



Toward Resilient Algorithms and Applications

Michael A. Heroux, SNL

Collaborator: Mark Hoemmen

SAND Number: 2013-3039C



For this Talk: Assumptions about Future Systems

- Resilience will be an issue.
- Engineering resilience into the system will be expensive?
 - Development cost.
 - Power consumption.
 - Performance degradation.
- Possible fault mitigation strategies:
 - Retain reliability levels of today's systems, e.g. SCR.
 - Permit faults to percolate up to user level:
 - What is the default behavior?
 - What are the fault management models?

Reliability: Progress and Regress

“ . . . I remarked to Dennis [Ritchie] that easily half the code I was writing in Multics was error recovery code. He said, ‘We left all that stuff out. If there’s an error, we have this routine called panic, and when it is called, the machine crashes, and you holler down the hall, “Hey, reboot it.” ’ ”

– *Tom van Vleck, Multics developer, circa 1973*

We have the privilege of thinking of a computer as a *reliable, digital* machine.

- *Today’s typical user*



Progress: Users' View of the System Now

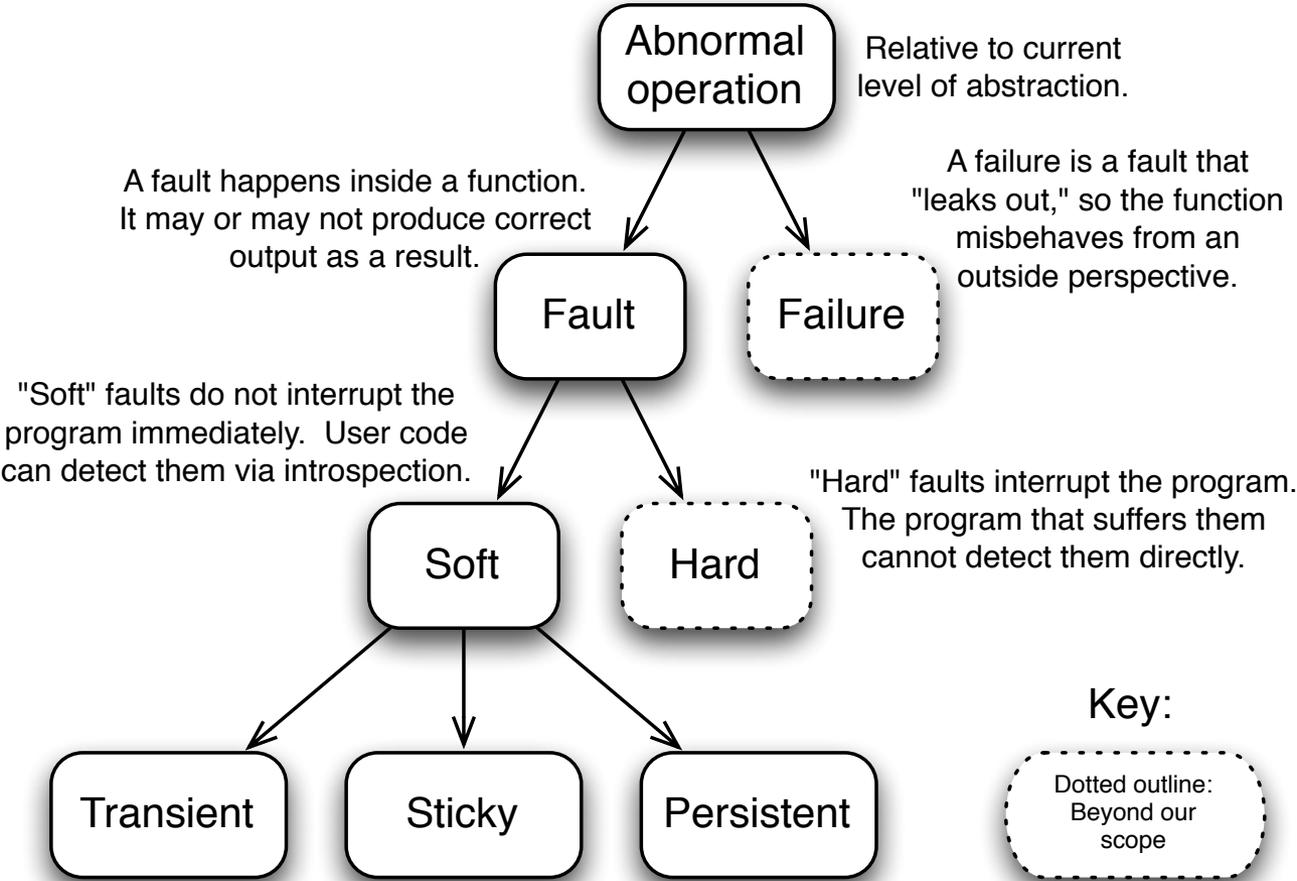
- “All nodes up and running.”
- Certainly nodes fail, but invisible to user.
- No need for me to be concerned.
- Someone else’s problem.



Regress: Users' View of the System Future

- Nodes in one of four states.
 1. Dead.
 2. Dying (perhaps producing faulty results).
 3. Reviving.
 4. Running properly:
 - a) Fully reliable or...
 - b) Maybe still producing an occasional bad result.

Fault Terminology





Reliability Model

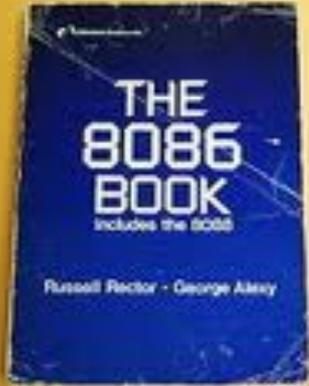
- Can't reason about code behavior without a model
- Current model: “Fail-stop”
 - System tries to detect all soft faults
 - ***Turn all detected soft faults into hard faults***
- Our basic model: “Sandbox”
 - Isolate unreliable computation in a box
 - Reliable code invokes box as a function
- Additional desired features of a model
 - Detection: report faults to application
 - Transience: refresh / recompute unreliable data periodically
 - Embed into type system: compiler can help you reason
- Our challenge goal:
 - ***Turn all detected hard faults into soft faults***



Hard Error Futures

- C/R will continue as dominant approach:
 - Global state to global file system OK for small systems.
 - Large systems: State control will be localized, use SSD.
- Checkpoint-less restart:
 - Requires full vertical HW/SW stack co-operation.
 - Very challenging.
 - Stratified research efforts not effective.

Resilience Trends Today: An X86 Analogy



- Published June 1980
- Sequential ISA.
- Preserved today.
- Illusion:
 - Out of order exec.
 - Branch prediction.
 - Shadow registers.
 - ...
- Cost: Complexity, energy.



Global checkpoint restart

- Preserve the illusion:
 - reliable digital machine.
 - CP/R model: Exploit latent properties.
- SCR: Improve performance 50-100%.
- NVRAM, etc.
- More tricks are still possible.
- End game predicted many times.

Resilient applications

- Expose the reality:
 - Fault-prone analog machine.
 - New fault-aware approaches.
- New models:
 - Programming, machine, execution.
- New algorithms:
 - Relaxed BSP.
 - LFLR.
 - Selective reliability.

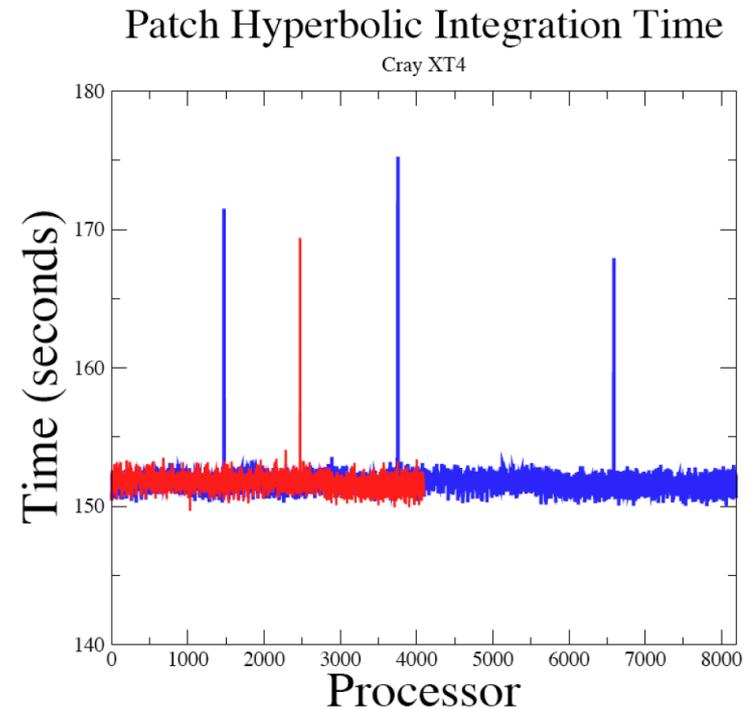


Resilience Problems: Already Here, Already Being Addressed, Algorithms & Co-design Are Key

- Already impacting performance: Performance variability.
 - HW fault prevention and recovery introduces variability.
 - Latency-sensitive collectives impacted.
 - MPI non-blocking collectives + new algorithms address this.
- Localized failure:
 - Now: local failure, global recovery.
 - Needed: local recovery (via persistent local storage).
 - MPI FT features + new algorithms: Leverage algorithm reasoning.
- Soft errors:
 - Now: Undetected, or converted to hard errors.
 - Needed: Apps handle as performance optimization.
 - MPI reliable messaging + PM enhancement + new algorithms.
- *Key to addressing resilience: algorithms & co-design.*

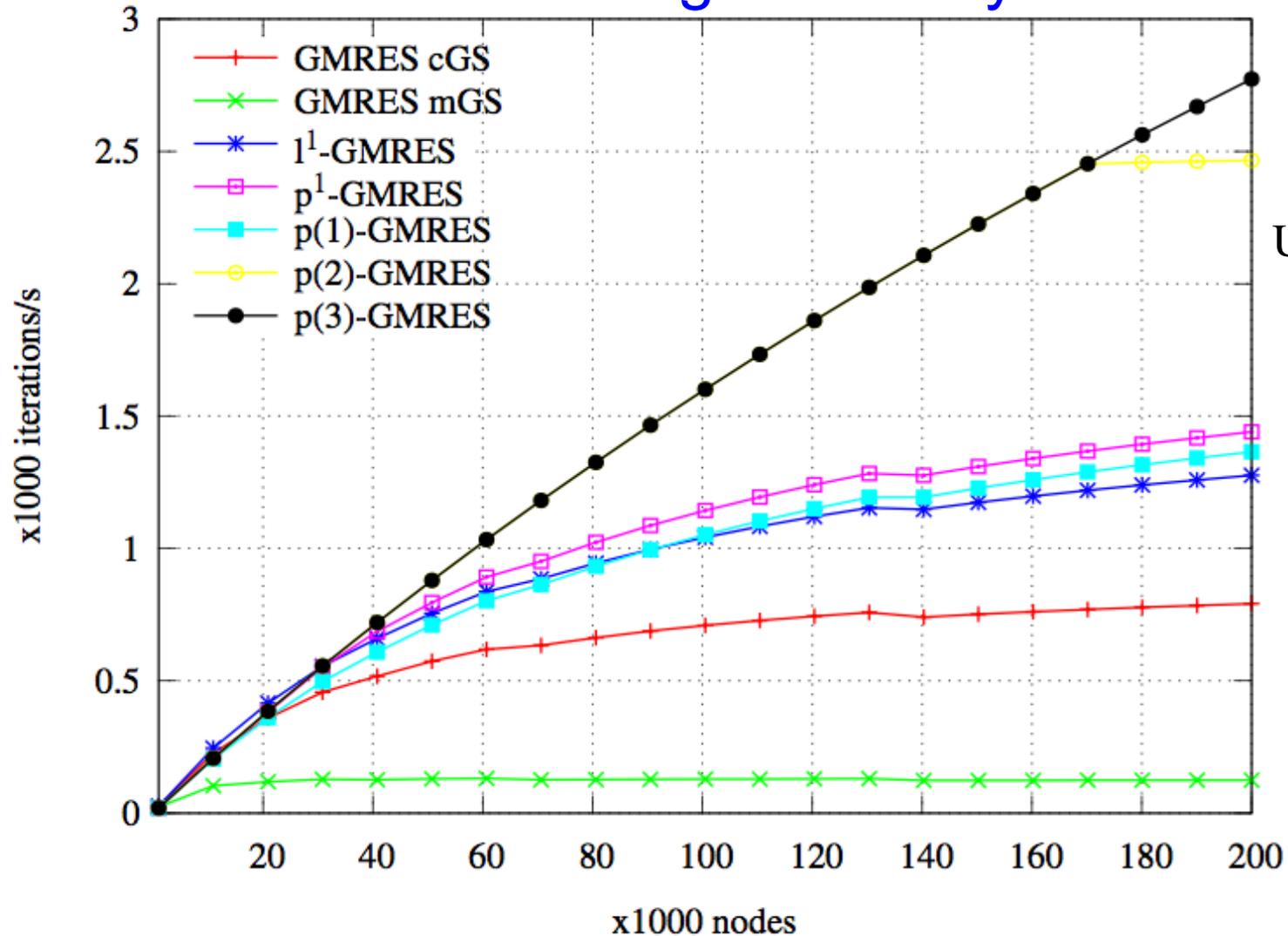
Resilience Issues Already Here

- First impact of unreliable HW?
 - Vendor efforts to hide it.
 - Slow & correct vs. fast & wrong.
- Result:
 - Unpredictable timing.
 - Non-uniform execution across cores.
- Blocking collectives:
 - $t_c = \max_i \{t_i\}$



Brian van Straalen, DOE Exascale Research
Conference, April 16-18, 2012. *Impact of persistent
ECC memory faults.*

Latency-tolerant Algorithms + MPI 3: Recovering scalability



Hiding global communication latency in the GMRES algorithm on massively parallel machines,

P. Ghysels T.J. Ashby K. Meerbergen W. Vanroose, Report 04.2012.1, April 2012,

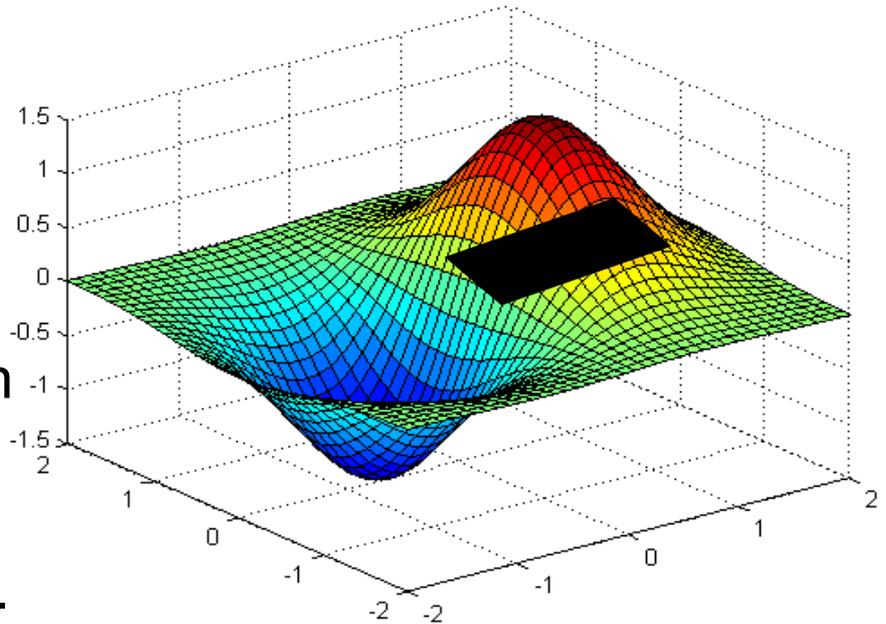


What is Needed to Support Latency Tolerance?

- MPI 3 (SPMD):
 - Asynchronous global and neighborhood collectives.
- A “relaxed” BSP programming model:
 - Start a collective operation (global or neighborhood).
 - Do “something useful”.
 - Complete the collective.
- The pieces are coming online.
- With new algorithms we can recover some scalability.

Enabling Local Recovery from Local Faults

- Current recovery model:
Local node failure,
global kill/restart.
- Different approach:
 - App stores key recovery data in persistent local (per MPI rank) storage (e.g., buddy, NVRAM), and registers recovery function.
 - Upon rank failure:
 - MPI brings in reserve HW, assigns to failed rank, calls recovery fn.
 - App restores failed process state via its persistent data (& neighbors’?).
 - All processes continue.





Local Recovery from Local Faults Advantages

- Enables fundamental algorithms work to aid fault recovery:
 - Straightforward app redesign for explicit apps.
 - Enables reasoning at approximation theory level for implicit apps:
 - What state is required?
 - What local discrete approximation is sufficiently accurate?
 - What mathematical identities can be used to restore lost state?
 - Enables practical use of many exist algorithms-based fault tolerant (ABFT) approaches in the literature.



What is Needed for Local Failure Local Recovery (LFLR)?

- LFLR realization is non-trivial.
- Programming API (but not complicated).
- Lots of runtime/OS infrastructure.
 - Persistent storage API (frequent brainstorming outcome).
- Research into messaging state and recovery.
- New algorithms, apps re-work.
- But:
 - Can leverage global CP/R logic in apps.
- This approach is often considered next step in beyond CP/R.



*Resilient Algorithms:
A little reliability*, please.*

*A system is reliable if it behaves correctly often enough that you don't have an automated, sophisticated way to handle failures.

Every calculation matters

Soft Error Resilience

Description	Iters	FLOPS	Recursive Residual Error	Solution Error
All Correct Calcs	35	343M	4.6e-15	1.0e-6
Iter=2, $y[1] += 1.0$ SpMV incorrect Ortho subspace	35	343M	6.7e-15	3.7e+3
$Q[1][1] += 1.0$ Non-ortho subspace	N/C	N/A	7.7e-02	5.9e+5

- Small PDE Problem: ILUT/GMRES
- Correct result: 35 Iters, 343M FLOPS
- 2 examples of a **single** bad op.
- Solvers:
 - 50-90% of total app operations.
 - Soft errors most likely in solver.
- Need new algorithms for soft errors:
 - Well-conditioned wrt errors.
 - Decay proportional to number of errors.
 - Minimal impact when no errors.

- New Programming Model Elements:
 - **SW-enabled, highly reliable:**
 - **Data storage, paths.**
 - **Compute regions.**
- Idea: *New algorithms with minimal usage of high reliability.*
- First new algorithm: FT-GMRES.
 - Resilient to soft errors.
 - Outer solve: Highly Reliable
 - Inner solve: “bulk” reliability.
- General approach applies to many algorithms.

FT-GMRES Algorithm

Input: Linear system $Ax = b$ and initial guess x_0

$r_0 := b - Ax_0$, $\beta := \|r_0\|_2$, $q_1 := r_0/\beta$

for $j = 1, 2, \dots$ until convergence **do**

Inner solve: Solve for z_j in $q_j = Az_j$

$v_{j+1} := Az_j$

for $i = 1, 2, \dots, k$ **do**

$H(i, j) := q_i^* v_{j+1}$, $v_{j+1} := v_{j+1} - q_i H(i, j)$

end for

$H(j+1, j) := \|v_{j+1}\|_2$

Update rank-revealing decomposition of $H(1:j, 1:j)$

if $H(j+1, j)$ is less than some tolerance **then**

if $H(1:j, 1:j)$ not full rank **then**

Try recovery strategies

else

Converged; return after end of this iteration

end if

else

$q_{j+1} := v_{j+1}/H(j+1, j)$

end if

$y_j := \operatorname{argmin}_y \|H(1:j+1, 1:j)y - \beta e_1\|_2$ \triangleright GMRES projected problem

$x_j := x_0 + [z_1, z_2, \dots, z_j]y_j$ \triangleright Solve for approximate solution

end for

“Unreliably” computed.

Standard solver library call.

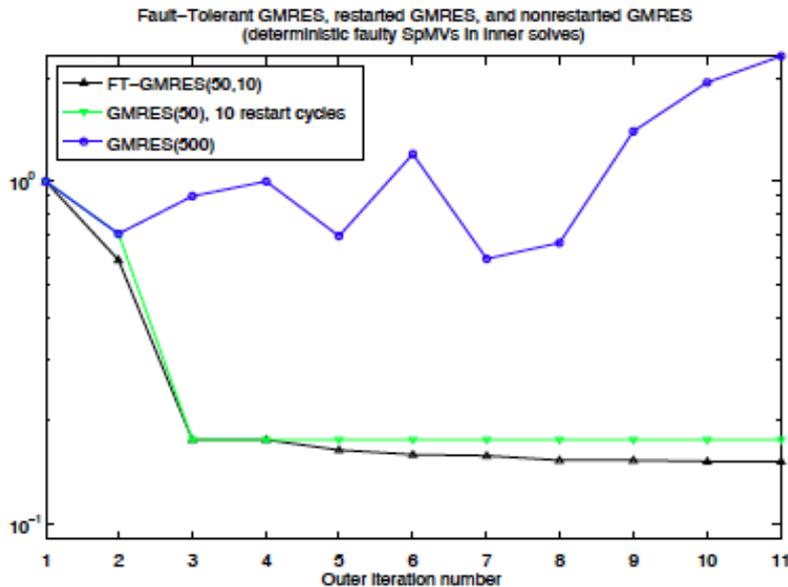
Majority of computational cost.

\triangleright Orthogonalize v_{j+1}

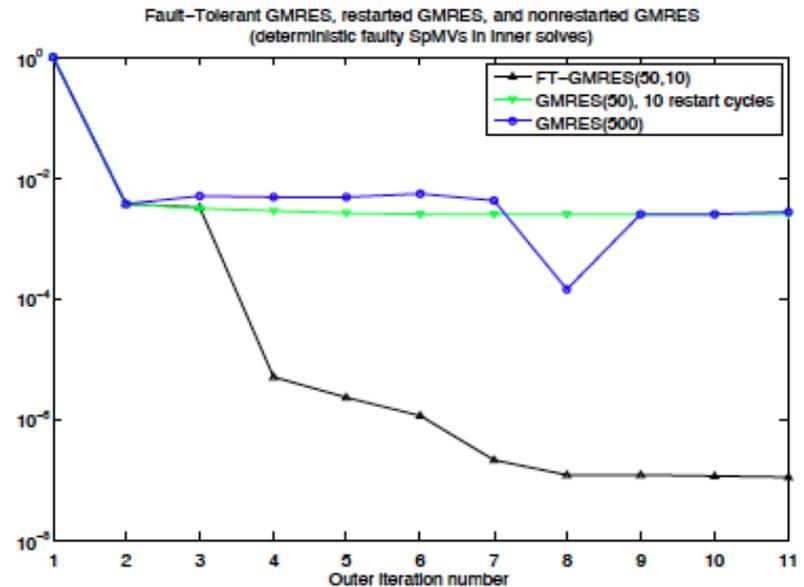
Captures true linear operator issues, AND
Can use some “garbage” soft error results.

Selective reliability enables “running through” faults

- ▶ FT-GMRES can run through faults and still converge.
- ▶ Standard GMRES, with or without restarting, cannot.



FT-GMRES vs. GMRES on Ill_Stokes (an ill-conditioned discretization of a Stokes PDE).



FT-GMRES vs. GMRES on mult_dcop_03 (a Xyce circuit simulation problem).



Desired properties of FT methods

- Converge eventually
 - No matter the fault rate
 - Or it detects and indicates failure
 - Not true of iterative refinement!
- Convergence degrades gradually as fault rate increases
 - Easy to trade between reliability and extra work
- Requires as little reliable computation as possible
- Can exploit fault detection if available
 - e.g., if no faults detected, can advance aggressively



Selective Reliability Programming

- Standard approach:

- System over-constrains reliability
- “Fail-stop” model
- Checkpoint / restart
- Application is ignorant of faults

- New approach:

- System lets app control reliability
- Tiered reliability
- “Run through” faults
- App listens and responds to faults



What is Needed for Selective Reliability?

- A lot, lot.
- A programming model.
- Algorithms.
- Lots of runtime/OS infrastructure.
- Hardware support?

- Containment domains a good start.
 - Need a “Drive fast, I feel lucky” mode for execution within a CD.



High reliability mode: Default or not?



Charon Complexity

- Charon: Device simulation code.
- SLOCCOUNT (tool from David A. Wheeler).
 - Charon physics: 191,877 SLOC.
 - Charon + nevada framework 414,885 SLOC
 - Charon_TPL 4,022,296 SLOC
- Library dependencies:
 - 25 Trilinos package.
 - 15 other TPLs.
- Expose faults to applications? **NO!**
 - Means libraries/compiler/runtime must handle them.
 - Are we preparing for this?



Strawman Resilient Exascale System

- Best possible global CP/R:
 - Maybe, maybe not.
 - Multicore permitted simpler cores.
 - Resilient apps may not need more reliable CP/R.
 - “Thanks, but we’ve outgrown you.”
- Async collectives:
 - Workable today.
 - Make robust. Educate developers.
 - Expect big improvements when apps adapt to relaxed BSP.
- Support for LFLR:
 - Next milestone.
 - FT in MPI: Didn’t make into 3.0...
- Selective reliability.
- Containment domains.
- Lots of other clever work: e.g., flux-limiter, UQ, ...



Conclusions

- Preserving the illusion of computers as reliable digital devices is expensive:
 - Engineering, TCO, ...
 - Also: Performance variability.
- Asynchronous approaches can mitigate some variability.
- Preserving global CP/R is expensive:
 - Engineering, infrastructure.
 - Analogy: Sequential x86.
- We should permit faults to occur during execution:
 - If runtime/power costs are high for hiding them *and*
 - We have a means to select reliability levels.



Conclusions

- Algorithms can handle soft errors:
 - Detection is straight-forward in many cases.
 - Majority of computation can occur in low-reliability mode.
 - We can even make use of garbage results.
- Make highly reliable data/computation the default.
 - Low reliability should be a performance optimization.
 - Inter-node activities (i.e., MPI) should be highly reliable.
- Future programming, machine, execution models:
 - Help apps reason about and express fault-resilient algorithms.
 - Give us markup for reliability attributes: Data and computation.
 - Give us tools for fine-grain state checkpoint, app-driven state recovery.
- Long-term goal: Make hard faults into soft faults.
 - Resilience tools and introspection could greatly reduce failures.