Local Failure Local Recovery: Toward Scalable Resilient Parallel Programming Model

Keita Teranishi, Sandia National Laboratories, California, USA
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%

System reliability is hard to predict

That means many failures could happen all of sudden.

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

Courtesy to Gupta et al, “Failures in large scale systems: long-term measurement, analysis, and implications,” SC17
Different components have different reliability

- Different reliability per Components
  - CPU, GPU, HBM, DDR, NVRAM, Network
- Even for the same component type, reliability differs among different manufacturers
- How to manage complex interaction between components?
  - Are errors and failures contained within a component?
  - Which software component manage these? Runtime, OS or Middleware?
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 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)
  - 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**
    - Shrunken 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**
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)
S3D Modifications

- Only 35 new, changed, or rearranged lines in S3D code
Global Online Recovery – Results

### MTBF

- **Production**: 2.6 h
- **Global recovery**: 189 s
- **Global recovery**: 94 s
- **Global recovery**: 47 s

### Total overhead

- **Production**: 31 %
- **Global recovery**: 10 %
- **Global recovery**: 15 %
- **Global recovery**: 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)
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:
  - Checkpointing
  - Process/data recovery
  - Rollback
- 4096 cores + 640 spare cores
- Right-most bar, global recovery with MTBF of 47s
- Local recovery has scalability advantages over global recovery

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

<table>
<thead>
<tr>
<th>Recovery Type</th>
<th>MTBF</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Production (2.6h)</td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>Global recovery (189s)</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>Global recovery (47s)</td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>Local recovery (45s)</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>Local recovery (20s)</td>
<td></td>
<td>25</td>
</tr>
</tbody>
</table>
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

```cpp
hlib::asyncAwait ( lambda,
    hclib_future_t *f1, ..,
    hclib_future_t *f4);
```

Task Launch with Replication

```cpp
diamond::asyncAwait_check<N> ( lambda,
    hclib::promise<int> out,
    hclib_future_t *f1, ..,
    hclib_future_t *f4);
```

Task Launch with Replay

```cpp
replay::asyncAwait_check<N>(
    lambda, hclib::promise<int> out,
    std::function<int(void*)> error_check_fn, void * params,
    hclib_future_t *f1, .. ,
    hclib_future_t *f4);
```
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)

- 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 or more) – Create three tasks and check for error in puts
  - Two out of three outputs should match for success
Task Replay

Detect

Replay

Detected

Up to N times

replay::async_assert_check<\texttt{N}> ( lambda,
hclib::promise<int> out, \texttt{std::function<int(void*)>} error_check_fn, void * \texttt{params}, hclib_future_t *f1,
\ldots, 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 \textbf{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

\[
abft::async_await_check ( \text{lambda}, \ hclib::promise<int> out, \ \text{std::function<int(void*)> error_check_fn, void * params, hclib_future_t *f1, .. , hclib_future_t *f4, ABFT_lambda});
\]

- 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 \text{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

![Overhead of resilience in the absence of failures](chart.png)
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

crankseg_1, 32 tiles

crankseg_1, 128 tiles
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

Parallel Execution Runtime (Pthread, OpenMP, CUDA etc.)

Intel Multicore
Intel Accelerator
NVIDIA GPU
AMD Multicore/APU
IBM Power
ARM

Adding Resilience Support

Kokkos::View< Data Type. Execution Space, Memory Space, .... >

DRAM
HBM
GPU Device Memory
NVRAM
Checkpoint System
HDF5
C++ IO
Kokkos Ecosystem

Kokkos Tools
- Debugging
- Profiling
- Tuning

Kokkos Kernels
- Linear Algebra Kernels
- Graph Kernels

Kokkos Core
- Parallel Execution
- Parallel Data Structures

Kokkos Remote Spaces
- PGAS
- IO

Science and Engineering Applications
- Trilinos

Kokkos Support
- Documentation
- Tutorials
- Bootcamps
- App support

Multi-Core
- Many-Core
- APU
- CPU + GPU

Courtesy of Christian Trott
Parallel Programming using Kokkos

- Provide parallel loop operations using C++ language features
- Conceptually, the usage is no more difficult than OpenMP. The annotations just go in different places.
Kokkos Core Abstractions

Resilience/redundancy in both abstractions

- Resilient Kokkos provides “resilient” data and execution spaces to enable resilience/fault tolerance without major modification in application program source.
Productive Resilience Support using Kokkos

```cpp
Kokkos::View<double *, Kokkos::resilience> m_data(1000);
for (i = 0; i < n; i++) {
    KokkosResilience::checkpoint( *resilience_context, "final", n, [=]() mutable
    { // Automatically checkpoint all active Kokkos::Views
        Kokkos::parallel_for(rp, KOKKOS_LAMBDA(const int i)
        {
            m_data(i)=i; // It’s Kokkos::View. No need to bind to checkpoint storage
        })
    }, KokkosResilience::filter::nth_iteration_filter< 10 >{} );
}

VELOC_Mem_protect(0, &i, 1, sizeof(int)); // Bind every single memory allocation
VELOC_Mem_protect(1, h, M * nbLines, sizeof(double)
VELOC_Mem_protect(2, g, M * nbLines, sizeof(double));
int v = VELOC_Restart_test("heatdis", 0);
if (v > 0) {
    VELOC_Restart_test is returning
    assert(VELOC_Restart("heatdis", v) == VELOC_SUCCESS);
} else
    i = 0;
while (i < n) {
    // iteratively compute the heat distribution
    // (5): checkpoint every K iterations
    if (i % K == 0)
        assert(VELOC_Checkpoint("heatdis", i) ==
            VELOC_SUCCESS);
    // increment the number of iterations
    i++;
}
```
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) = ... ;
});

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) = ... ;
});

Checkpoint "loop_1,A"

Replication

Automatic Checkpointing
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 resilience 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.)