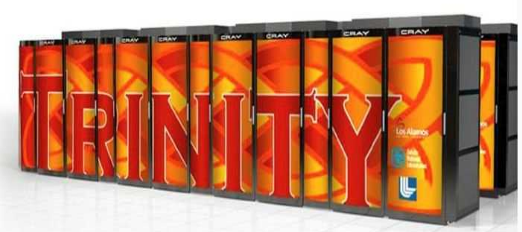
SAND2017-7991PE



Toward Resilient Task Parallel Programming Models

Keita Teranishi

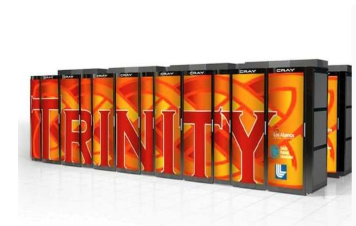
AICS Café at RIKEN AICS, Kobe 08/02/2017

Funded by ASC CSSE Program.

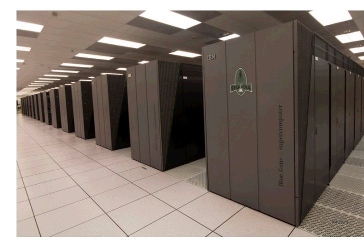
Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. SAND NO. 2011-XXXXP

Need for Scalable Resilience for HPC Applications

- Today, we see large HPC systems suffer frequent failures
  - MTBF 0.5-7 days (failure = lost job)
    - Global file system crashes
    - 60-80% of failures are due to software (*)
- Future systems are expected to be less reliable
  - User expectations
    - 75-95% node utilization (30-40% in enterprise computing)
    - Tightly coupled massively parallel applications
  - More components (and shrinking of each component)
    - Today: Millions of threads, Several Peta bytes of memory
    - Exascale: Billions of threads, 100+ Peta bytes of memory
  - Limited Power Budget
    - Today (US): 5-10MW for 10-20Peta Flops
    - Exascale systems: 20-30MW for 1Exa Flops !!

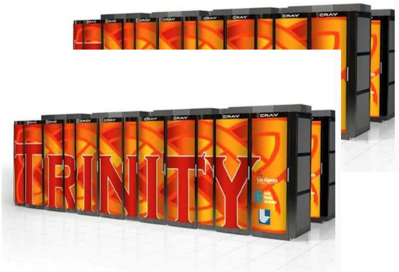






Hardware and System Based Resilience and Fault Mitigation

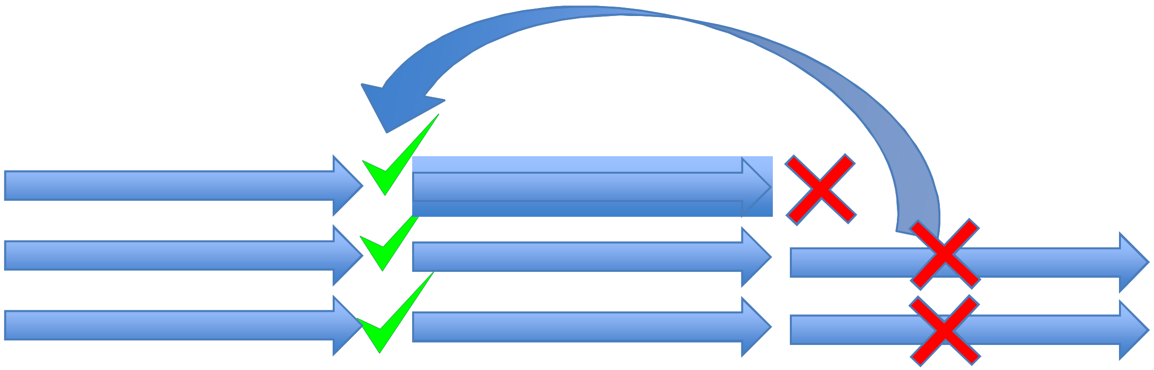
- **Redundancy**
  - Weather centers purchase two identical systems
  - N-Modular Redundancy (typically N=3)
  - Parity bits (ECC) in memory
    - Recent advances in Chipkill (RAID-5 like redundancy) improves the memory protection dramatically
  - Replicate Threads/Processes
- **System Cooling**
  - Lowering the temperature reduces the error rate
    - Node temperature of the K-computer is kept at 30C
    - 85C for average data centers
- **System Level Checkpointing**
  - OS stores process images to persistent storage
  - Recovery is done through loading the image



Application/Runtime Approach

- System/Hardware Approach Ignores
  - Application specific failure characteristics and patterns
  - Application specific failure mitigation and recovery
  - User's capability to manage failures
- Application/Runtime approach can handle different types of failures in different granularities
  - Soft Failures (such as Silient Data Corruptions)
  - Hard Failure (such as process crash)
- Our Goal: Programming Model Support for HPC application resilience
  - Libraries
  - Language Extention

Coordinated Checkpoint and Restart (C/R)

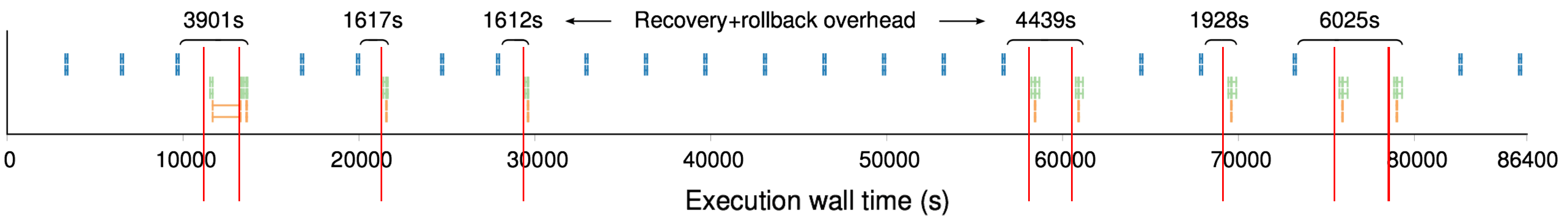


- Periodically write the state of application to secondary storage
  - Coordination (synchronization) is involved among execution streams
  - Rollback to the checkpoint when failure occurs
  - Triggers all execution streams to rollback or restart

- Performance depends on IO bandwidth

- ECP funds to the advanced C/R library (VeloC project at ANL and LLNL)
  - Better IO usage for performance improvement
  - Support of non-MPI code and multiple HPC middleware systems (MOAB, SLRUM)

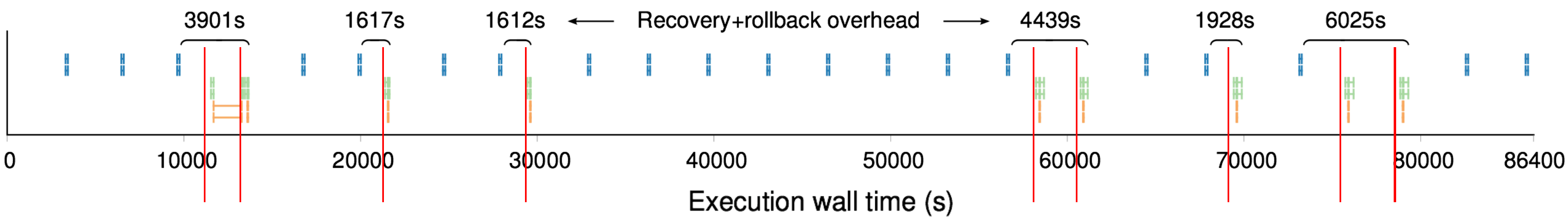
Motivating Use Case – S3D Production Runs



- 24-hour tests using Titan (**125k cores**)
  - Reported MTBF of 8 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) has to be done to the PFS

		Total cost
Checkpoint (per timestep)	**55 s**	1.72 %
Restarting processes	**470 s**	5.67 %
Loading checkpoint	**44 s**	1.38 %
Rollback overhead	**1654 s**	22.63 %
Total overhead		**31.40 %**

Motivating Use Case – Problem Summary



- Current **checkpoint cost, ~1 min**
- Total **recovery+rollback cost, ~36 min**

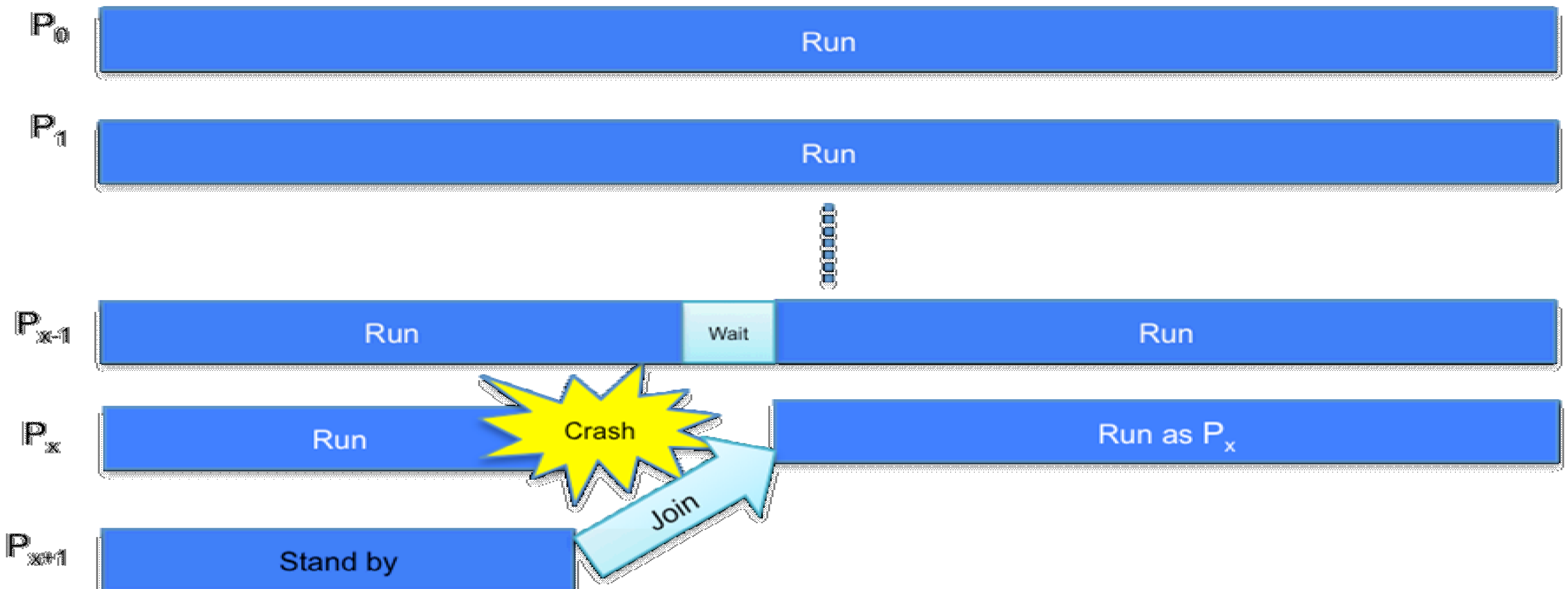
Infeasible

Traditional C/R or runtime-based offline techniques are

- *not efficient in current systems*
- *not possible in future systems*

Our Solution: Online Failure Recovery

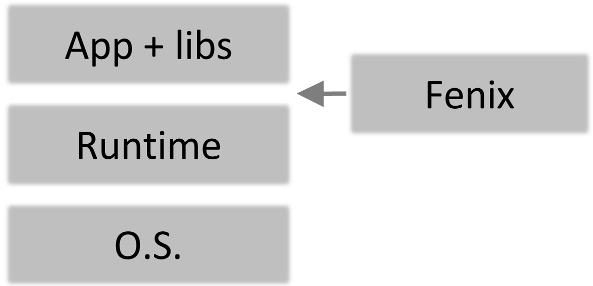
- Software framework to augment existing apps with resilience capability
  - The remaining processes stay alive with **isolated** process/node failure
  - Multiple implementation options for recovery
    - Roll-back, roll-forward, asynchronous, algorithm specific, etc.
  - **Hot Spare Process for recovery**



Solution #1 :Global Online Recovery

1. Process recovery: Recover failures without promoting to job failures

- Framework for online, semi-transparent recovery
- Targets SPMD, message passing applications
- Tolerates **hard failures (spare or spawned ranks)**
- Keep process memory (may contain valuable data or checkpoints)



2. Data recovery: Optimize checkpointing

- Store application-specific data in-memory
- **Coordinate checkpoint creation implicitly**
  - Fully coordinated checkpointing ⟶ consistency, but barrier-like constructs
  - Uncoordinated checkpointing ⟶ efficient, but no consistency guarantees

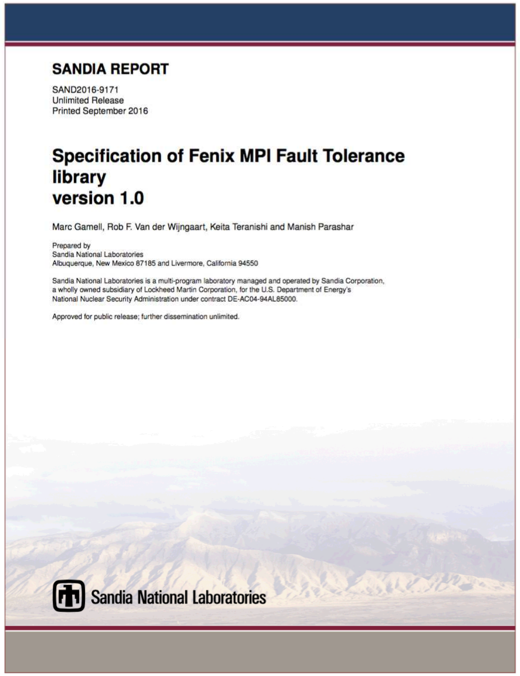
Implicitly Coordinated Checkpointing

Applications know consistent points!

  - Implicit coordination: create consistent checkpoints without communication
  - Consistency guaranteed by **checkpointing at the same "logical" time**

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 (prototype of MPI Fault Tolerance)** to shield the users from low-level MPI features
    - Process recovery through spare process pool
    - Process failure is checked at PMPI layer and recovery happens 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.

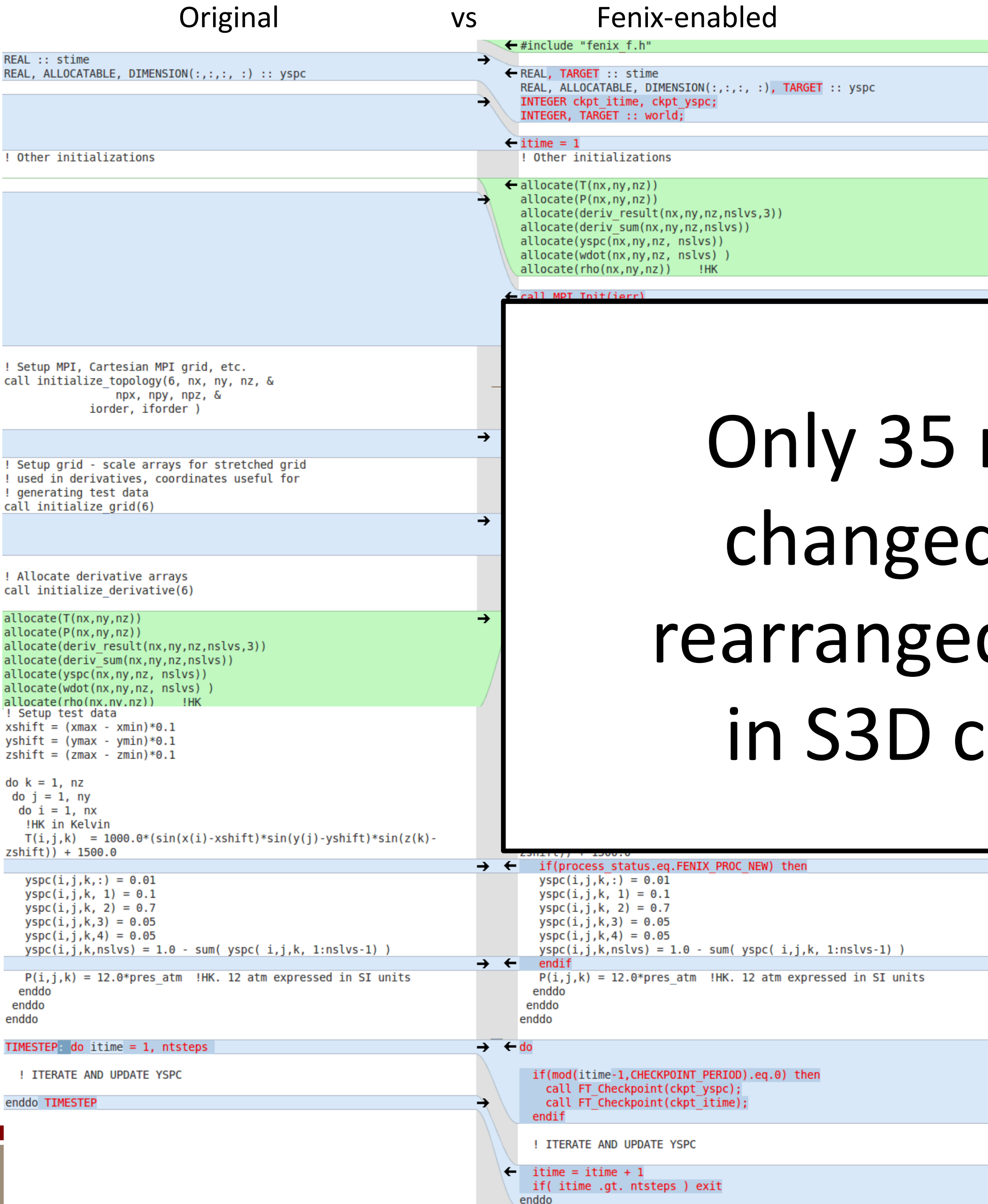
Sandia National Laboratories

S3D Modifications

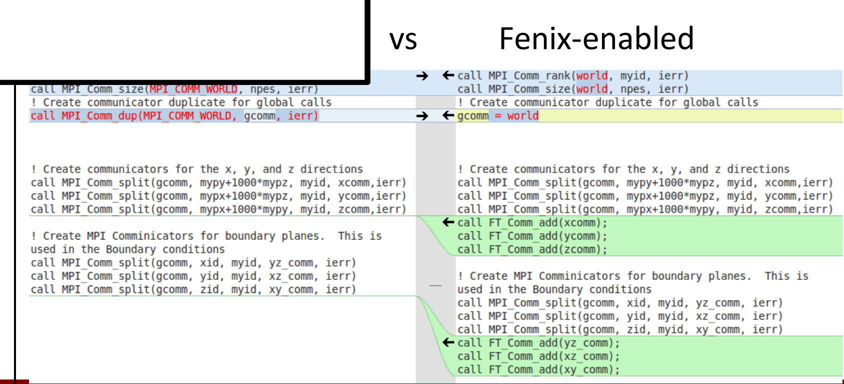
- Only 35 new, changed, or rearranged lines in S3D code

nction

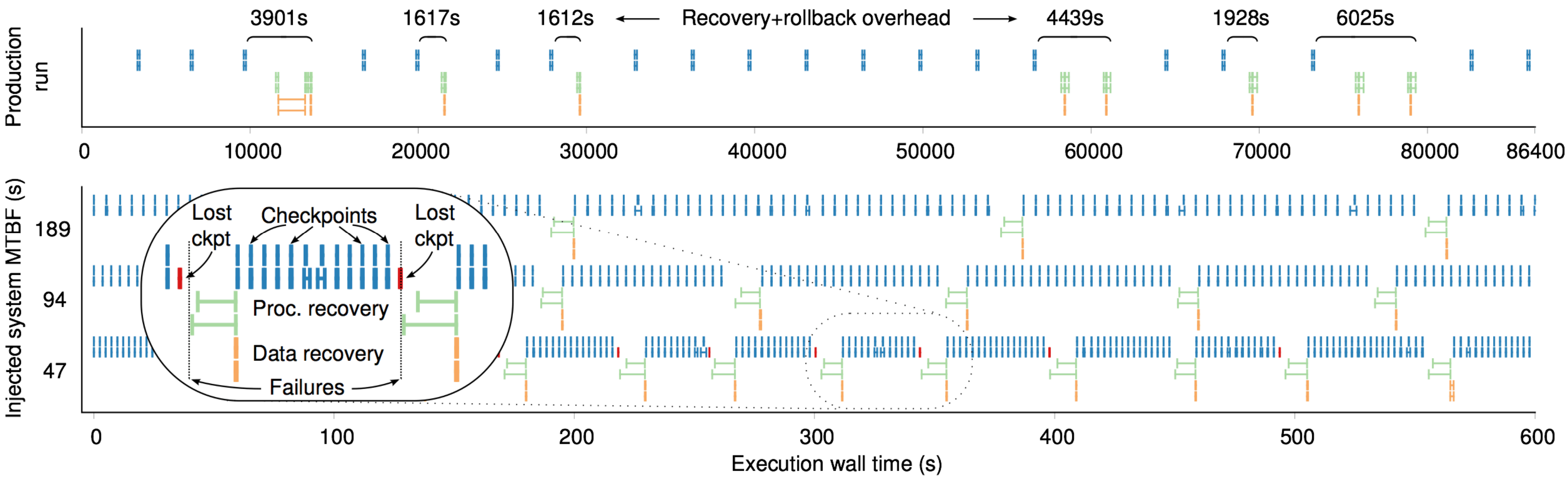
y module



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



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)

Solution #2: Local Online Recovery

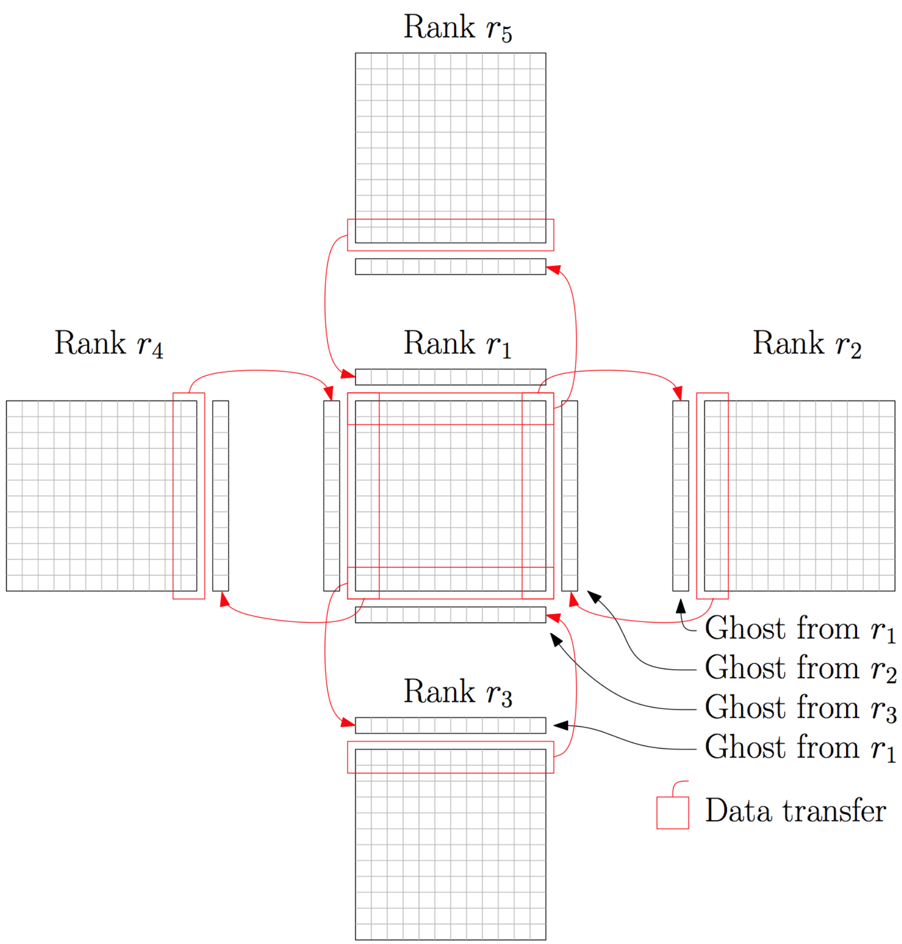
- Fenix-1.0 is the first step toward local recovery
  - Avoid global termination and restart
  - All processes rollback to the Fenix_Init() call
  - Natural for algorithms and applications that makes collective calls frequently
- Some applications fit more scalable recovery model
  - Stencil Computation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

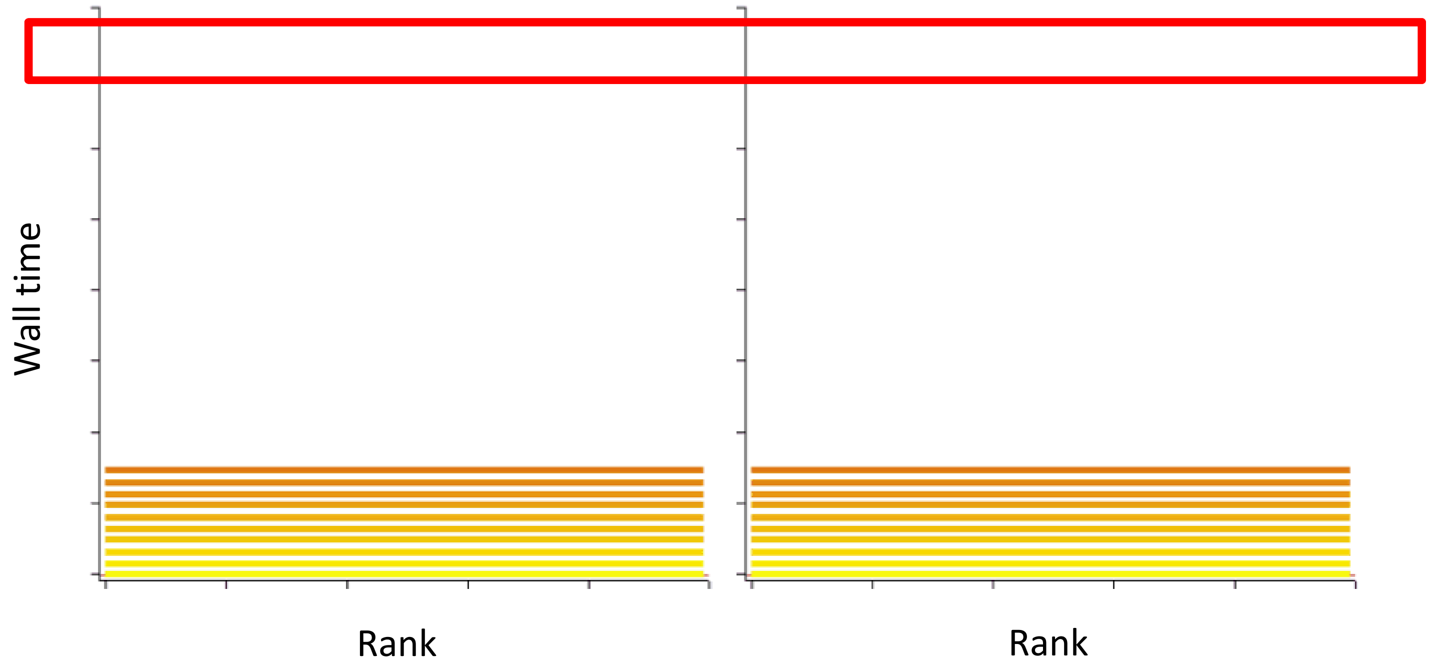
Target: Stencil-based Scientific Applications



- 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

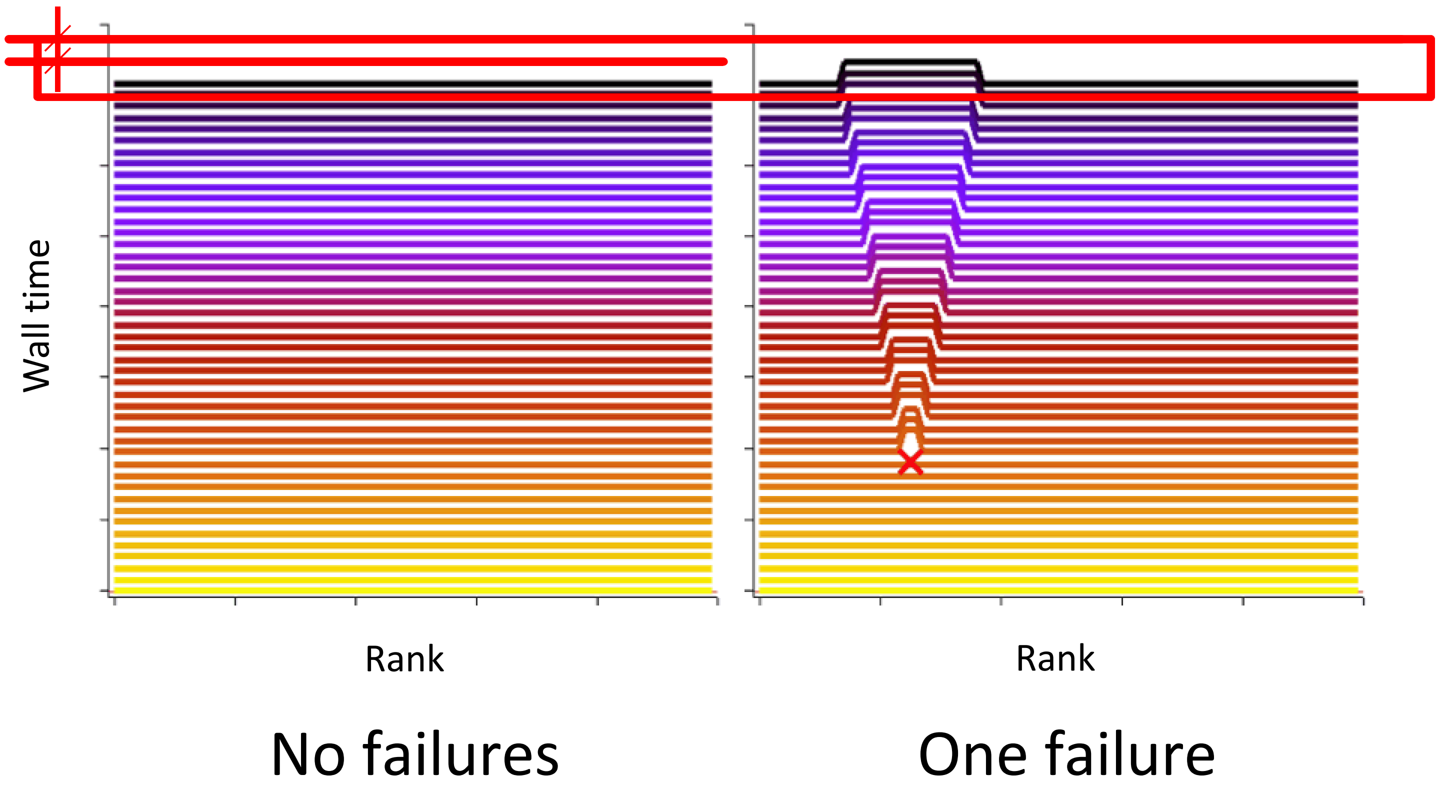


No failures

One failure

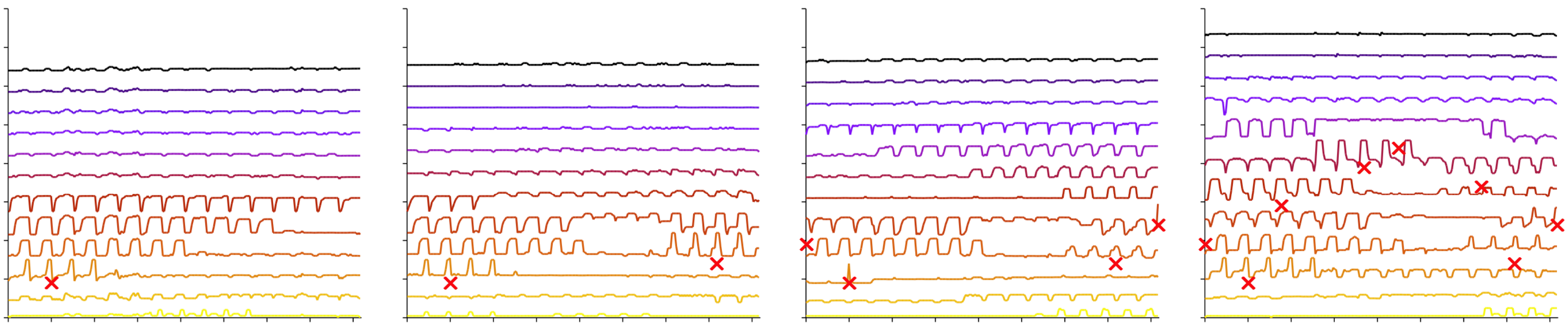
Effect of Multiple Failures with Local Recovery

Simulated execution of a 1D PDE

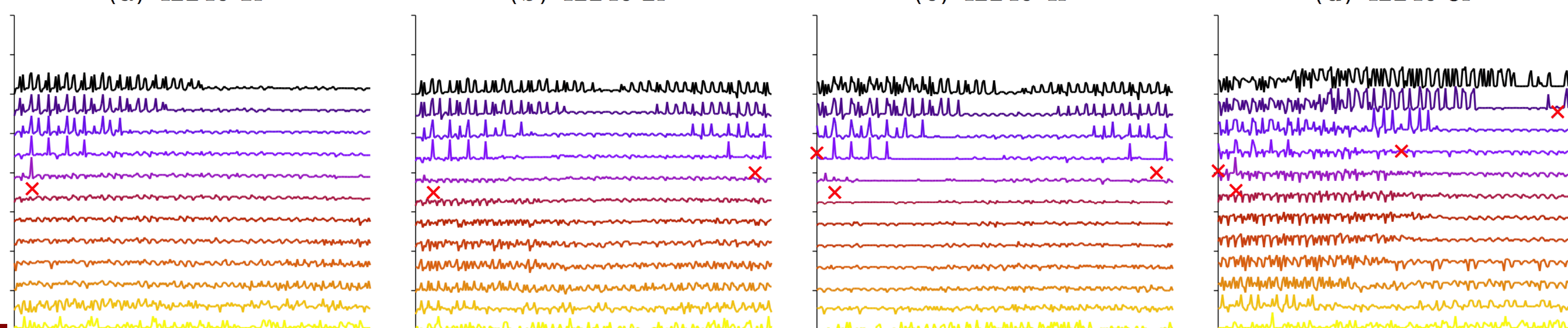


Experimental Evaluation with S3D

- 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

Experimental Evaluation

Goal

- Evaluate local recovery techniques using S3D on Titan to show
  - Low overhead while recovering from node failures every 5 seconds
  - Failure recovery is scalable
  - Recovery overhead is not proportional to system size

Experiments

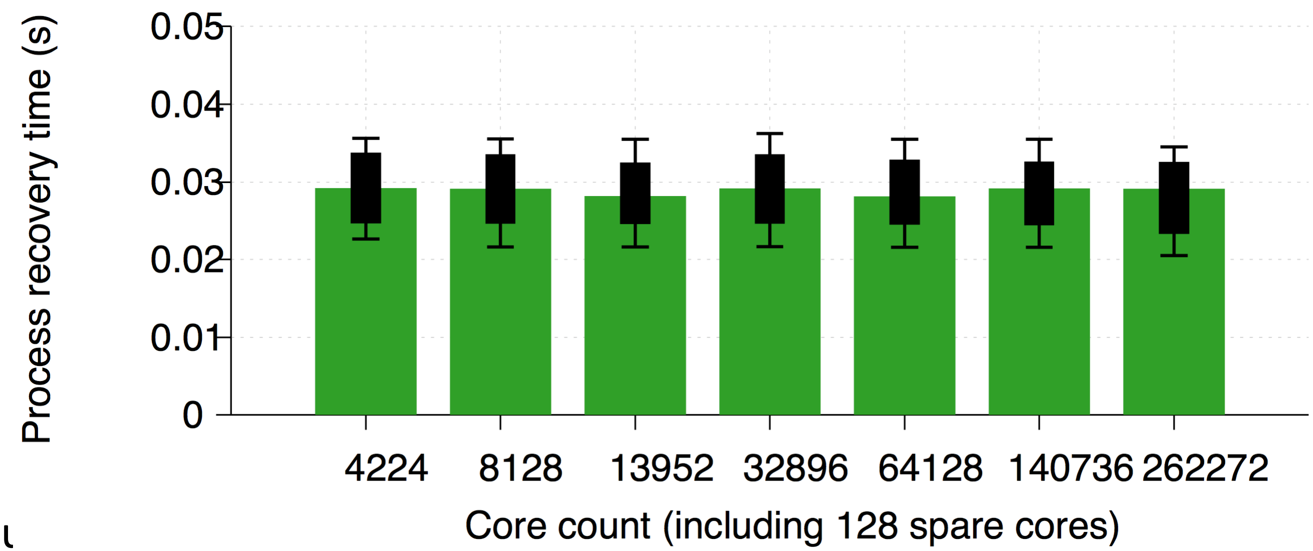
- **Recovery scalability** up to 262272 cores
- **Total overhead** of fault tolerance

Methodology

- Study the overheads related to the recovery processes
- Compare local vs global recovery
- Failure recovery cost can be decomposed into:
  - **Environment recovery**
  - Checkpoint fetching from neighbor (scalable, 130MB/core)
  - Rollback cost (average of 1/2 iteration time, O(2.5 seconds), scalable)

Recovery Scalability

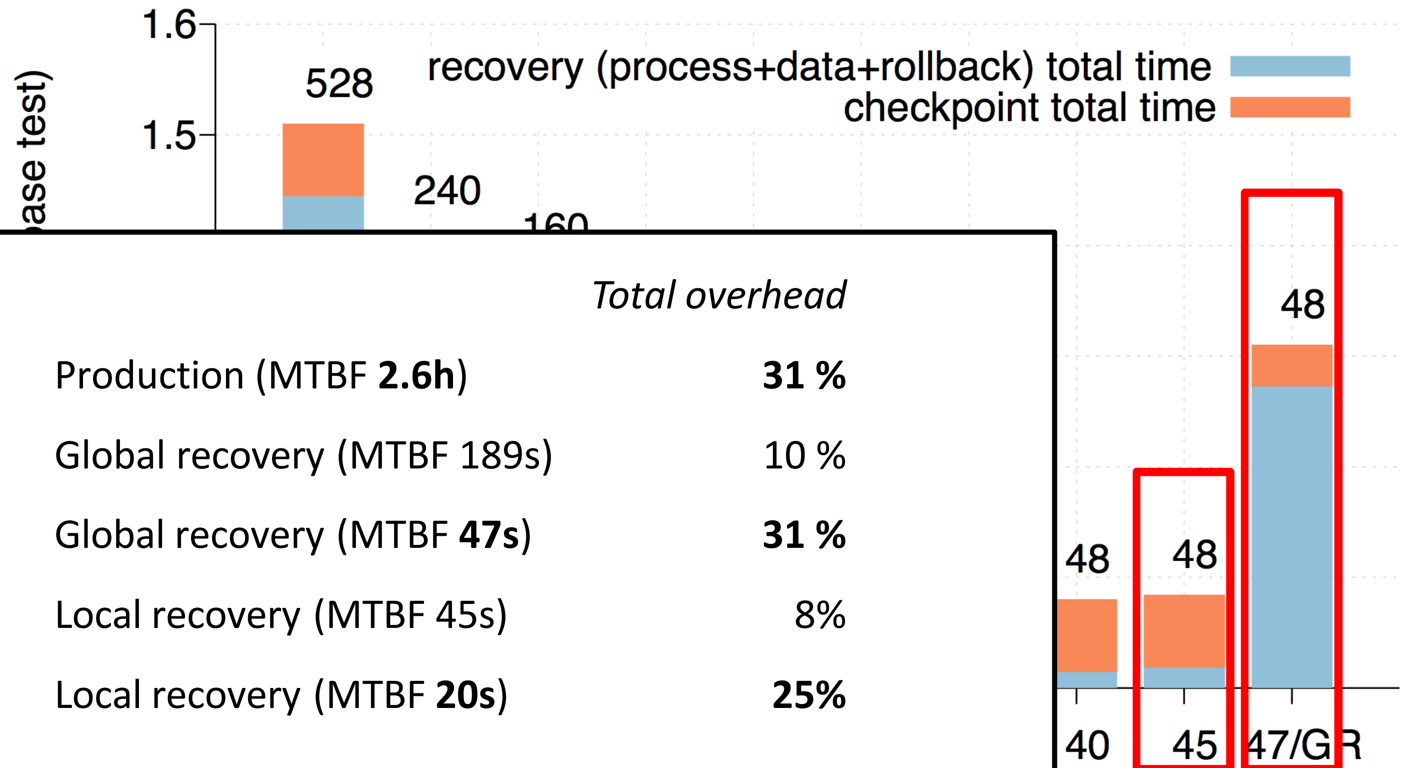
- Using MTBF of 10s
- Core count from 4224 to 262272 (including 128 spare cores)
- Result shows the average recovery time for all failures injected.



- Conclı
  - Process recovery time is independent of system size
  - Good scalability

Total Overhead of Fault Tolerance

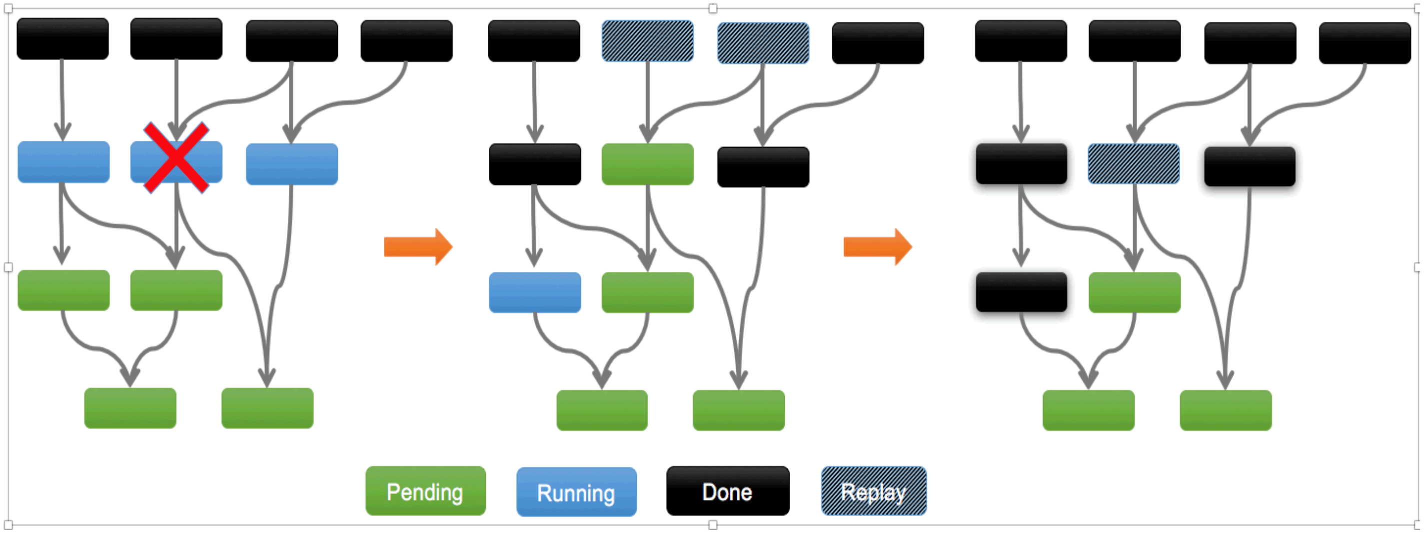
- End-to-end time vs failure-free, checkpoint-free time
- Overall overhead:
  - Checkpoint
  - Process/da recovery
  - Rollback
- 4096 cores + spare cores
- Right-most bar global recovery with MTBF of 4
- Local recovery has scalability advantages over global recovery



	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%)**

Solution #3: Toward Resilient Asynchronous Many Task (AMT) Parallel Execution Model

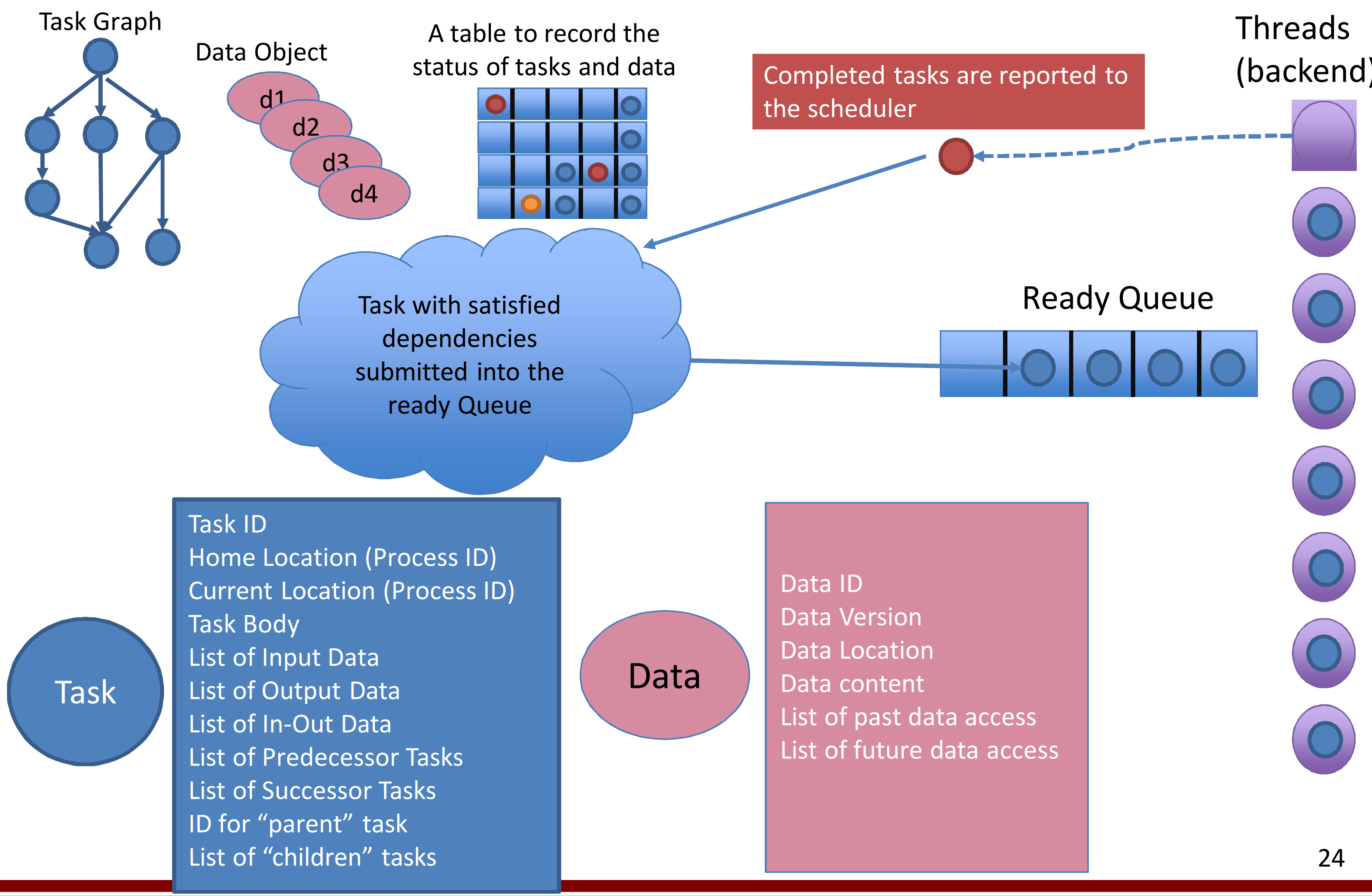


- AMT allows
  - Concurrent task execution
  - Overlap of communication and computation
  - Over-decomposition of Data
- Node/Process Failure is manifested as loss of task and data
  - Generic model for online local recovery
  - Recovery is done through task replay

Asynchronous Many Task Model (AMT)

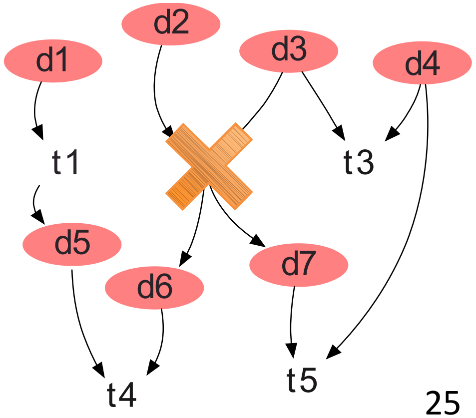
- Static vs. Dynamic AMT
  - Static: Task DAG is constructed before executing task
    - All task dependencies are known to the scheduler
  - Dynamic: Task DAG can be altered during task execution
    - Conditional task launch
    - Increases the scheduling overhead due to extra book keeping to prepare conditional task launch
- Distributed vs. On-node AMT
  - Distributed
    - Data movement across processes is handled by AMT runtime
  - On-node
    - MPI or its equivalent is exposed to the users
    - Performance concerns with MPI for multithreading

Example of AMT architecture



Resilience in AMT

- Task and Data are allowed to keep extra information to assist recovery
- Task represents a transaction in a workflow
- Failure within a program means failure of a task
  - Failure containment is relatively simple (do not launch dependent tasks)
  - Recovery is done through task replay
  - Failure is mitigated by replication of tasks and data
- Like MPI/SPMD model, availability of high performance persistent storage is essential for scalability



Potential Benefit of AMT Resilience

- Online Recovery
  - Fail-stop does not need to shut down the whole program
- Local Recovery
  - Rollback happens within a task or sub-DAG
  - Overlap of recovery and non-recovery tasks
    - Tasks that are not dependent on the recovery tasks can continue
    - Overlapping allows failure-masking
      - Delays due to multiple task failures are masked by overlapping execution of non-dependent tasks
    - Task/Work stealing
  - Recovery may block the progress of some pending tasks
  - The scheduler could allocate the resources to the other tasks

Challenges in AMT Resilience

- What is necessary to enable task replay?
  - Tasks?
    - Possible to make individual tasks self-recoverable?
  - Scheduler?
    - How to schedule recovery tasks?
    - How to replay/resubmit failed tasks?
- What information is necessary to retain a work-flow (subgraph) of the lost/corrupted tasks
  - Predecessor and Successor information
  - How many generations need to be kept?

Challenge in AMT Resilience

- Static vs. Dynamic DAG
  - Static DAG
    - Tasks and the workflow is fixed in “task parallel” region of computation
    - Less flexible
    - Dependencies can be analyzed before running tasks
    - Less info necessary for scheduling
    - Less information to enable task replay
  - Dynamic DAG
    - Task is created on-fly (conditional statement in a task triggers new task launch)
    - More flexible
    - Dependency information is processed on-fly
    - Scheduler needs flexible data structure to bookkeep the retired tasks and new tasks
    - More information is required to enable task replay

- Distributed AMT vs. On-Node AMT
  - Resilience mechanism for remote data access
    - Anything other than Message Logging?
    - Will abstraction help?
  - Integration with (ongoing) MPI’s fault tolerance framework

Major Approaches: Replication and Replay

- Task Replication Model
  - Analogue to N-modular redundancy
  - Proactive resilience
    - Failure may not trigger fail-stop or task replay
  - Replication cost needs to be controlled

- Task Replay Model
  - Replay tasks when they are failed.
  - Task flow allows local rollback for local failure

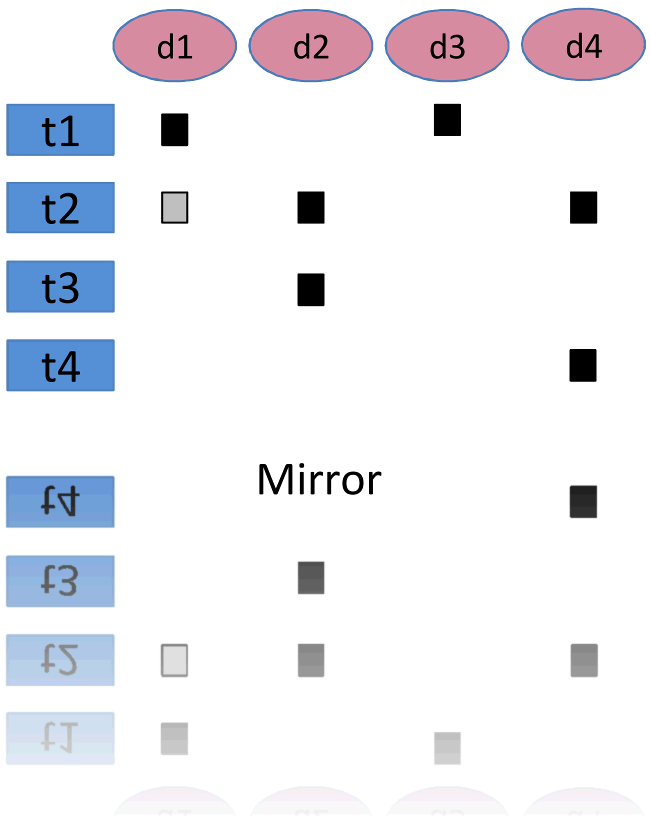
Major Approaches: Scheduler and Transaction

- Resilient Scheduler Model
  - Control infrastructure (scheduler) monitors the state of tasks and data
  - Correct the state if necessary
    - Resilient Task Collection (data parallel computation)
    - Resilient Task Scheduler (dependency aware)

- Resilient Transactional Model
  - The initial data of the task is stored in persistent storage
  - Task replays itself if it does not meet the “success” condition
  - Task is self-healing; however, hard to recover from loss of tasks
    - Containment Domains
    - Resilient Tasks

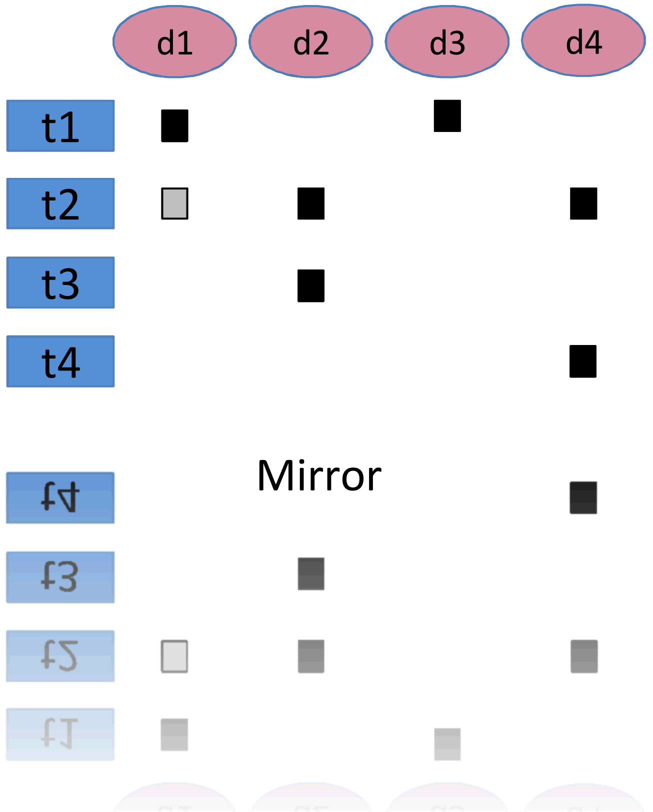
Task Collection (TC)

- Work by Ma & Krishnamoorthy PNNL
- Designed for work-stealing of data parallel computation
  - Record of tasks and associated data operations
  - Idle processes steal tasks by updating their metadata in the collection
- TC allows identifying lost tasks and their operations
  - Individual task info is mirrored
  - Replication of control information



Task Collection (TC)

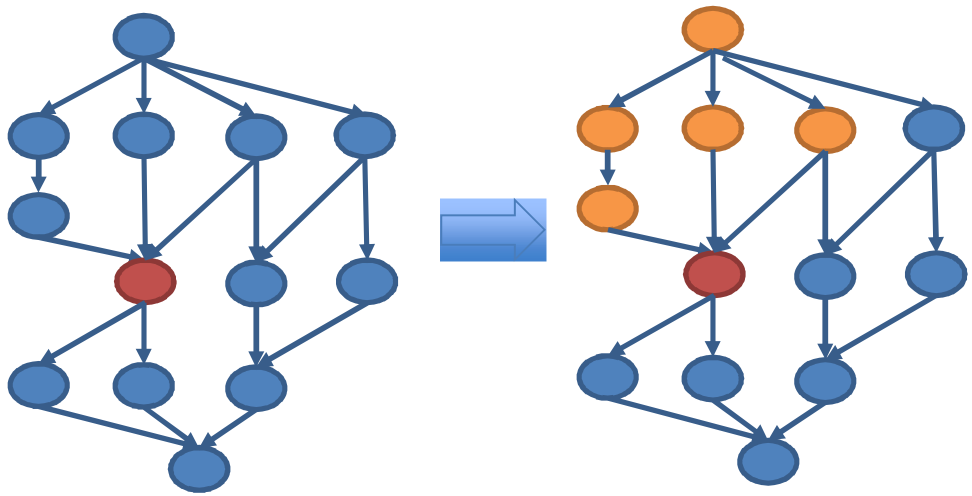
- TC records the history of all “data” transactions for each task
  - No message/update content
- Collective recovery
  - Lazy recovery is light-weight
    - Let all tasks finish and check for corrupted tasks
    - Resubmit all corrupted tasks
    - Cannot prevent failure propagation
    - In the worst case, all tasks are re-executed
  - More bookkeeping allows quick recovery
    - More overhead with the absence of failures
- Multiple TC can be used to manage multiple data parallel tasks



Drawbacks of Resilient TC

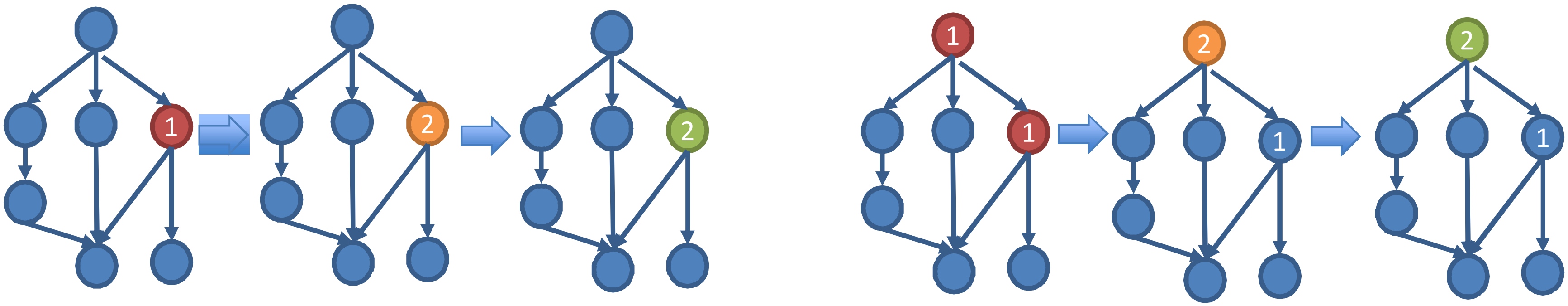
- Not applicable for arbitrary task dependencies
  - The order of data accesses implicitly describes the dependency
  - Extra information is necessary
- Collective operations can be expensive at large scale

Fault Tolerant Static Task Scheduling



- Work by Cao, Herault, Bosilca and Dongarra at UTK
- Use **parameterized task graph (PTG)** to trace back all predecessors of failed tasks until the persistent input data (checkpoint) is reached
- Periodic task-based checkpointing and algorithm based fault tolerance reduce the number of tasks to be re-executed
- Applicable for distributed AMT
  - Failure notification is assumed underneath the runtime

Fault Tolerant Dynamic Task Scheduling



- Work by Kurt & G. Agrawal(OSU), Krishnamoorthy (PNNL) and K Agrawal (Washington U)
- Extend dynamic task graph scheduling implemented on the top of work-stealing data parallel tasking runtime (Cilk)
- Runtime Scheduler monitors the status of tasks
  - Try-Catch block to access task status
  - Correct the state of failed task and then resubmit a failed task using a new “reincarnation” number
  - Input data block error could trigger the recovery of predecessor tasks (tasks executed in the past); the current task is pulled out from the queue
- Impose a few constraints in Task graph
  - Graph is not expanded beyond the tasks being executed and their direct successors

Fault Tolerant Dynamic Task Scheduling

- Task and Data need to contain more information than the static task scheduling
  - Reincarnation (EPIC) of tasks
  - Flexible data structure for data dependency information
  - Data versioning
- Some information may not be available
  - Future data accesses

Task

- Task ID
- Home Location (Process ID)
- Current Location (Process ID)
- Task Body
- List of Input Data
- List of Output Data
- List of In-Out Data
- **List of Predecessor Tasks**
- **List of Successor Tasks**
- ID for "parent" task
- List of "children" tasks
- **ID for reincarnation**

Data

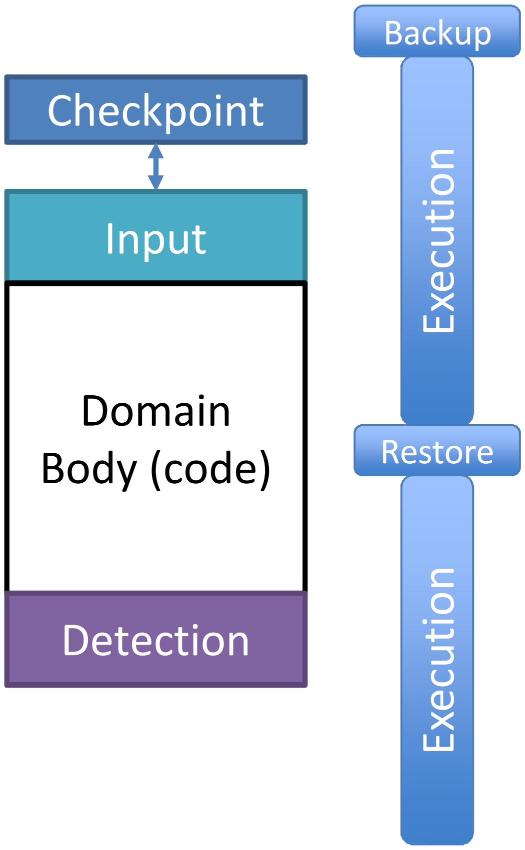
- Data ID
- **Data Version**
- Data Location
- Data content
- **List of past data access**
- (List of future data access)

Drawbacks of Fault Tolerant Scheduling

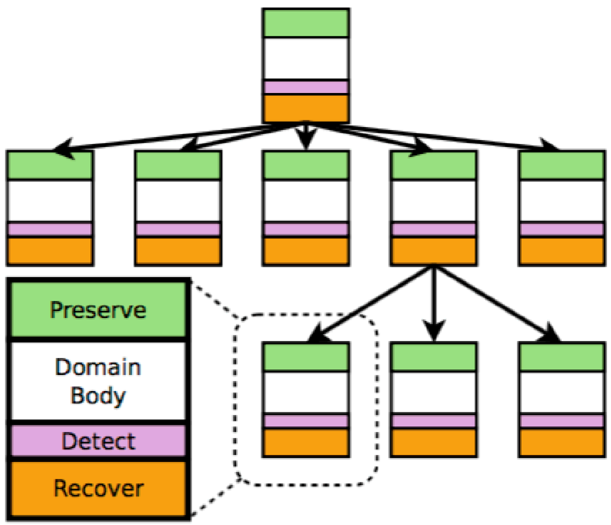
- No coverage for hard failures (loss of tasks and data blocks)
  - Duplication of scheduling information is essential
  - One idea is mapping some schedule information to task-collection to handle hard failures
- Large cost in meta data management (global lock) for dynamic task graphs and work stealing support
- Assumption of the persistence of control information
  - How to recover the loss/corruption of control information?

Transaction Model: Task Self-Replay

- By BSC on resilient OmpSs
- Add checkpoint/restart capability to every task
  - Checkpoint before the execution of body
  - When task is not successful, repeat the same task
  - Input data is derived checkpoint
  - DARMA team did similar work using checksums
- Assume all failures can be contained within single tasks
- Unclear how to cover a loss of tasks (process/node failures)
- Possible to support Node-AMT+MPI model
  - Receiver-based message logging



Hierarchical Transaction Model: Containment Domains



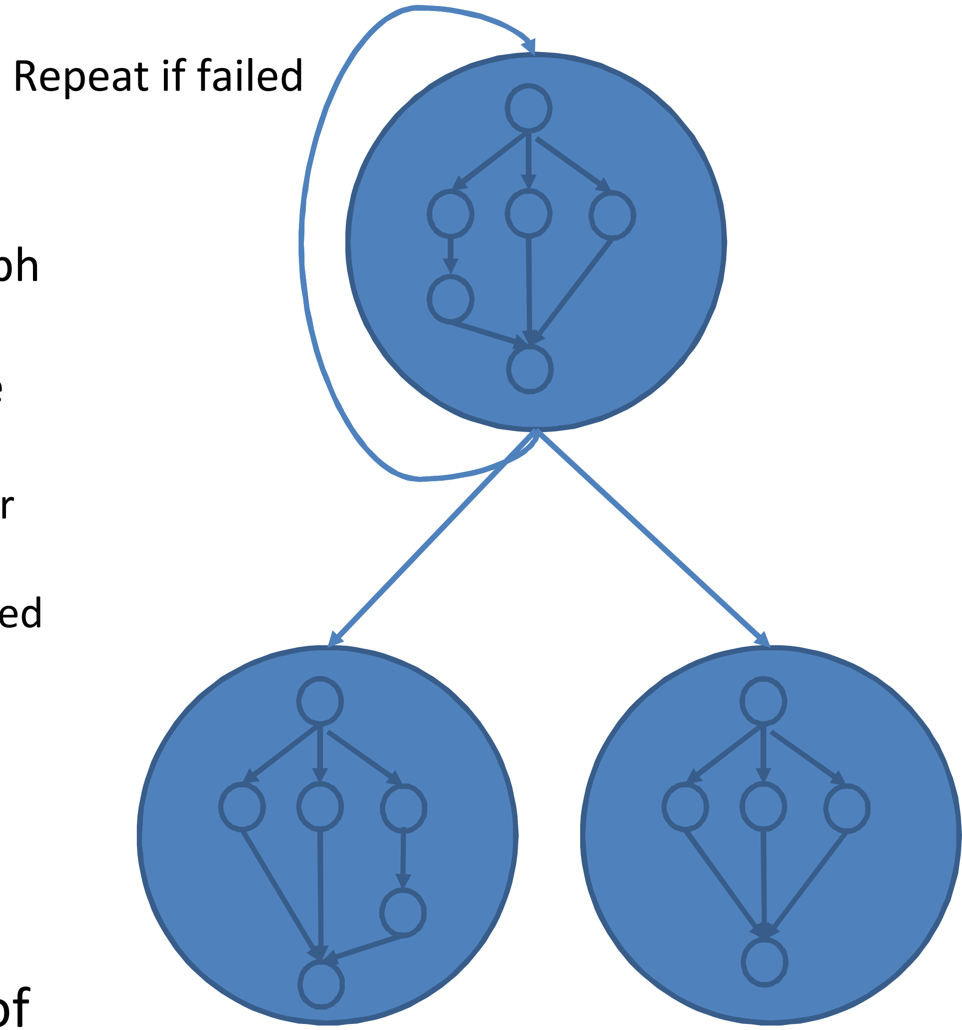

```
void task<inner> SpMV( in M, in Vi, out Ri) {
 cd = create_CD(parentCD);
 add_to_CD_via_copy(cd, M, …);
 forall(…) reduce(…)
   SpMV(M[…],Vi[…],Ri[…]);
 commit_CD(cd);
}

void task<leaf> SpMV(…) {
 cd = create_CD(parentCD);
 add_to_CD_via_copy(cd, M, …);
 add_to_CD_via_parent(cd, Vi, …);
 for r=0..N
  for c=rowS[r]..rowS[r+1] {
    Ri[r]+=M[c]*veci[cIdx[c]];
    check {fault<fail>(c > prevC);}
    prevC=c;
  }
 commit_CD(cd);
}
```

Courtesy: Mattan Erez at UT Austin

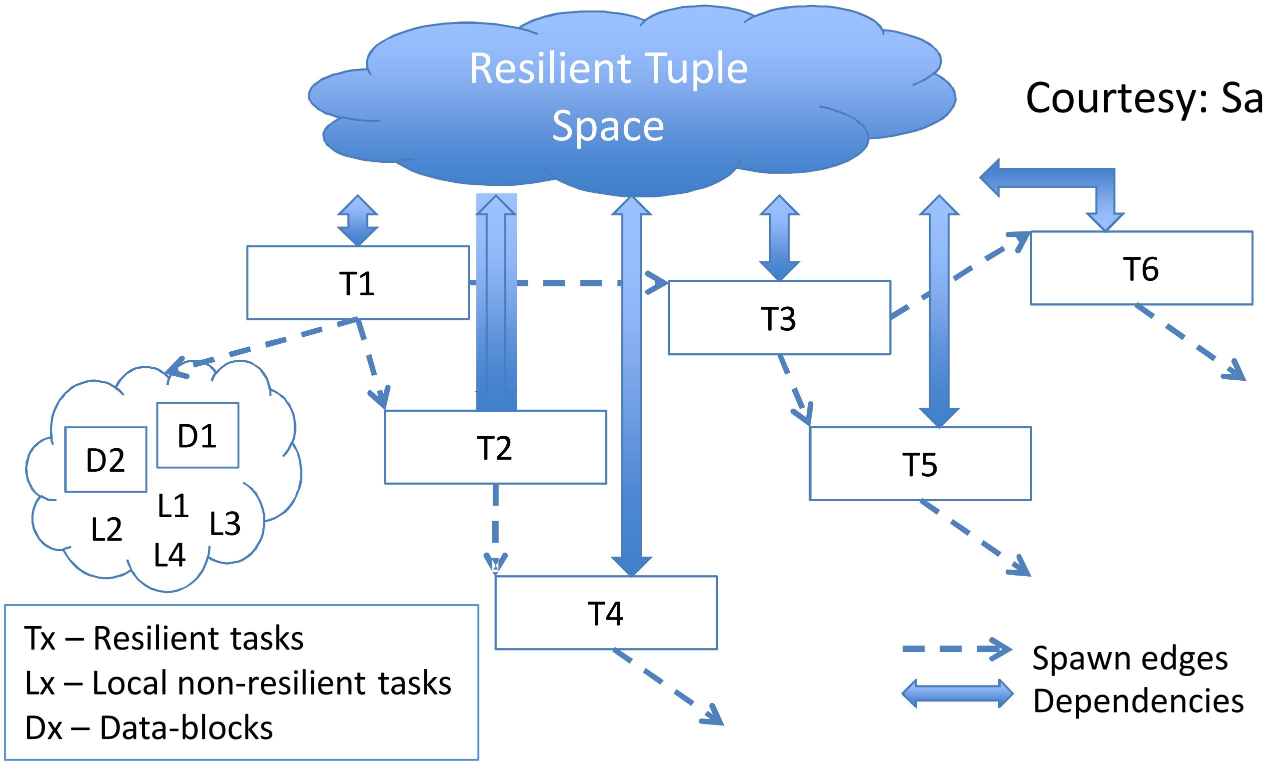
- Each domain defines
  - Data to be protected
  - Failure detection
  - Body of the code
  - Recovery is done through replay
- Hierarchical task representation allows localized recovery
- **Many AMTs do not support hierarchical composition of tasks**

Selective Transaction Model

- Work by Rice U + Sandia
- Selective Transaction Model
  - Create a big task that holds a graph of children tasks
  - Re-execute big tasks when failure occurs
    - Parent task stays in the scheduler until all children finish
    - All children tasks are not protected
  - Can be seen as a variant of Containment Domains
- Like CDs, runtime needs to support parent-children task model
- Reduce the potential overhead of task execution latency
  - Less frequent checkpointing



Ongoing work: SNL, Rice U and Rutgers U



Courtesy: Sanjay Chatterjee

- Resilient version of Open Community Runtime (OCR)
  - Resilient tasks can spawn non-resilient tasks
  - Resilient tasks are replayable upon crash
  - Resilient task and resilient data objects are maintained by resilient data warehouse (resilient tuple space)

Conclusion

- Scalable Application Recovery at Scale
  - Extend Fault-Tolerant MPI prototype
    - Hot spare processess
    - In-meory checkpoiting
  - Application specific message logging to allow localized online recovery

- Future work explore resilience in AMT runtime
  - Require vertical integration
  - Lots of opportunities on the horizon

Acknowledgement

- Janine Bennett, Robert Clay, Michael Heroux, Nicole Slattengren and Jeremiah Wilke (Sandia National Labs)
- Marc Gamell and Rob van der Wijngaart (Intel)
- Sanjay Chatterjee and Vivek Sarkar (Rice U)
- Manish Parashar (Rutgers U)
- George Bosilca, Aurélien Bouteiller and Thomas Herault (University of Tennessee, Knoxville)