Building the Next Generation of Parallel Applications and Libraries

Michael A. Heroux
Scalable Algorithms Department
Sandia National Laboratories

Collaborators:
SNL Staff: [B.|R.] Barrett, E. Boman, R. Brightwell, H.C. Edwards, A. Williams
SNL Postdocs: M. Hoemmen, S. Rajamanickam,
MIT Lincoln Lab: M. Wolf
ORNL staff: Chris Baker

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.
Quiz (True or False)

1. MPI-only has the best parallel performance.
2. Future parallel applications will not have MPI_Init().
3. Use of “markup”, e.g., OpenMP pragmas, is the least intrusive approach to parallelizing a code.
4. All future programmers will need to write parallel code.
5. DRY is not possible across CPUs and GPUs.
6. CUDA and OpenCL will be footnotes in computing history.
7. Extended precision is too expensive to be useful.
8. Resilience will be built into algorithms.
10. Global SIMT is sufficient parallelism for scientific computing.
Trilinos Background & Motivation
Trilinos Contributors

Current Contributors

Chris Baker
Ross Bartlett
Pavel Bochev
Erik Boman
Lee Buermann
Todd Coffey
Eric Cyr
David Day
Karen Devine
Clark Dohrmann
David Gay
Glen Hansen
David Hensinger
Mike Heroux
Mark Hoemmen
Russell Hooper
Jonathan Hu
Sarah Knepper
Patrick Knupp
Joe Kotulska
Jason Kraftcheck
Rich Lehoucq
Nicole Lemaster
Kevin Long
Karla Morris
Chris Newman
Kurtis Nusbaum
Ron Oldfield
Mike Parks
Roger Pawlowski
Brent Perschbacher
Kara Peterson
Eric Phipps
Siva Rajamanickam
Denis Ridzal
Lee Ann Riesen
Damian Rouson
Andrew Salinger
Nico Schlömer
Chris Siefert
Greg Sjaardema
Bill Spotz
Heidi Thornquist
Ray Tuminaro
Jim Willenbring
Alan Williams
Michael Wolf

Past Contributors

Paul Boggs
Jason Cross
Michael Gee
Esteban Guillen
Bob Heaphy
Ulrich Hetmaniuk
Robert Hoekstra
Vicki Howle
Kris Kampshoff
Tammy Kolda
Joe Outzen
Mike Phenow
Paul Sexton
Ken Stanley
Marzio Sala
Cedric Chevalier
Target Problems: PDES and more…

PDES

Inhomogeneous Fluids

Circuits

And More…
Target Platforms: Any and All
(Now and in the Future)

- Desktop: Development and more…
- Capability machines:
  - Cielo (XE6), JaguarPF (XT5), Clusters
  - Titan (Hybrid CPU/GPU).
  - Multicore nodes.
- Parallel software environments:
  - MPI of course.
  - threads, vectors, CUDA OpenCL, …
  - Combinations of the above.
- User “skins”:
  - C++/C, Python
  - Fortran.
  - Web.
Evolving Trilinos Solution

- Trilinos\(^1\) is an evolving framework to address these challenges:
  - Fundamental atomic unit is a *package*.
  - Includes core set of vector, graph and matrix classes (Epetra/Tpetra packages).
  - Provides a common abstract solver API (Thyra package).
  - Provides a ready-made package infrastructure:
    - Source code management (git).
    - Build tools (Cmake).
    - Automated regression testing.
    - Communication tools (mail lists, trac).
  - Specifies requirements and suggested practices for package SQA.
- In general allows us to categorize efforts:
  - Efforts best done at the Trilinos level (useful to most or all packages).
  - Efforts best done at a package level (peculiar or important to a package).
- Allows package developers to focus only on things that are unique to their package.

1. Trilinos loose translation: “A string of pearls”
Transforming Computational Analysis To Support High Consequence Decisions

- Forward Analysis
  - Accurate & Efficient Forward Analysis
  - Robust Analysis with Parameter Sensitivities
  - Optimization of Design/System
  - Quantify Uncertainties/Systems Margins
  - Optimization under Uncertainty
  - Systems of systems

Each stage requires greater performance and error control of prior stages:
Always will need: more accurate and scalable methods.
more sophisticated tools.
Trilinos Download History: 19525 Total

Trilinos Download History Mar 2005 - May 2011

Number of Downloads

Month

Mar-05 Jun-05 Sep-05 Dec-05 Mar-06 Jun-06 Sep-06 Dec-06 Mar-07 Jun-07 Sep-07 Dec-07 Mar-08 Jun-08 Sep-08 Dec-08 Mar-09 Jun-09 Sep-09 Dec-09 Mar-10 Jun-10 Sep-10 Dec-10 Mar-11

Colors:
- 10.6
- 10.4
- 10.2
- 10.0
- 9.0
- 8.0
- 7.0
- 6.0
- 5.0
Registered User by Region

Registered Users by Region (6158 Total)

- Europe: 2306
- US (except Sandia): 2100
- Sandia (includes unregistered): 944
- Asia: 361
- Americas (except US): 295
- Australia/NZ: 84
- Africa: 68
Registered Users by Type

Registered Users by Type
(6158 Total)

University; 3698
Government; 912
Personal; 742
Industry; 660
Other; 146
maherou@jaguar13:/ccs/home/maherou> module avail trilinos

------------------------------------------ /opt/cray/modulefiles ------------------------------------------
trilinos/10.0.1(default) trilinos/10.2.0

------------------------------------------ /sw/xt5/modulefiles ------------------------------------------
trilinos/10.0.4 trilinos/10.2.2 trilinos/10.4.0 trilinos/8.0.3 trilinos/9.0.2
Capability Leaders: Layer of Proactive Leadership

- **Areas:**
  - Framework, Tools & Interfaces (J. Willenbring).
  - Software Engineering Technologies and Integration (R. Bartlett).
  - Discretizations (P. Bochev).
  - Scalable Linear Algebra (M. Heroux).
  - Linear & Eigen Solvers (J. Hu).
  - Nonlinear, Transient & Optimization Solvers (A. Salinger).
  - Scalable I/O: (R. Oldfield)

- Each leader provides strategic direction across all Trilinos packages within area.
# Trilinos Package Summary

<table>
<thead>
<tr>
<th>Objective</th>
<th>Package(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meshing &amp; Discretizations</td>
<td>STKMesh, Intrepid, Pamgen, Sundance, ITAPS, Mesquite</td>
</tr>
<tr>
<td>Time Integration</td>
<td>Rythmos</td>
</tr>
<tr>
<td>Automatic Differentiation</td>
<td>Sacado</td>
</tr>
<tr>
<td>Mortar Methods</td>
<td>Moertel</td>
</tr>
<tr>
<td>Linear algebra objects</td>
<td>Epetra, Jpetra, Tpetra, Kokkos</td>
</tr>
<tr>
<td>Interfaces</td>
<td>Thyra, Stratimikos, RTOp, FEI, Shards</td>
</tr>
<tr>
<td>Load Balancing</td>
<td>Zoltan, Isorropia</td>
</tr>
<tr>
<td>“Skins”</td>
<td>PyTrilinos, WebTrilinos, ForTrilinos, Ctrilinos, Optika</td>
</tr>
<tr>
<td>C++ utilities, I/O, thread API</td>
<td>Teuchos, EpetraExt, <strong>Kokkos</strong>, Triutils, ThreadPool, Phalanx</td>
</tr>
<tr>
<td>Iterative linear solvers</td>
<td>AztecOO, Belos, Komplex</td>
</tr>
<tr>
<td>Direct sparse linear solvers</td>
<td>Amesos, Amesos2</td>
</tr>
<tr>
<td>Direct dense linear solvers</td>
<td>Epetra, Teuchos, Pliris</td>
</tr>
<tr>
<td>Iterative eigenvalue solvers</td>
<td>Anasazi, Rbgen</td>
</tr>
<tr>
<td>ILU-type preconditioners</td>
<td>AztecOO, IFFPACK, Ifpack2</td>
</tr>
<tr>
<td>Multilevel preconditioners</td>
<td>ML, CLAPS</td>
</tr>
<tr>
<td>Block preconditioners</td>
<td>Meros, Teko</td>
</tr>
<tr>
<td>Nonlinear system solvers</td>
<td>NOX, LOCA</td>
</tr>
<tr>
<td>Optimization (SAND)</td>
<td>MOOCHO, Aristos, TriKota, Globipack, Optipack</td>
</tr>
<tr>
<td>Stochastic PDEs</td>
<td>Stokhos</td>
</tr>
</tbody>
</table>
Observations and Strategies for Parallel Software Design
Three Design Points

- Terascale Laptop: Uninode-Manycore
- Petascale Deskside: Multinode-Manycore
- Exascale Center: Manynode-Manycore
Basic Concerns: Trends, Manycore

• Stein’s Law: *If a trend cannot continue, it will stop.*
  Herbert Stein, chairman of the Council of Economic Advisers under Nixon and Ford.

• Trends at risk:
  – Power.
  – Single core performance.
  – Node count.
  – Memory size & BW.
  – Concurrency expression in existing Programming Models.
  – Resilience.
Observations

- MPI-Only is not sufficient, except ... much of the time.
- Near-to-medium term:
  - MPI+[OMP|TBB|Pthreads|CUDA|OCL|MPI]
  - Long term, too?
- Concern:
  - Best hybrid performance: 1 MPI rank per UMA core set.
  - UMA core set size growing slowly ➔ Lots of MPI tasks.
- Long-term:
  - Something hierarchical, global in scope.
- Conjecture:
  - Data-intensive apps need non-SPDM model.
  - Will develop new programming model/env.
  - Rest of apps will adopt over time.
  - Time span: 10-20 years.
What Can we Do Right Now?

• Study why MPI was successful.
• Study new parallel landscape.
• Try to cultivate an approach similar to MPI (and others).
MPI Impresssions
MPI: It Hurts So Good

- Observations
  - “assembly language” of parallel computing
  - lowest common denominator
    - portable across architectures
    - upfront effort required
- Concerns

So What Would Life Be Like Without MPI?

“MPI is often considered the “portable assembly language” of parallel computing, …”

Tim Stitts, CSCS
SOS14 Talk
March 2010

Dan Reed, Microsoft
Workshop on the Road Map for the Revitalization of High End Computing
June 16-18, 2003

Looking Forward to a New Age of Large-Scale Parallel Programming and the Demise of MPI
...hopes and dreams of an HPC educator.
3D Stencil in NAS MG

MPI Reality
Tramonto
WJDC
Functional

- New functional.
- Bonded systems.
- 552 lines C code.


How much MPI-specific code?
dft_fill_wjdc.c
MPI-specific code
source_pp_g.f

MFIX: Multiphase Flows with Interphase eXchanges (https://www.mfix.org/)

- MPI-callable, OpenMP-enabled.
- 340 Fortran lines.
- No MPI-specific code.
- Ubiquitous OpenMP markup (red regions).
Reasons for MPI Success?

- Portability? Yes.
- Standardized? Yes.
- Momentum? Yes.
- Separation of many Parallel & Algorithms concerns? Big Yes.

- Once framework in place:
  - Sophisticated physics added as serial code.
  - Ratio of science experts vs. parallel experts: 10:1.

- Key goal for new parallel apps: Preserve this ratio
Single Program Multiple Data (SPMD) 101
2D PDE on Regular Grid (Standard Laplace)
2D PDE on Regular Grid (Helmholtz)

\[-\nabla u - \sigma u = f \quad (\sigma \geq 0)\]
2D PDE on Regular Grid (4\textsuperscript{th} Order Laplace)
More General Mesh and Partitioning
SPMD Patterns for Domain Decomposition

• Halo Exchange:
  – Conceptual.
  – Needed for any partitioning, halo layers.
  – MPI is simply portability layer.
  – Could be replaced by PGAS, one-sided, …

• Collectives:
  – Dot products, norms.

• All other programming:
  – Sequential!!!
Computational Domain Expert Writing MPI Code
Computational Domain Expert Writing Future Parallel Code
Evolving Parallel Programming Model
Parallel Programming Model: Multi-level/Multi-device

- **Inter-node/inter-device** (distributed) parallelism and resource management
- **Node-local control flow** (serial)
- **Intra-node (manycore)** parallelism and resource management
- **Stateless computational kernels** run on each core

**Network of computational nodes**
- Computational node with manycore CPUs and/or GPGPU

**Message Passing**
- Threading

Adapted from slide of H. Carter Edwards
Domain Scientist’s Parallel Palette

- MPI-only (SPMD) apps:
  - Single parallel construct.
  - Simultaneous execution.
  - Parallelism of even the messiest serial code.
- MapReduce:
  - Plug-n-Play data processing framework - 80% Google cycles.
- Pregel: Graph framework (other 20%)
- Next-generation PDE and related applications:
  - Internode:
    - MPI, yes, or something like it.
    - Composed with intranode.
  - Intranode:
    - Much richer palette.
    - More care required from programmer.
- What are the constructs in our new palette?
Obvious Constructs/Concerns

• Parallel for:
  forall (i, j) in domain {…}
  – No loop-carried dependence.
  – Rich loops.
  – Use of shared memory for temporal reuse, efficient device data transfers.

• Parallel reduce:
  forall (i, j) in domain {
    xnew(i, j) = …;
    delx+= abs(xnew(i, j) - xold(i, j));
  }
  – Couple with other computations.
  – Concern for reproducibility.
Other construct: Pipeline

• Sequence of filters.
  • Each filter is:
    – Sequential (grab element ID, enter global assembly) or
    – Parallel (fill element stiffness matrix).

• Filters executed in sequence.

• Programmer’s concern:
  – Determine (conceptually): Can filter execute in parallel?
  – Write filter (serial code).
  – Register it with the pipeline.

• Extensible:
  – New physics feature.
  – New filter added to pipeline.
TBB Pipeline for FE assembly

Serial Filter
- Launch elem-data from mesh
- FE Mesh
  - E1
  - E2
  - E3
  - E4

Parallel Filter
- Compute stiffnesses & loads
- Element-stiffness matrices computed in parallel
- Stiffnesses & loads
  - E1
  - E2
  - E3
  - E4

Several Serial Filters in series
- Assemble rows of stiffness into global matrix
- Assemble rows of stiffness
  - 0,1,2
  - 3,4,5
  - 6,7,8

Global Matrix

Each assembly filter assembles certain rows from a stiffness, then passes it on to the next assembly filter.
Alternative
TBB Pipeline for FE assembly

Launch elem-data from mesh → Compute stiffnesses & loads → Assemble rows of stiffness into global matrix

Serial Filter  Parallel Filter  Parallel Filter

FE Mesh

E1  E2

E3  E4

Element-stiffness matrices computed in parallel

Each parallel call to the assembly filter assembles all rows from the stiffness, using locking to avoid race conflicts with other threads.

Global Matrix
**Base-line FE Assembly Timings**

Problem size: 80x80x80 == 512000 elements, 531441 matrix-rows
The finite-element assembly performs 4096000 matrix-row sum-into operations
(8 per element) and 4096000 vector-entry sum-into operations.

MPI-only, no threads. Linux dual quad-core workstation.

<table>
<thead>
<tr>
<th>Num-procs</th>
<th>Assembly-time Intel 11.1</th>
<th>Assembly-time GCC 4.4.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.80s</td>
<td>1.95s</td>
</tr>
<tr>
<td>4</td>
<td>0.45s</td>
<td>0.50s</td>
</tr>
<tr>
<td>8</td>
<td>0.24s</td>
<td>0.28s</td>
</tr>
</tbody>
</table>
FE Assembly Timings

Problem size: 80x80x80 == 512000 elements, 531441 matrix-rows
The finite-element assembly performs 4096000 matrix-row sum-into operations (8 per element) and 4096000 vector-entry sum-into operations.

No MPI, only threads. Linux dual quad-core workstation.
Other construct: Thread team

- Multiple threads.
- Fast barrier.
- Shared, fast access memory pool.
- Example: Nvidia SM
- X86 more vague, emerging more clearly in future.
• Observe: Iteration count increases with number of subdomains.
• With scalable threaded smoothers (LU, ILU, Gauss-Seidel):
  – Solve with fewer, larger subdomains.
  – Better kernel scaling (threads vs. MPI processes).
  – Better convergence, More robust.
• Exascale Potential: Tiled, pipelined implementation.
• Three efforts:
  – Level-scheduled triangular sweeps (ILU solve, Gauss-Seidel).
  – Decomposition by partitioning
  – Multithreaded direct factorization

Thread Team Advantages

• Qualitatively better algorithm:
  – Threaded triangular solve scales.
  – Fewer MPI ranks means fewer iterations, better robustness.

• Exploits:
  – Shared data.
  – Fast barrier.
  – Data-driven parallelism.
Finite Elements/Volumes/Differences and parallel node constructs

- Parallel for, reduce, pipeline:
  - Sufficient for vast majority of node level computation.
  - Supports:
    - Complex modeling expression.
    - Vanilla parallelism.
  - Must be “stencil-aware” for temporal locality.

- Thread team:
  - Complicated.
  - Requires true parallel algorithm knowledge.
  - Useful in solvers.
Programming Today for Tomorrow’s Machines
Parallel Programming in the small:
- Focus: writing sequential code fragments.
- Programmer skills:
  - 10%: Pattern/framework experts (domain-aware).
  - 90%: Domain experts (pattern-aware)
Languages needed are already here.
- Exception: Large-scale data-intensive graph?
for ((i,j,k) in points/elements on subdomain) {
    compute coefficients for point (i,j,k)
    inject into global matrix
}

Notes:
• User in charge of:
  – Writing physics code.
  – Iteration space traversal.
  – Storage association.
• Pattern/framework/runtime in charge of:
  – SPMD execution.
pipeline <i,j,k> {
    filter(addPhysicsLayer1<i,j,k>);
    ...
    filter(addPhysicsLayer<n<i,j,k>);
    filter(injectIntoGlobalMatrix<i,j,k>);
}

Notes:
• User in charge of:
  – Writing physics code (filter).
  – Registering filter with framework.
• Pattern/framework/runtime in charge of:
  – SPMD execution.
  – Iteration space traversal.
    o Sensitive to temporal locality.
  – Filter execution scheduling.
  – Storage association.
• Better assignment of responsibility (in general).
Quiz (True or False)

1. MPI-only has the best parallel performance.
2. Future parallel applications will not have MPI_Init().
3. Use of “markup”, e.g., OpenMP pragmas, is the least intrusive approach to parallelizing a code.
4. All future programmers will need to write parallel code.
Portable Multi/Manycore Programming
Trilinos/Kokkos Node API
• Goal: Don’t repeat yourself (DRY).
• Every parallel programming environment supports basic patterns: parallel_for, parallel_reduce.
  – OpenMP:
    #pragma omp parallel for
    for (i=0; i<n; ++i) {y[i] += alpha*x[i];}
  – Intel TBB:
    parallel_for(blocked_range<int>(0, n, 100), loopRangeFn(…));
  – CUDA:
    loopBodyFn<<< nBlocks, blockSize >>> (…);
• How can we write code once for all these (and future) environments?
Tpetra and Kokkos

**Tpetra** is an implementation of the Petra Object Model.
- Design is similar to Epetra, with appropriate deviation.
- Fundamental differences:
  - heavily exploits templates
  - utilizes hybrid (distributed + shared) parallelism via Kokkos Node API

**Kokkos** is an API for shared-memory parallel nodes
- Provides parallel_for and parallel_reduce skeletons.
- Support shared memory APIs:
  - ThreadPool Interface (TPI; Carter Edwards’s pthreads Trilinos package)
  - Intel Threading Building Blocks (TBB)
  - NVIDIA CUDA-capable GPUs (via Thrust)
  - OpenMP (*implemented by Radu Popescu/EPFL*)
Generic Shared Memory Node

- Abstract inter-node comm provides DMP support.
- Need some way to portably handle SMP support.
- Goal: allow code, once written, to be run on any parallel node, regardless of architecture.

**Difficulty #1:** Many different memory architectures
- Node may have multiple, disjoint memory spaces.
- Optimal performance may require special memory placement.

**Difficulty #2:** Kernels must be tailored to architecture
- Implementation of optimal kernel will vary between archs
- No universal binary ➔ need for separate compilation paths

**Practical goal:** Cover 80% kernels with generic code.
Kokkos Node API

- **Kokkos** provides two main components:
  - **Kokkos memory model** addresses Difficulty #1
    - Allocation, deallocation and efficient access of memory
    - **compute buffer**: special memory used for parallel computation
    - New: Local Store Pointer and Buffer with size.
  - **Kokkos compute model** addresses Difficulty #2
    - Description of kernels for parallel execution on a node
    - Provides stubs for common parallel work constructs
    - Currently, **parallel for loop** and **parallel reduce**

- Code is developed around a polymorphic Node object.
- Supporting a new platform requires only the implementation of a new **node type**.
Kokkos Memory Model

• A generic node model must at least:
  – support the scenario involving distinct device memory
  – allow efficient memory access under traditional scenarios

• Nodes provide the following memory routines:

  ArrayRCP<T> Node::allocBuffer<T>(size_t sz);
  void Node::copyToBuffer<T>(T * src, ArrayRCP<T> dest);
  void Node::copyFromBuffer<T>(ArrayRCP<T> src, T * dest);
  ArrayRCP<T> Node::viewBuffer<T>(ArrayRCP<T> buff);
  void Node::readyBuffer<T>(ArrayRCP<T> buff);
Kokkos Compute Model

• How to make shared-memory programming generic:
  – Parallel reduction is the intersection of \texttt{dot()} and \texttt{norm1()}.
  – Parallel for loop is the intersection of \texttt{axpy()} and mat-vec.
  – We need a way of fusing kernels with these basic constructs.

• Template meta-programming is the answer.
  – This is the same approach that Intel TBB and Thrust take.
  – Has the effect of requiring that Tpetra objects be templated on Node type.

• Node provides generic parallel constructs, user fills in the rest:

```cpp
template <class WDP>
void Node::parallel_for(
    int beg, int end, WDP workdata);

Work-data pair (WDP) struct provides:
• loop body via WDP::execute(i)
```

```cpp
template <class WDP>
WDP::ReductionType Node::parallel_reduce(
    int beg, int end, WDP workdata);
```

Work-data pair (WDP) struct provides:
• reduction type WDP::ReductionType
• element generation via WDP::generate(i)
• reduction via WDP::reduce(x, y)```
## Example Kernels: axpy() and dot()

<table>
<thead>
<tr>
<th><strong>Template</strong></th>
<th><strong>Example</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template &lt;class WDP&gt;</code></td>
<td><code>void Node::parallel_for(int beg, int end, WDP workdata );</code></td>
</tr>
<tr>
<td><code>template &lt;class T&gt;</code></td>
<td><code>struct AxpyOp {</code></td>
</tr>
<tr>
<td></td>
<td><code>const T * x;</code></td>
</tr>
<tr>
<td></td>
<td><code>T * y;</code></td>
</tr>
<tr>
<td></td>
<td><code>T alpha, beta;</code></td>
</tr>
<tr>
<td></td>
<td><code>void execute(int i)</code></td>
</tr>
<tr>
<td></td>
<td><code>{ y[i] = alpha*x[i] + beta*y[i]; }</code></td>
</tr>
<tr>
<td></td>
<td><code>};</code></td>
</tr>
<tr>
<td><code>AxpyOp&lt;double&gt; op;</code></td>
<td><code>op.x = ...; op.alpha = ...;</code></td>
</tr>
<tr>
<td><code>op.y = ...; op.beta = ...;</code></td>
<td><code>node.parallel_for&lt; AxpyOp&lt;double&gt; &gt; (0, length, op);</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Template</strong></th>
<th><strong>Example</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template &lt;class WDP&gt;</code></td>
<td><code>WDP::ReductionType Node::parallel_reduce(int beg, int end, WDP workdata );</code></td>
</tr>
<tr>
<td><code>template &lt;class T&gt;</code></td>
<td><code>struct DotOp {</code></td>
</tr>
<tr>
<td></td>
<td><code>typedef T ReductionType;</code></td>
</tr>
<tr>
<td></td>
<td><code>const T * x, * y;</code></td>
</tr>
<tr>
<td></td>
<td><code>T identity() { return (T)0; }</code></td>
</tr>
<tr>
<td></td>
<td><code>T generate(int i) { return x[i]*y[i]; }</code></td>
</tr>
<tr>
<td></td>
<td><code>T reduce(T x, T y) { return x + y; }</code></td>
</tr>
<tr>
<td></td>
<td><code>};</code></td>
</tr>
<tr>
<td><code>DotOp&lt;float&gt; op;</code></td>
<td><code>op.x = ...; op.y = ...;</code></td>
</tr>
<tr>
<td><code>float dot;</code></td>
<td><code>dot = node.parallel_reduce&lt; DotOp&lt;float&gt; &gt; (0, length, op);</code></td>
</tr>
</tbody>
</table>
Compile-time Polymorphism

Kokkos functor (e.g., AxpyOp)

+SerialNode

Serial Kernel

+TpiNode

pthread Kernel

+ThrustNode

Thrust Kernel

+FutureNode

Future Kernel
Kokkos Node API vs Native Implementation
Axpy, len=10K, float, int data

Node Type, Prob Size, # Threads

- float Kokkos init time
- float native init time
- float Kokkos sum time
- float native sum time
- int Kokkos init time
- int native init time
- int Kokkos sum time
- int native sum time

Time (sec)
What’s the Big Deal about Vector-Vector Operations?

Examples from OOQP (Gertz, Wright)

\[ y_i \leftarrow y_i + \alpha x_i z_i \quad , i = 1 \ldots n \]

\[ y_i \leftarrow \begin{cases} 
    y^\text{min} - y_i & \text{if } y_i < y^\text{min} \\
    y^\text{max} - y_i & \text{if } y_i > y^\text{max} \\
    0 & \text{if } y^\text{min} \leq y_i \leq y^\text{max} 
\end{cases} , i = 1 \ldots n \]

\[ y_i \leftarrow y_i / x_i \quad , i = 1 \ldots n \]

\[ \alpha \leftarrow \{ \max \alpha : x + \alpha d \geq \beta \} \]

Example from TRICE (Dennis, Heinkenschloss, Vicente)

\[ d_i \leftarrow \begin{cases} 
    (b - u)^{1/2} & \text{if } w_i < 0 \text{ and } b_i < +\infty \\
    1 & \text{if } w_i < 0 \text{ and } b_i = +\infty \\
    (u - a)^{1/2} & \text{if } w_i \geq 0 \text{ and } a_i > -\infty \\
    1 & \text{if } w_i \geq 0 \text{ and } a_i = -\infty 
\end{cases} , i = 1 \ldots n \]

Many different and unusual vector operations are needed by interior point methods for optimization!

Example from IPOPT (Waechter)

\[ x_i \leftarrow \begin{cases} 
    \left( x_i^L + \frac{(x_i^U - x_i^L)}{2} \right) & \text{if } \bar{x}_i > \bar{x}_i^U \\
    \bar{x}_i^L & \text{if } x_i < \bar{x}_i^L \\
    \bar{x}_i^U & \text{if } x_i > \bar{x}_i^U 
\end{cases} , i = 1 \ldots n \]

\[ \bar{x}_i^L = \min \left( x_i^L + \eta \left( x_i^U - x_i^L \right) x_i^L + \delta \right) \]

\[ \bar{x}_i^U = \max \left( x_i^L - \eta \left( x_i^U - x_i^L \right) x_i^U - \delta \right) \]

Currently in MOOCHO:

> 40 vector operations!
Tpetra RTI Components

• Set of stand-alone non-member methods:
  – unary_transform<UOP>(Vector &v, UOP op)
  – binary_transform<BOP>(Vector &v1, const Vector &v2, BOP op)
  – reduce<G>(const Vector &v1, const Vector &v2, G op_glob)
  – binary_pre_transform_reduce<G>( Vector &v1,
      const Vector &v2,
      G op_glob)

• These are non-member methods of Tpetra::RTI which are loosely coupled with Tpetra::MultiVector and Tpetra::Vector.

• Tpetra::RTI also provides Operator-wrappers:
  – class KernelOp<..., Kernel > : Tpetra::Operator<...>
  – class BinaryOp<...,BinaryOp> : Tpetra::Operator<...>
// isn’t this nicer than a bunch of typedefs?
auto &platform = Tpetra::DefaultPlatform::getDefaultPlatform();
auto comm = platform.getComm();
auto node = platform.getNode();

// create Map and some Vector objects
Tpetra::global_size_t numGlobalRows = ...;
auto map = createUniformContigMapWithNode<int, int>(numGlobalRows, comm, node);
const size_t numLocalRows = map->getNodeNumElements();
auto x = Tpetra::createVector<float>(map),
       y = Tpetra::createVector<float>(map),
       z = Tpetra::createVector<double>(map),
       w = Tpetra::createVector<double>(map);

// parallel initialization of x[i] = 1.0 using C++0x lambda function
Tpetra::RTI::unary_transform( *x, [](float xi){return 1.0f;} );
// parallel initialization of y[i] = x[i]
Tpetra::RTI::binary_transform( *y, *x, [](float, float xi) {return xi;} );
// parallel y[i] = x[i] + y[i]
Tpetra::RTI::binary_transform( *y, *x, std::plus<float>() );
// parallel single precision dot(x,y)
fresult = Tpetra::RTI::reduce( *x, *y, reductionGlob<ZeroOp<float>>(
                                 std::multiplies<float>(),
                                 std::plus<float>() ));
Future Node API Trends

• TBB provides very rich pattern-based API.
  – It, or something very much like it, will provide environment for sophisticated parallel patterns.

• Simple patterns: FutureNode may simply be OpenMP.
  – OpenMP handles parallel_for, parallel_reduce fairly well.
  – Deficiencies being addressed.
  – Some evidence it can beat CUDA.

• OpenCL practically unusable?
  – Functionally portable.
  – Performance not.
  – Breaks the DRY principle.
Additional Benefits of Templates
Multiprecision possibilities

- Tpetra is a templated version of the Petra distributed linear algebra model in Trilinos.
  - Objects are templated on the underlying data types:
    \[
    \text{MultiVector}\langle\text{scalar}=\text{double}, \text{localOrdinal}=\text{int}, \text{globalOrdinal}=\text{localOrdinal}\rangle \ldots \\
    \text{CrsMatrix}\langle\text{scalar}=\text{double}, \text{localOrdinal}=\text{int}, \text{globalOrdinal}=\text{localOrdinal}\rangle \ldots \\
    \]
  - Examples:
    \[
    \text{MultiVector}\langle\text{double}, \text{int}, \text{long int}\rangle \ V; \\
    \text{CrsMatrix}\langle\text{float}\rangle \ A; \\
    \]

  Speedup of float over double in Belos linear solver.

<table>
<thead>
<tr>
<th>Scalar</th>
<th>float</th>
<th>double</th>
<th>double-double</th>
<th>quad-double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solve time (s)</td>
<td>2.6</td>
<td>5.3</td>
<td>29.9</td>
<td>76.5</td>
</tr>
<tr>
<td>Accuracy</td>
<td>$10^{-6}$</td>
<td>$10^{-12}$</td>
<td>$10^{-24}$</td>
<td>$10^{-48}$</td>
</tr>
</tbody>
</table>

Arbitrary precision solves using Tpetra and Belos linear solver package
FP Accuracy Analysis: 
FloatShadowDouble Datatype

```cpp
class FloatShadowDouble {

public:
    FloatShadowDouble() {
        f = 0.0f;
        d = 0.0;
    }
    FloatShadowDouble(const FloatShadowDouble & fd) {
        f = fd.f;
        d = fd.d;
    }

    inline FloatShadowDouble operator+=(const FloatShadowDouble & fd) {
        f += fd.f;
        d += fd.d;
        return *this;
    }

    inline std::ostream & operator<<(std::ostream & os, const FloatShadowDouble & fd) {
        os << fd.f << "f " << fd.d << "d"; return os;
    }

};

• Templates enable new analysis capabilities
• Example: Float with “shadow” double.
```
FloatShadowDouble

Sample usage:
#include “FloatShadowDouble.hpp”
Tpetra::Vector<FloatShadowDouble> x, y;
Tpetra::CrsMatrix<FloatShadowDouble> A;
A.apply(x, y); // Single precision, but double results also computed, available

Initial Residual = 455.194f  455.194d
Iteration = 15  Residual = 5.07328f  5.07618d
Iteration = 30  Residual = 0.00147022f  0.00138466d
Iteration = 45  Residual = 5.14891e-06f  2.09624e-06d
Iteration = 60  Residual = 4.03386e-09f  7.91927e-10d
ifndef TPETRA_POWER_METHOD_HPP
#define TPETRA_POWER_METHOD_HPP

#include <Tpetra_Operator.hpp>
#include <Tpetra_Vector.hpp>
#include <Teuchos_ScalarTraits.hpp>

namespace TpetraExamples {

/** \brief Simple power iteration eigensolver for a Tpetra::Operator. */

template <class Scalar, class Ordinal>
Scalar powerMethod(const Teuchos::RCP<const Tpetra::Operator<Scalar,Ordinal>> &A,
                   int niters,
                   typename Teuchos::ScalarTraits<Scalar>::magnitudeType tolerance,
                   bool verbose)
{
    typedef typename Teuchos::ScalarTraits<Scalar>::magnitudeType Magnitude;
    typedef Tpetra::Vector<Scalar,Ordinal> Vector;

    if ( A->getRangeMap() != A->getDomainMap() ) {
        throw std::runtime_error("TpetraExamples::powerMethod(): operator must have domain and range maps that are equivalent.");
    }
}

}
// create three vectors, fill z with random numbers
Teuchos::RCP<Vector> z, q, r;
q = Tpetra::createVector<Scalar>(A->getRangeMap());
r = Tpetra::createVector<Scalar>(A->getRangeMap());
z = Tpetra::createVector<Scalar>(A->getRangeMap());
z->randomize();
//
Scalar lambda = 0.0;
Magnitude normz, residual = 0.0;
// power iteration
for (int iter = 0; iter < niter; ++iter) {
    normz = z->norm2(); // Compute 2-norm of z
    q->scale(1.0/normz, *z); // Set q = z / normz
    A->apply(*q, *z); // Compute z = A*q
    lambda = q->dot(*z); // Approximate maximum eigenvalue: lambda = dot(q, z)
    if (iter % 100 == 0 || iter + 1 == niter) {
        r->update(1.0, *z, -lambda, *q, 0.0); // Compute A*q - lambda*q
        residual = Teuchos::ScalarTraits<Scalar>::magnitude(r->norm2() / lambda);
        if (verbose) {
            std::cout << "Iter = " << iter << " Lambda = " << lambda << " Residual of A*q - lambda*q = " << residual << std::endl;
        }
    }
    if (residual < tolerance) { break; } return lambda; } // end of namespace TpetraExamples
Placement and Migration
Placement and Migration

• MPI:
  – Data/work placement clear.
  – Migration explicit.

• Threading:
  – It’s a mess (IMHO).
  – Some platforms good.
  – Many not.
  – Default is bad (but getting better).
  – Some issues are intrinsic.
Data Placement on NUMA

• Memory Intensive computations: Page placement has huge impact.
• Most systems: First touch (except LWKs).
• Application data objects:
  – Phase 1: Construction phase, e.g., finite element assembly.
  – Phase 2: Use phase, e.g., linear solve.
• Problem: First touch difficult to control in phase 1.
• Idea: Page migration.
Data placement experiments

• MiniApp: HPCCG (Mantevo Project)
• Construct sparse linear system, solve with CG.
• Two modes:
  – Data placed by assembly, not migrated for NUMA
  – Data migrated using parallel access pattern of CG.
• Results on dual socket quad-core Nehalem system.
Weak Scaling Problem

- MPI and conditioned data approach comparable.
- Non-conditioned very poor scaling.
Page Placement summary

• MPI+OpenMP (or any threading approach) is best overall.
• But:
  – Data placement is big issue.
  – Hard to control.
  – Insufficient runtime support.
• Current work:
  – Migrate on next-touch (MONT).
  – Considered in OpenMP (next version).
  – Also being studied in Kitten (Kevin Pedretti).
• Note: This phenomenon especially damaging to OpenMP common usage.
Resilient Algorithms:
A little reliability, please.
The privilege to think of a computer as a reliable, digital machine.

“At 8 nm process technology, it will be harder to tell a 1 from a 0.”

(W. Camp)
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.
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.
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.
Soft Error Futures

- Soft error handling: A legitimate algorithms issue.
- Programming model, runtime environment play role.
Consider GMRES as an example of how soft errors affect correctness

• Basic Steps
  1) Compute Krylov subspace (preconditioned sparse matrix-vector multiplies)
  2) Compute orthonormal basis for Krylov subspace (matrix factorization)
  3) Compute vector yielding minimum residual in subspace (linear least squares)
  4) Map to next iterate in the full space
  5) Repeat until residual is sufficiently small

• More examples in Bronevetsky & Supinski, 2008
Why GMRES?

• Many apps are implicit.
• Most popular (nonsymmetric) linear solver is preconditioned GMRES.
• Only small subset of calculations need to be reliable.
  – GMRES is iterative, but also direct.
Every calculation matters

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.

M. Heroux, M. Hoemmen
FTGMRES Results

Fault-Tolerant GMRES, restarted GMRES, and nonrestarted GMRES
(deterministic faulty SpMVs in inner solves)

- FT–GMRES(50,10)
- GMRES(50), 10 restart cycles
- GMRES(500)

Outer iteration number
Quiz (True or False)

5. DRY is not possible across CPUs and GPUs.
6. Extended precision is too expensive to be useful.
7. Resilience will be built into algorithms.
Bi-Modal: MPI-only and MPI+[X|Y|Z]
Parallel machine with \( p = m \times n \) processors:
- \( m \) = number of nodes.
- \( n \) = number of shared memory processors per node.

Two ways to program:
- Way 1: \( p \) MPI processes.
- Way 2: \( m \) MPI processes with \( n \) threads per MPI process.

New third way:
- “Way 1” in some parts of the execution (the app).
- “Way 2” in others (the solver).
Multicore Scaling: App vs. Solver

Application:
- Scales well (sometimes superlinear)
- MPI-only sufficient.

Solver:
- Scales more poorly.
- Memory system-limited.
- MPI+threads can help.

* Charon Results: Lin & Shadid TLCC Report
MPI-Only + MPI/Threading: $Ax=b$

App passes matrix and vector values to library data classes

All ranks store $A, x, b$ data in memory visible to rank 0

Library solves $Ax=b$ using shared memory algorithms on the node.
MPI Shared Memory Allocation

Idea:
• Shared memory alloc/free functions:
  – MPI_Comm_alloc_mem
  – MPI_Comm_free_mem
• Predefined communicators:
  MPI_COMM_NODE – ranks on node
  MPI_COMM_SOCKET – UMA ranks
  MPI_COMM_NETWORK – inter node
• Status:
  – Available in current development branch of OpenMPI.
  – First “Hello World” Program works.
  – Incorporation into standard still not certain. Need to build case.
  – Next Step: Demonstrate usage with threaded triangular solve.
• Exascale potential:
  – Incremental path to MPI+X.
  – Dial-able SMP scope.

int n = …;
double* values;
MPI_Comm_alloc_mem(  
    MPI_COMM_NODE, // comm (SOCKET works too)  
n*sizeof(double),  // size in bytes  
MPI_INFO_NULL, // placeholder for now  
&values); // Pointer to shared array (out)

// At this point:
// - All ranks on a node/socket have pointer to a shared buffer (values).
// - Can continue in MPI mode (using shared memory algorithms) or
// - Can quiet all but one:
int rank;
MPI_Comm_rank(MPI_COMM_NODE, &rank);
if (rank==0) { // Start threaded code segment, only on rank 0 of the node
 ...
}

MPI_Comm_free_mem(MPI_COMM_NODE, values);

Collaborators: B. Barrett, Brightwell, Wolf - SNL; Vallee, Koenig - ORNL
Algorithms and Meta-Algorithms
Communication-avoiding iterative methods

- **Iterative Solvers:**
  - Dominant cost of many apps (up to 80+% of runtime).

- **Exascale challenges for iterative solvers:**
  - Collectives, synchronization.
  - Memory latency/BW.
  - Not viable on exascale systems in present forms.

- **Communication-avoiding (s-step) iterative solvers:**
  - Idea: Perform $s$ steps in bulk ($s=5$ or more):
    - $s$ times fewer synchronizations.
    - $s$ times fewer data transfers: Better latency/BW.
  - Problem: Numerical accuracy of orthogonalization.

- **New orthogonalization algorithm:**
  - Tall Skinny QR factorization (TSQR).
  - Communicates less and more accurate than previous approaches.
  - Enables reliable, efficient s-step methods.

- **TSQR Implementation:**
  - 2-level parallelism (Inter and intra node).
  - Memory hierarchy optimizations.
  - Flexible node-level scheduling via Intel Threading Building Blocks.
  - Generic scalar data type: supports mixed and extended precision.

**TSQR capability:**
- Critical for exascale solvers.
- Part of the Trilinos scalable multicore capabilities.
- Helps all iterative solvers in Trilinos (available to external libraries, too).
- Staffing: Mark Hoemmen (lead, post-doc, UC-Berkeley), M. Heroux
- Part of Trilinos 10.6 release, Sep 2010.

**Multicore Orthogonalization Performance**

- 1Mx10 matrix, 8-core Nehalem, Intel Compilers

LAPACK – Serial, MGS –Threaded modified Gram-Schmidt

- LAPACK QR
- MGS
- TSQR
Advanced Modeling and Simulation Capabilities: Stability, Uncertainty and Optimization

- Promise: 10-1000 times increase in parallelism (or more).

- Pre-requisite: High-fidelity “forward” solve:
  - Computing families of solutions to similar problems.
  - Differences in results must be meaningful.
## Advanced Capabilities: Readiness and Importance

<table>
<thead>
<tr>
<th>Modeling Area</th>
<th>Sufficient Fidelity?</th>
<th>Other concerns</th>
<th>Advanced capabilities priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seismic</td>
<td>Yes.</td>
<td>None as big.</td>
<td>Top.</td>
</tr>
<tr>
<td><em>S. Collis, C. Ober</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shock &amp; Multiphysics (Alegra)</td>
<td>Yes, but some concerns.</td>
<td>Constitutive models, material responses maturity.</td>
<td>Secondary now. Non-intrusive most attractive.</td>
</tr>
<tr>
<td><em>A. Robinson, C. Ober</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiphysics (Charon)</td>
<td>Reacting flow w/ simple transport, device w/ drift diffusion, …</td>
<td>Higher fidelity, more accurate multiphysics.</td>
<td>Emerging, not top.</td>
</tr>
<tr>
<td><em>J. Shadid</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>K. Pierson</em></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Advanced Capabilities: Other issues

• Non-intrusive algorithms (e.g., Dakota):
  – Task level parallel:
    • A true peta/exa scale problem?
    • Needs a cluster of 1000 tera/peta scale nodes.

• Embedded/intrusive algorithms (e.g., Trilinos):
  – Cost of code refactoring:
    • Non-linear application becomes “subroutine”.
    • Disruptive, pervasive design changes.

• Forward problem fidelity:
  – Not uniformly available.
  – Smoothness issues.
  – Material responses.
Advanced Capabilities:
Derived Requirements

• Large-scale problem presents collections of related subproblems with forward problem sizes.

• Linear Solvers: \( Ax = b \rightarrow AX = B, \ Ax^i = b^i, \ A^i x^i = b^i \)
  – Krylov methods for multiple RHS, related systems.

• Preconditioners:
  – Preconditioners for related systems.

\[ A^i = A_0 + \Delta A^i \]

• Data structures/communication: \( \text{pattern}(A^i) = \text{pattern}(A^j) \)
  – Substantial graph data reuse.
Accelerator-based Scalability Concerns

Global Scope Single Instruction Multiple Thread (SIMT) is too Restrictive
If FLOPS are free, why are we making them cheaper?
Larry Wall:
Easy things should be easy, hard things should be possible.

Why are we making easy things easier and hard things impossible?
Explicit/SIMT vs. Implicit/Recursive Algorithms

Explicit/SIMT:
- Explicit formulations.
- Jacobi prec.

Implicit/Recursive:
- Implicit formulations.
- Multilevel prec.
Problems with Accelerator-based Scalability

- Global SIMT is the only approach that really works well on GPUs, but:
  - Many of our most robust algorithms have no apparent SIMT replacement.
  - Working on it, but a lot to do, and fundamental issues at play.
- SMs might be useful to break SIMT mold, but:
  - Local store is way too small.
  - No market reason to make it bigger.
- Could consider SIMT approaches, but:
  - Broader apps community moving the other way:
    - Climate: Looking at implicit formulations.
    - Embedded UQ: Coupled formulations.
- Accelerator-based apps at risk?
  - Isolation from the broader app trends.
  - Accelerators good, but in combination with strong multicore CPU.
Summary

- Some app targets will change:
  - Advanced modeling and simulation: Gives a better answer.
  - Kernel set changes (including redundant computation).
- Resilience requires an integrated strategy:
  - Most effort at the system/runtime level.
  - C/R (with localization) will continue at the app level.
  - Resilient algorithms will mitigate soft error impact.
  - Use of validation in solution hierarchy can help.
- Building the next generation of parallel applications requires enabling domain scientists:
  - Write sophisticated methods.
  - Do so with serial fragments.
  - Fragments hoisted into scalable, resilient fragment.
- Success of manycore will require breaking out of global SIMT-only.
Quiz (True or False)

1. MPI-only has the best parallel performance.
2. Future parallel applications will not have MPI_Init().
3. Use of “markup”, e.g., OpenMP pragmas, is the least intrusive approach to parallelizing a code.
4. All future programmers will need to write parallel code.
5. DRY is not possible across CPUs and GPUs
6. CUDA and OpenCL may be footnotes in computing history.
7. Extended precision is too expensive to be useful.
8. Resilience will be built into algorithms.
10. Global SIMT is sufficient parallelism for scientific computing.