

# Recent Trends and Challenges for High Performance Sparse Linear Algebra

Michael A. Heroux  
Sandia National Laboratories

Contributors: Mark Hoemmen, Siva Rajamanickam

<https://trilinos.github.io>



Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.



# Outline

- Brief Overview of Trilinos.
- Some lessons learned (I think).
- On-node parallelism.
- Embedded Resilience.
- Non-accelerated systems return.

# *Trilinos Overview*

# What is Trilinos?

- Object-oriented software framework for...
- Solving big complex science & engineering problems.
- Large collection of reusable scientific capabilities.
- More like LEGO™ bricks than Matlab™.



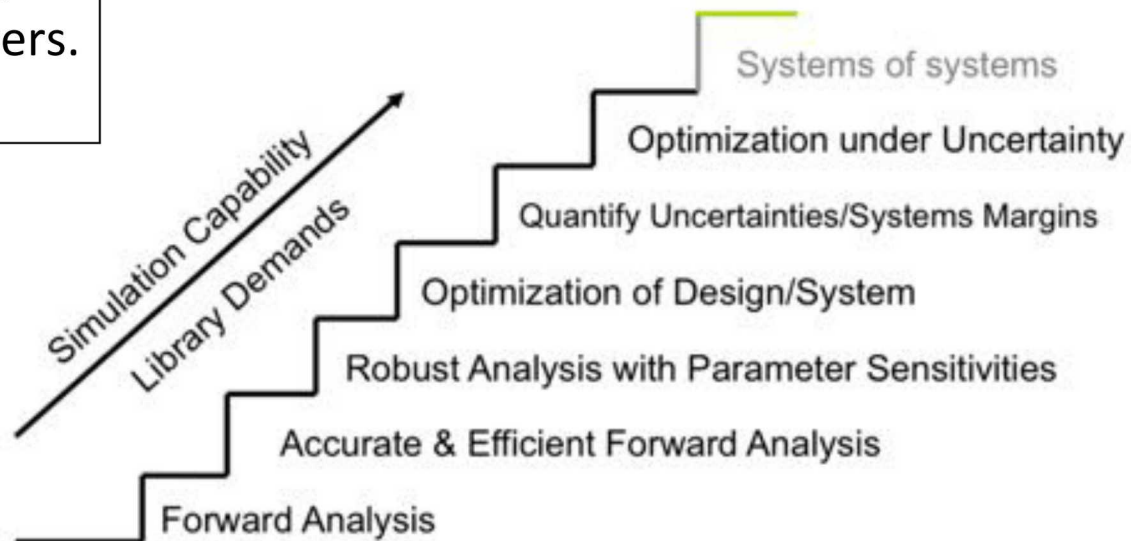


## Optimal Kernels to Optimal Solutions:

- ◆ Geometry, Meshing
- ◆ Discretizations, Load Balancing.
- ◆ Scalable Linear, Nonlinear, Eigen, Transient, Optimization, UQ solvers.
- ◆ Scalable I/O, GPU, Manycore

- ◆ 60+ Packages.
- ◆ Distributions:
  - ◆ GitHub repo.
  - ◆ Cray LIBSCI, Linux
- ◆ Thousands of Users.
- ◆ Worldwide distribution.

## Laptops to Leadership systems



Each stage requires *greater performance and error control* of prior stages:  
**Always will need: more accurate and scalable methods.  
more sophisticated tools.**

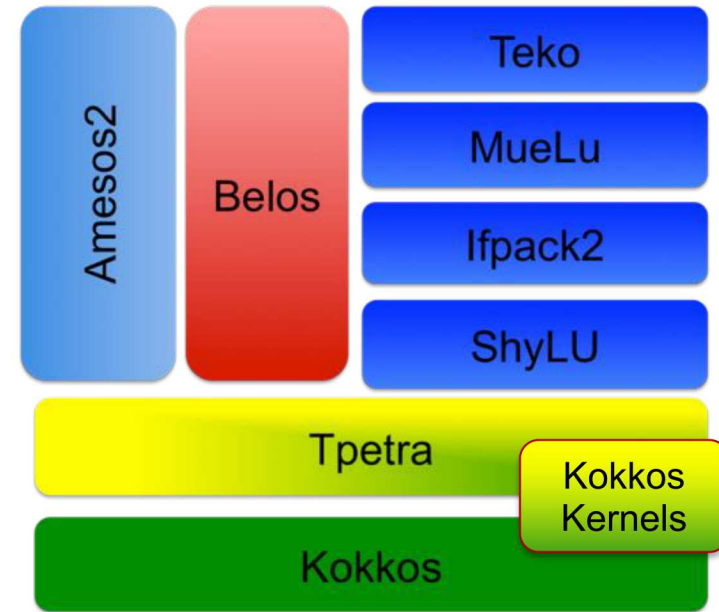


# Trilinos Highlights

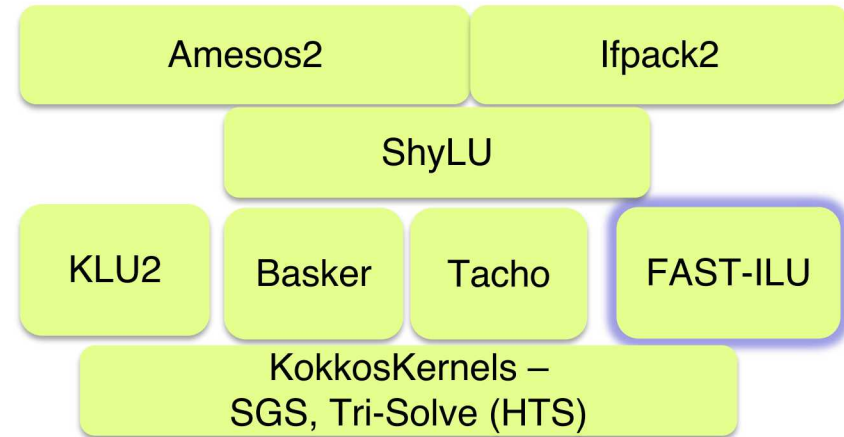
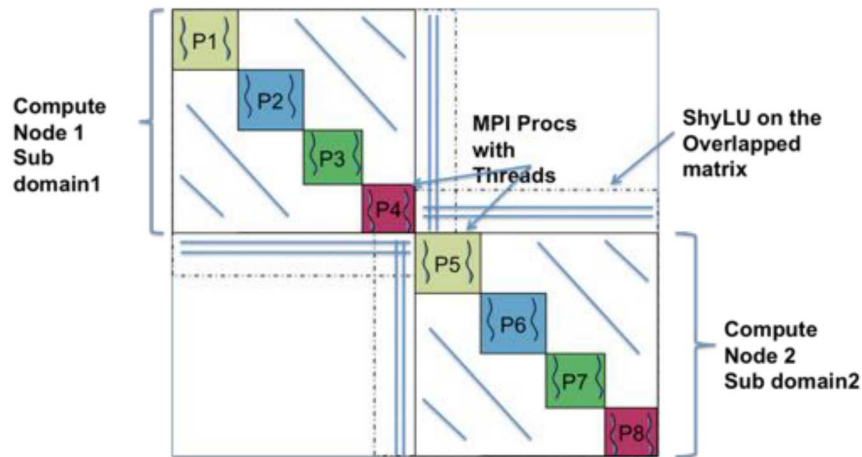
- Huge library of algorithms
  - Linear and nonlinear solvers, preconditioners, ...
  - Optimization, transients, sensitivities, uncertainty, ...
- Solid support for multicore & hybrid CPU/GPU
  - Built into the new Tpetra linear algebra objects
    - Therefore into iterative solvers with zero effort!
  - Unified intranode programming model: Kokkos
  - Spreading into the whole stack:
    - Multigrid, sparse factorizations, element assembly...
- Support for mixed and arbitrary precisions
  - Don't have to rebuild Trilinos to use it
- Support for flexible 2D sparse partitioning
  - Useful for graph analytics, other data science apps.

# Trilinos linear solvers

- Sparse linear algebra (Kokkos/KokkosKernels/Tpetra)
  - Threaded construction, Sparse graphs, (block) sparse matrices, dense vectors, parallel solve kernels, parallel communication & redistribution
- Iterative (Krylov) solvers (Belos)
  - CG, GMRES, TFQMR, recycling methods
- Sparse direct solvers (Amesos2)
- Algebraic iterative methods (Ifpack2)
  - Jacobi, SOR, polynomial, incomplete factorizations, additive Schwarz
- Shared-memory factorizations (ShyLU)
  - LU, ILU(k), ILUt, IC(k), iterative ILU(k)
  - Direct+iterative preconditioners
- Segregated block solvers (Teko)
- Algebraic multigrid (MueLu)



# ShyLU and Subdomain Solvers : Overview



## MPI+X based subdomain solvers

Decouple the notion of one MPI rank as one subdomain: Subdomains can span multiple MPI ranks each with its own subdomain solver using X or MPI+X

**Subpackages of ShyLU:** Multiple Kokkos-based options for on-node parallelism

**Basker** : LU or ILU (t) factorization

**Tacho**: Incomplete Cholesky - IC (k)

**Fast-ILU**: Fast-ILU factorization for GPUs

**KokkosKernels**: Coloring based Gauss-Seidel (M. Deveci), Triangular Solves (A. Bradley)



# *Some Lessons Learned*

# *Simultaneous* heterogeneous execution is hard

- HPCG on Trinity
- 9380 Haswell, 9984 KNL compute nodes.
  - Haswell
    - Processor dimensions: 27x42x17
    - Local grid dimensions: **160x160x112**
  - KNL
    - Processor dimensions: 27x42x34
    - Local dimensions: **160x160x152**
- HPCG result: 546 TF/s (4<sup>th</sup> at ISC18).
  - Previous 180 TF/s for Haswell only.
- Key Point: For sparse codes, it's about the memory system.
- **For accelerated systems, *simultaneous* heterogeneous execution seems unwise: Keep all computation on the GPUs.**

HPCG on SIERRA (Power9's + 4 Voltas):

- About 10% of performance is from Power9's
- Summit 6 GPUs: Power9's less important.
- Both:
  - Code complexity challenging.
  - Runtime system complexity (MPI).
  - Work partitioning.

# Porting to accelerated systems

## 2-phase strategy: TaihuLight

- Management Processing Element (MPE)
  - 64-bit RISC core
  - support both user and system modes
  - 256-bit vector instructions
  - 32 KB L1 i-cache, 32 KB L1 data, both 4 way set associative.
  - 256 KB L2 cache, 8 way set associative
  
- Computing Processing Element (CPE)
  - 8x8 CPE mesh
  - 64-bit RISC core
  - support only user mode
  - 256-bit vector instructions
  - 64 KB Scratch Pad Mem (1 private per CPE)
    - Can be configured as explicit local mem or SW managed \$.
  - Each CPE has own 16KB direct mapped i-cache.

### Initial port:

- Vanilla MPI, 1 rank per MPE
- 23.2 GF/s /core
- 4 vector FMA
- 2 pipes
- 16 Flops/cycle FMA
- Peak: 2/65 of node peak

### Subsequent optimization:

- Offload any work to CPEs
- 11.6 GF/s /core
- 4 vector FMA
- 1 pipe
- 8 Flops/cycle FMA

# CAM-SE to TaihuLight: 2017 Gordon Bell Finalist

- CAM-SE: Spectral Element Atmospheric dynamical core
  - Reported:
    - 754,129 SLOC.
    - 152,336 SLOC modified for TaihuLight (20%).
    - 57,709 SLOC added (8%).
    - 12+ team members.
  - Challenges:
    - Reusability of code seems low: Much of the optimization is specific to Sunway CPE processor.
    - Translation effort difficult to accomplish while still delivery science results: Disruptive.
  - Other notable example: Uintah (see Dec 2017 ASCAC talk)
    - Separation of runtime concerns seems to really help, but app-specific.

# Some Observations from these Efforts

- Even the simplest simultaneous heterogeneous execution is difficult.
  - **Best option seems to keep all significant computation on accelerator.**
  - More generally: Heterogeneous execution is fine, if it's not simultaneous.
- MPI-backbone approach is very attractive.
  - **Initial app port to host backbone, hotspot optimization.**
  - Investment in portable programming expressions seems essential.
  - Separation of functionality expression and work/data mapping seems essential.



# Pattern for parallel dynamic allocation

- Pattern:
  1. Count / estimate allocation size; may use Kokkos parallel\_scan
  2. Allocate; use Kokkos::View for best data layout & first touch
  3. Fill: parallel\_reduce over error codes; if you run out of space, keep going, count how much more you need, & return to (2)
  4. Compute (e.g., solve the linear system) using filled data structures
- Compare to Fill, Setup, Solve sparse linear algebra use pattern
- Semantics change: Running out of memory not an error!
  - Always return: Either no side effects, or correct result
  - Callers must expect failure & protect against infinite loops
  - Generalizes to other kinds of failures, even fault tolerance
- **Thread-scalable execution of mundane code is “straightforward” but hard work.**

# *On-node data & execution*

# Sparse LA Challenges vs Dense

- Dynamic tasking:
  - Fine grain work migration is not effective.
    - Cost of data migration is too high.
    - Best to make sure data are mapped to make most efficient use of bandwidth.
  - Coarse grain can work:
    - Encapsulate memory allocation, initialization and computation.
    - Assures co-location of work and data.
- Indirect addressing:
  - Gather/scatter.
  - Atomic writes vs. coloring.
- Data structure polymorphism:
  - "Sparse" encompasses many kinds of problems.
  - Architecture details impact data structure choices.
- **Response: Encapsulate data/work in polymorphic layer**

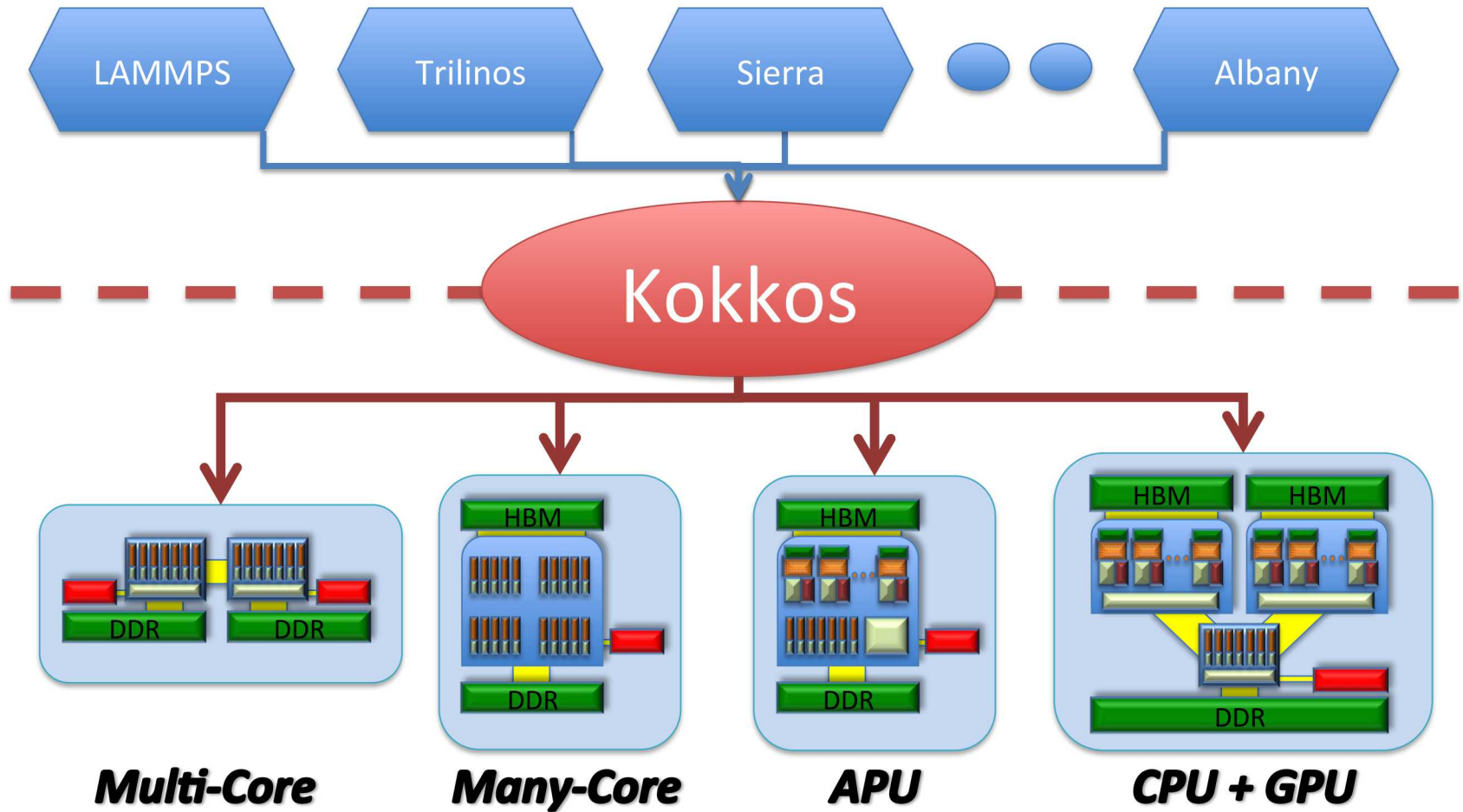
# Must support multiple architectures Sandia National Laboratories

- Systems to support
  - Trinity (Intel Haswell & KNL)
  - Sierra: NVIDIA GPUs + IBM multicore CPUs
  - Astra: Arm manycore
  - Plus “everything else”
- 3 different architectures
  - ~~Multicore CPUs (big cores)~~
  - Manycore CPUs (small cores)
  - GPUs (highly parallel)
- MPI only, & MPI + threads
  - Threads don't always pay on non-GPU architectures today
  - Porting to threads must not slow down the MPI-only case

**CORAL**  
COLLABORATION  
OAK RIDGE • ARGONNE • LIVERMORE



# Kokkos: *Performance, Portability, & Productivity*





*Goal: One Code gives good performance on every platform*

- Machine model:
  - N execution space + M memory spaces
  - NxM matrix for memory access performance/possibility
  - Asynchronous execution allowed
- Implementation approach
  - A C++ template library
  - C++11 now required
  - Target different back-ends for different hardware architecture
  - Abstract hardware details and execution mapping details away
- Distribution
  - Open Source library
  - Available on GitHub
- Long Term Vision
  - Move features into the C++ standard

**Execution Pattern:** `parallel_for`, `parallel_reduce`, `parallel_scan`, `task`, ...

**Execution Policy :** how (and where) a user function is executed

E.g., data parallel range : concurrently call `function(i)` for  $i = [0..N)$

User's function is a C++ functor or C++11 lambda

**Execution Space :** where functions execute

Encapsulates hardware resources; e.g., cores, GPU, vector units, ...

**Memory Space :** where data resides

➤ AND what execution space can access that data

Also differentiated by access performance; e.g., latency & bandwidth

**Memory Layout :** how data structures are ordered in memory

➤ provide mapping from logical to physical index space

**Memory Traits :** how data shall be accessed

➤ allow specialisation for different usage scenarios (read only, random, atomic, ...)

# Execution Pattern

```
#include <Kokkos_Core.hpp>
#include <cstdio>

int main(int argc, char* argv[]) {
    // Initialize Kokkos analogous to MPI_Init()
    // Takes arguments which set hardware resources (number of threads, GPU Id)
    Kokkos::initialize(argc, argv);

    // A parallel_for executes the body in parallel over the index space, here a simple range 0<=i<10
    // It takes an execution policy (here an implicit range as an int) and a functor or lambda
    // The lambda operator has one argument, and index_type (here a simple int for a range)
    Kokkos::parallel_for(10, [=](int i){
        printf("Hello %i\n", i);
    });

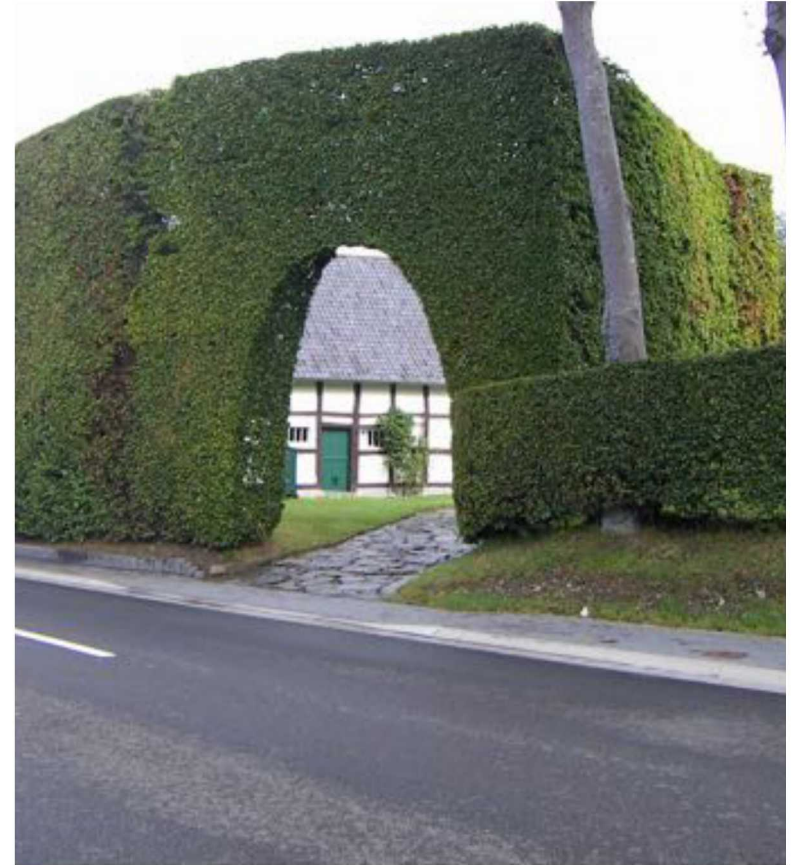
    // A parallel_reduce executes the body in parallel over the index space, here a simple range 0<=i<10 and
    // performs a reduction over the values given to the second argument
    // It takes an execution policy (here an implicit range as an int); a functor or lambda; and a return value
    double sum = 0;
    Kokkos::parallel_reduce(10, [=](int i, int& lsum) {
        lsum += i;
    }, sum);
    printf("Result %lf\n", sum);

    // A parallel_scan executes the body in parallel over the index space, here a simple range 0<=i<10 and
    // Performs a scan operation over the values given to the second argument
    // If final == true lsum contains the prefix sum.
    double sum = 0;
    Kokkos::parallel_scan(10, [=](int i, int& lsum, bool final) {
        if(final) printf("ScanValue %i\n", lsum);
        lsum += i;
    });

    Kokkos::finalize();
}
```

# Kokkos protects us against...

- Hardware divergence
- Programming model diversity
- Threads at all
  - `Kokkos::Serial` back-end
  - Kokkos' semantics require vectorizable (ivdep) loops
  - Expose parallelism to exploit later
  - Hierarchical parallelism model encourages exploiting locality
- Kokkos protects our HUGE time investment of porting Trilinos



Kokkos is our hedge



# Other Node-Parallel Abstractions

- An underlying node-parallel data/loop abstraction seems necessary for sparse computations:
  - To reduce redundant coding for different targets.
  - To provide good data placement strategies.
- Kokkos is one approach. Others include:
  - RAJA.
  - OCCA.
  - OpenACC.
  - OpenMP (ramping up).
- Without some abstraction, these things are especially hard:
  - Porting to different architectures.
  - Execution on heterogeneous architectures.



# *Resilient Sparse Solvers*

# Our Luxury in Life (wrt FT/Resilience)

The privilege to think of a computer as a  
*reliable, digital* machine.

Conjecture: This privilege will persist through  
Exascale.

Reason: Vendors will not give us a unreliable  
system until we are ready to use one, and  
we will not be ready by 2020 – 2023.

# Take away message

If we want unreliable systems,  
we must work harder on resilience.



# Four Resilience Programming Models

- Relaxed Bulk Synchronous (rBSP)
- **Skeptical Programming. (SP)**
- Local-Failure, Local-Recovery (LFLR)
- **Selective (Un)reliability (SU/R)**

*Toward Resilient Algorithms and Applications*  
Michael A. Heroux arXiv:1402.3809v2 [cs.MS]  
<https://arxiv.org/abs/1402.3809>

# Skeptical Programming

*I might not have a reliable digital machine*

- Expect rare faulty computations
- Use analysis to derive cheap “detectors” to filter large errors
- Use numerical methods that can absorb *bounded error*

**Algorithm 1:** GMRES algorithm

```
for  $l = 1$  to do
   $\mathbf{r} := \mathbf{b} - \mathbf{A}\mathbf{x}^{(j-1)}$ 
   $\mathbf{q}_1 := \mathbf{r} / \|\mathbf{r}\|_2$ 
  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.
- FEM ideas:
  - Invariant subspaces.
  - Conservation principles.
  - More generally:
    - pre-conditions, post-conditions, invariants.



Note: These same ideas are useful for the Artifact Evaluation Appendix, used by SC18 Tech Papers Program.

# Every calculation matters

## Soft Error Resilience



Description	Iters	FLOPS	Recursive Residual Error	Solution Error
All Correct Calcs	35	343 M	4.6e-15	1.0e-6
Iter=2, $y[1] += 1.0$ SpMV incorrect Ortho subspace	35	343 M	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.

*Fault-tolerant linear solvers via selective reliability,*

Patrick G. Bridges, Kurt B. Ferreira,  
Michael A. Heroux, Mark Hoemmen  
arXiv:1206.1390v1 [math.NA]

# 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**

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

**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**

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

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

**end for**

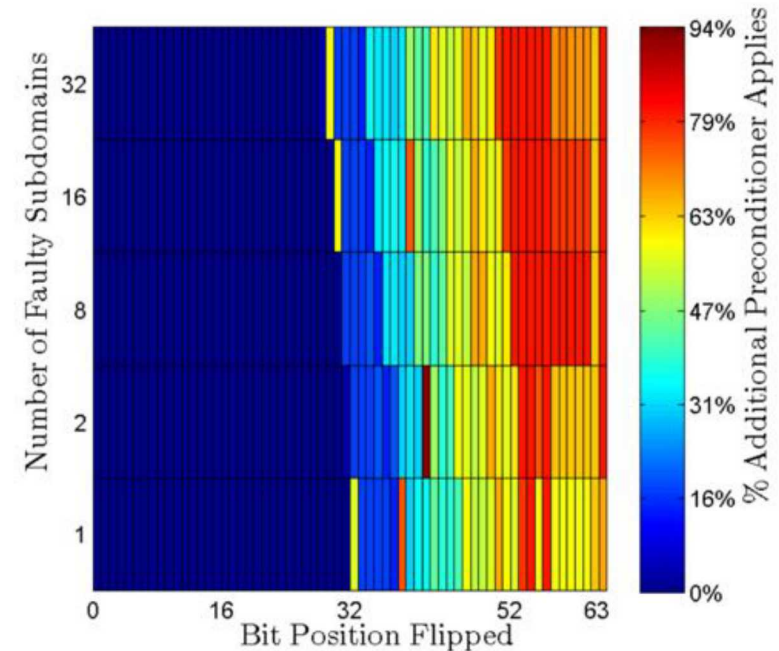
“Unreliably” computed.  
Standard solver library call.  
Majority of computational cost.

▷ Orthogonalize  $v_{j+1}$

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

# Selective Reliability Highlights

- The Selective Reliability Model is Implementable, even today.
- SDC (Bit-flips) are not equally impactful.
- Bit protection can be selective:
  - Integers, Pointers – Use high bits.
  - FP – Exponent bits.
- (Lowlight) Work in this area is stalled.
- Waiting for the next call-to-arms.
- Beyond Moore may be next opportunity (unreliable digital).



James Elliott, Mark Hoemmen, and Frank Mueller. 2015. A Numerical Soft Fault Model for Iterative Linear Solvers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '15)*. ACM, New York, NY, USA, 271-274. DOI=<http://dx.doi.org/10.1145/2749246.2749254>

# *NON-ACCELERATOR SYSTEMS*



# Non-accelerate systems didn't disappear: Sandia National Laboratories

## Sandia “Astra” System

- Cavium ThunderX2 ARM based system.
- System performance: 2.3 PF.
- Number of nodes: 2,592
- Cores/node: 56 (2 socket, 28 cores)
- Better than average memory performance (8-channel design.)
- Total # cores: 145152
- Core spec:
  - 2GHz
  - 2 128-bit FP units, 4 fmadds
  - 8 DP flops/core/clock/
- Also: Diskless storage system.

# ThunderX3 SVE (Future capability)

- **Scalable vector length** allowing each implementation to choose the amount of parallelism.
- **Rich addressing modes** including non-linear data accesses.
- **Per-lane predication** allowing vectorization of loops with complex control flow.
- **Predicate-driven loop control and management** to reduce vectorization overhead relative to scalar code.
- **Horizontal operations** for reducible loop-carried dependencies.
- **Vector partitioning and software-managed speculation** to vectorize loops with data-dependent exits.
- **Scalarized intra-vector sub-loops** to allow vectorizing loops with complex loop-carried dependencies.

Alejandro Rico, José A. Joao, Chris Adeniyi-Jones, and Eric Van Hensbergen. 2017. ARM HPC Ecosystem and the Reemergence of Vectors: Invited Paper. In Proceedings of the Computing Frontiers Conference (CF'17). ACM, New York, NY, USA, 329-334. DOI: <https://doi.org/10.1145/3075564.3095086>

# Final Take-Away Points

Development for accelerated sparse solvers in full swing.

Critical mass of sparse solver code exists, continues to grow.

Simultaneous heterogeneous execution is hard.

Sequenced heterogeneous is OK, but need code generation tools for multiple targets (e.g., Kokkos).

Intra-node parallelism is still biggest challenge:

Kokkos provides vehicle for reasoning and implementing on-node parallel.

Eventual goal: Search and replace Kokkos:: with std::

Node-parallel algorithms are already available.

Fully node parallel execution is hard work.

# Take Away points, cont.

- Resilience will be an issue, really.
  - But only as we are ready to adapt algorithms and codes.
  - The longer we delay, the more likely we will have a future system installed but never accepted.
- Already is:
  - Performance variability is result.
    - Latency tolerant algorithms are key.
  - Delays in system delivery, others.
- Checkpoint/restart will continue to improve:
  - NVRAM, Compression
  - Moving away from this model is very expensive.
- Embedded soft error detection/correction could be useful:
  - Skeptical programming (meta-computations).
  - Selective reliability (managing key bits and pointers).
  - Does not require any special support outside of solvers.

# Take Away points, cont.

- Manycore systems are re-emerging.
- Far from exascale-ready.
- Astra: 2.3 PF on 2600 nodes.
- Exascale possible with 430X:
  - Combination node/core/VL increases.
  - Example (440X):
    - Nodes: 2600 → 250,000
    - Cores: 56 → 64
    - VL: 4 → 16
- Targeting accelerators and manycore with same code base:
  - Kokkos-like abstractions needed.
  - Will still be challenging.
- Plenty of work to do, and we didn't talk about asynchronous tasking!