Present and Future Computing Requirements Trilinos Libraries for Scalable, Resilient Manycore Computations

Michael A. Heroux
Sandia National Laboratories

NERSC ASCR Requirements for 2017
January 15, 2014
LBNL

1. Project Description

PI: Michael Heroux, Sandia

- Summarize your project(s) and its scientific objectives through 2017
  - Advanced solvers:
    - Tightly coupled multi-physics.
    - Embedded nonlinear analysis, optimization and UQ.
  - Algorithms, data classes for scalable manycore systems:
    - Extract fine-grain data parallelism.
    - Low-rank approximations for off-diagonal blocks.
    - Linear solvers in service of advanced solvers.
  - Resilient computations:
    - Progress in presence of performance variability.
    - Local failure-local recovery.
    - Detect/correct soft errors.

1. Project Description (cont.)

Present/future focus:

- Coupled multi-physics:
  - CASL: Drekar uses 32 Trilinos packages.
  - Preconditioners: Physics-based utilizing ML, Ifpack, SuperLU,..
  - Rapid app development in Albany: first concept to scalable app < 1 year.
  - 2017: 6+ apps giving optimal solutions with error bars.
- Scalable, unstructured single DOF MG solves:
  - Critical to scalability now and future.
  - 2017: Scalable manycore smoothers, continued alg progress.
- Beginning-to-end Trilinos/component-based apps:
  - Albany today.
  - 2017: App consists of definition of physics (the "business rules"). Coordinated, parametrized use of many interoperable, reusable components.

2. Computational Strategies

- We approach this problem computationally at a high level by:
  - Vertical stack of interoperable components: Geometry-to-Analysis.
  - Horizontal suites of interchangeable components: Swap-in functionality.

- The codes we use are:
  - Direct sparse: SuperLU, MUMPS, etc.
  - Partitioning: ParMetis, Skotch, etc.
  - BLAS, LAPACK, etc.
  - Wrappers to Hypre, PETSc functionality.

<u>2. Computational Strategies </u>(cont.)

- These codes are characterized by these algorithms:
  - Unstructured problems.
  - PDEs, circuits, medium range integral formulations (classical DFTs, Peridynamics)
- Our biggest computational challenges are:
  - Effective use of manycore/accelerators.
  - Continued solver scaling.
  - Resilience.
- Our parallel scaling is limited by:
  - Varies by app: Load imbalance, lack of algorithmic scalability, strong scaling limits, lack of need.
- We expect our computational approach and/or codes to change (or not) by 2017 in this way: More multi-physics, opt, UQ.

3. Current HPC Usage: N/A.

4. HPC Requirements for 2017: N/A

5. Strategies for New Architectures (1 of 2)

- Does your software have CUDA/OpenCL directives; if yes, are they used, and if not, are there plans for this?
  - CUDA: Yes; OpenCL: No (maybe never).
- Does your software run in production now on Titan using the GPUs?
  - Yes, Denovo.
- Does your software have OpenMP directives now; if yes, are they used, and if not, are there plans for this?
  - Yes, optional. Modest use, increasing dramatically with Intel MIC.
- Does your software run in production now on Mira or Sequoia using threading?
  - No.
- Is porting to, and optimizing for, the Intel MIC architecture underway or planned?
  - Yes, underway. Significant effort.

5. Strategies for New Architectures (2 of 2)

- Have there been or are there now other funded groups or researchers engaged to help with these activities?
  - Current ASC funding for data structures/software. Current ASCR/RX-Solvers for algorithms.
- If you answered "no" for the questions above, please explain your strategy for transitioning your software to energy-efficient, manycore architectures
  - N/A.
- What role should NERSC play in the transition to these architectures?
  - Occasional access to resources for scaling studies has been and would be very helpful.
- What role should DOE and ASCR play in the transition to these architectures?
  - DOE/NNSA: near-to-medium term, production oriented. DOE/ASCR: long-term, high risk/high payoff.
- Other needs or considerations or comments on transition to manycore:
  - Continued activities focused on “disruptive” approaches, e.g., Parallex//XPI/HPX.

5. Special I/O Needs

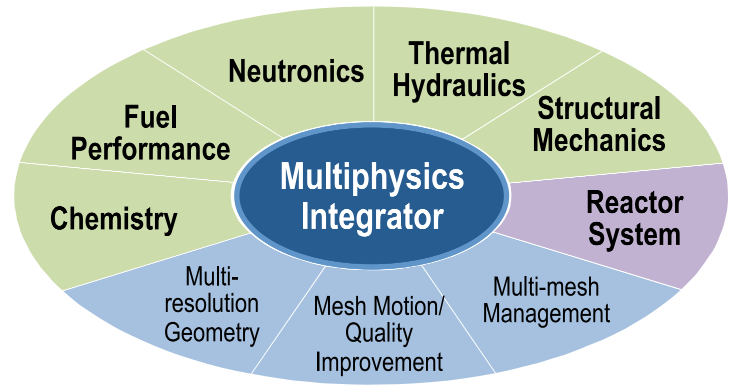
- Does your code use checkpoint/restart capability now?
  - Trilinos/Trios package provide I/O functionality.
  - 2017: Compatible data containers for compute & analytics.

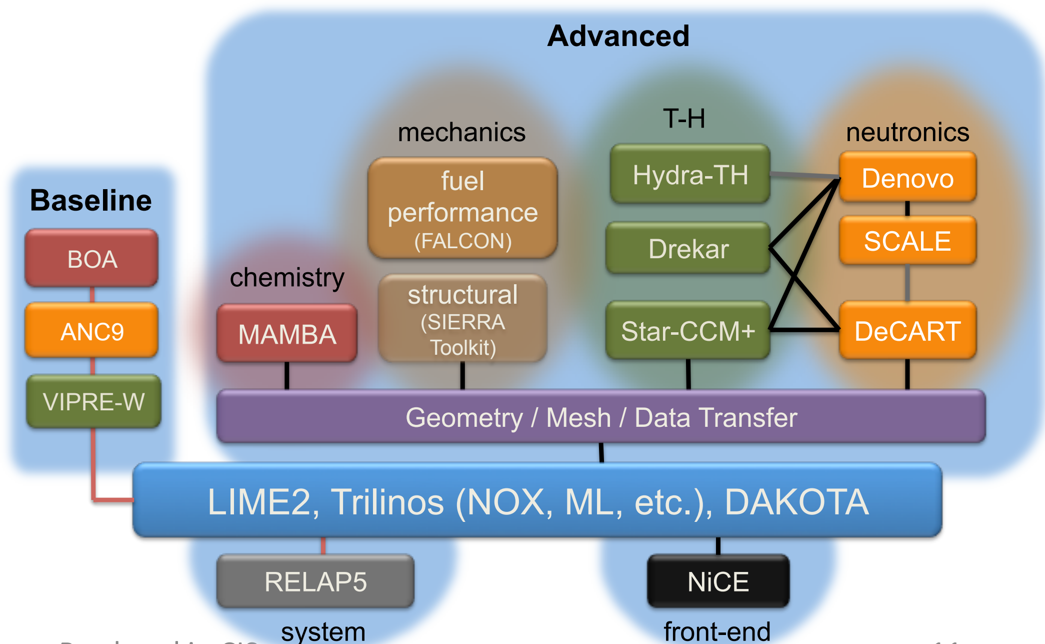
- Do you foresee that a burst buffer architecture would provide significant benefit to you or users of your code?
  - Yes, for library features.
  - Dual use as persistent store component for LFLR resilience.

Details: Multi-physics

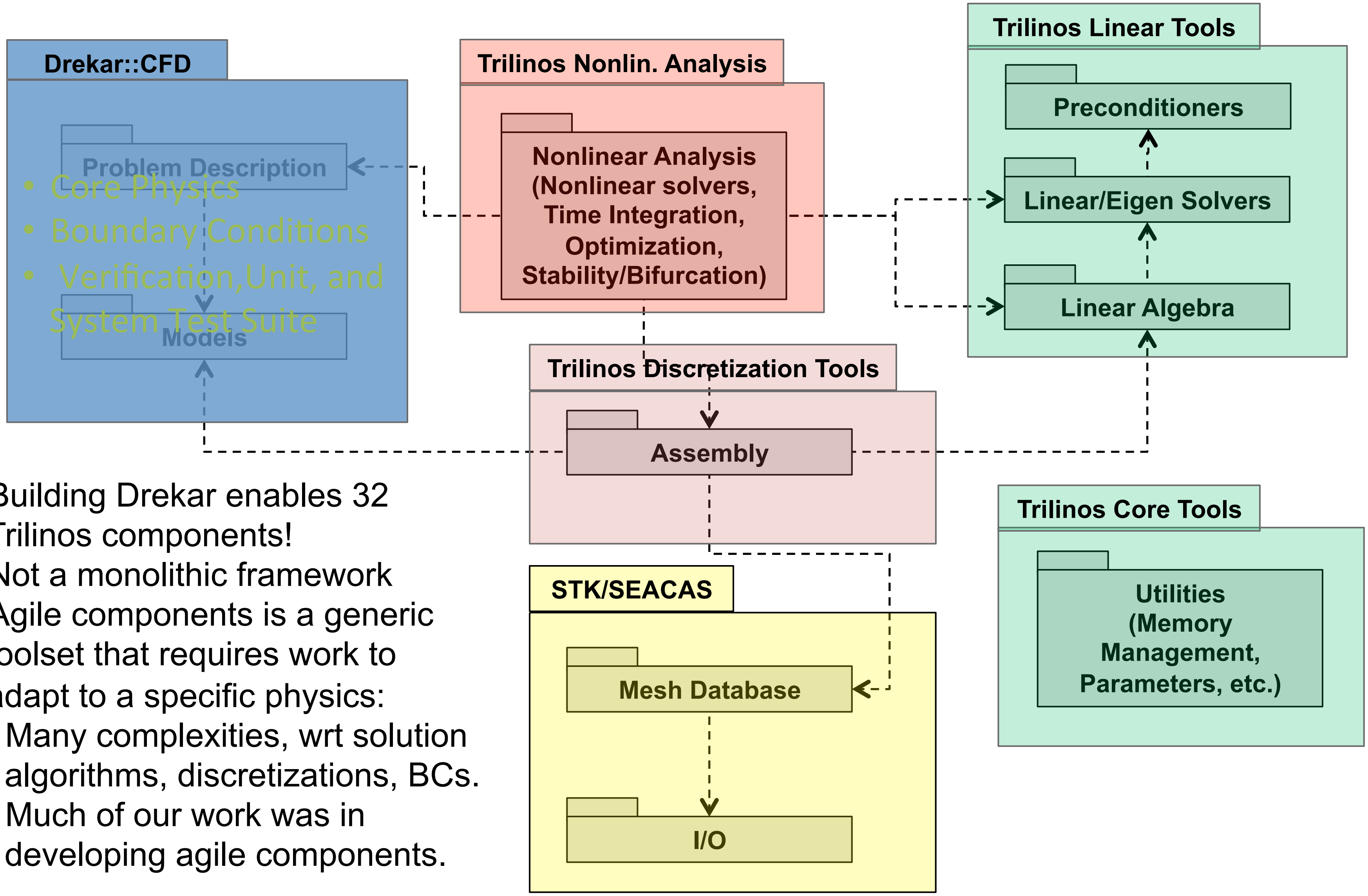
Developing a New Turbulent CFD Component (Drekar::CFD)

- Major CASL Driver is to adapt DOE high performance computing (HPC) technology for use in U.S. Nuclear industry.
- Laboratory app. codes have intellectual property/export control restrictions
- Commercial CFD is a critical part of CASL (CD-Adapco, ASCOMP GmbH)
- CASL advanced CFD addresses limits in commercial codes:
  - Scalability
  - Proprietary code base limits **efficient** multiphysics integration
  - Uncertainty quantification techniques are typically limited to "black-box" sampling
  - Publically available to all partners/NRC
  - Advanced physics models





Drekar::CFD Software Design
(UML Package Interaction Diagram)



- Building Drekar enables 32 Trilinos components!
- Not a monolithic framework
- Agile components is a generic toolset that requires work to adapt to a specific physics:
  - Many complexities, wrt solution algorithms, discretizations, BCs.
  - Much of our work was in developing agile components.

Details: Manycore algorithms/ containers

Kokkos implementation algorithm:

- 1) Replace array allocations with Kokkos::Views (in Host space)
- 2) Replace array access with Kokkos::Views
- 3) Replace functions with Functors, run in parallel on Host
- 4) Set device to ‘Cuda’, ‘OpenMP’ or ‘Threads’ and run on specified Device

FELIX_ViscosityFO_Def.hpp

```
for (std::size_t cell=0; cell < workset.numCells; ++cell) {
  for (std::size_t qp=0; qp < numQPs; ++qp) {
    //evaluate non-linear viscosity, given by Glen's law, at quadrature points
    epsilonEqpSq  = Ugrad(cell,qp,0,0)*Ugrad(cell,qp,0,0); //epsilon_xx^2
    epsilonEqpSq += Ugrad(cell,qp,1,1)*Ugrad(cell,qp,1,1); //epsilon_yy^2
    epsilonEqpSq += Ugrad(cell,qp,0,0)*Ugrad(cell,qp,1,1); //epsilon_xx*epsilon_yy
    epsilonEqpSq += 1.0/4.0*(Ugrad(cell,qp,0,1) + Ugrad(cell,qp,1,0))*(Ugrad(cell,qp,0,1) + Ugrad(cell,qp,1,0)); //epsilon_xy^2
    epsilonEqpSq += 1.0/4.0*Ugrad(cell,qp,0,2)*Ugrad(cell,qp,0,2); //epsilon_xz^2
    epsilonEqpSq += 1.0/4.0*Ugrad(cell,qp,1,2)*Ugrad(cell,qp,1,2); //epsilon_yz^2
    epsilonEqpSq += ff; //add regularization "fudge factor"
    mu(cell,qp) = factor*pow(epsilonEqpSq,  power); //non-linear viscosity, given by Glen's law
  }
}
```

Viscosity Kokkos kernel

```
template < typename ScalarType, class DeviceType >
class Viscosity {
 Array2 mu_;
 Array4 U_;
 int numQPs_;
 ScalarType ff_;
 ScalarType factor_;
 ScalarType power_;

 public:
 typedef DeviceType device_type;

 Viscosity (Array2 &mu,
           Array4 &u,
           int numQPs,
           ScalarType ff,
           ScalarType factor,
           ScalarType power)
  : mu_(mu)
  , U_(u)
  , numQPs_(numQPs)
  , ff_(ff)
  , factor_(factor)
  , power_(power){}

 KOKKOS_INLINE_FUNCTION
 void operator () (std::size_t i) const
 {
  ScalarType ep=0.0;
  for (std::size_t j=0; j<numQPs_; j++)
  {
   ep=U_(i, j,0,0)*U_(i,j,0,0);
   ep +=U_(i, j,1,1)*U_(i,j,1,1);
   ep +=U_(i, j,0,0)*U_(i,j,1,1);
   ep +=1.0/4.0*(U_(i, j,0,1)+ U_(i,j,1,0))*(U_(i,j,0,1)+U_(i,j,1,0));
   ep +=1.0/4.0*U_(i,j,0,2)*U_(i,j,0,2);
   ep +=1.0/4.0*U_(i,j,1,2)*U_(i,j,1,2);
   ep +=ff_;
   mu_(i,j) = factor_*pow(ep, power_);
  }
 }
};
```

Evaluation environments

Compton:

- 42 nodes:
  - Two 8-core Sandy Bridge Xeon E5-2670 @ 2.6GHz (HT activated) per node,
  - 24GB (3*8Gb) memory per node,
  - Two Pre-production KNC 2 per node.

Shannon:

- 32 nodes:
  - Two 8-core Sandy Bridge Xeon E5-2670 @ 2.6GHz (HT deactivated) per node,
  - 128GB DDR3 memory per node,
  - 2x NVIDIA K20x per node

Kokkos::Cuda on Shannon

Viscosity Host_time = 0.654771	Viscosity Device_time = 0.000481
Body Force Host_time = 0.014789	Body Force Device_time = 0.000451
Residual Host_time = 0.636981	Residual Device_time = 0.000536

Kokkos::Threads on Shannon

Viscosity Host_time = 0.69962	Viscosity Device_time = 0.045445
Body Force Host_time = 0.017365	Body Force Device_time = 0.002276
Residual Host_time = 0.565082	Residual Device_time = 0.040913

numThreads =2, numCores =8

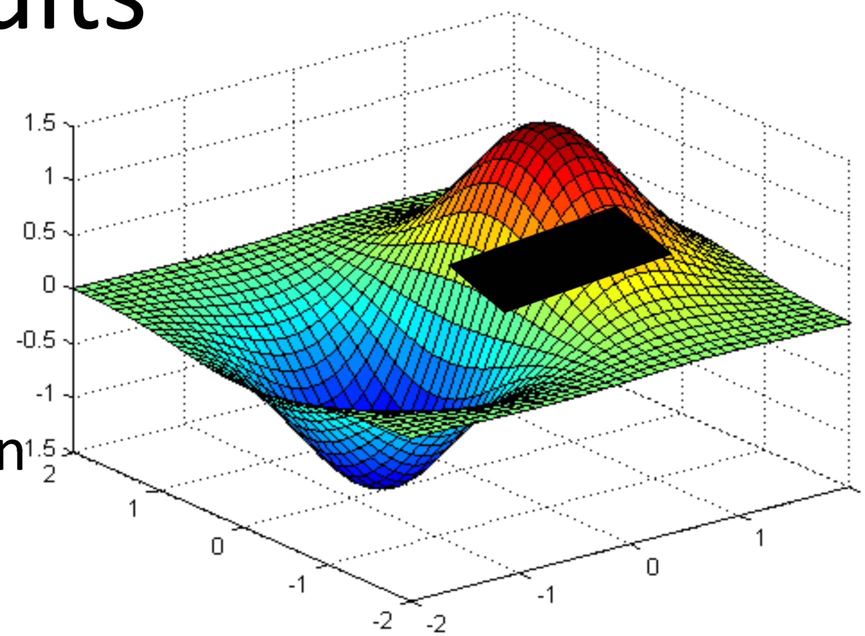
Kokkos::OpenMP on Compton (MIC)

Viscosity Host_time = 7.41132	Viscosity Device_time = 0.019931
Body Force Host_time = 1.18717	Body Force Device_time = 0.010295
Residual Host_time = 35.458	Residual Device_time = 0.130741

numThreads =4, numCores =56
numCells=10000, numWorkSet=100

Details: Resilience Models

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.

LFLR Algorithm Opportunities & Challenges

- 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.

First LFLR Example

- Prototype LFLR Transient PDE solver.
- Simulated process lost.
- Simulated persistent store.
- Over-provisioned MPI ranks.
-

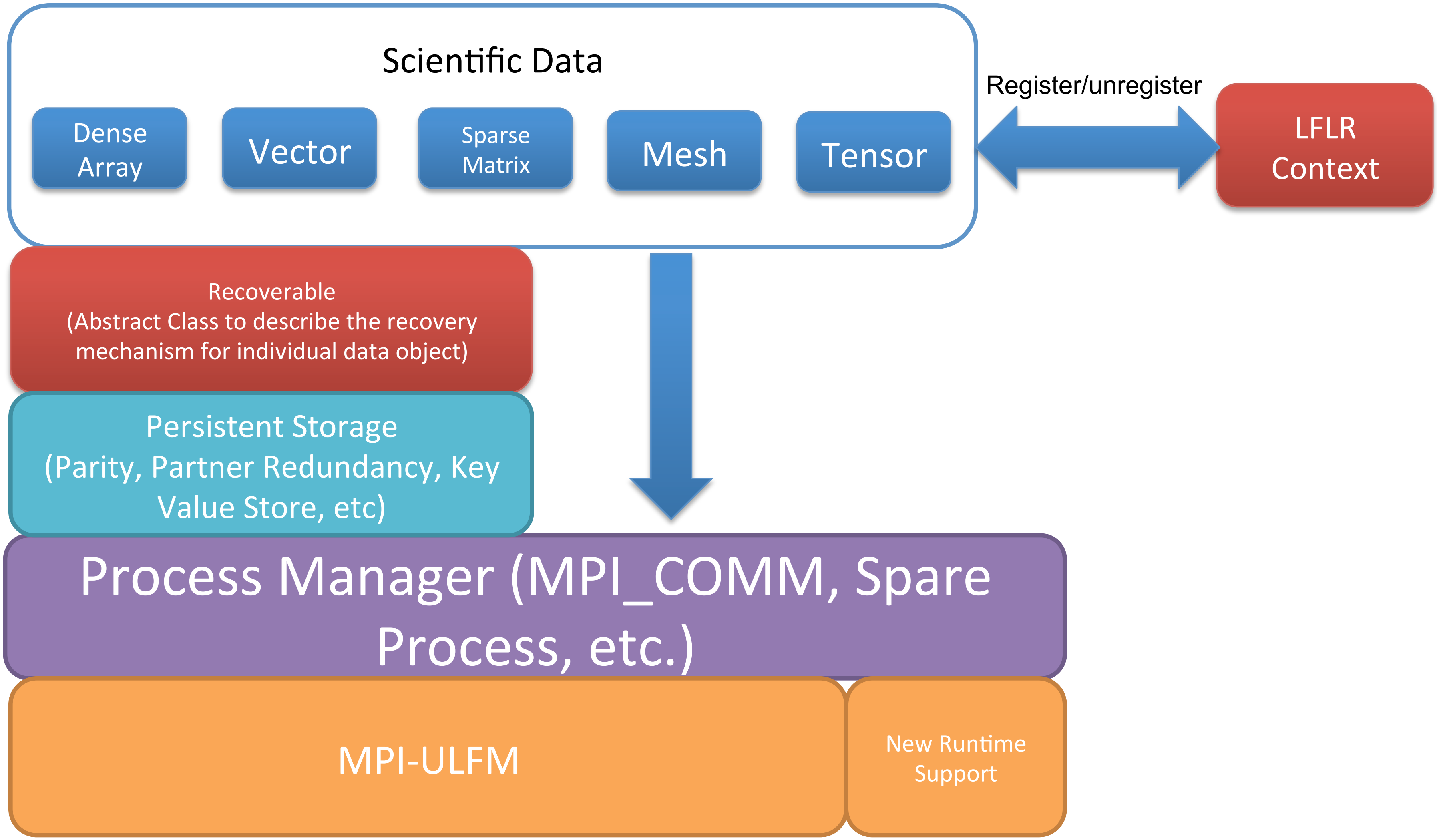
Data/work recovery time

Persistent store time

# of Processes	CG	READ	WRITE	ALL
4	2.64	0.008	0.01	2.77
8	5.39	0.09	0.012	5.83
16	7.84	0.008	0.013	7.99
32	9.9	0.008	0.014	10.04
64	12.56	0.009	0.0145	12.76
128	16.99	0.0085	0.015	17.14
256	21.6	0.009	0.016	21.76
512	28.75	0.009	0.015	28.91

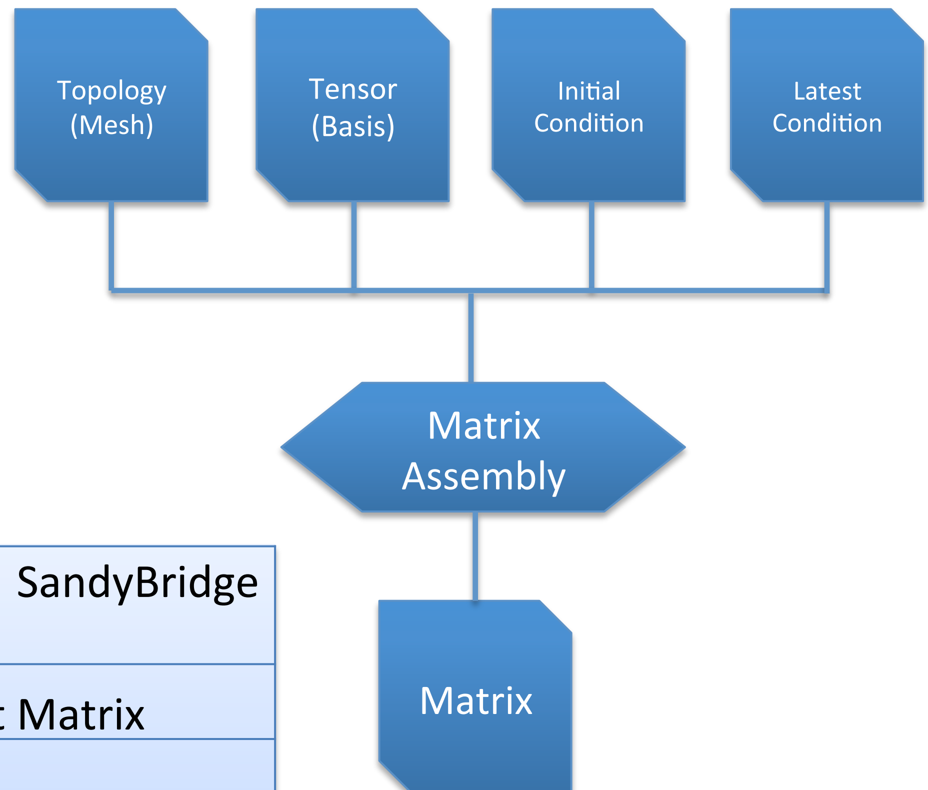
Results from explicit variant of Mantevo/MiniFE, Keita Teranishi

Design of LFLR



Data Recovery from Computation

- Lots of scientific objects are dependent on more compact data objects
  - Higher abstraction of mathematical model
- Can be recovered through inexpensive computation
  - 90%+ storage reduction in miniFE
  - Some refactoring in scientific objects
    - Put them “recoverable” subclass
  - Increase roll-back overhead



	miniFE: 512x512x512: 1024 SandyBridge CPU Cores (FDR IB)	
	With Matrix	Without Matrix
Storage per core	53.94 MB	2.1 MB
Regenerate overhead	(in memory) 0.1 sec (in global file system) 5 sec+	(in memory + compute) 0.6 sec

Every calculation matters

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.

Soft Error Resilience

- 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.

Skeptical Programming

I might not have a reliable digital machine

- Expect rare faulty computations
- Use analysis to derive <u>cheap</u> "detectors" to filter large errors
- Use numerical methods that can absorb *bounded error*

Algorithm 1: GMRES algorithm

for $l = 1$ to do
  $\mathbf{r} := \mathbf{b} - \mathbf{A}\mathbf{x}^{(j-1)}$
  $\mathbf{q}_1 := \mathbf{r}/\|r\|_2$
  for $j = 1$ to $restart$ do
    $\mathbf{w}_0 := \mathbf{A}\mathbf{q}_j$
    for $i = 1$ to j do
      $h_{i,j} := \mathbf{q}_i \cdot \mathbf{w}_{i-1}$
      $\mathbf{w}_i := \mathbf{w}_{i-1} - h_{i,j}\mathbf{q}_i$
    end
    $h_{j+1,j} := \|\mathbf{w}\|_2$
    $\mathbf{q}_{j+1} := \mathbf{w}/h_{j+1,j}$
    Find $\mathbf{y} = \min \|\mathbf{H}_j\mathbf{y} - \|\mathbf{b}\|\mathbf{e}_1\|_2$
    Evaluate convergence criteria
    Optionally, compute $\mathbf{x}_j = \mathbf{Q}_j\mathbf{y}$
  end
end

GMRES

Theoretical Bounds on the Arnoldi Process

$$\|\mathbf{w}_0\| = \|\mathbf{A}\mathbf{q}_j\| \leq \|\mathbf{A}\|_2\|\mathbf{q}_j\|_2$$
$$\|\mathbf{w}_0\| \leq \|\mathbf{A}\|_2 \leq \|\mathbf{A}\|_F$$

From isometry of orthogonal projections,

$$|h_{i,j}| \leq \|\mathbf{A}\|_F$$

- $h_{i,j}$ form Hessenberg Matrix
- Bound only computed once, valid for entire solve

Evaluating the Impact of SDC in Numerical Methods
J. Elliott, M. Hoemmen, F. Mueller, SC'13

What is Needed for Skeptical Programming?

- Skepticism.
- Meta-knowledge:
  - Algorithms,
  - Mathematics,
  - Problem domain.
- Nothing else, at least to get started.

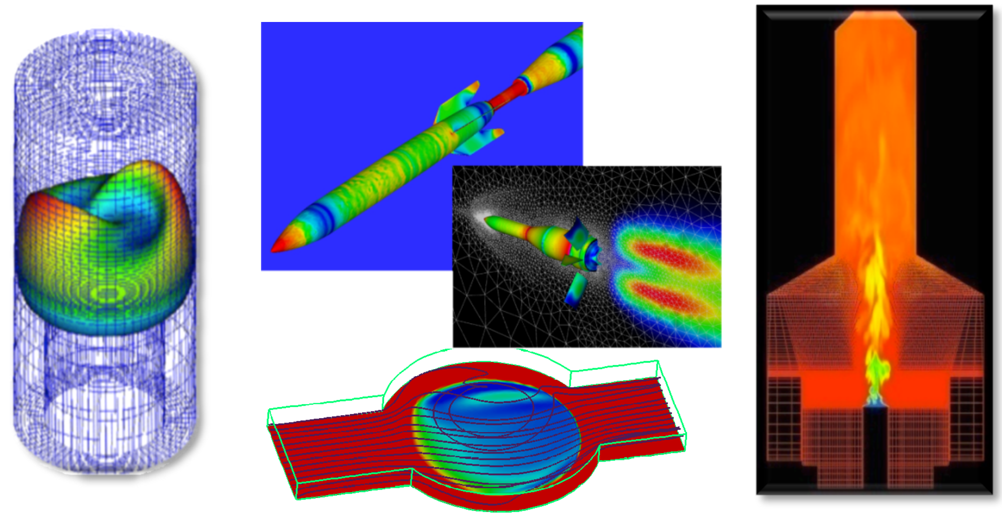
6. Summary

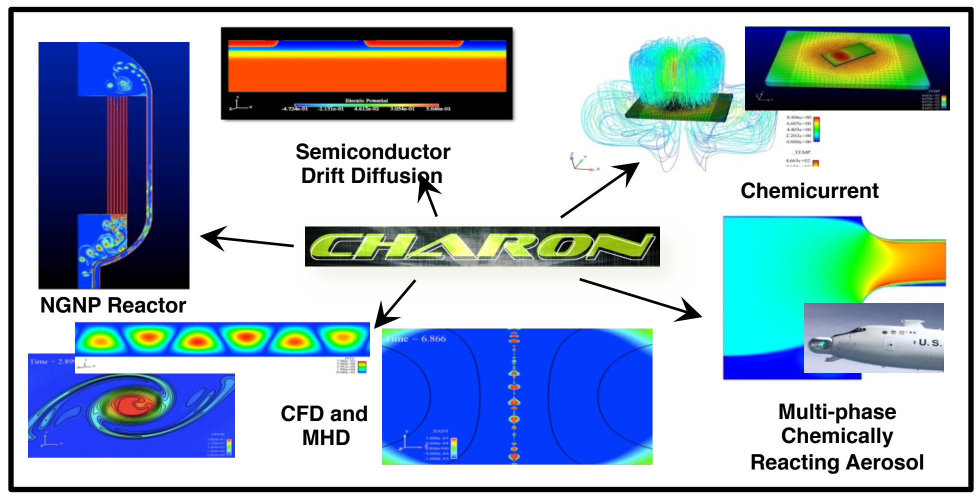
- What new science results might be afforded by improvements in NERSC computing hardware, software and services?
  - New NERSC capabilities would benefit all of the stratetic directions for Trilinos (although too much reliability could be a problem ☺.
- Recommendations on NERSC architecture, system configuration and the associated service requirements needed for your science
  - We are preparing for all reasonable architectures. Given other trends, we are tending to focus on MIC more at this time.
- NERSC generally refreshes systems to provide on average a 2X performance increase every year. What significant scientific progress could you achieve over the next 5 years with access to 32X your current NERSC allocation?
  - N/A.
- What "expanded HPC resources" are important for your project?
  - N/A.
- General discussion
  - Cray relationship: Ongoing, focus on old (Epetra) and new (Tpetra) stack.

Extras

Are we really starting from scratch?

- **No!** Leveraging/growing the agile components base!
- Sandia has a 20+ year history in HPC turbulent multiphase reacting flow solvers
- **Case Study**: ASC and ASCR funding recently generalized multiphysics assembly kernels into agile components
  - Ideas explored in Charon and SIERRA/Aria codes
  - Abstracted to generic software package
  - Now forms the core for assembly in Albany, Drekar::CFD, Drekar::MHD, Charon2, and Paradigm





Rapid Implementation of New Physics Using Graph-based Assembly Process

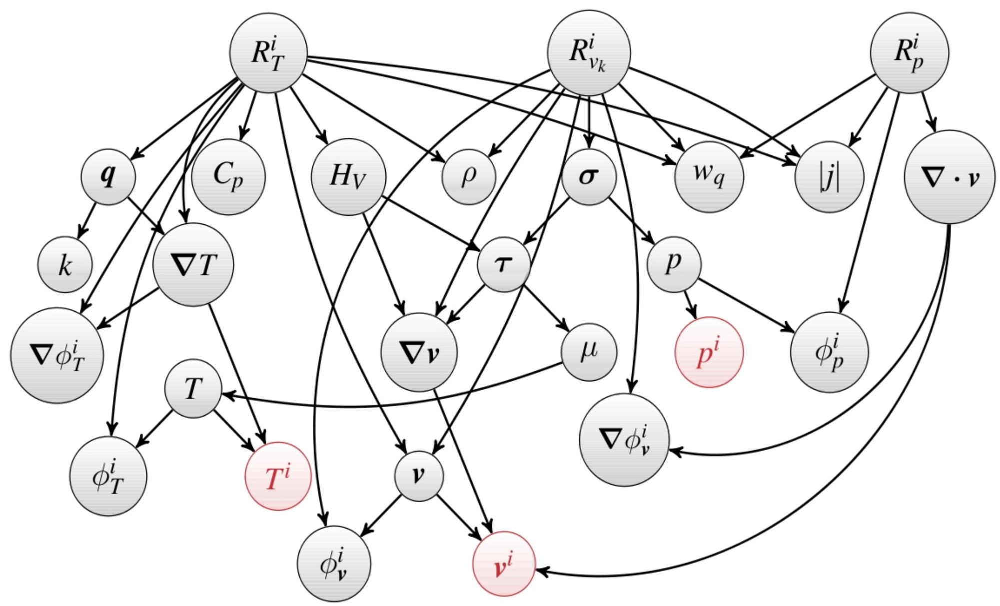
- Competing/Complementary Discretization Technology:
  - Symbolics and code generation: FEniCS/UFL/Dolphin/FIAT, Liszt
  - Symbolics in C++ → DSEL: Sundance
  - Graph-based assembly: Unitah
  - Graph-based assembly + TBGP: Drekar, Albany, SIERRA/Aria
  - Traditional coding of physics loops: Libmesh, Deal.II

- Advantages
  - Template-based Generic Programming
  - Automated dependency tracking
  - Extreme flexibility: easy to add/swap equations and models, test in isolation
  - User controlled granularity
  - Multi-core research: workset/alg. Decomposition
  - TPL integration
  - Debugging

$$R_T^i = \sum_{e=1}^{N_e} \sum_{q=1}^{N_q} \left[\left(\rho C_p \boldsymbol{v} \cdot \boldsymbol{\nabla} T - H_V \right) \phi_T^i - \boldsymbol{q} \cdot \boldsymbol{\nabla} \phi_T^i \right] w_q |j| = 0$$

$$R_{v_k}^i = \sum_{e=1}^{N_e} \sum_{q=1}^{N_q} \left[\rho \boldsymbol{v} \cdot \boldsymbol{\nabla} \boldsymbol{v} \phi_{\boldsymbol{v}}^i + \boldsymbol{\sigma} : \boldsymbol{\nabla} \left(\phi_{\boldsymbol{v}}^i \boldsymbol{e}_k \right) \right] w_q |j| = 0$$

$$R_p^i = \sum_{e=1}^{N_e} \sum_{q=1}^{N_q} \boldsymbol{\nabla} \cdot \boldsymbol{v} \phi_p^i w_q |j| = 0$$



Notz, Pawlowski, Sutherland; TOMS in press

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

- LFLR realization is non-trivial.
- Programming API (but not complicated). ULFM helps.
- Lots of runtime/OS infrastructure.
  - Persistent storage API (frequent brainstorming outcome).
- Research into messaging state and recovery? No.
- 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.

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.

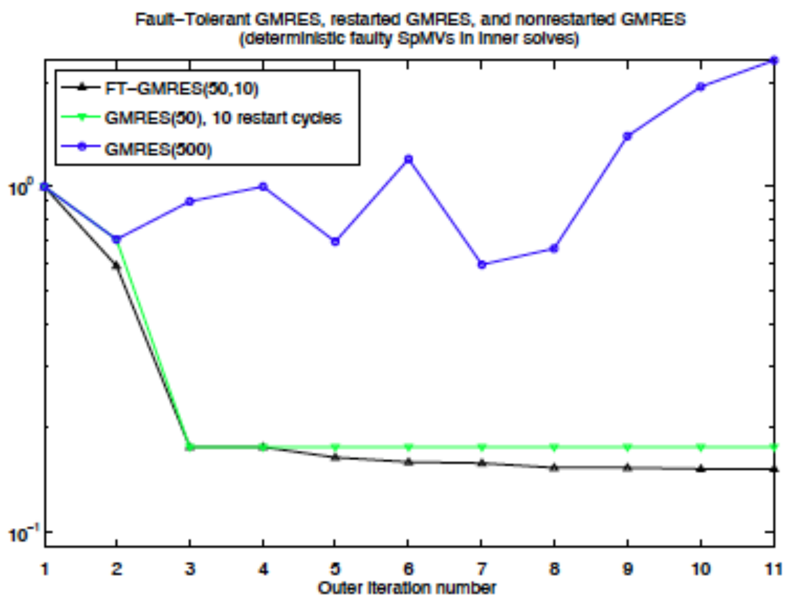
Captures true linear operator issues, AND
Can use some "garbage" soft error results.

What is Needed for Selective Reliability?

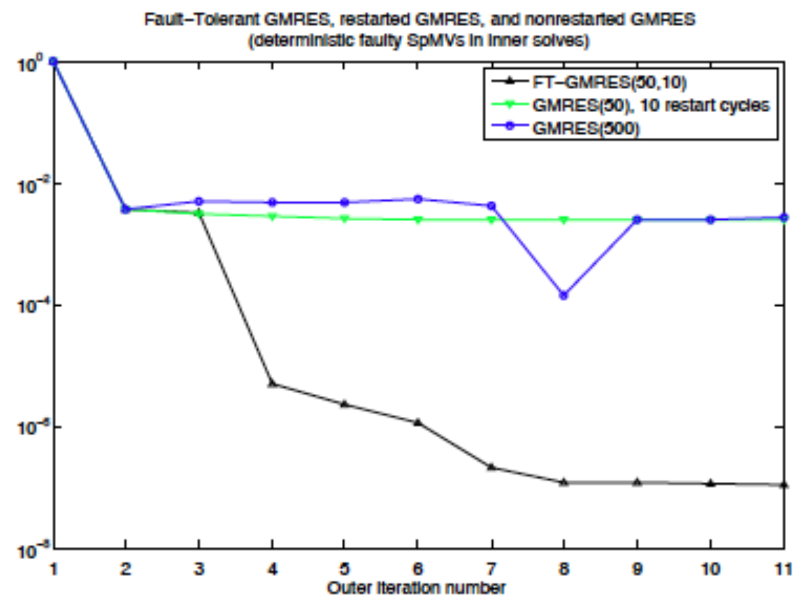
- A lot, lot.
- A programming model.
- Algorithms.
- Lots of runtime/OS infrastructure.
- Hardware support?

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