SAND2013-3039C

Toward Resilient Algorithms and Applications

Michael A. Heroux, SNL

Sandia National Laboratories is a multi-program laboratory operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin company, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Defining Resilience

Resilience is about keeping the application workload running to a correct solution in a timely and efficient manner in spite of frequent hard (i.e., unrecoverable) errors and soft (i.e., recoverable) errors

Observations

- Faults occur continuously throughout the stack from hardware, OS, runtime, to the application
- The hardware can not be expected to solve everything
- Fault detection needed at each level of the stack
- Holistic fault tolerance involves the coordinated recovery across the levels and components of the stack

Petascale faults already “continuous”

Transient error bit flips in memory are the first place we are experiencing continuous failure in petascale systems.

Av. 1 flip/min/TB (350/min on Jaguar)

Transient Memory Faults in Jaguar

- Jaguar has a lot of memory (362 TB)
- It endures over 8,000 single bit errors an hour (corrected by ECC)
- Cosmic rays (neutron flux) can account for only 1/2 of these transient faults (talk to Geist)
- It has a double bit error about every 24 hours (beats DRAM FIT rate)

Resilience isn't just about bit flips

Failures can come from any hardware or software component

And it doesn't have to have anything to do with cosmic rays.

We often consider that failures are due to growing complexity of multi-core CPUs and GPUs with millions of transistors. Here is a warning story that it can be any little thing:

Jaguar's most recent resilience challenge was a simple voltage regulator module.

- –There are over 18 thousand of these in Jaguar

Analysis – failure randomly occurs AFTER job finishes not during

- – try poll vs idle to avoid temp variation (cooling down of part)

Worked for a few weeks but failures started again

- – Tweak to lower voltage (less than a volt)
- – Jaguar stabilized

Resilience is constant struggle

What Capabilities are missing for Resilience Goals?

Primary Gap: Knowledge of Error types and rates

- Permanent, transient, gradual, silent (undetected)
- Hard to solve a problem that is not understood

Prediction (hard, holy grail)

Always needed

- Detection (everyone's responsibility HW, SW, App)
- Notification (who notifies, who is told, and when)
- Recovery (integrated effort across stack)

Standard Fault Model and API

- (not needed for chkpt/restart but needed for run through)

To deliver resilience solution also need to consider:

Usability, Efficiency, and Portability

Resilience Research Gap Analysis

Priorities

Early Start Research

Address immediate needs

1. Local checkpoint techniques for saving and restoring state need to be developed into practical solutions that are near-term, minimizing application changes

Understand the problem

2. Errors types, rates, fault root causes, and propagation need to be understood and characterized, including understanding the rate and types of silent errors on future systems
3. There is a need for an application fault model, that identifies:
   - What types of faults or degradations will be detected
   - What and how the components will be notified
   - What supported actions will be available to the components to recover or adapt to degradations of different types
4. Need standard fault test suite or metrics to stress resilience solutions and compare them fairly

Gaps preventing resilience solutions

These need to be addressed to create extreme-scale system solutions

5. No effective fault prediction due to lack of knowledge of errors, root cause, or volume of data to develop statistical models
6. System software is not designed to confine errors/faults, to avoid or limit their propagation, or to recover from them when possible
7. There is no communication or coordination across the stack in error/fault detection and management, nor coordination for preventive or corrective actions.
8. No resilience in the programming models. MPI and PGAS do not offer a paradigm for resilient programming.
9. Present Applications are not fault tolerant nor fault aware. (Next few slides describe how)

Paradigm Shift is Coming

Fault rate is growing exponentially therefore faults will eventually become continuous.

Faults will be continuous and across all levels from HW to Apps (no one level can solve the problem -- solution must be holistic)

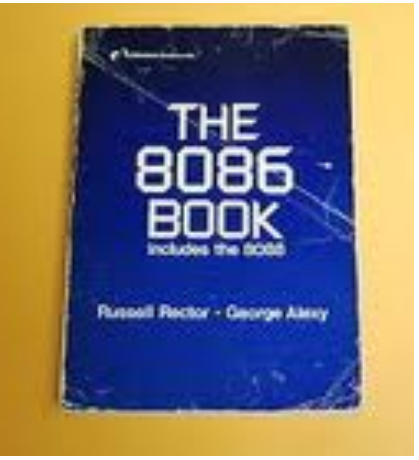
Expectations should be set accordingly with users and developers

Self-healing system software and application codes needed

Development of such codes requires a fault model and a framework to test resilience at scale

Validation in the presence of faults is critical for scientists to have faith in the results generated by exascale systems

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.

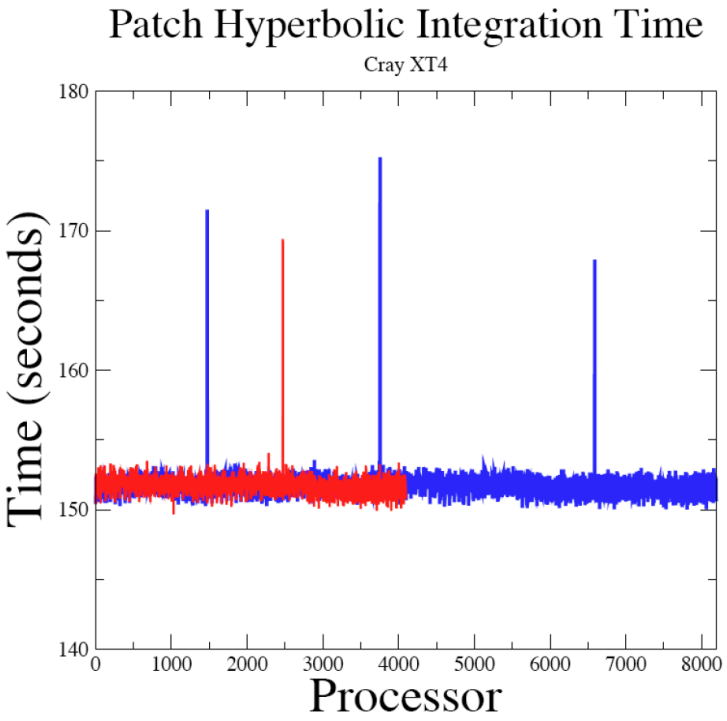
Sandia National Laboratories

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.*

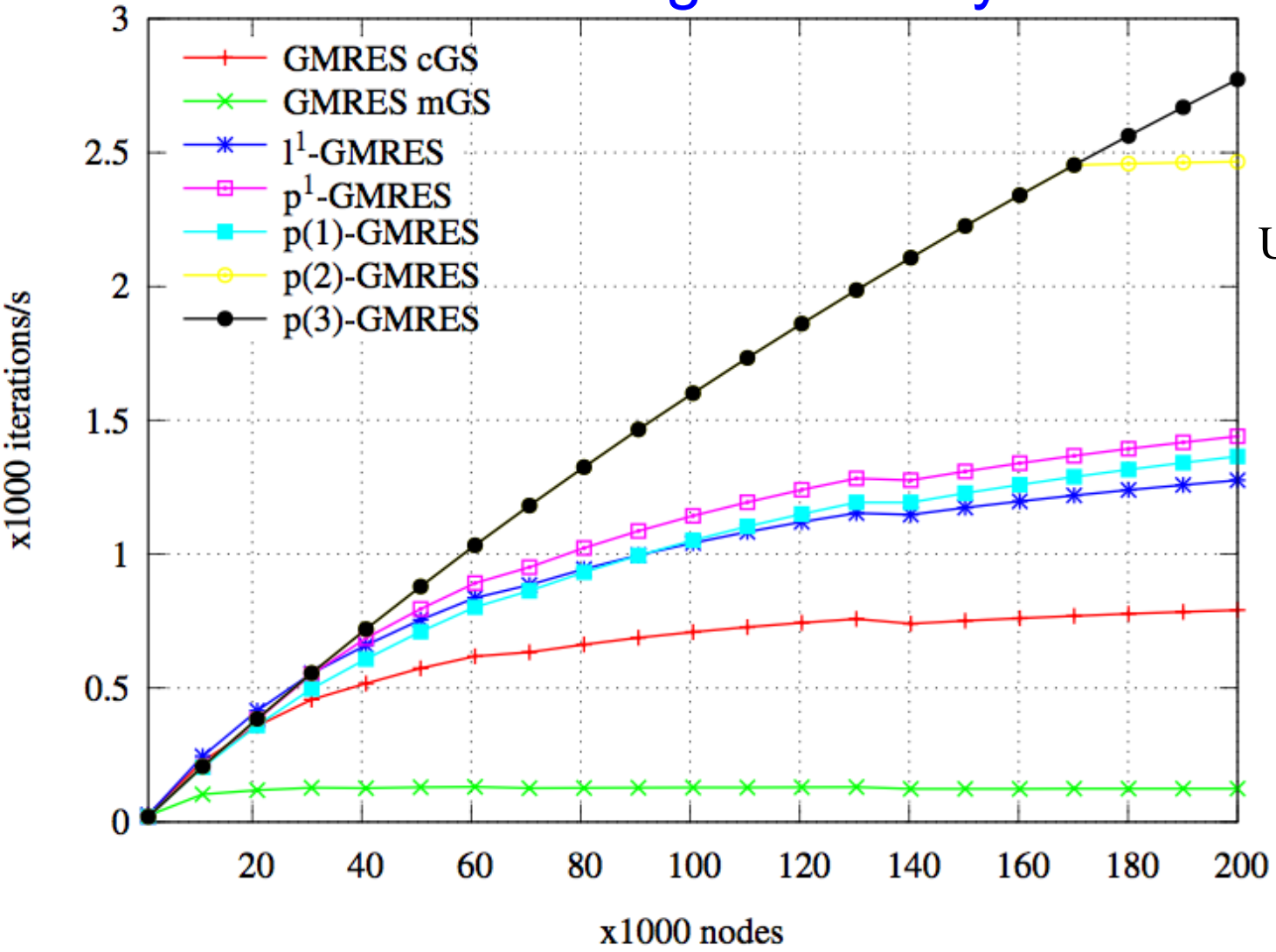
Resilience Issues Already Here

- 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



Up is good

***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, ExaScience Lab Intel Labs Europe**

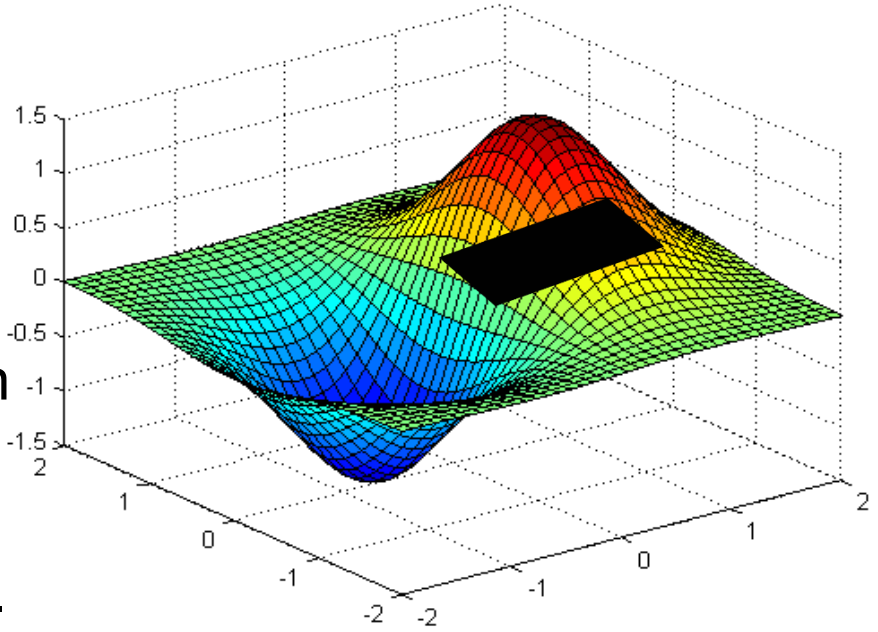
Sandia National Laboratories

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.

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 $j = 1, 2, \ldots$ until convergence **do**

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

  $v_{j+1} := Az_j$

  for $i = 1, 2, \ldots, k$ **do** ▷ Orthogonalize v_{j+1}

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

  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

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

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

end for

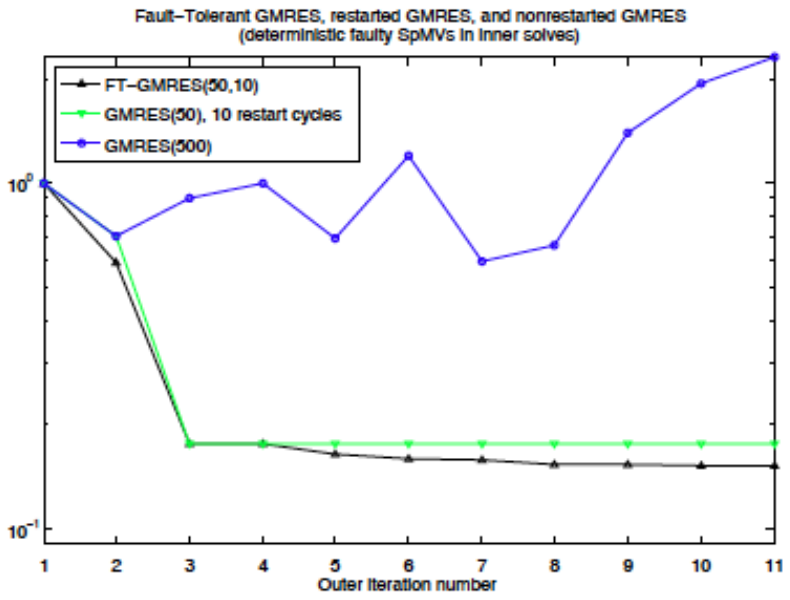
"Unreliably" computed.
Standard solver library call.
Majority of computational cost.

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

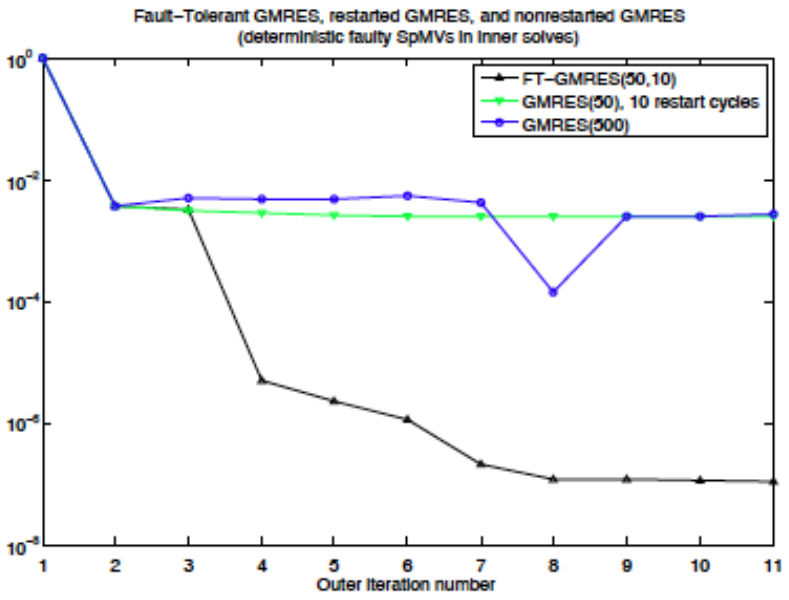
Sandia National Laboratories

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.

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.
- Asyncronous 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

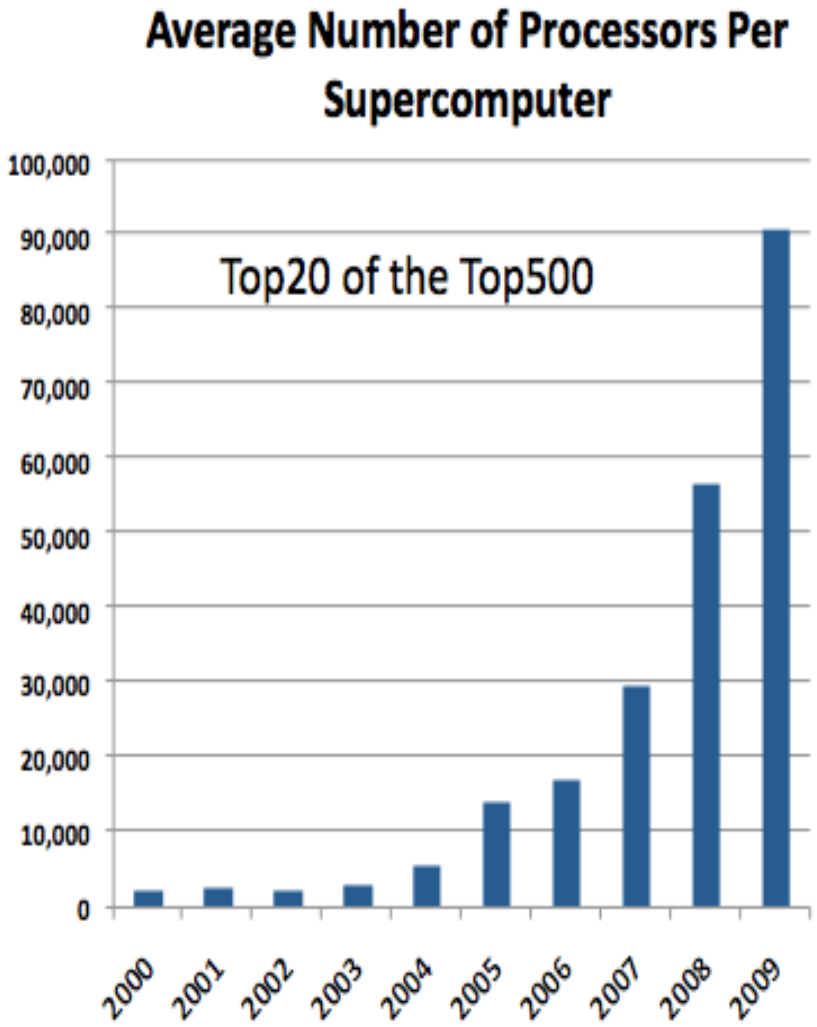
- Algorithms can handle soft errors:
  - Detection is straight-forward in many cases.
  - Majority of computation can occur in low-reliability mode.
  - We can even make use of garbage results.
- Make highly reliable data/computation the default.
  - Low reliability should be a performance optimization.
  - 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.

Backup

Exponential growth – Shock and Awe

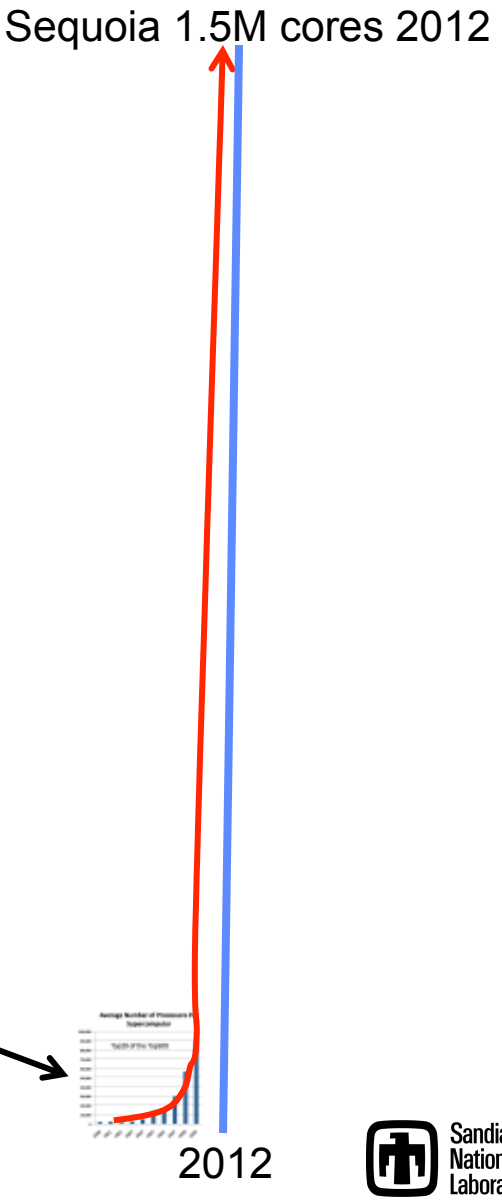
Challenges

- Fundamental assumptions of applications and system software design did not anticipate exponential growth in parallelism
- Number of system components increasing faster than component reliability (FIT has been constant)
- Assuming silent errors are constant fraction of the total errors, then Undetected error rates increasing



What is scary about exponential growth

- 20 PF Sequoia system just installed at LLNL IBM BlueGene/Q with over 1.5 million cores.
- Exponentials can quickly overwhelm us.
- One year all is OK. The next year error rate is out of control. Like hitting a wall.
- Evolutionary improvements in SW and HW don't scale "walls" very well
- Resilience solutions may require more revolutionary approaches



Creating Fault tolerant and Fault Aware predictive and non-predictive Apps

Simple set of flexible, powerful capabilities:

Capabilities: [task, workID, work phase] can specify

- persistent state storage
- ability to recover this state (even neighbors)
- notification (what happens)
- log_messages(start, stop, delete)

“What happened” examples:

- Lost data
- Lost resources
- Lost capability eg. degradation [BW, proc speed, memory]

Persistent state storage for predictive apps is assumed “local”. For non-predictive apps the storage can be “anywhere”

Transactional Model for Faults
Sometimes called “Sandbox”

Provide additional capabilities:

- Reliable storage (array) doesn’t need to be persistent
- Reliable execution (start, stop)

Many programming styles possible