SAND2013-4916C

Recent Trends in Numerical Linear Algebra: Avoiding Communication and Tolerating Computer Hardware Faults

Mark Hoemmen
mhoemme@sandia.gov

Sandia National Laboratories[1]

20 June 2013

[1] Sandia National Laboratories is a multiprogram 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.

Credits

- Haim Avron, for inviting me to speak
- This talk includes contributions by colleagues at
  - University of California Berkeley
  - Sandia National Laboratories
  - University of New Mexico
  - Texas Tech University
  - North Carolina State University
- Research funded by the US Department of Energy, the University of California, Microsoft, and Intel

Who am I and what do I do?

- R&D staff at Sandia National Laboratories

Research:

- Fault-tolerant numerical algorithms
- Communication-avoiding iterative solvers
- Shared-memory parallel programming models

Development: 

- `trilinos.sandia.gov`
- Distributed parallel sparse linear algebra
- Iterative linear solvers & eigensolvers
- Tutorials and helping users

Purpose of this tutorial

- Show communication-avoiding (CA) and fault-tolerant (FT) as special cases of *architecture-aware* algorithms
- Justify CA & FT via computer hardware trends
- Compare risk of CA vs. FT algorithms development
  - "Trends" $\Rightarrow$ predicting the future $\Rightarrow$ risk
- Introduce small part of CA & FT research
- Point you to software & research resources

Architecture-aware algorithms

- *Aware*: Designed with computer architecture in mind
- *Architecture*: Hardware or system aspects affecting
  - Time to solution ("performance")
  - Correctness (getting the right answer)
  - Peak power or total energy
- *In mind*:
  - Use hardware knowledge to improve these metrics
  - Not necessarily specific to particular hardware
  - Designed for hardware trends and abstractions
- "Communication-avoiding" and "fault tolerant" are just examples of architecture-aware algorithms

Architecture-aware numerical linear algebra

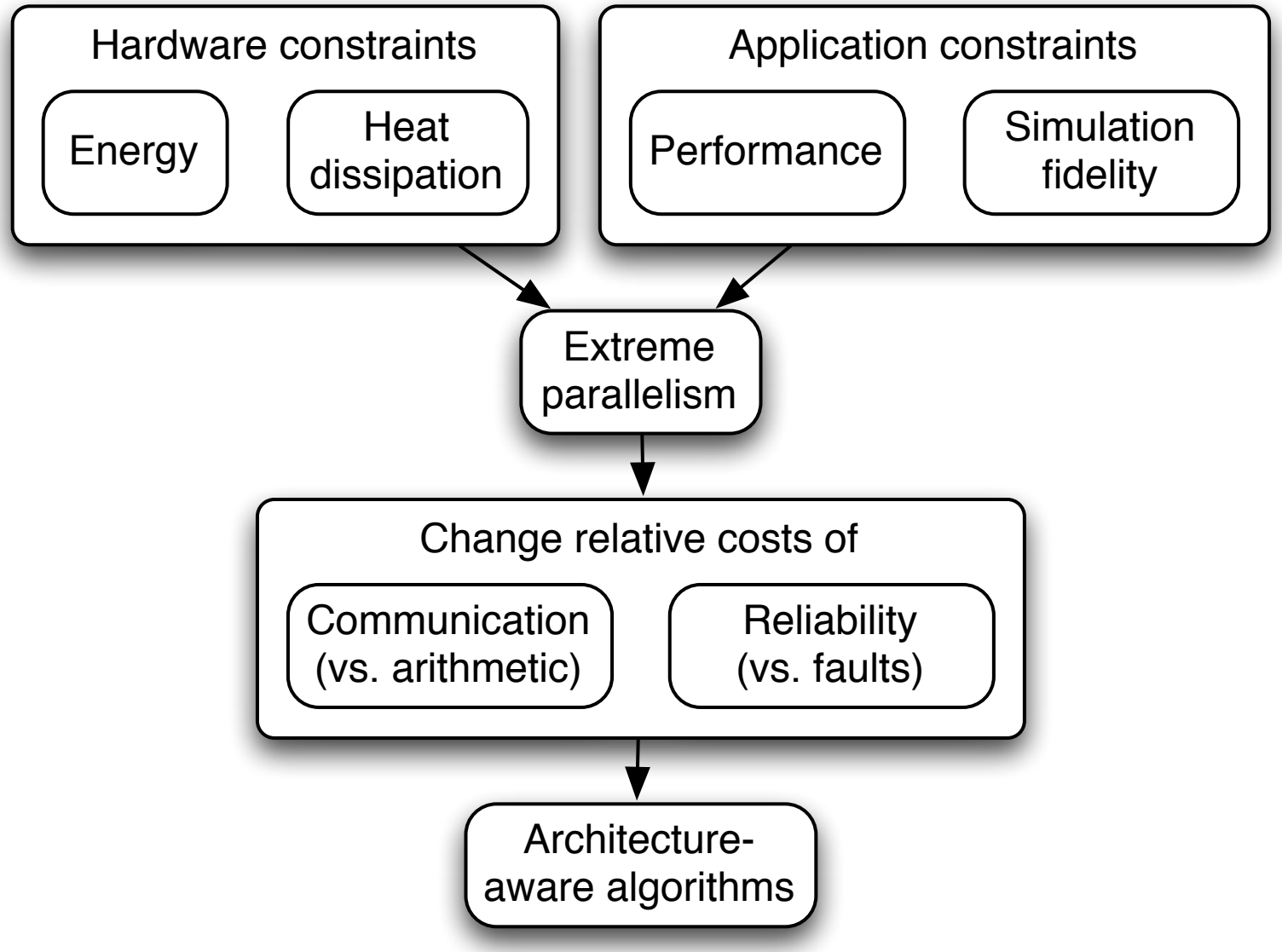
- Linear algebra historically tied to computation
  - Gauss: Astronomy & geodesy
  - Early computer scientists (Turing & von Neumann) contributed to rounding error analysis of linear systems
- Human "computers" also have "architecture"
  - Small short-term memory (like electronic computers!)
  - Memory hierarchy: head, paper, file cabinet
  - Slow arithmetic, fast memory load, slow store
  - Make mistakes in both arithmetic and storage

Carl Friedrich Gauss

Alan Turing

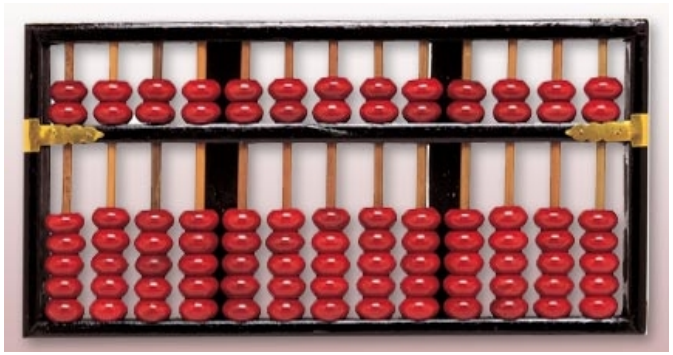
John von Neumann

Modern computers require architecture awareness



Avoiding communication

Computers do two things

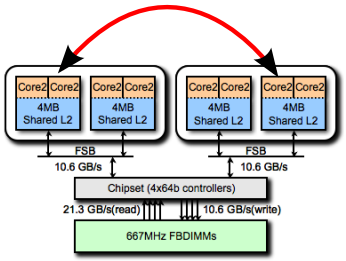


Arithmetic

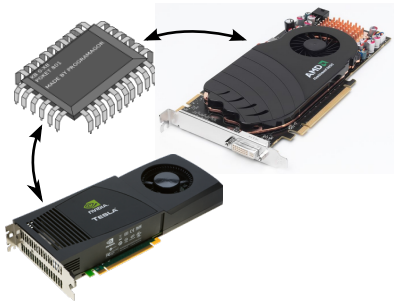
Communication

Communication is data movement

- Parallel: between processors

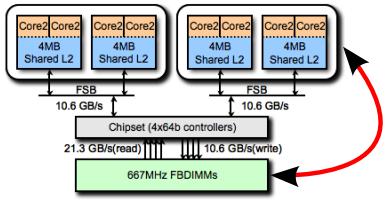


Between cores on a node, or nodes of a cluster

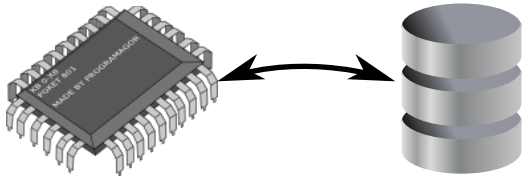


Between core(s) and coprocessor(s)

- Sequential: between levels of memory hierarchy



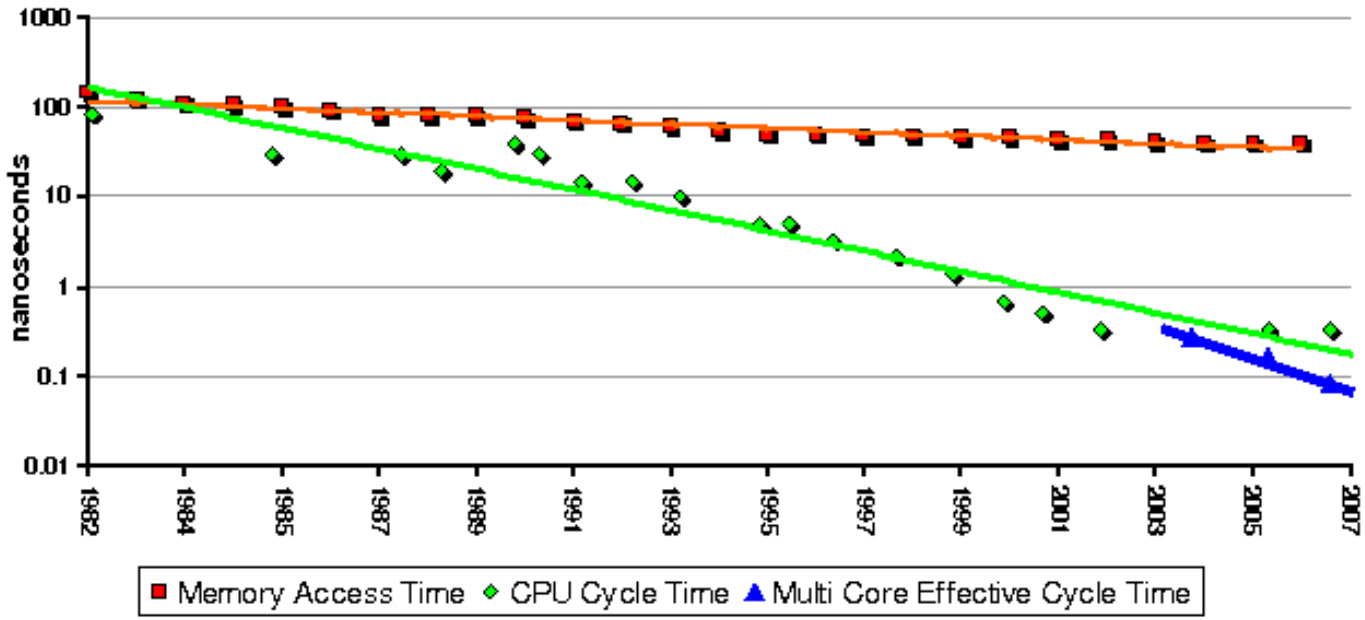
Cache to DRAM



DRAM to disk

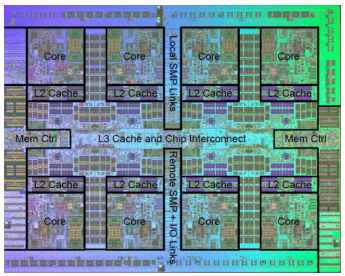
Communication slow relative to arithmetic

- Floating-point rate $\gg$ 1/bandwidth $\gg$ latency
- Relative to arithmetic, communication gets *exponentially slower* over time
- Example: memory latency & floating-point throughput
  - Comparable in 1982
  - Memory latency $\approx 1000\times$ slower in 2007
- Symptoms: *memory wall* and *power wall*

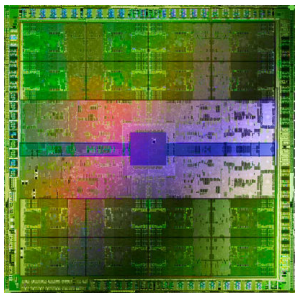


Symptom 1: Memory wall

- Memory latency & bandwidth can't keep up with flop rate
- Ratio flop/s vs. how fast memory or bus can feed it:



IBM POWER7: ≈ 20

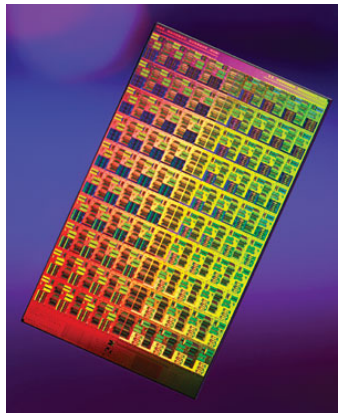


NVIDIA Fermi: ≈ 200

- True throughout memory hierarchy and between processors
- 1 MPI msg. $\approx 10^4$ flops
- 1 disk read $\approx 10^7$ flops

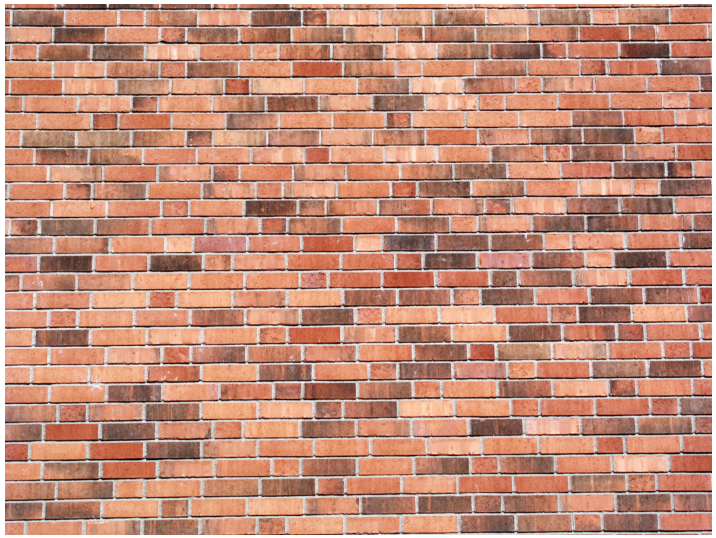
Symptom 2: Power wall

- Your cell phone will be 1000-way parallel
- Your cluster will be 10^9 - way parallel



- Can't make sequential processors faster (or they'll melt)
- Thus everything is parallel now
- Memory capacity not scaling with # cores
  - No on-node weak scaling
- More parallelism $\Longrightarrow$ more communication

Memory wall + power wall = brick wall



- Old algorithms will be *slower* on new hardware
- Solution: new algorithms – communicate less
- *Communication-avoiding* algorithms

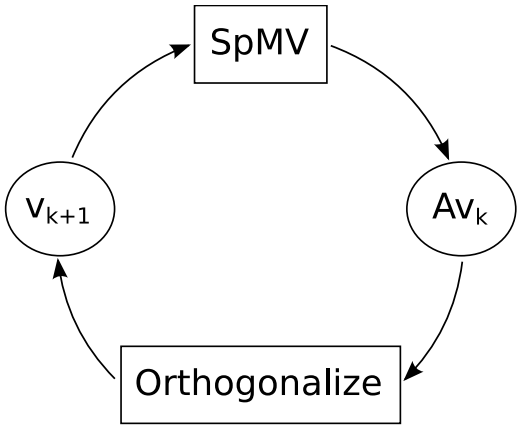
Algorithm example: CA-GMRES

Standard Krylov methods are communication-bound

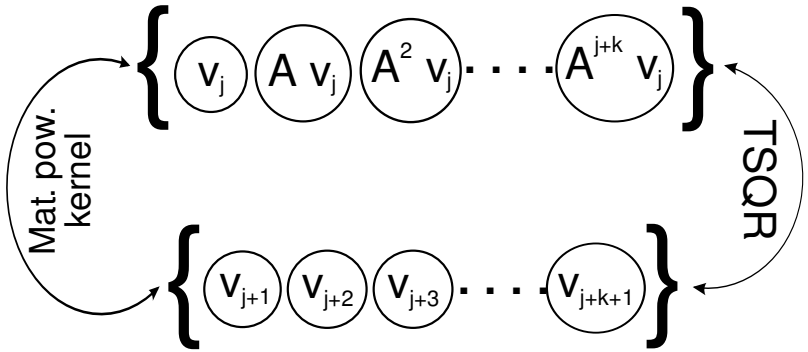
- Alternate between 2 or 3 kernels:
  - Sparse matrix-vector multiplication (SpMV)
  - (Perhaps) preconditioning (looks like SpMV)
  - Vector-vector operations (AXPY, DOT)
- All these kernels are *communication-bound*
  - SpMV dominated by
    - Reading the sparse matrix (bandwidth)
    - Passing messages and synchronizing (latency)
  - Vector-vector operations dominated by
    - Reading the vectors (bandwidth)
    - Parallel reductions (latency)

Communication-avoiding approach

- Data dependency between kernels in standard method
  - Can't hide communication
  - Can't use faster kernels
- Break data dependency $\Rightarrow$ can use faster kernels



Data dependency in standard Krylov methods



Data dependency in communication-avoiding methods

Benefits

Illustrate with GMRES: for s iterations:

	Standard GMRES	CA-GMRES
Messages	$O(s \log P)$	$O(\log p)$
Read A	s times	1 time
Read vectors	$O(s^2)$	$O(s)$

- Under certain conditions on the matrix A
  - If A partitions with a low surface-to-volume ratio:
  - e.g., low-dimensional PDE discretizations
  - Significant benefit even without this condition

Idea: build CA-GMRES out of new kernels

1. **Matrix powers kernel**: Given A and v, compute Av, A^2v, $\ldots$, A^sv, for about the same communication cost as $A \cdot v$
   - Same $\approx$ # messages as $A \cdot v$
   - Read the matrix $1 + o(1)$ time(s)
2. **Tall Skinny QR** (TSQR): $[v_1, \ldots, v_{s+1}] = Q \cdot R$
   - $O(1)$ parallel reductions, not s or s^2
   - Read and write vectors $O(1)$ times
   - As accurate as Householder QR
3. **Block Gram-Schmidt** (BGS): Gram-Schmidt, but on "blocks" of contiguous columns
   - Amortize reductions over whole blocks; use BLAS 3
   - Combine with TSQR for block orthogonalizations
   - Can do reorthogonalization efficiently & accurately

Build CA-GMRES and other CA iterative methods from these.

Details omitted to save time

- Developing other CA iterative methods
- Avoiding accuracy loss due to rounding error
- Implementing TSQR and matrix powers kernel efficiently
- See reports and papers at `bebop.cs.berkeley.edu`
- Please see §1.5-1.6 of my PhD dissertation for related work

Categories of CA algorithms & optimizations

- Problem categories for which CA algorithms exist
  - Dense linear algebra (factorizations) & tensors
  - Sparse linear algebra (iterative linear & eigensolvers)
  - General nested loops (not just linear algebra!)
- Categories of CA optimizations
  - "Tall skinny" QR, LU, etc.
    - Not just orthogonalizations for iterative solvers...
    - Panel factorizations for general dense factorizations
  - "Matrix powers kernel" for iterative solvers
    - Same setup code as in domain decomposition
    - Also applies to Chebyshev smoothing for multigrid
  - "3-D" & "2.5-D" matrix multiply & factorizations
    - Add redundant parallel storage
    - Reduce parallel communication

CA software and research resources

- Many algorithms already developed or in development
- Trilinos: Kernels for iterative linear solvers and eigensolvers
- PLASMA: Dense one-sided factorizations
- Research resources:
  - UC Berkeley "BeBOP" research group: `bebop.cs.berkeley.edu`
  - LAPACK Working Notes: `http://www.netlib.org/lapack/lawns/`
- See David Gleich's talk on MapReduce TSQR later today!

Risk of developing CA solutions

Little to no risk

- Memory locality is implied by physics (speed of light)
- Even low-power processors these days have caches
- LAPACK motivated by locality, started in late 1980's
- Many CA algorithms have public open-source implementations

Tolerating faults

Correct computer hardware is going away

- "The privilege to think of a computer as a *reliable*, *digital* machine"[2] with correct arithmetic and data storage
- Current computers experience
  - Process failures during parallel run
  - Performance variation due to hardware fault correction
- Future hardware may be silently incorrect
- Why? Power & performance
  - "At 8nm . . . it will be harder to tell a 1 from a 0"[3]
  - More parallel components (more things to fail)
  - Consumer applications drive hardware development
  - Improving hardware reliability costs performance or power

[2] M. Heroux, Sandia
[3] W. Camp, Sandia

Why we care about silently wrong results

Simulations for high consequence decisions

- Decisions like
  - Existential threats
  - Weather disasters
  - Civil engineering
- High fidelity & tight time constraints
- Hard or impossible to validate experimentally

Asteroid impact that may have killed the dinosaurs (Image credit: NASA).

Correct hardware costs performance or power

Two strategies for handling faults in hardware

Redundancy to detect & correct faults

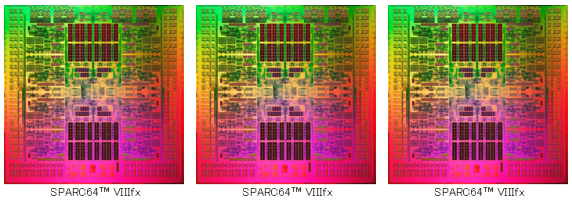
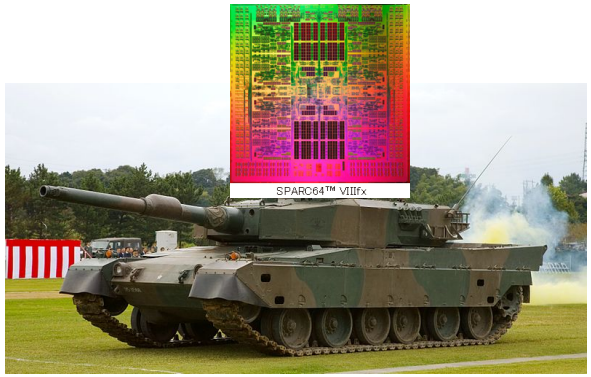


Image credits: Fujitsu, Wikipedia.

Robustness: Absorb fault-causing events



- Real hardware combines both
- Robustness costs performance *and* power
- Redundancy costs power *or* performance
  - In space (more data / computation units)
  - In time (run it $\geq$ 2x on same hardware)

Power: hard constraint, decided initially

- Can I power & cool it?
- Often decided first
- For big computers: geography, facilities
- For mobile devices: shape, weight, anatomy
- 2x time / space replication means 2x energy / power

"[S]caling today's systems to an exaflop level would consume more than a gigawatt of power, roughly the output of the Hoover Dam" [1,2]

1. "The opportunities and challenges of exascale computing," Fall 2010, US DOE Office of Science.
2. Image credit: US Bureau of Reclamation.

Alternative: Selective reliability

- Power-constrained only if system must hide all faults
  - ECC memory & "run it twice" for correctness
  - Global checkpoint-restart for process failure
- What if algorithms could take some of that burden?
- Selective reliability
  - Hardware + system let algorithm adjust reliability
  - Protect data & computation only where & when needed
  - Less *peak power* if you protect less in *space*
  - Less *total energy* if you protect less in *time*
- Requires codesign: cross-discipline communication
  - You influence hardware, system, & algorithm resilience
  - Could I make correctness cheaper if I move it up a level?
  - Algorithm-based approaches only worthwhile if they save power or energy over lower-level approaches

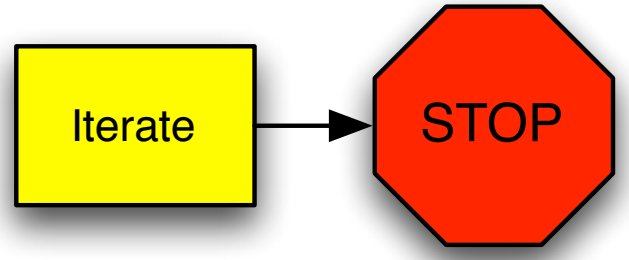
Reliability model lets us prove algorithms correct

- Reliability model: when & where faults may occur
- If faults can happen anywhere or anytime:
  - Can't prove algorithms correct
  - Can't even do time replication (esp. if faults "sticky")
- Analogy: rounding error in floating-point arithmetic
  - IEEE 754 & 854 provide invariants (e.g., exact rounding)
  - Invariants let me bound my algorithm's error
  - Language lets me control error (by declaring precision)
- Programmers cannot provide invariants themselves!
  - Just like IEEE 754, the hardware & system must help

"Floating-point code is just like any other code: it helps to have provable facts on which to depend" (David Goldberg).

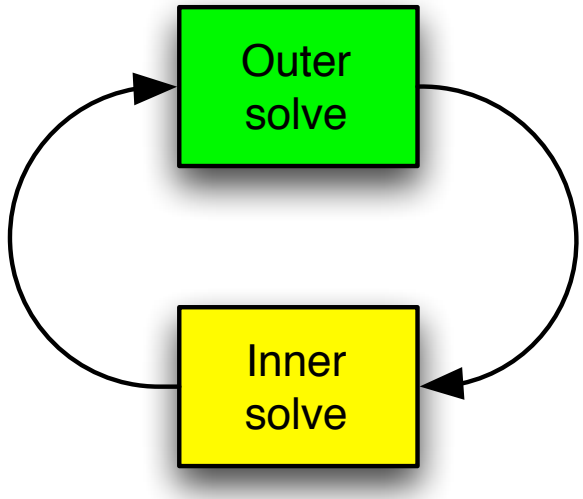
Reliability models

Current model: Fail-stop



- System tries to detect all soft faults
- Turn all detected soft faults into hard faults
- Detected local faults become global
- Checkpoint / restart is the only recovery model

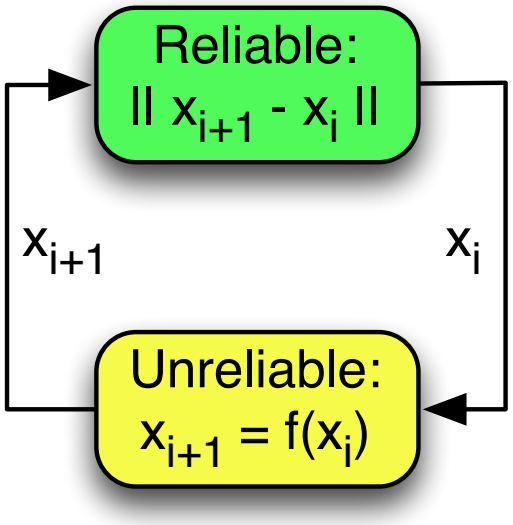
Better model: Sandbox



- Isolate unreliable data & computation in a box
- Reliable code invokes box
- Local faults stay local
- App gets flexibility to define recovery model

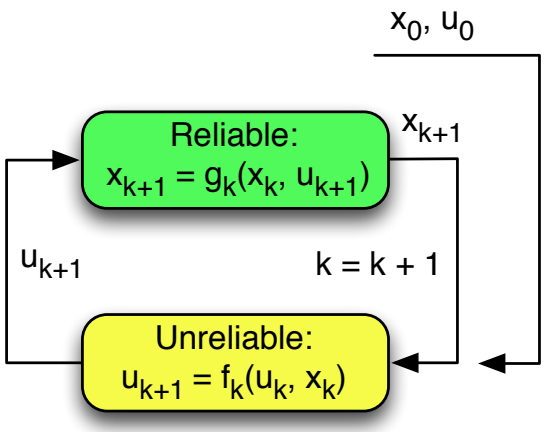
Example: Sandbox model for fixed-point iteration

- Solve $f(x) = x$: iterate $x_{i+1} = f(x_i)$ until $\|x_{i+1} - x_i\| < \epsilon$
- If faults can happen anywhere, can't trust convergence test
- Protect convergence test & put f evaluation in sandbox
  - If test passes, we know x_{i+1} is correct
  - If enough iterations pass without a fault, we will converge
  - If f time-consuming, then most time in unreliable mode
- This generalizes to nearly any iterative algorithm!



How can we guarantee convergence?

- Above fixed-point iteration might never converge
- Use sandbox model to "drive" convergence
- Generic iteration for solving $g(x) = x$:



- x_k is current approximate solution; u_k is auxiliary
- Initial guesses x_0, u_0
- for k in $0, 1, 2, \ldots$:
  - $u_{k+1} = f_k(u_k, x_k)$
  - $x_{k+1} = g_k(x_k, u_{k+1})$

- f_k may experience faults; g_k may not
- g_k must converge eventually, for any sequence of f_k, or clearly detect if it can't converge

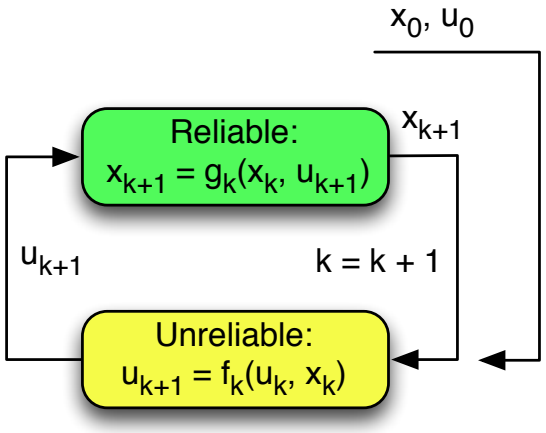
Brief review of iterative linear solvers

- Solve linear system $Ax = b$
  - Gaussian elimination not feasible / too expensive
  - Can compute matrix-vector products $v = Au$
  - Have M ("preconditioner") with $AM \approx I$

1. Pick initial guess x_0, compute $r_0 = b - Ax_0$
2. x_k is x_0 plus linear combination of $(AM)^k r_0$
3. Compute search directions via matrix-vector products
4. Repeat until norm of $r_k = b - Ax_k$ is small

How does this map to solving $Ax = b$?

Imagine above iteration as an iterative solver for $Ax = b$:



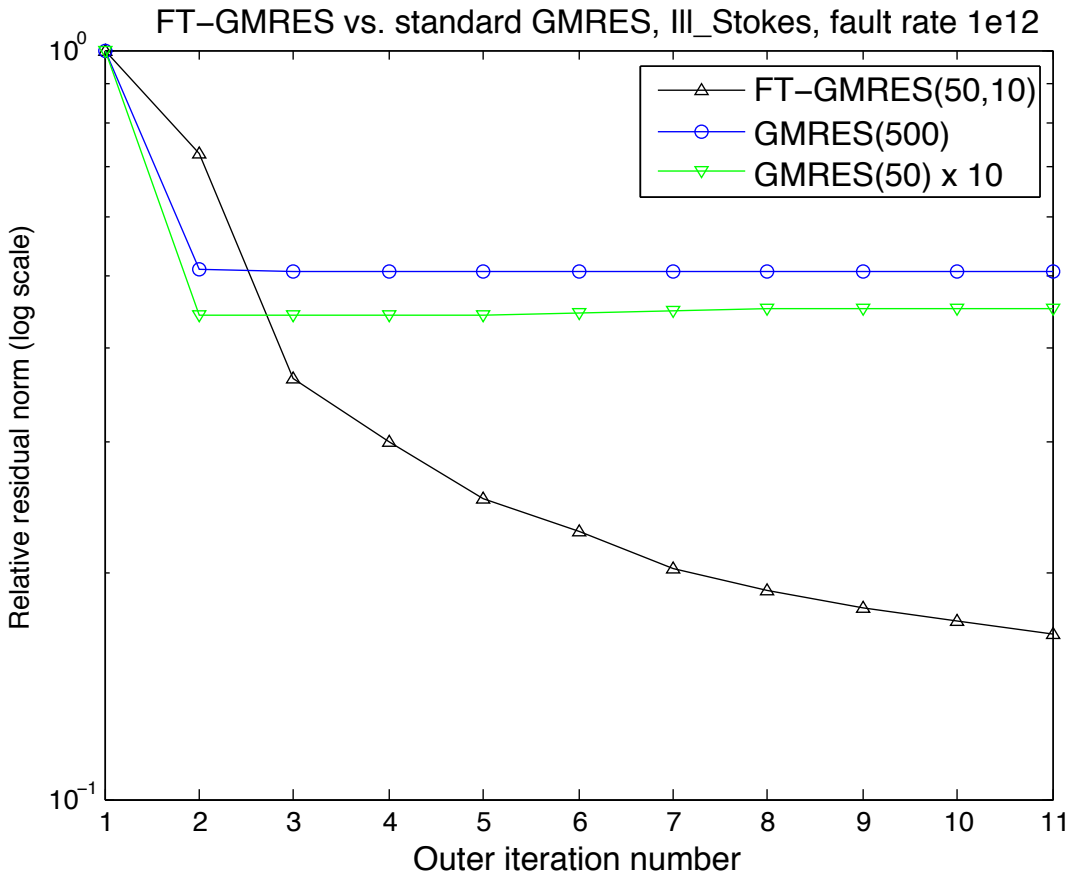
- for k in $0, 1, 2, \ldots$:
  - $u_{k+1} = f_k(u_k, x_k)$
  - $x_{k+1} = g_k(x_k, u_{k+1})$

- f_k: apply the preconditioner
  - May be different each k, perhaps because of faults
- $\Rightarrow$ Need a "flexible" method
- Monotonic convergence $\Rightarrow$ (Flexible) GMRES

Fault-Tolerant GMRES

- "Outer solver" g_k is Flexible GMRES, run reliably
  - Converges eventually (or tells you otherwise)
    - Detect failure cheaply and locally
    - ($O(k^2)$ arithmetic operations at iteration k)
    - Same failure condition as without faults
  - Flexible: Allows arbitrary preconditioner changes
    - Any bit in floating-point word may flip
    - Fault = "changing" preconditioner
- "Inner solver" f_k: your current solver & preconditioner
  - Should take most of the time
  - Preconditions the outer solver
- Can reuse existing software stack for inner solver

FT-GMRES can run through faults



Trilinos-based prototype. FT-GMRES vs. GMRES on lll_Stokes. ILUT preconditioning; 10 outer iterations, 50 inner iterations.

Other ideas

- New and growing research area
- Situations against which algorithms must prepare
  - Data / arithmetic corruption
  - Loss of a parallel process
  - Performance variation of different parallel processes

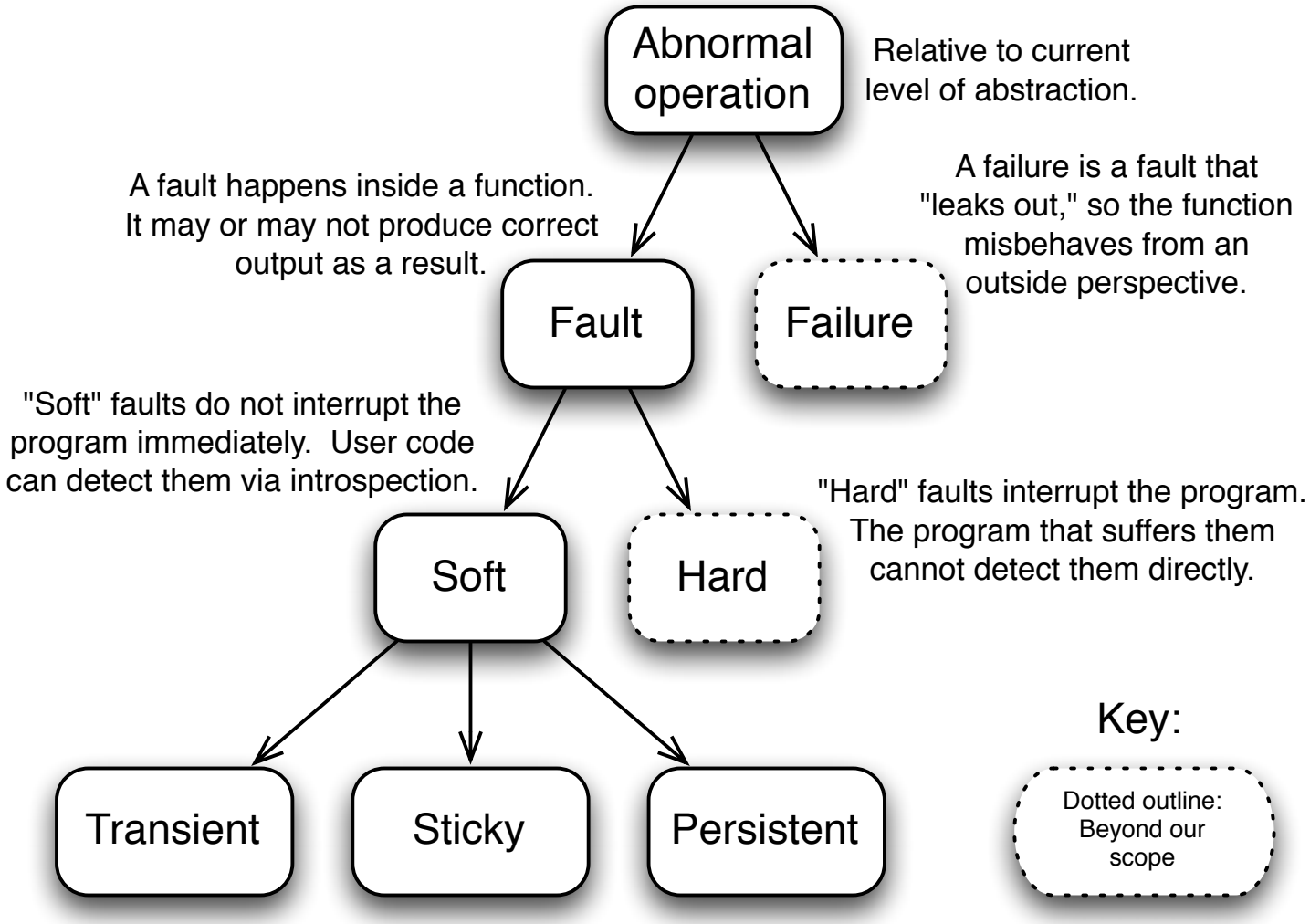
Risk of developing FT solutions: it depends on...

- Fault type
  - Likely: Per-process performance variation
  - Likely: Single process failure in a parallel job
  - Less certain: Incorrect data or arithmetic
    - Today: often either kills process or does nothing
    - Hardware designers pushing limits of correctness
    - But FT less established, so software developers less willing to accept more faulty hardware
- Algorithm class & performance requirements
  - MapReduce-like algorithms are easier to make FT
    - Jobs are independent or very loosely coupled
    - MapReduce system hides restarting failed jobs
    - Can also use redundant execution for correctness
  - HPC-like algorithms are harder to make FT
    - Tightly coupled parallel processes
    - Correctness is often a global property
    - Users unwilling to accept redundant execution

Thanks!

Extra slides

Definitions: fault, failure, soft or hard



Predecessors to CA algorithms

1. The "Matrix Perspective"
2. Blocked (a.k.a. "BLAS 3") factorizations

The "Matrix Perspective"

- Study matrices as independent abstractions
  - Not as determinants or systems of polynomials
- Innumerable contributors throughout history
- James Joseph Sylvester (1814-1897) named "matrix"
- Arthur Cayley (1821-1895)
  - Made matrices themselves an object of study
  - First to denote a matrix with a single letter
  - Defined matrix multiply and inverse
- Alston Scott Householder (1904-1993)
- "Decompositional approach to matrix computations"
  - "Toolbox" of factorizations for solving problems
  - Simplifies rounding and perturbation error analysis
  - Makes software easier to write and more general

Blocked factorizations

- Data movement speeds didn't grow w/ arithmetic speeds
  - Even on 1950s computers, especially when solving problems too big to fit in "fast" memory
  - Memory hierarchies (caches) to exploit *locality*
  - Matrix-matrix multiply has asymptotically higher *flop-to-byte ratio* than {matrix,vector}-vector operations
- Reformulate algorithms to use matrix-matrix operations
  - Optimized implementations of matrix-matrix operations
  - Factorizations that exploit them ("blocked" algorithms)
- Could call these "architecture-aware algorithms"
  - Explicitly designed with computer architecture in mind
  - Same (or greater!) arithmetic operation counts

Krylov subspace methods

- Numerical methods for solving linear algebra problems
  - Linear systems ($Ax = b$)
  - Least-squares problems ($\min_x \|Ax - b\|_2$)
  - Eigenvalue problems ($Ax = \lambda x$)
- Examples:
  - $Ax = b$: CG, MINRES, GMRES, BiCGSTAB
  - $Ax = \lambda x$: Arnoldi, $\{\text{symmetric}, \text{nonsymmetric}\}$ Lanczos
- Project A onto *Krylov subspace*, solve projected problem
- Krylov methods are *iterative*
  - Grow search space dimension with each iteration
  - Solution is approximate and (hopefully) converges

Who is Krylov?

- Alexei Nikolaevich Krylov (1863–1945)
- Russian naval engineer & applied mathematician
- Invented Krylov methods to solve symmetric eigenproblems
- Concerned with costs of computation
  - *Counted flops* (vs. symmetric Jacobi)
  - We also count, but what's expensive has changed...

A. N. Krylov in 1910s

Original GMRES

1: v_1 is scaled initial residual
2: **for** $k = 1$ to s **do**
3: $w_k = A \cdot v_k$ ▷ Sparse matrix-vector multiplication
4: **for** $j = 1$ to k **do** ▷ Modified Gram-Schmidt
5: $H_{jk} := \langle v_j, w_k \rangle$ ▷ Inner product
6: $w_k := w_k - v_j H_{jk}$
7: **end for**
8: $H_{k+1,k} := \|w_k\|_2$ ▷ Vector norm
9: $v_{k+1} := w_k / H_{k+1,k}$
10: **end for**
11: Compute solution using H

New kernels

1. **Matrix powers kernel**: Given A and v, compute Av, A^2v, $\ldots$, A^sv, for about the same communication cost as $A \cdot v$
   - Same $\approx$ # messages as $A \cdot v$
   - Read the matrix $1 + o(1)$ time(s)
2. **Tall Skinny QR** (TSQR): $[v_1, \ldots, v_{s+1}] = Q \cdot R$
   - $O(1)$ parallel reductions, not s or s^2
   - Read and write vectors $O(1)$ times
   - As accurate as Householder QR
3. **Block Gram-Schmidt** (BGS): Gram-Schmidt, but on "blocks" of contiguous columns
   - Amortize reductions over whole blocks; use BLAS 3
   - Combine with TSQR for block orthogonalizations
   - Can do reorthogonalization efficiently & accurately

Build new iterative methods from these.

New version of GMRES

- Idea: Any Krylov subspace basis will do
- Let's pick one not requiring dot products

1: $W = [v_1, Av_1, A^2 v_1, \ldots, A^s v_1]$ ▷ Matrix powers kernel
2: Compute $QR = W$ ▷ TSQR factorization
3: Compute upper Hessenberg $H(1 : s+1, 1 : s)$ using R
4: Compute solution using $H(1 : s+1, 1 : s)$

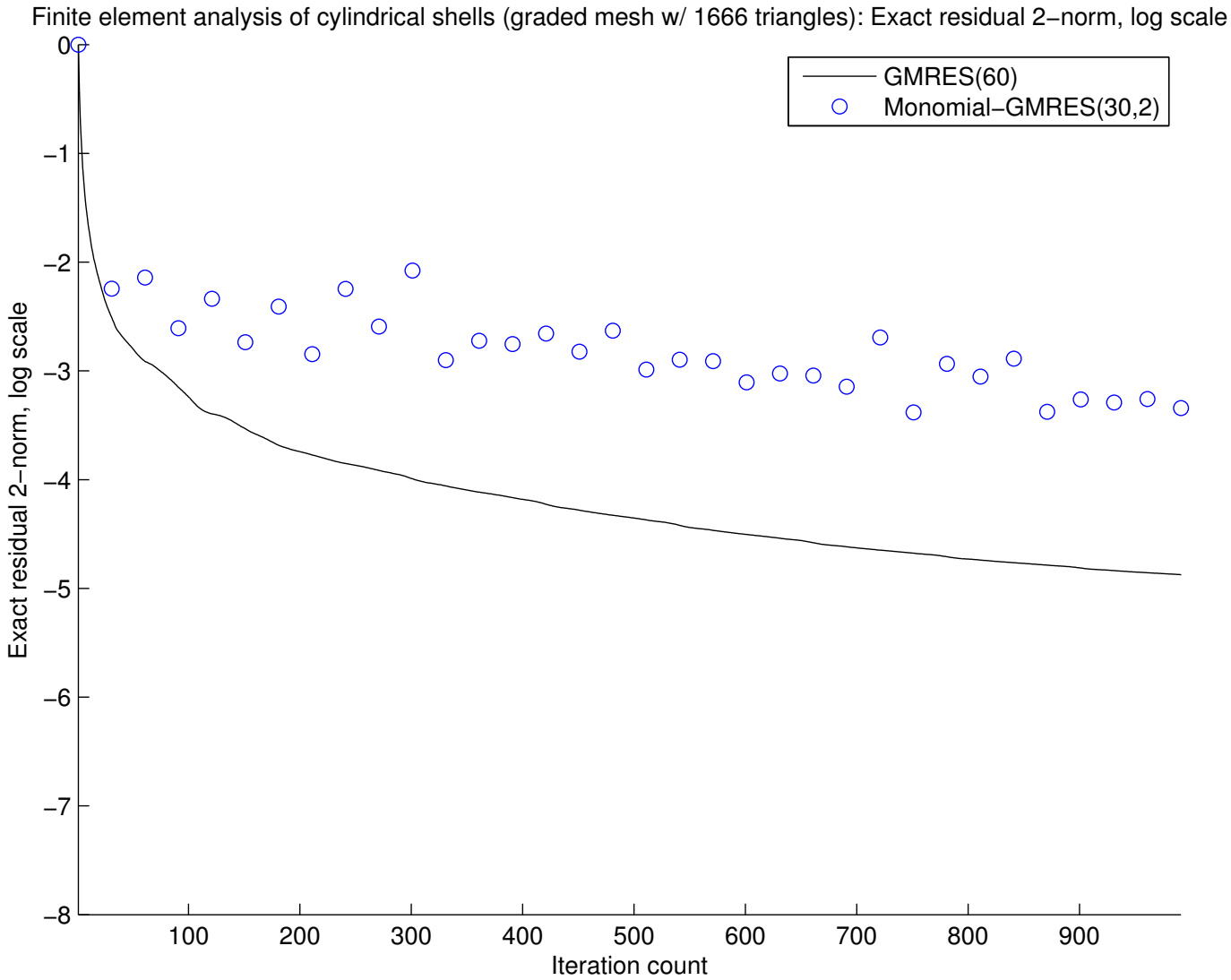
Basis v, Av, A^2v, ... unstable

- v, Av, A^2v, ... looks familiar...

Basis v, Av, A^2v, ... unstable

- v, Av, A^2v, ... looks familiar...
- It's the power method!
  - *Converges* to principal eigenvector of A
  - Bad: "converging" $\Longrightarrow$ NOT independent!
- Basis condition number *exponential* in s

Basis v, Av, A^2v, ... unstable

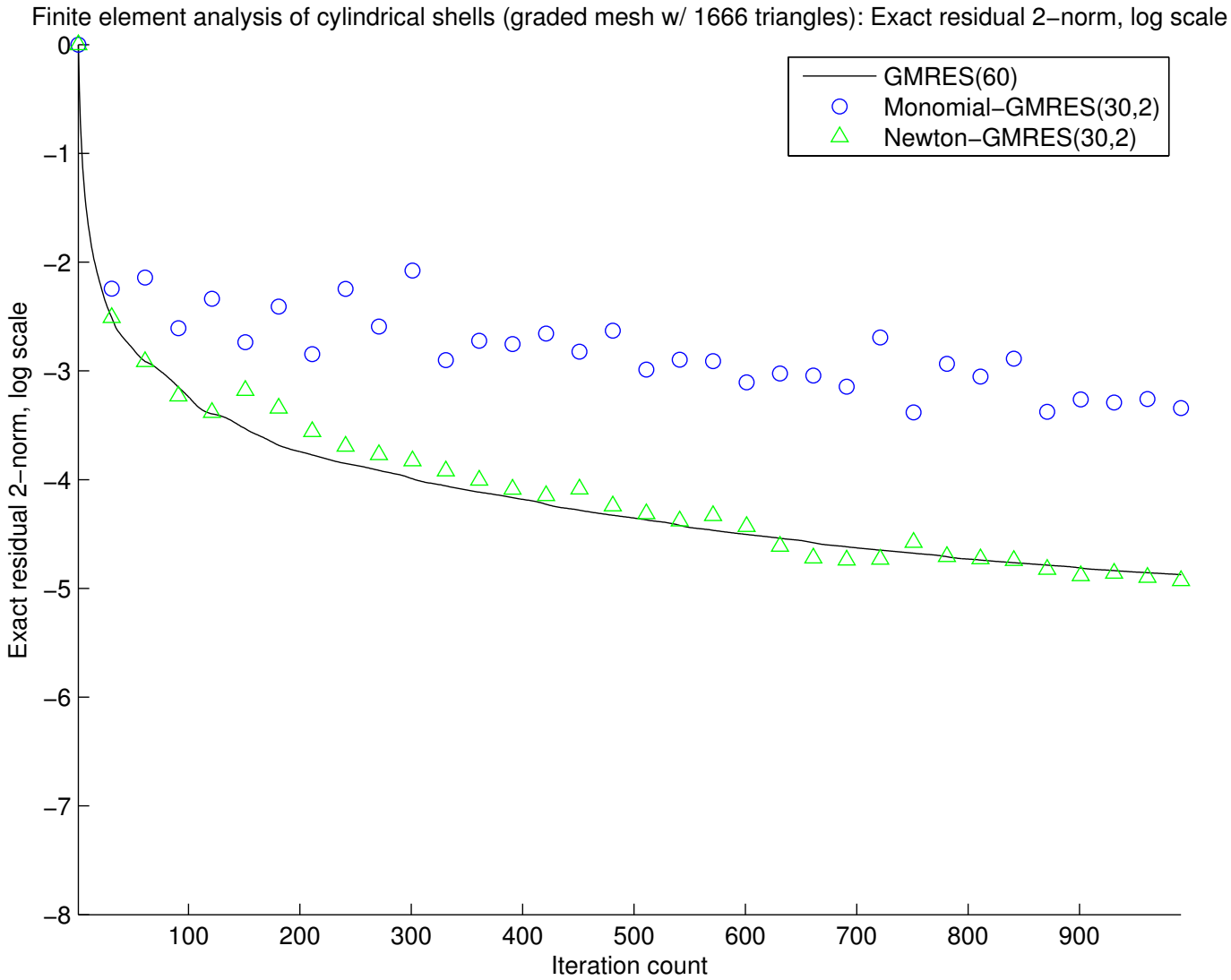


Matrix `s3rmt3m3` from Matrix Market: FEM analysis of cylindrical

Pick a different basis

- Intuition from polynomial interpolation
  - Monomial basis gets linearly dependent fast
  - a.k.a. "Vandermonde matrices are ill-conditioned"
- Use a different basis, e.g. Newton basis
  $W = [v, (A - \theta_1 I)v, (A - \theta_2 I)(A - \theta_1 I)v, \ldots]$
  - θ_k: "shifts" from Ritz values
  - "For free" from the Krylov method itself
- Same optimizations work as for v, Av, A^2v, ... basis

Newton basis improves convergence



Same matrix as before, monomial and Newton bases