SAND2014-18249PE

Toward Local Failure Local Recovery (LFLR) Resilience Model Using MPI-ULFM

Keita Teranishi and Michael A. Heroux

Sandia National Laboratories

EuroMPI/ASIA'14, Kyoto, Japan

September 11, 2014

SAND# 2014-17339 PE

Sandia National Laboratories

Exceptional service in the national interest

U.S. DEPARTMENT OF ENERGY

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

Overview

Sandia National Laboratories

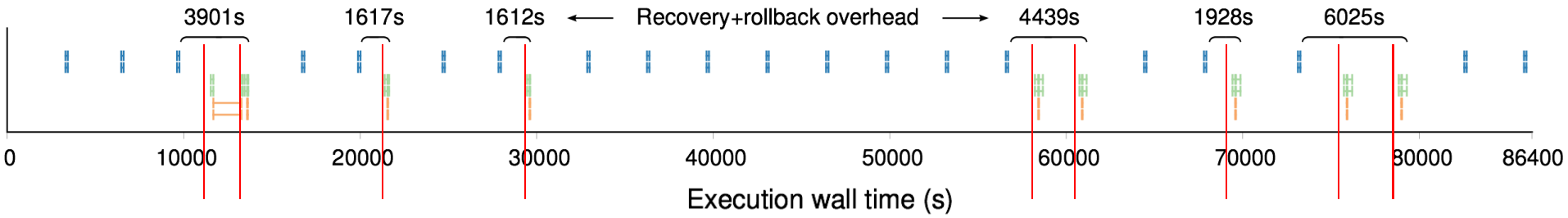
- **Motivation for Local Failure-Local Recovery (LFLR)**
- Architecture for LFLR
- Application Recovery
- Results
- Discussion
- Conclusions

Motivation for LFLR:

Sandia National Laboratories

- Current practice of Checkpoint/Restart is global response to single node (local) failure
  - Kill all processes (global terminate), then restart
  - Dependent on Global File system
  - SCR (LLNL) is fast, but adheres global recovery
- Single node failures are predominant
  - 85% on LLNL clusters (Moody et al. 2010)
  - 60-90% on Jaguar/Titan (ORNL)
- Need for scalable, portable and application agnostic solution
  - **Local Failure Local Recovery Model (LFLR)**
  - **SPMD model**

Example – S3D production runs



- 24-hour tests using Titan (125k cores)
- 9 process/node failures over 24 hours
- Failures are promoted to **job failures**, causing all 125k processes to exit.

As a result, checkpoint (5.2 MB/core) has to be done to the PFS

- Checkpoint data: **55 s** Total: 1.72 %
- Restarting processes: **470 s** Total: 5.67 %
- Loading checkpoint: **44 s** Total: 1.38 %
- Rollback overhead: **1654 s** Total: 22.63 %
- Total overhead due to fault tolerance: **31.40 %**

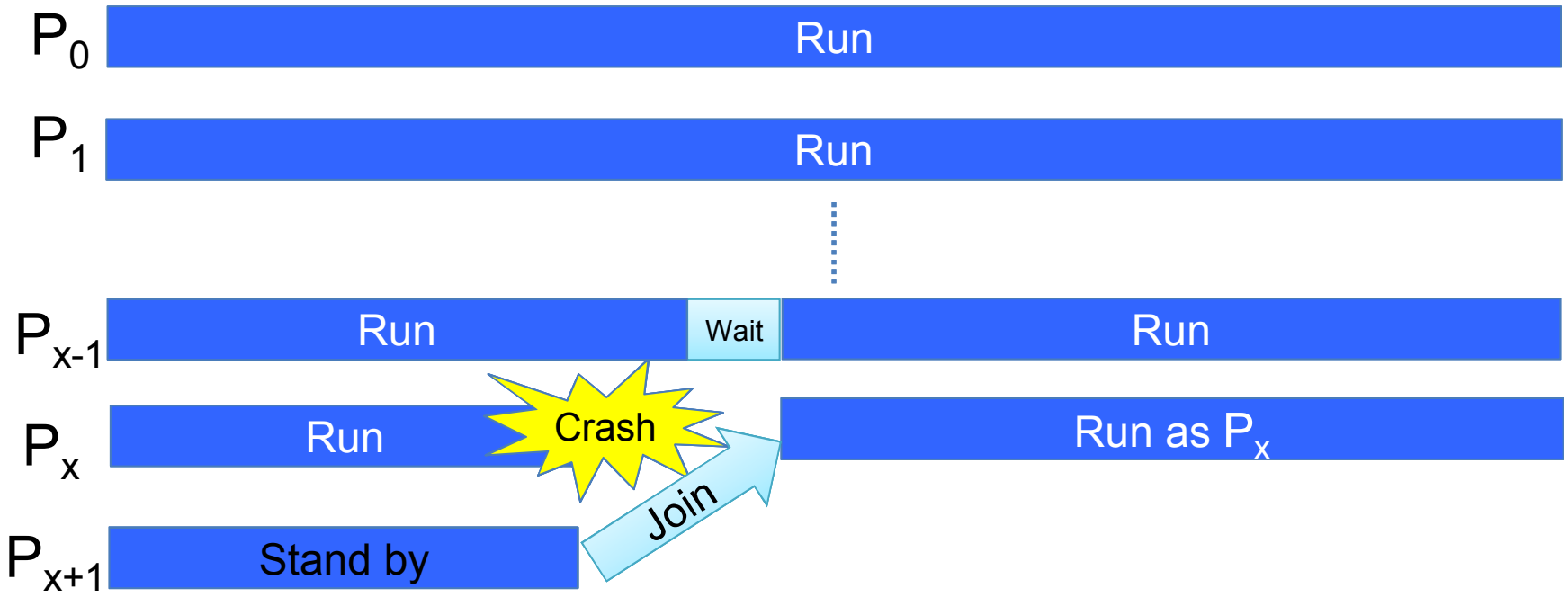
- **Courtesy of Hemanth Kolla.**
- ***Exploring Automatic Online Failure Recovery for Scientific Applications at Extreme Scales*, M. Gamell, M, Parashar, D. Katz, H. Kolla and J. Chen, to appear in SC14.**

TODAY: 20-30%
Future: ???%

LFLR: Alternative Solution for Checkpoint/Restart

Sandia National Laboratories

- LFLR overcomes the shortcomings of C/R
  - **The remaining processes stay alive with single process failures**
  - Application-based
  - Multiple implementation options
    - Roll-back, roll-forward, asynchronous, algorithm specific, etc.
  - Active Hot Spare Process for recovery



Overview

Sandia National Laboratories

- Executive Summary
  - Overview
  - Milestones Completed
- Motivation for Local Failure-Local Recovery (LFLR)
- **Architecture for LFLR**
- Application Recovery
- Results
- Discussion
- Conclusions

4 requirements to enable LFLR

Sandia National Laboratories

1. Runtime and middleware that permit parallel program execution to continue under process failures.

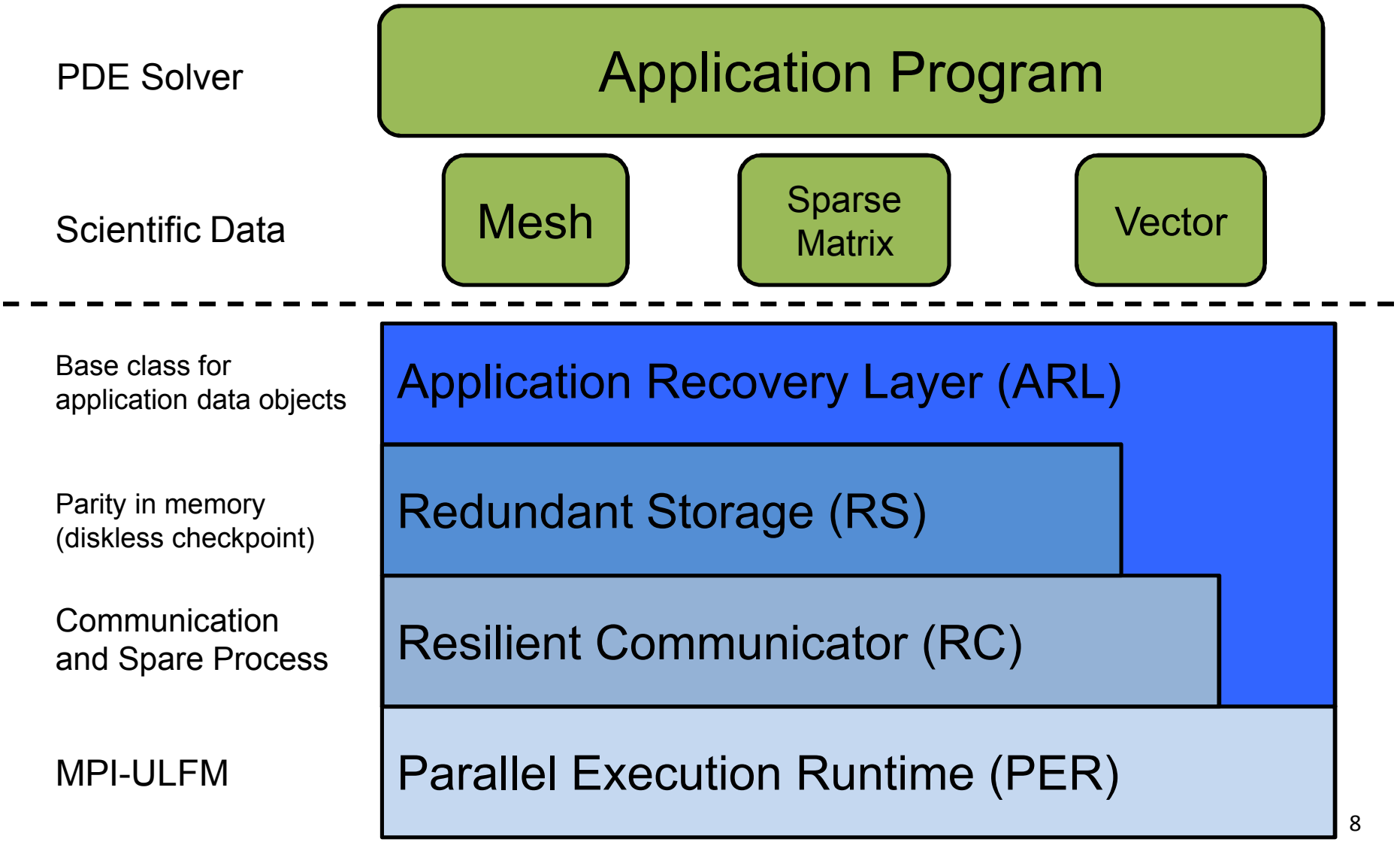
2. Runtime and middleware that provide replacement processes for the failed ones, and allow application programs to query the status of processes.

3. Redundant persistent storage for restoring the data associated with failed process.

4. Tools and framework to build application specific recovery schemes.

Application and Architecture Agnostic Framework to enable LFLR

Sandia National Laboratories



Parallel Execution Runtime (PER): Running Applications Through Process Failures

Sandia National Laboratories

- Requirement: The remaining process can run through the failure of single (or a few) processes
- LFLR's backend
- Possible to replace any resilient runtime other than MPI
  - Ongoing work: Our in-house runtime built for UGNI (Cray XE/XC)

Resilient MPI for Parallel Execution Runtime

Sandia National Laboratories

- Several Fault Tolerant MPI implementations
  - FT-MPI (Fagg et al.), LA-MPI (LANL) and NR-MPI (NUDT, China) makes process recovery up on failure
- MPI-ULFM provides a minimum set of APIs for resilience
  - No process recovery
  - Users are responsible for writing application recovery
  - Several MPI calls for fixing MPI communicator
    - MPI_Comm_agree : Check the global status of MPI_Comm
    - MPI_Comm_revoke: Invalidate MPI Communicator
    - MPI_Comm_shrink: Fix MPI Communicator removing dead process
  - Error notification when receiving message from lost process
  - Prototype code is available at http://fault-tolerance.org

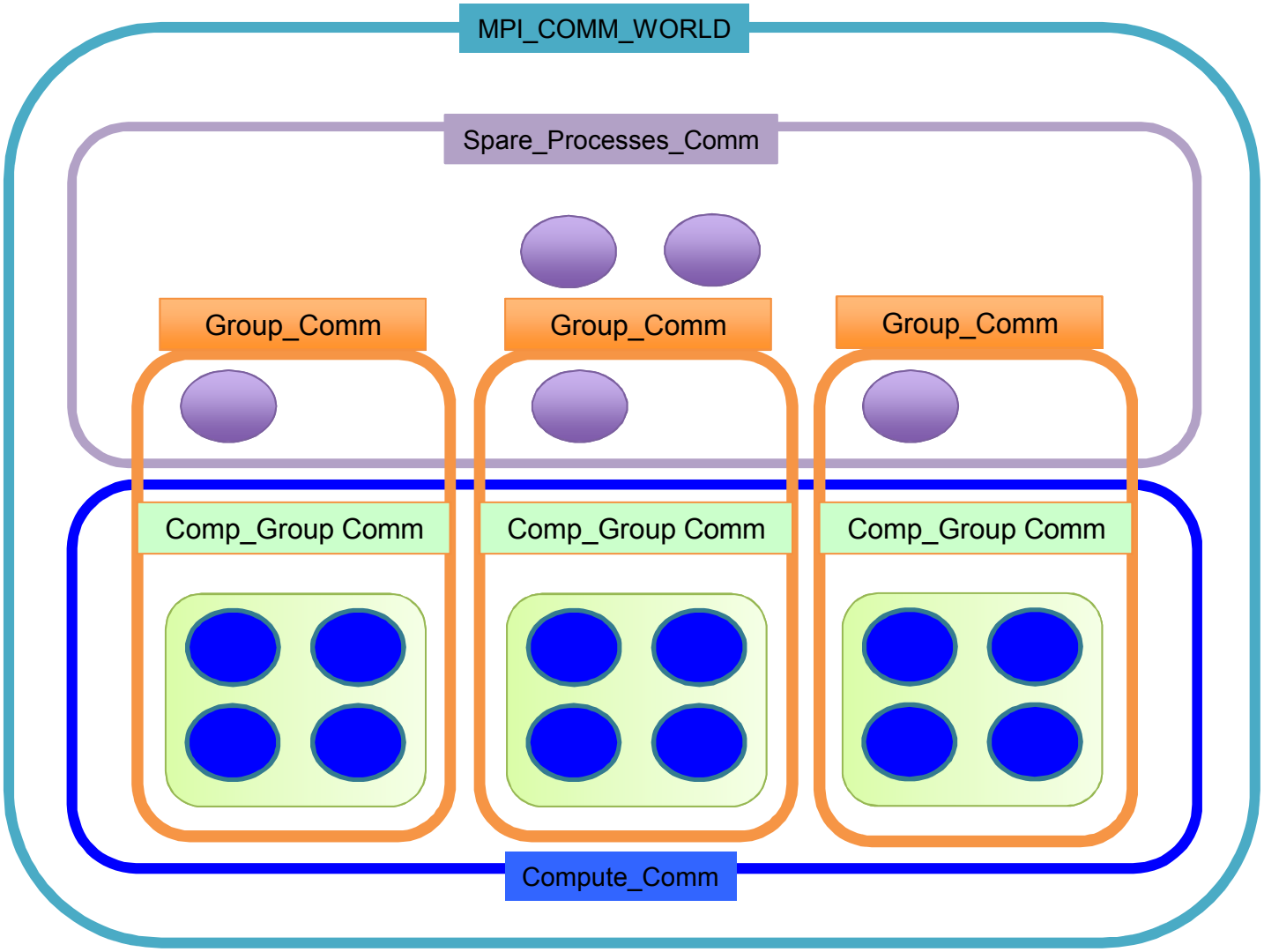
Resilient Communicator (RC): Handles Message Passing and Spare Process Management

Sandia National Laboratories

- Process Management
  - Spare Processes
  - Process Grouping for scalable commit and restore
- Direct access to MPI communicator
  - For integration with the existing code
  - Allows MPI's message passing call and failure notification
- Wrapper Functions
  - Support Non-ULFM capability
    - Algorithm selection for resilient collective
    - Status check, etc.

Resilient Communicator: Current Design

Sandia National Laboratories



Redundant Storage (RS): Efficient Persistent Data Storage to Restore the Data from Lost Process

Sandia National Laboratories

- Application data needs to be stored in persistent storage
  - Accessible after process loss
  - Data needs to be placed/copied outside the process
- Several Options
  - Memory for the executable
    - Diskless checkpointing
      - RAID-like redundancy
  - Local disk
    - Caching
    - RAM, NVRAM and disk
    - Involves system configuration to set the size of /tmp
  - Burst Buffer
    - For NERSC-8 and Trinity system
    - Need good APIs to access it!
  - Global File Systems

Redundant Storage API (Tentative)

Sandia National Laboratories

register(void *data , size_t size , RC, distribution, storage_option)

commit()

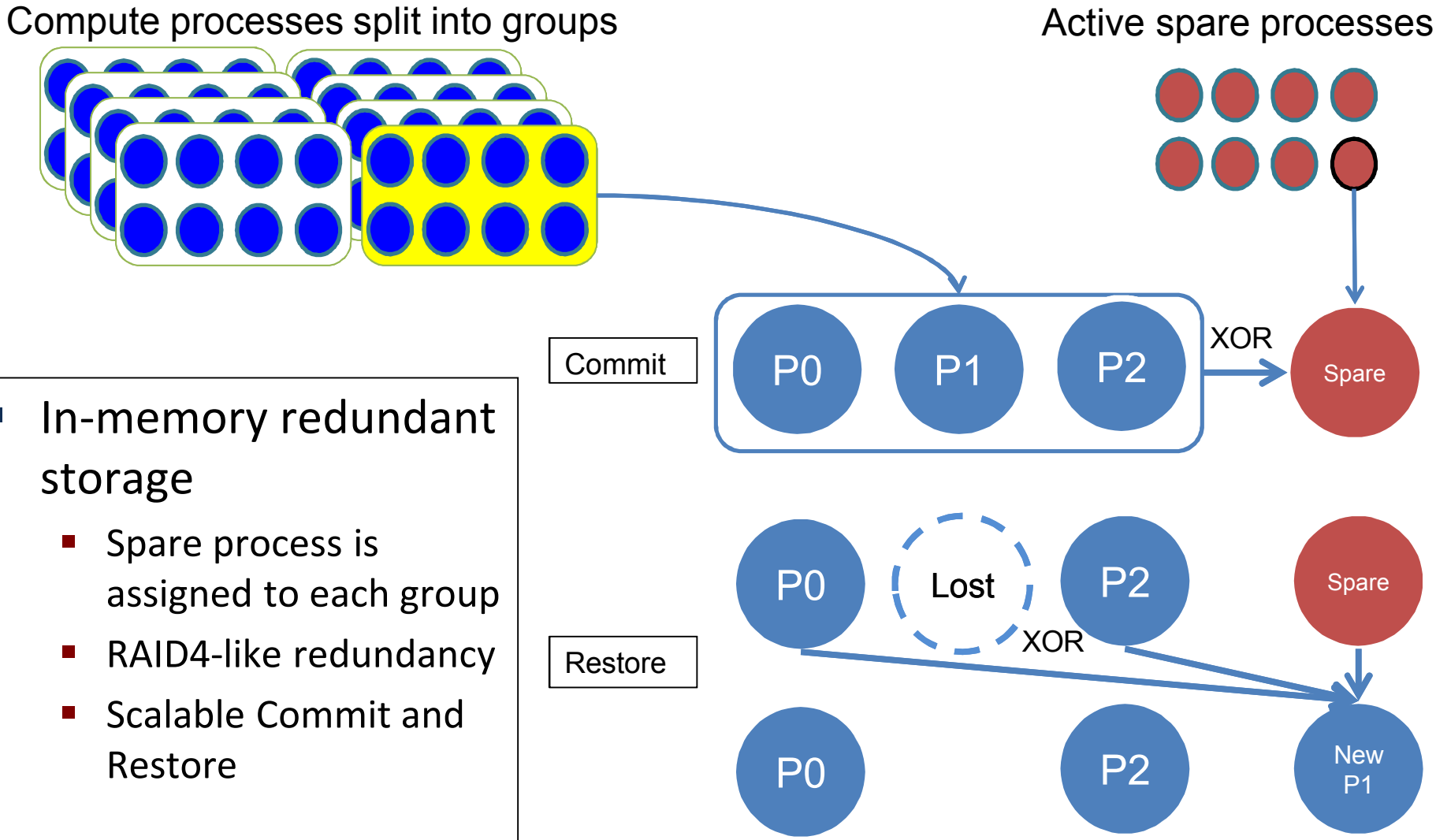
restore(void *data, size_t *size)

unregister( )

- “distribution” defines the type of data distribution to help flexible recovery of the data in a particular piece of memory
  - Distributed (Each process owns its local piece)
  - Shared (Shared by multiple processes)
  - Local (Exclusive to single rank)
- We provide options for the underlying storage scheme.

Scalable Recovery using Diskless Checkpointing and Active Spare

Sandia National Laboratories

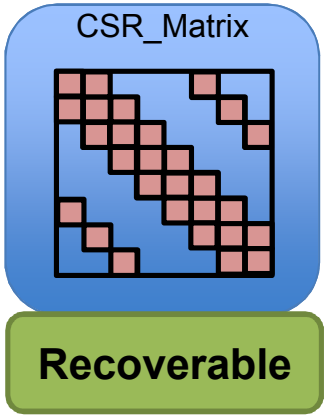


- In-memory redundant storage
  - Spare process is assigned to each group
  - RAID4-like redundancy
  - Scalable Commit and Restore

Application Resilience Layer (ARL) : Builds Application Specific Recovery Scheme

Sandia National Laboratories

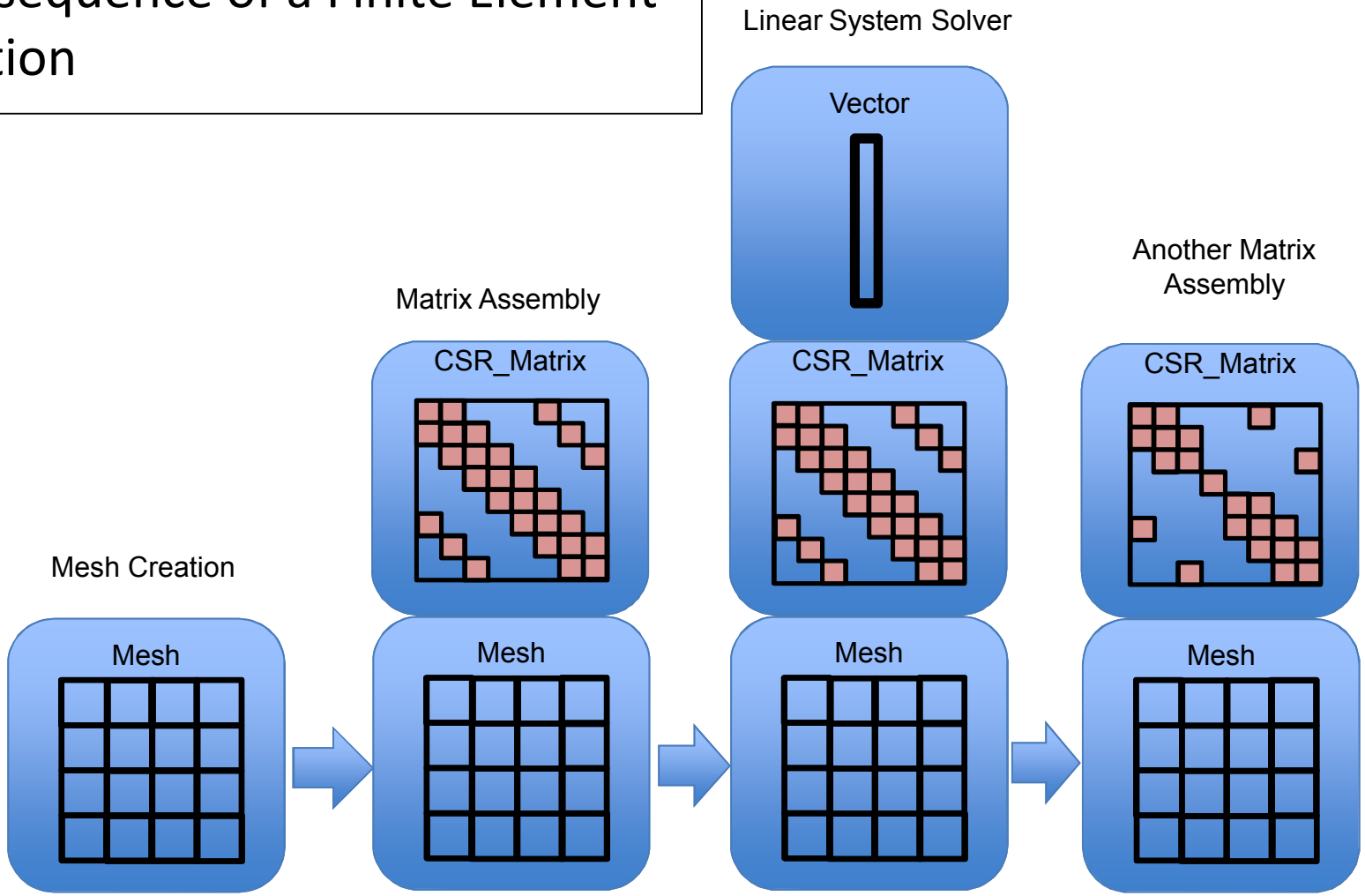
- 2 Components
  - LFLR_registry
  - recoverable
- LFLR_registry monitors the allocation of scientific data objects
  - Provides **stack of pointers for scientific objects**
  - Scientific object is pushed to LFLR_registry by its constructor
    - Popped when destroying (by destructor)
- **recoverable** is a base class for scientific data objects
  - Orchestrate **stack of Redundant Storage objects** to perform application specific data recovery
  - Allows the user to describe the application specific data recovery scheme
  - Recovery method can be implemented without RS if necessary



ARL: LFLR_Registory Manages Stack of Data Objets

Sandia National Laboratories

Typical sequence of a Finite Element application



Integrating ARL to the existing Scientific Data Structure

Sandia National Laboratories

```
struct CSR_Matrix {
        std::vector row_offset;
        std::vector col_index;
        std::vector coefs;









}
```

```
struct CSR_Matrix :: public recoverable {
        std::vector row_offset;
        std::vector col_index;
        std::vector coefs;

        CSR_Matrx( LFLR_registry *reg) {
                myreg = reg;
                myreg.push(this);
        }

        ~CSR_Matrix() { myreg.pop(); …}

        int commit() {
                rsstack.commit();
        }
        int restore() {
                rsstack.restore();

        }
}
```

Overview

Sandia National Laboratories

- Motivation for Local Failure-Local Recovery (LFLR)
- Architecture for LFLR
- **Application Recovery**
- Results
- Discussion
- Conclusions

Application Recovery needs to Recover 3 Entities

Sandia National Laboratories

- Process (RC)
- Application Data (RS, ARL)
- Application State
  - Skeletonize Program for Spare Process

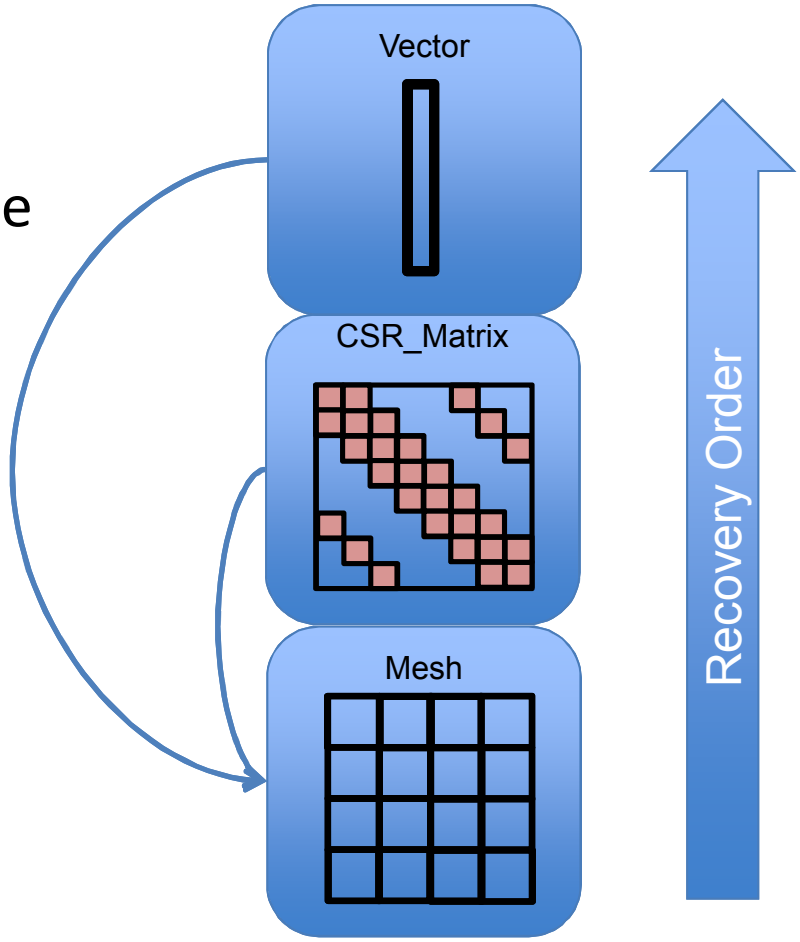
Process Recovery

Sandia National Laboratories

- Error Detection
  - MPI-ULFM's capability
  - Once Detected: Set RC.state = false;
- Error Notification
  - OMPI_Comm_agree(MPI_Comm) to check the status of all the process associated with the MPI communicator
  - Ongoing Work:
    - More involvement of spare process for each group
    - MPI_Comm_revoke to allow lazy notification
- Recovery
  - Several MPI-ULFM calls
    - OMPI_Comm_shrink to eliminate lost process
    - OMPI_Comm_create to include spare process to computing process
    - OMPI_Comm_create and MPI_Group_create to modify other sub-communicators

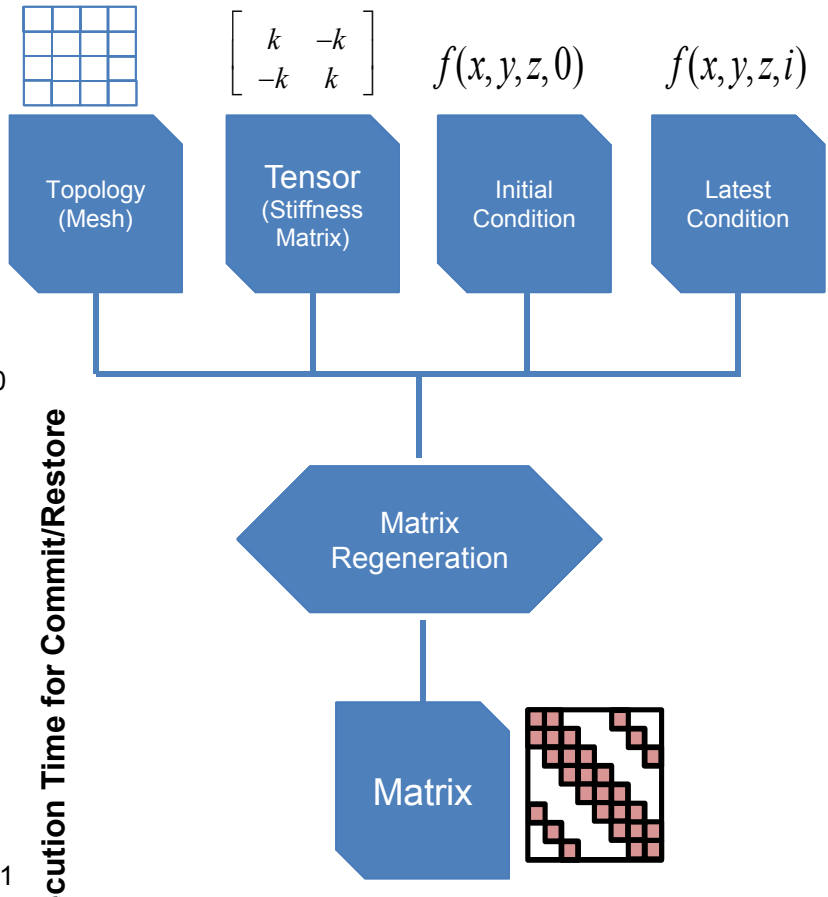
Data Recovery Involves LFLR_Registory

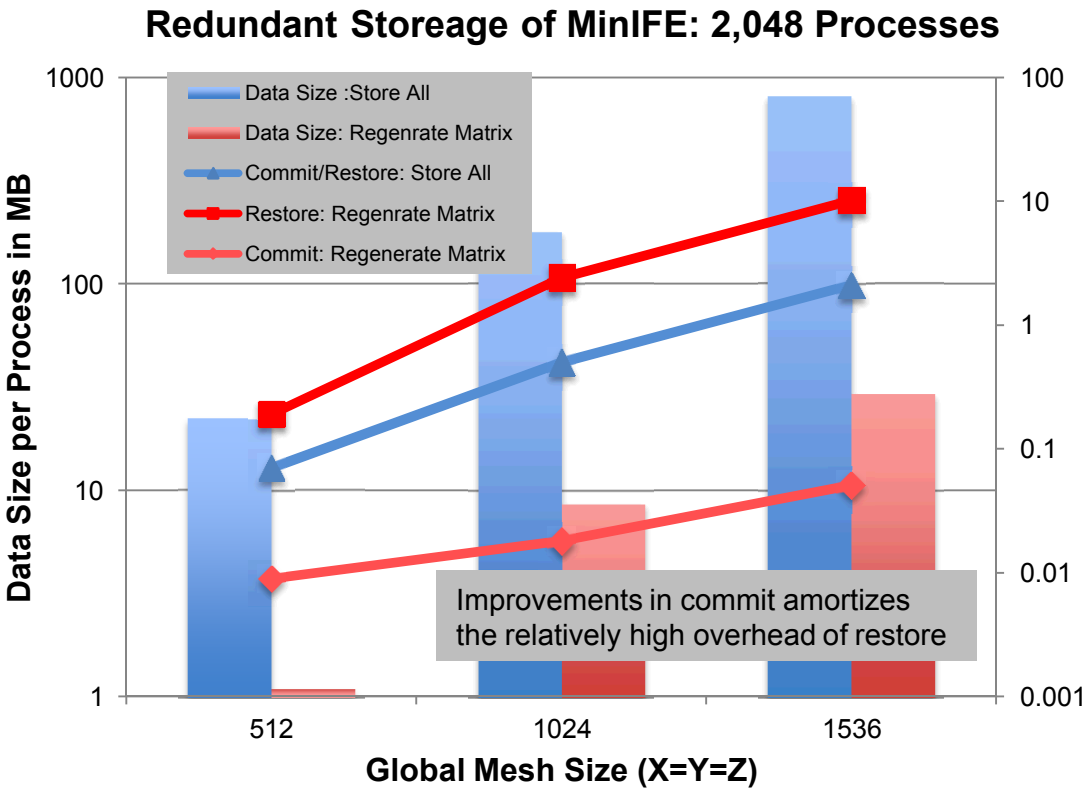
- Recovery Order is bottom-up of the stack in LFLR_registory
- The data in the bottom can be the source for the those above
- Recovery the data required to execute the latest state of application



Leveraging the Application for Efficient Storage Reduction

- Exploit Data Dependencies of Application data
- Recovery through inexpensive local computation (Example: MiniFE)
  - Reuse the existing Matrix Assembly code
  - Localized matrix regeneration
  - Substantial storage reduction





Code Transformation to allow spare process to obtain the latest state of the application

Sandia National Laboratories

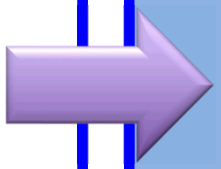
```
Compute(comm,data) {
        CSR_Matrix A(comm);

        for …  {
                Vector x(comm);

                // Do real computation

        }

}
```



```
Compute(rcomm, registry, data, flg) {
        CSR_Matrix A(rcomm,&registry);
        registry.commit();
        for ( i = 0; …  {
                Vector x(rcomm,&registry);
                registry.commit();
                if( flag == true ) {
                        // Do real computation
                }
                if( failed ) {
                        rcomm,recover();
                        registrty.recover();
                        if(I am joining from spare) {
                                flag = true;
                        }
                        i--;
                }
        }  // x is unregistered
        // Check failure here
}
```

Overview

Sandia National Laboratories

- Executive Summary
  - Overview
  - Milestones Completed
- Motivation for Local Failure-Local Recovery (LFLR)
- Architecture for LFLR
- Application Recovery
- **Results**
- Discussion
- Conclusions

Preliminary Result: Experiment Settings

Sandia National Laboratories

- Time Stepping PDE
  - 20 steps, multiple linear system solve
- LFLR enabled
  - Vectors are stored in every time step
- Weak scaling
  - 64x64x64 for ULFM for 4 cores and increase the problem size (x*y*z) linearly
  - Cray Cluster with SandyBridge (2.6Mhz) 16 cores (2CPU) per node, QDR Infiniband
  - Process failure during linear system solve (2048 PEs)
    - MPI-ULFM with our own fix for resilient collective

Resilient Time-Stepping MiniFE

Sandia National Laboratories

Create Mesh M

Compute Matrix A out of M

Save M in Persistent Storage

Do until the last time step

b_i and b_{i-1} in Persistent Storage

Get new b_i from x_{i-1} (Update Boundary Condition)

Solve $Ax_i=b_i$ (Linear System Solution)

if the linear system solver fails, try the same iterative step

end do

Process loss is checked periodically

- **Local vector is stored with the subscript (iteration count) info**
- **Allow linear system solver to crash or end up with wrong solution**
  - Process loss
  - Convergence failure due to silent data corruption
- **Repeat the same iteration when linear system solver fails**
  - Need to get x_{i-1} and b_{i-1}

CG Iteration with Process Failure Check

Algorithm 2 Conjugate Gradient with Process Failure Detection

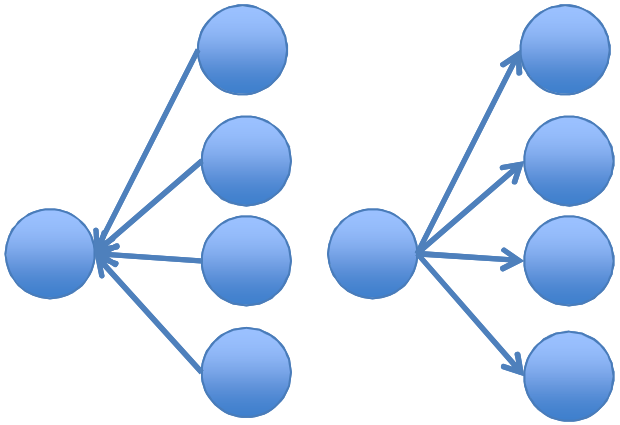
Input: A, b and initial guess x_0
$r_0 := b - Ax_0$, $x := x_0$, $p_0 := r_0$, $i := 0$
while $\|r_i\|$ is small enough or Failure **do**
  $q := Ap_i$ (Error Detection)
  $\alpha := \|r_i\|^2/(p_i^T q)$ (Error Detection)
  $x := x + \alpha p_i$
  $r_{i+1} := r_i - \alpha q_i$
  $r_{new} := \|r_{i+1}\|^2$ (Error Detection)
  $\beta := r_{new}/\|r_i\|^2$
  $p_{i+1} := r_{i+1} + \beta p_i$
  $i := i + 1$
  if (Process failure is found) **then**
    break
  end if
end while

Error Detection at MatVec and Dot Product

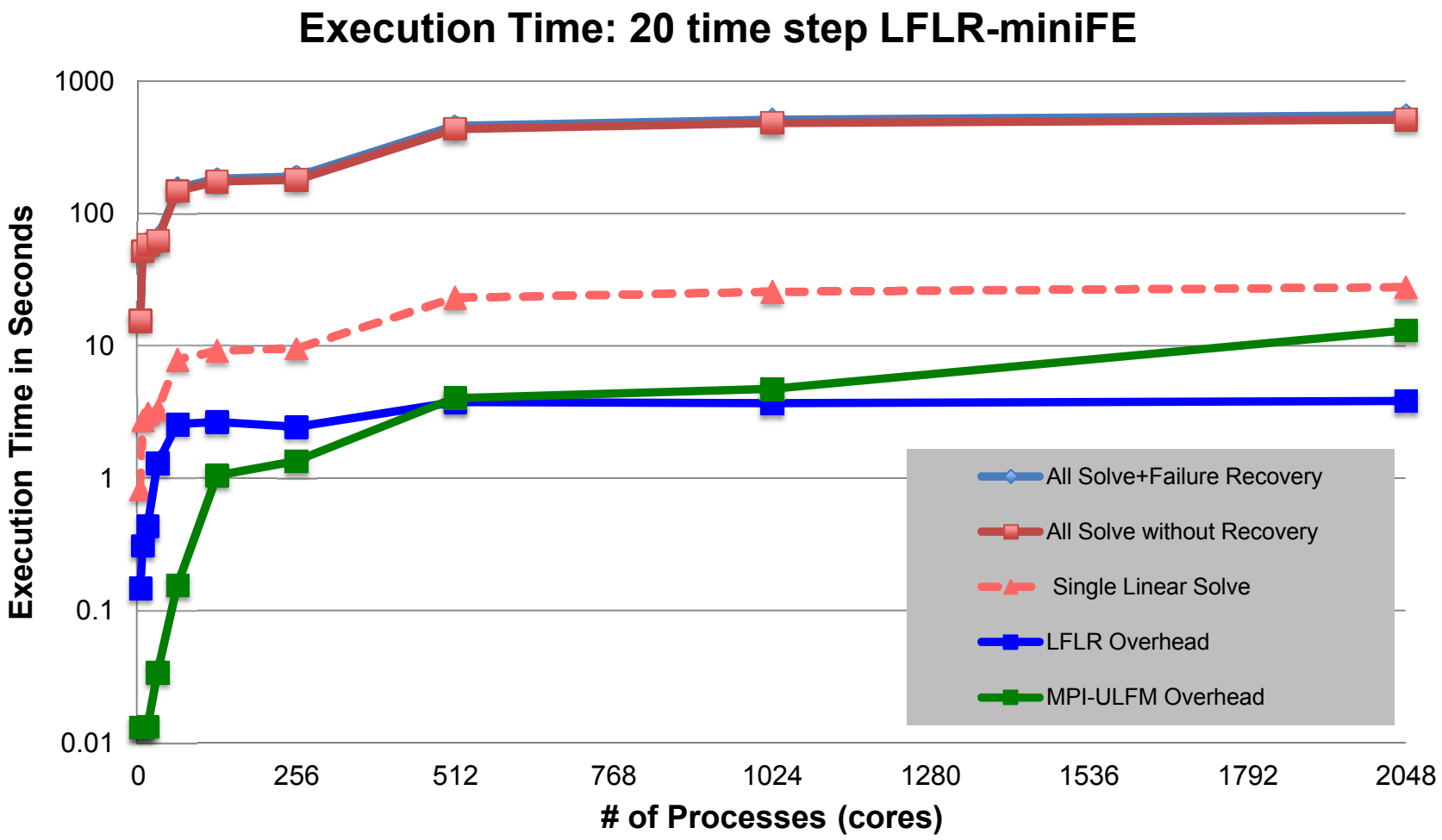
Performance Fix in MPI-ULFM

Sandia National Laboratories

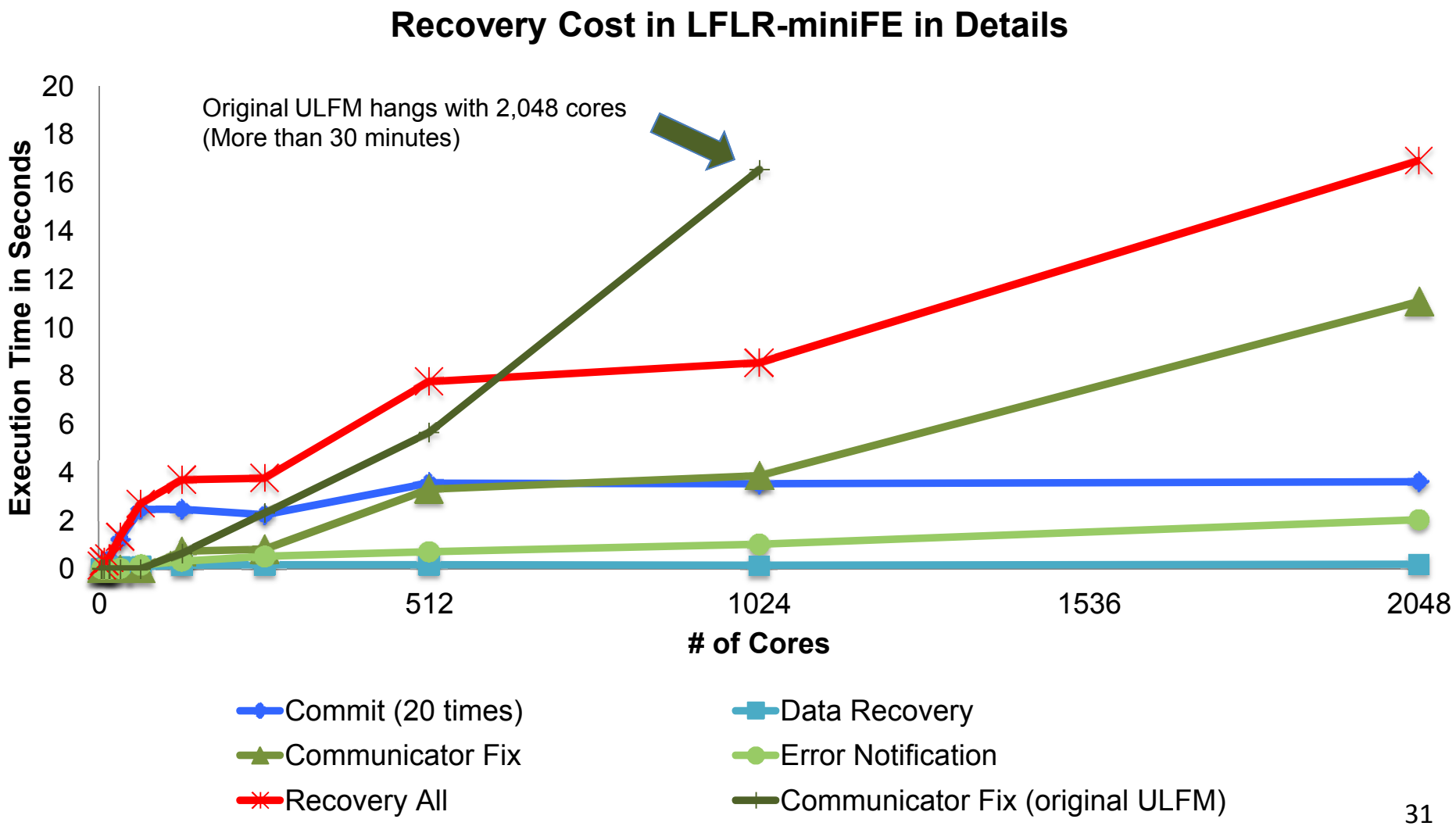
- Enable Tuned Collectives
- Enable Tree based resilient MPI_allreduce
- Replaced resilient Global Agreement protocol used for OMPI_Comm_shrink and MPI_Comm_create.
  - The original version was one-to-all and all-to-one
  - Applied tree-based version based on Hursey and Graham



Scalable Recovery: LFLR-enabled miniFE

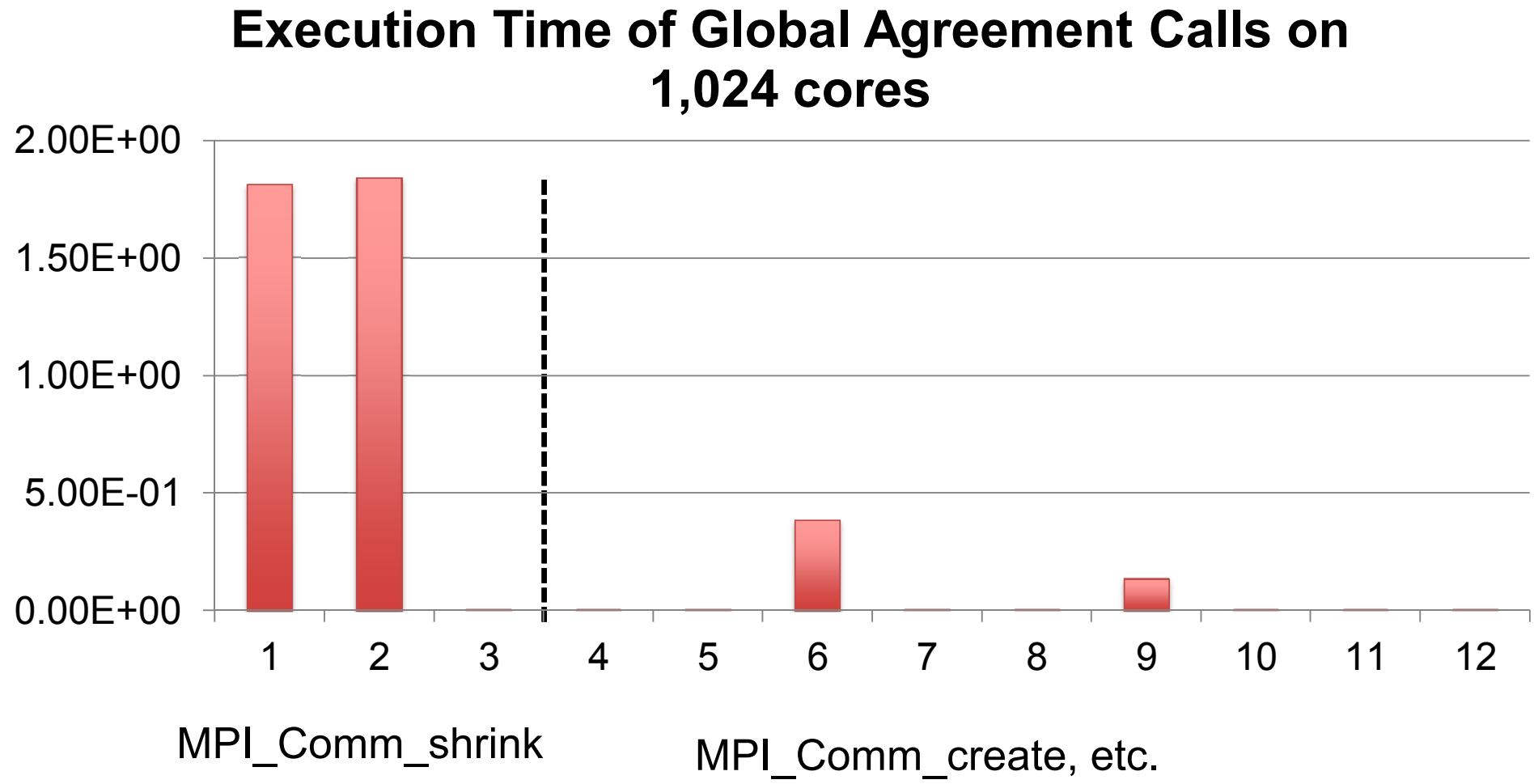


Scalability Issues of LFLR-minIFE: MPI-ULFM



Performance Fluctuation of Resilient Agreement with MPI-ULFM

Sandia National Laboratories



Overview

Sandia National Laboratories

- Motivation for Local Failure-Local Recovery (LFLR)
- Architecture for LFLR
- Application Recovery
- Results
- **Discussions**
- Conclusions

Why spare processes?

Sandia National Laboratories

- Simple model indicates small increase in computing nodes (less than 1%) to run 7 days.
  - Large ROI for Reliability
- Eliminates complications of running SPMD program with fewer processes
- Non-SPMD programming model may not require this requirement
  - Scheduler can assign data and task to maintain a good load balancing
  - Need Performance model w spare and w/o spare

Why not uncoordinated Checkpoint?

Sandia National Laboratories

- Uncoordinated Checkpoint allows asynchronous commit, and recovery does not involve global roll-back
  - Looks promising
- But ….
  - Realistic implementation involves good infrastructure support
    - Fast redundant and persistent storage
    - Spare process
    - Message logger
  - Complex protocol to minimize the amount of rollback for the remaining process
  - Need good understanding in the application performance (communication) patterns
    - Clustering (Cappello et al.)
    - Hybridize with Coordinated Checkpointing (Ferreira et al.)
    - LFLR allows application users to coordinate recovery scheme with the application!

Problem: Lack of Asynchronous Spare Process Assignment

Sandia National Laboratories

- OMPI_Comm_shrink and MPI_Comm_Create are blocking
  - It is possible to perform local recovery operation until the communicator is re-set.

Problem: Performance of Resilient Collective

Sandia National Laboratories

- The majority of the application recovery cost is resilient collective at this moment
- Error Detection requires good resilient collective

- Need better resilient collectives to satisfy application needs!
  - Needs to support multiple (simultaneous) process failures
    - MPI-ULFM focus all possible cases of failures
    - This includes massive loss of processes, **which we do not CARE!**
  - Needs special version that works **under a reasonable assumption**
    - Failure happens at node, blade or nodes associated with a single NIC
  - Spare process and grouping can help the performance

Conclusions

Sandia National Laboratories

- We leverage existing technologies to enable an LFLR model for SPMD code
  - Fault Tolerant MPI (MPI-ULFM)
  - Hot spare process
  - In-memory redundant storage is scalable
  - No access to global file system
- Scalable Communicator (Spare Process) Management is essential in future resilient MPI
  - Otherwise, we have to use other parallel programming runtime….
- Future Work
  - Performance study on large (Peta) scale systems
  - Recovery for catastrophic situations (e.g. many-node crash)
  - Explore different recovery semantics other than roll-back
    - Roll-forward

Acknowledgement

- George Bosilca, University of Tennessee, Knoxville
- Robert Clay, Sandia National Laboratories, CA
- Pedro Diniz, Information Science Institute, University of Southern California
- Mark Hoemmen, Sandia National Laboratories, NM

Resilient Communicator: Partial List of APIs

Sandia National Laboratories

- Initalization and Recovery

  int RC.init();

  int RC.recover();

- Access to MPI Communicator

  MPI_Comm * RC.getGlobalComm();

  MPI_Comm * RC.getSpareComm();

  MPI_Comm * RC.getComputeComm();

  MPI_Comm * RC.getGroupComm();

  MPI_Comm * RC.getComputeGroupComm();

- Wrapper Routines:

  int RC.GlobalSend(void *mem, size_t data_size, int rank);

  int RC.GlobalRecv(void *mem, sizt_t data_size, int rank);