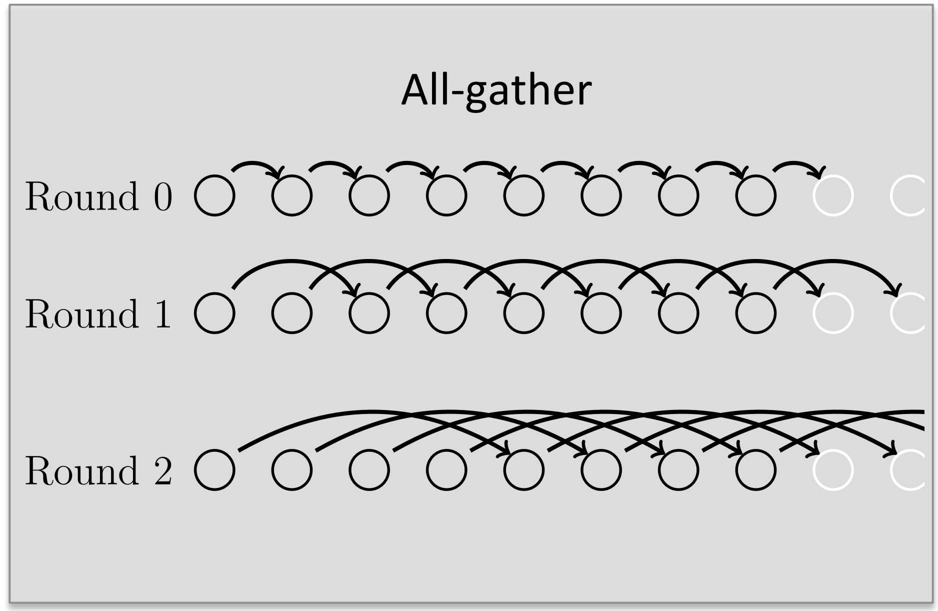
Tasks activated by callbacks when dependencies exist




Activate callback
join_counter = 0
Task ready
Task
A(0,0) x A(0,1)
join_counter=1
A(0,0)
Node 0
0xABC0F0
Put
A(0,1)
Does not exit
Task callback
DHT
A(1,3)
Node 1
0x835A00
Received metadata
A(0,1)
0xFAFAFA
Transport
Layer
Metadata put to
Node 0
Task
Compute block A(0,1)
DHT
Put data on Node 1
A(0,1)
0xFAFAFA

Fault-aware collectives





 = Send Message + Ping

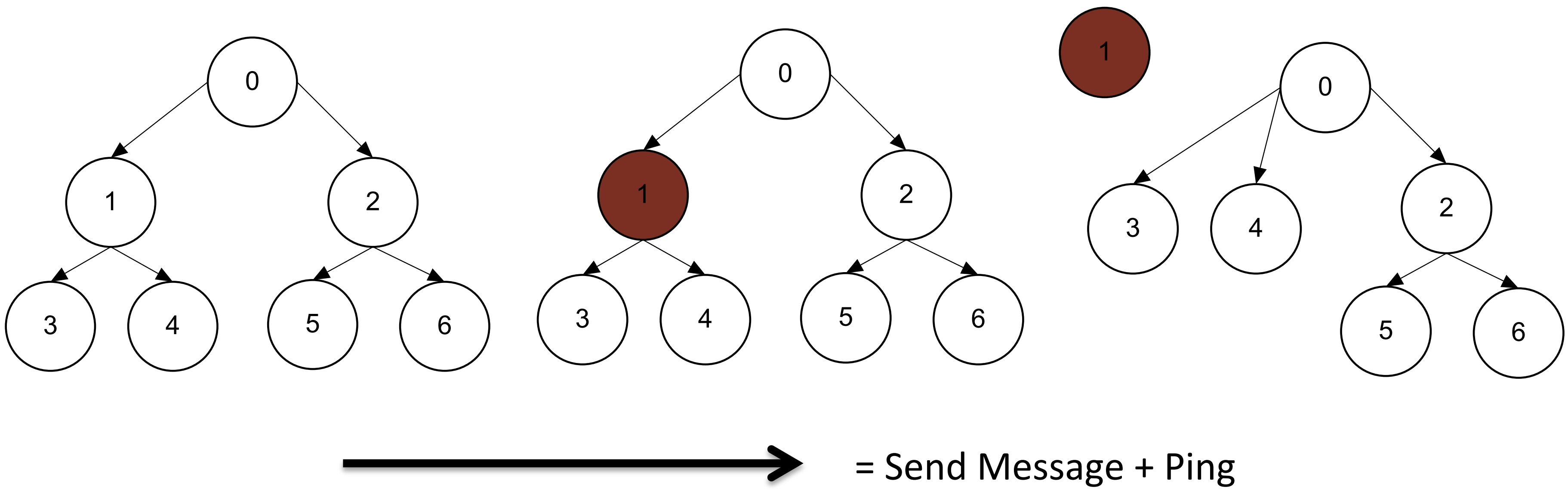
- Round partners are pinged (either timeout or RDMA NACK) to ensure alive
- If failure detected, every round of collective must still be executed (sending 0 byte fake messages)
- Only fault-aware – processes can exit with different error status, but guaranteed to finish and not deadlock waiting on dead nodes

<u>Send/recv Protocol</u>

1. Source sends RDMA header
2. Dest recvs RDMA header, executes RDMA get
3. Completion ack delivered to sender/receiver

RDMA get assumed resilient! Requires network layer support

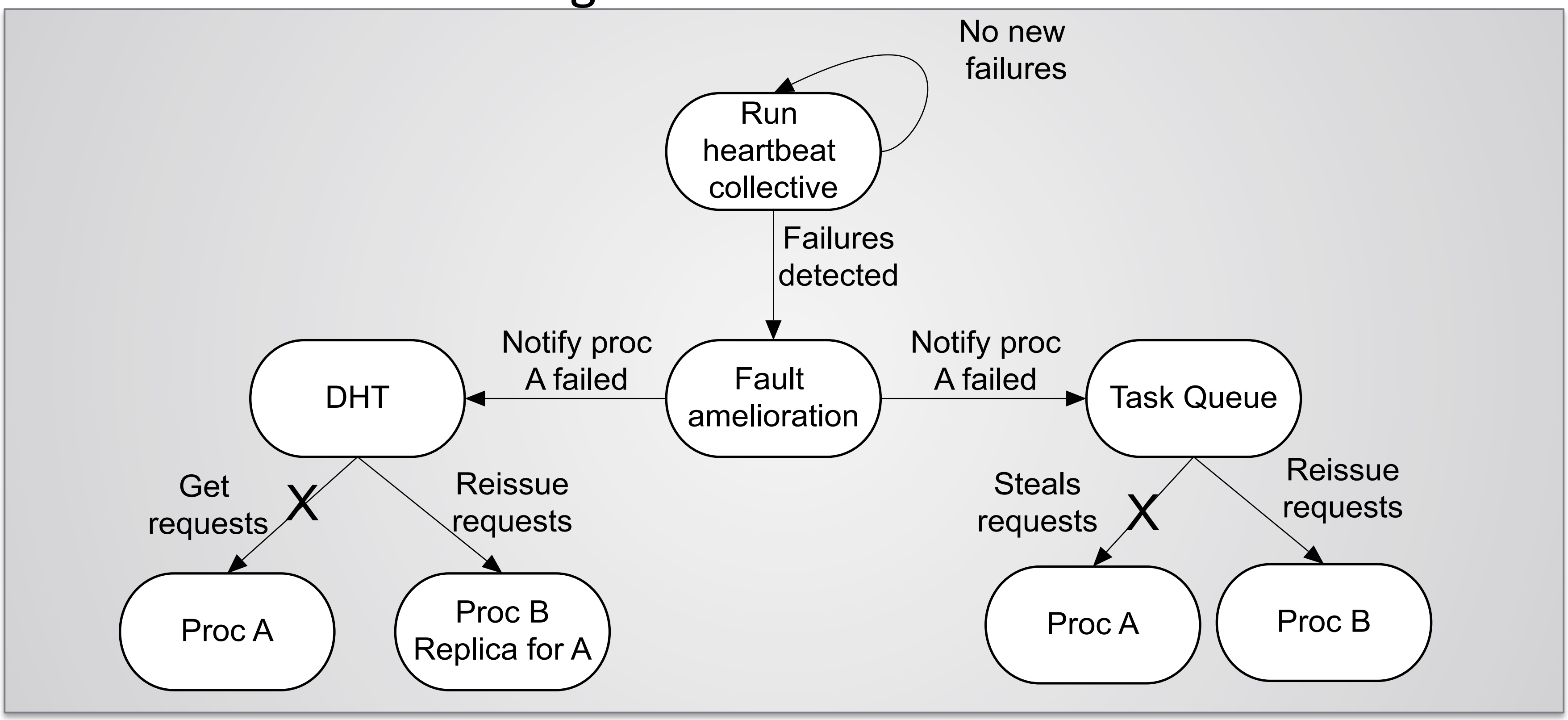
Fault-tolerant collectives: Resilient voting algorithm



- Basically same as algorithms from Hursey and Graham
- Votes passed up tree and merged on root
- Much simpler to assume root never fails – ways around it
- After failures detected, tree reconnects and votes reissued
- Can be used immediately after any fault-aware collective to vote on completion – makes any fault-aware collective fault-tolerant

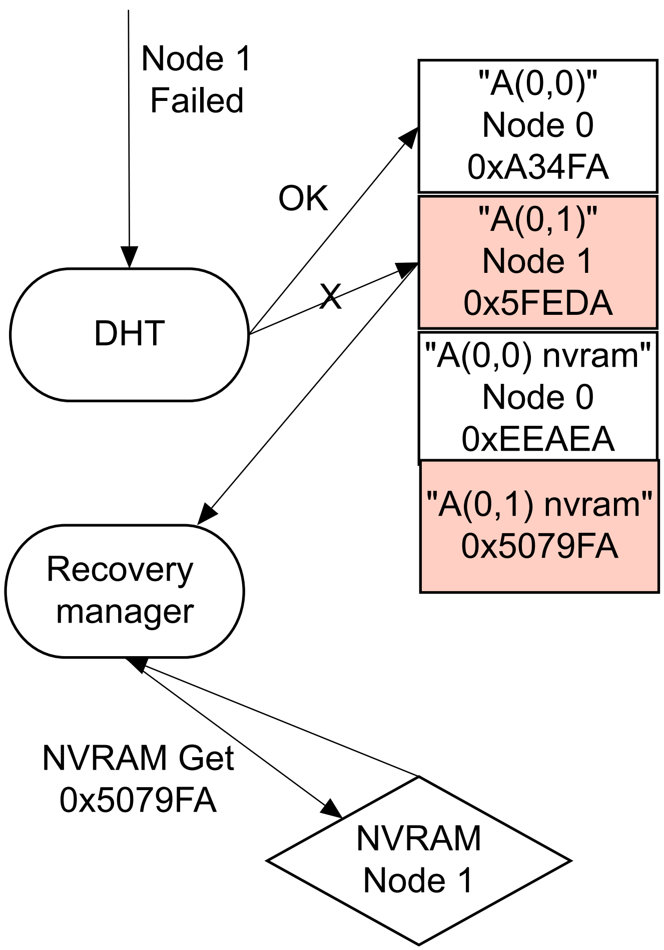
Heartbeat connects to DHT and task queue to respond to failures

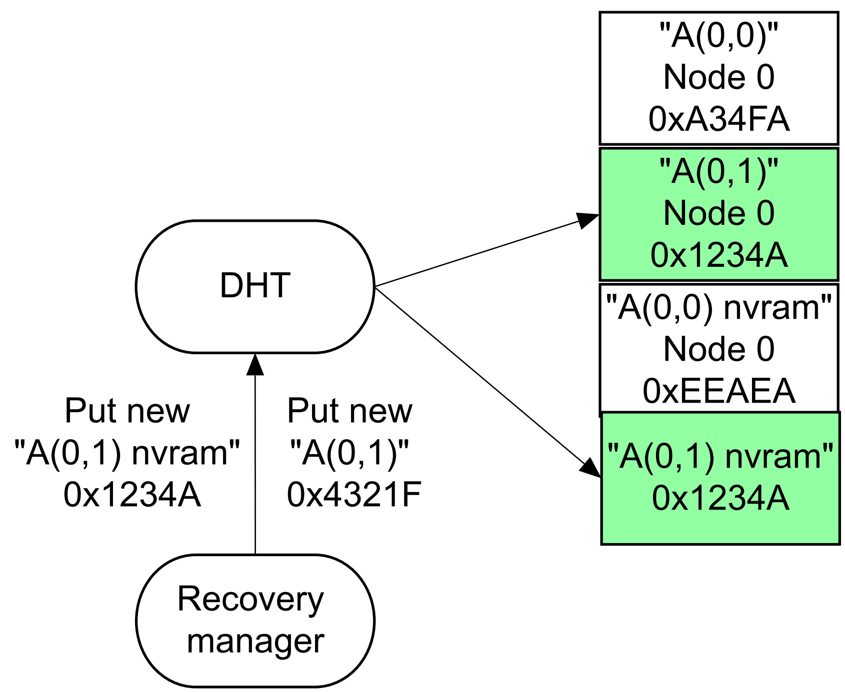
- Collectives are self-diagnosing; DHT, task queue, data backup managers need something to provide notifications of failures
- Fault-tolerant voting algorithm serves as "heartbeat" overlay network for detecting failures



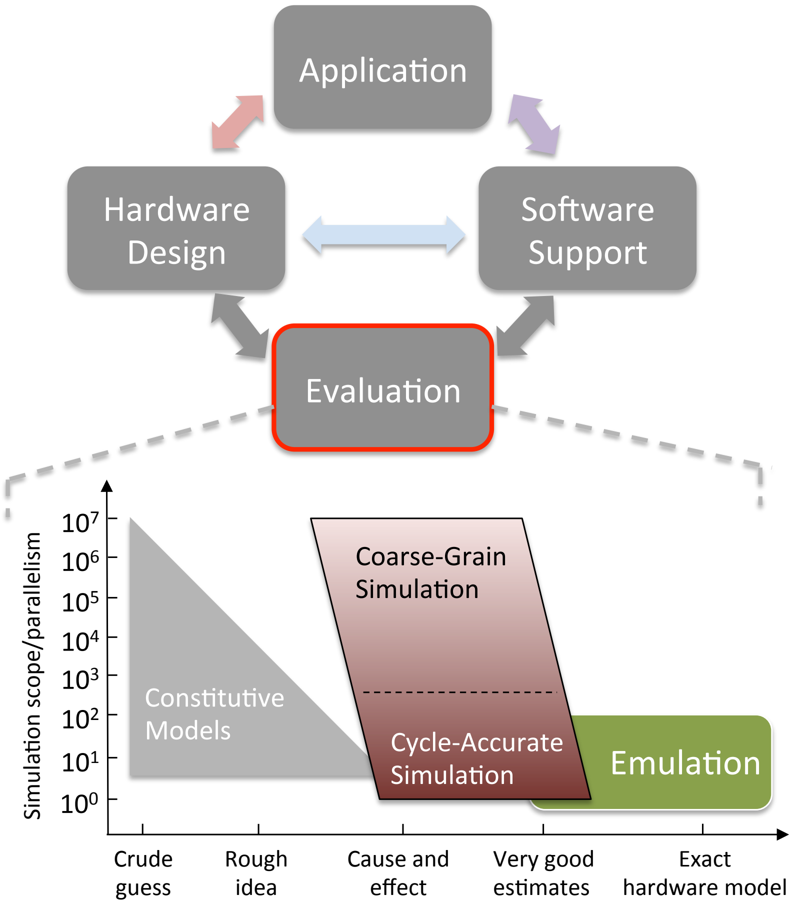
Transparent NVRAM fault tolerance with DHT

Recovery operations occur in background, no application awareness



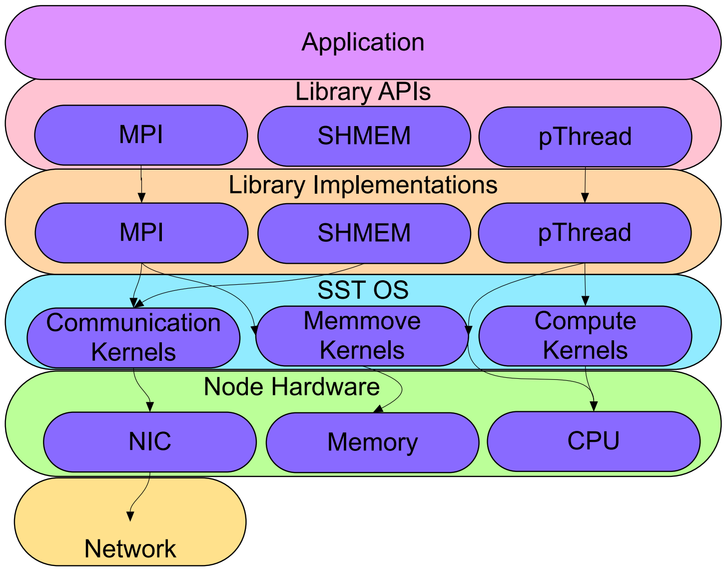


Why develop with a simulator?

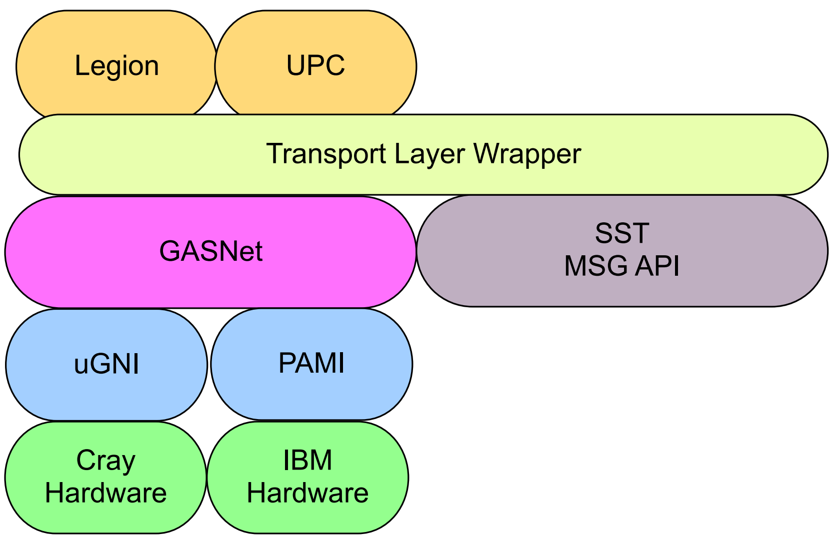


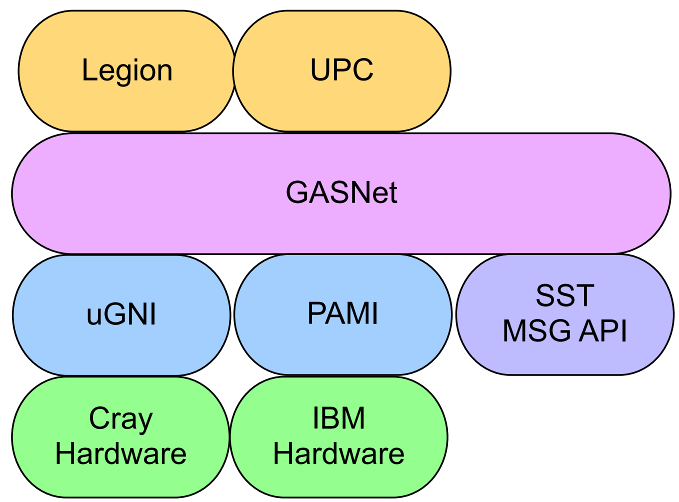
- Coarse-grained simulation explores system-level (load balancing, effects of failures)
- Think about overall structure without implementing every detail
- Rapidly iterate experiments (don't need to wait in queue)
- Co-design for speculative hardware
- Total control over when/where failures happen

SST Macroscale stack diagram



- SST is an on-line simulator
- Compile applications directly into SST libraries to simulate MPI/ pthreads/etc
- SST can link into runtime systems at two different levels: directly or indirectly as GASNet backend
- Illustrated for existing runtime systems like Legion and UPC





Compile-and-go simulation

```
int USER_MAIN(int argc, char **argv)
{
 MPI_Init(&argc, &argv);

 ...

 for (int iter=0; iter < niter; ++iter){
   MPI_Isend(left_block, nelems_left_block, MPI_DOUBLE,
       row_send_partner, row_tag, MPI_COMM_WORLD, &reqs[0]);
   MPI_Isend(right_block, nelems_right_block, MPI_DOUBLE,
       col_send_partner, col_tag, MPI_COMM_WORLD, &reqs[1]);
   MPI_Irecv(next_left_block, nelems_left_block, MPI_DOUBLE,
       row_recv_partner, row_tag, MPI_COMM_WORLD, &reqs[2]);
   MPI_Irecv(next_right_block, nelems_right_block, MPI_DOUBLE,
       col_recv_partner, col_tag, MPI_COMM_WORLD, &reqs[3]);

   do_dgemm('T', 'T', nrows, ncols, nlink, 1.0, left_block, nrows,
      right_block, ncols, 0, product_block, nrows);
  }

  ...

  MPI_Finalize();
}
```

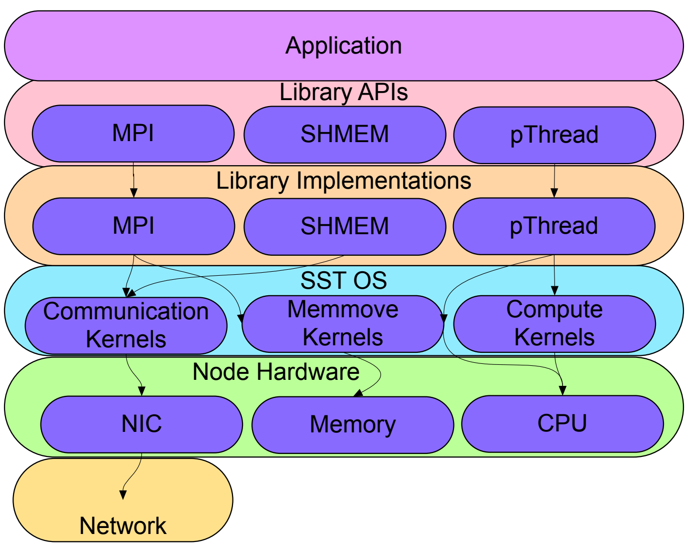
Linkage intercepts main and spawns user-space thread to simulate process

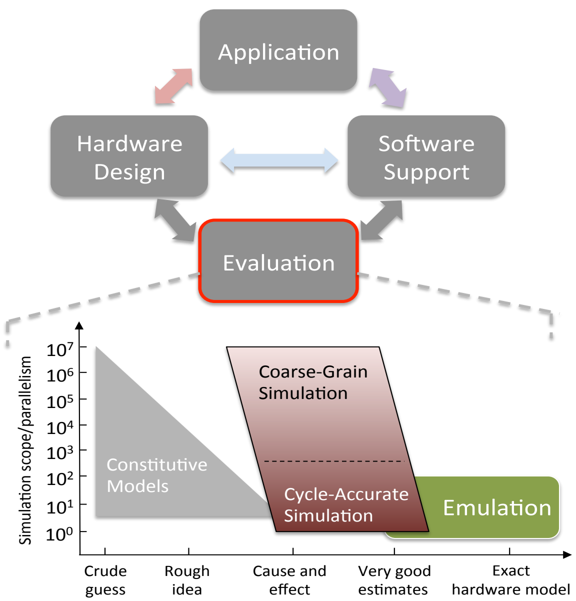
Linkage intercepts MPI calls and simulates send/recv time via congestion models

Linkage intercepts BLAS calls and estimates compute time without actually performing work

What are you giving up (or not) with simulation?

- NOT emulation – coarse-grained simulation
- No real computation, tasks just simulate time passing
- Coarse-grained network models (approximate treatment of congestion)
- Full runtime is executing – tasks are not actually run, but all task/data management is executing for real





We have implemented the DHARMA runtime system in SST

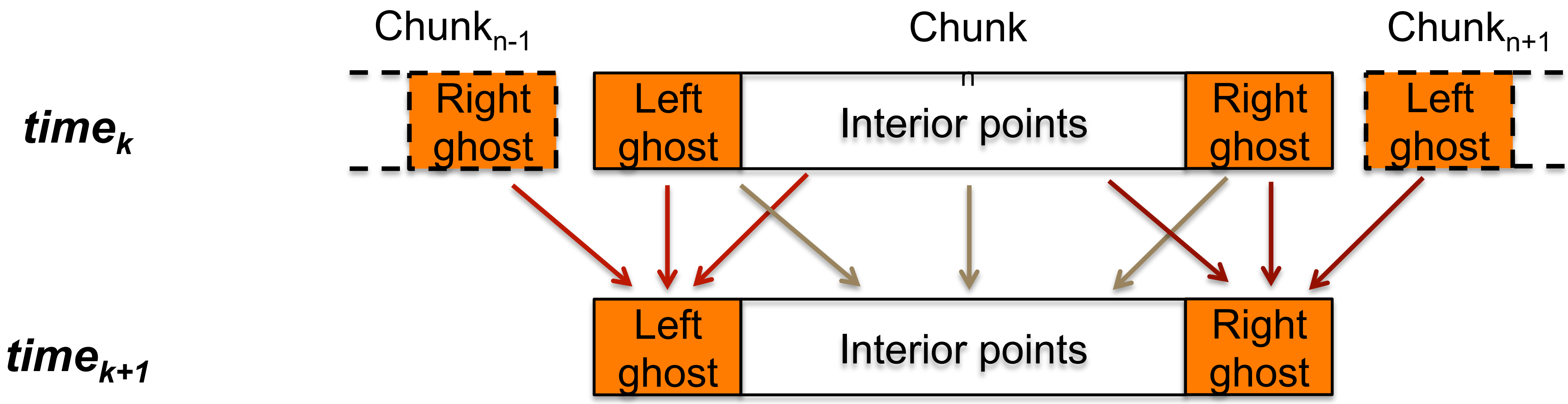
Sandia National Laboratories

- Coarse-grained simulation allows for system-level exploration
- Skeletonized mini-apps of explicit and implicit solver

Runtime studies	Algorithmic studies
Scalability with no faults (strong and weak)	Task-granularity and decomposition
Performance in the presence of faults	Classification of performance according to compute/ communication ratios
Node degradation tests	Algorithmic tradeoffs
Comparison against baseline MPI skeleton	Matrix assembly variants made possible by shift to many-task model

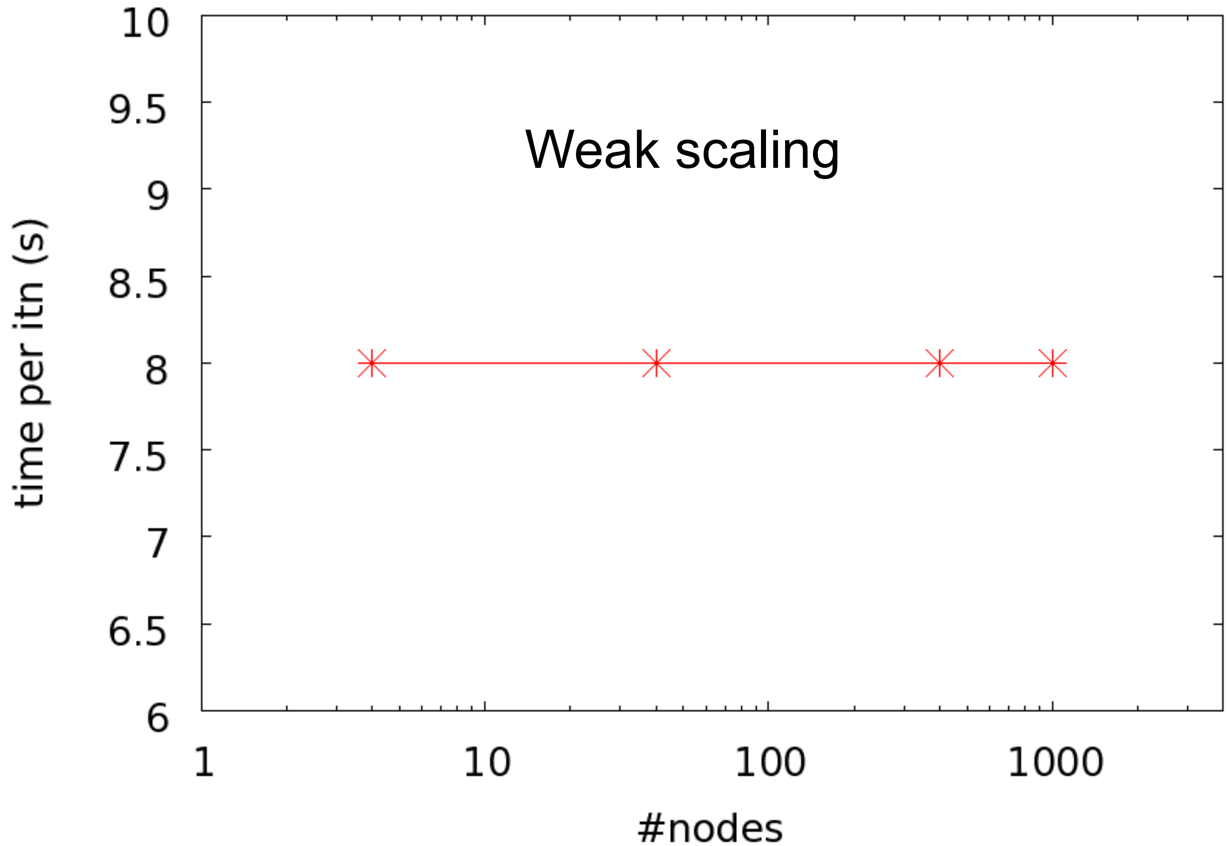
- Demonstration of full-scale implementation of run-time and associated mini-apps on capability-class system next year

Explicit 1D-PDE problem



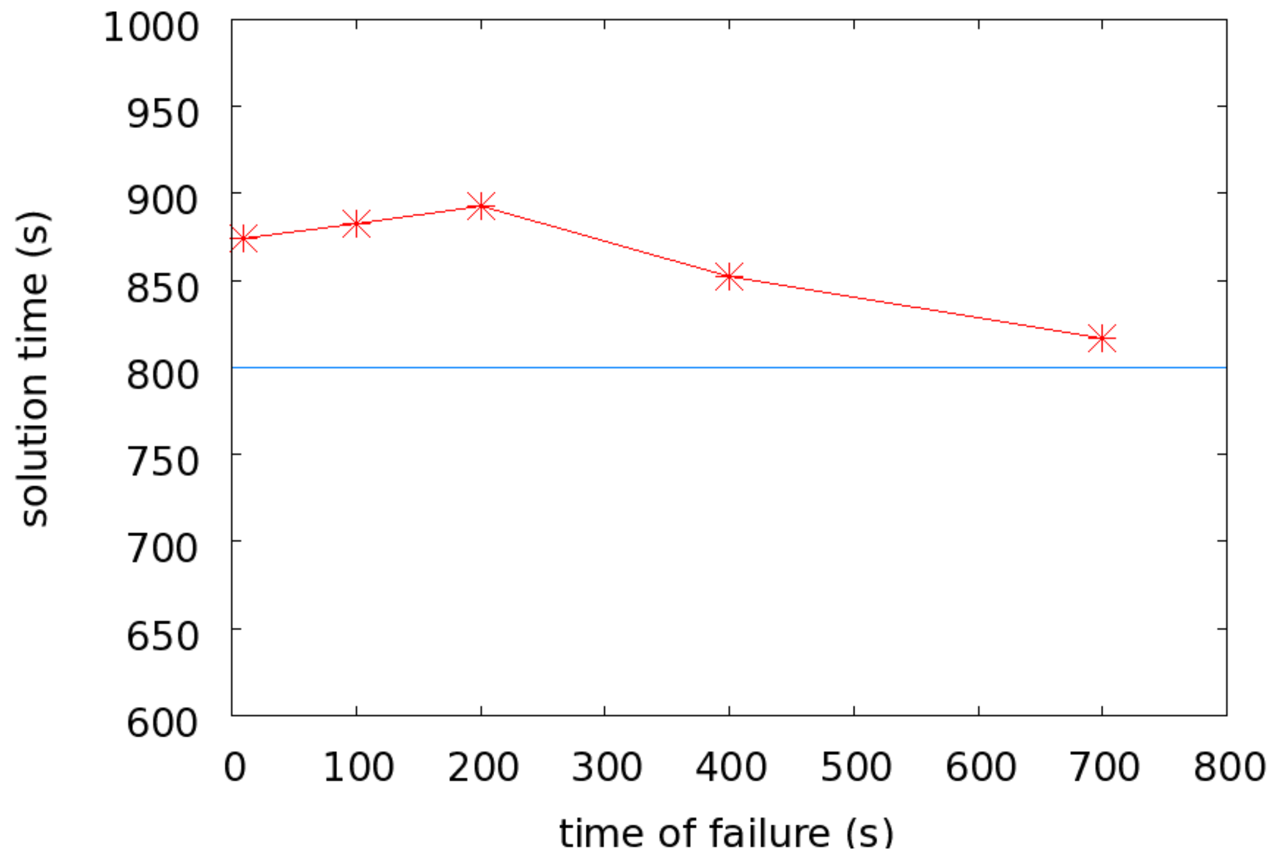
- Two tasks for each chunk: one for interior points, another for ghost points.
- Compute times based on typical S3D cost.

Baseline – no failures



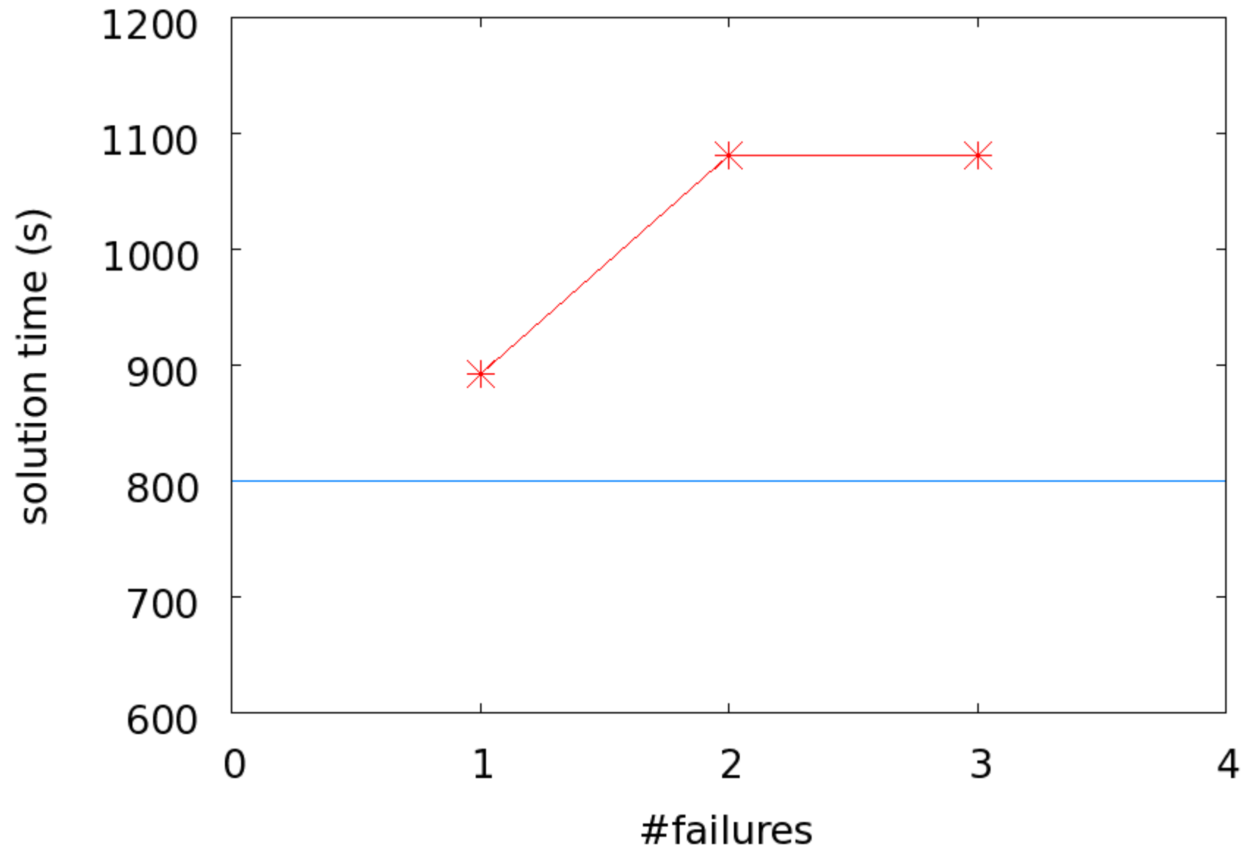
- Specs: 10 chunks per node, n_{int} = 8000, n_{gh} = 4, #itns = 2-100
- SST simulation shows perfect weak scaling of the problem with *dharma* runtime.

Impact of 1 failure



- Tests with just 1 node failure at various instants (#nodes=400)
- The time to solution non-monotonic with failure induction time.
- Early fault – more time to absorb. Late fault – less overhead.

Impact of multiple failures



- Tests with multiple failures (#nodes=400, #itns=100).
- 1st failure at 200s, 2nd at 400s and 3rd at 700s.
- Overhead not proportional to number of failures.

Summary

- DHARMA aims at tackling challenge of resilience in dynamic, adaptive, task-DAG world
- Distributed consistency problem at heart of runtime
- Many places to optimize
- Can we use application knowledge to structure runtime and make resilience strategy both as EFFICIENT and TRANSPARENT as possible?
- What's the burden to the programmer?