A User Perspective on Autotuning for Scalable Multicore Systems

Michael A. Heroux
Sandia National Laboratories
Trilinos Contributors

Chris Baker
Ross Bartlett
Pavel Bochev
Paul Boggs
Erik Boman
Cedric Chevalier
Todd Coffey
Eric Cyr
David Day
Karen Devine
Clark Dohrmann
Kelly Fermoyle
David Gay
Mike Heroux
Ulrich Hetmaniuk
Robert Hoekstra
Russell Hooper
Vicki Howle
Jonathan Hu
Joe Kotulski
Rich Lehoucq
Kevin Long
Kurtis Nusbaum
Roger Pawlowski
Brent Perschbacher
Eric Phipps
Lee Ann Riesen
Marzio Sala
Andrew Salinger
Chris Siefert
Bill Spotz
Heidi Thornquist
Ray Tuminaro
Jim Willenbring
Alan Williams

Past Contributors

Jason Cross
Michael Gee
Esteban Guillen
Bob Heaphy
Kris Kampshoff
Ian Karlin
Sarah Knepper
Tammy Kolda
Joe Outzen
Mike Phenow
Paul Sexton
Bob Shuttleworth
Ken Stanley
Background/Motivation
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:
  - Redstorm (XT3), JaguarPF (XT5), Clusters
  - Roadrunner (Cell-based).
  - Multicore nodes.
- Parallel software environments:
  - MPI of course.
  - threads, vectors, CUDA OpenCL, …
  - Combinations of the above.
- User “skins”:
  - C++/C, Python
  - Fortran.
  - Web, CCA.
Evolving Trilinos Solution

- Beyond a “solvers” framework
- Natural expansion of capabilities to satisfy application and research needs

\[ L(u) = f \]

Math. model

\[ L_h(u_h) = f_h \]

Numerical model

\[ u_h = L_h^{-1}f_h \]

Algorithms

- Discretization methods, AD, Mortar methods, …

- Convert to models that can be solved on digital computers
- Find faster and more efficient ways to solve numerical models

Numerical math

Algorithms

discretizations

methods

solvers

core

- Time domain
- Space domain
- Automatic diff.
- Domain dec.
- Linear
- Nonlinear
- Mortar methods
- Petra
- Utilities
- Eigenvalues
- Interfaces
- Optimization
- Load Balancing
Trilinos Statistics

## Trilinos Package Summary


<table>
<thead>
<tr>
<th>Objective</th>
<th>Package(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Discretizations</strong></td>
<td></td>
</tr>
<tr>
<td>Meshing &amp; Spatial</td>
<td>phdMesh, Intrepid, Pamgen, Sundance</td>
</tr>
<tr>
<td>Spatial Discretizations</td>
<td></td>
</tr>
<tr>
<td>Time Integration</td>
<td>Rythmos</td>
</tr>
<tr>
<td><strong>Methods</strong></td>
<td></td>
</tr>
<tr>
<td>Automatic Differentiation</td>
<td>Sacado</td>
</tr>
<tr>
<td>Mortar Methods</td>
<td>Moertel</td>
</tr>
<tr>
<td><strong>Core</strong></td>
<td></td>
</tr>
<tr>
<td>Linear algebra objects</td>
<td>Epetra, Jpetra, Tpetra</td>
</tr>
<tr>
<td>Abstract interfaces</td>
<td>Thyra, Stratimikos, RTOp</td>
</tr>
<tr>
<td>Load Balancing</td>
<td>Zoltan, Isorropia</td>
</tr>
<tr>
<td>“Skins”</td>
<td>PyTrilinos, WebTrilinos, Star-P, ForTrilinos, CTrilinos</td>
</tr>
<tr>
<td>C++ utilities, I/O, thread API</td>
<td>Teuchos, EpetraExt, Kokkos, Triutils, TPI</td>
</tr>
<tr>
<td><strong>Solvers</strong></td>
<td></td>
</tr>
<tr>
<td>Iterative (Krylov) linear solvers</td>
<td>AztecOO, Belos, Komplex</td>
</tr>
<tr>
<td>Direct sparse linear solvers</td>
<td>Amesos</td>
</tr>
<tr>
<td>Direct dense linear solvers</td>
<td>Epetra, Teuchos, Pliris</td>
</tr>
<tr>
<td>Iterative eigenvalue solvers</td>
<td>Anasazi</td>
</tr>
<tr>
<td>ILU-type preconditioners</td>
<td>AztecOO, IFPACK</td>
</tr>
<tr>
<td>Multilevel preconditioners</td>
<td>ML, CLAPS</td>
</tr>
<tr>
<td>Block preconditioners</td>
<td>Meros</td>
</tr>
<tr>
<td>Nonlinear system solvers</td>
<td>NOX, LOCA</td>
</tr>
<tr>
<td>Optimization (SAND)</td>
<td>MOOCHO, Aristos</td>
</tr>
<tr>
<td>Stochastic PDEs</td>
<td>Stokhos</td>
</tr>
</tbody>
</table>
About MPI

- MPI will be the primary inter-node programming model.
- Right ingredients:
  - Portable, ubiquitous.
  - Forced alignment of work/data ownership and transfer.
- Matches architectures:
  - Interconnects of best commercial node parts.
- Key point: Very few people write MPI calls.
  - Domain-specific abstractions.
  - Example: Epetra_MpiDistributor
    - 20 revisions since initial checkin in December 2001.
    - Only three developers made non-trivial changes in 8+ years.
    - No nontrivial changes in 4+ years. No changes in 2+ years.
- New languages:
  - Big fan of Co-Array Fortran (Have been for 15 years: F--).
  - Chapel looks good, esp. domains/distributions.
  - But tough uphill climb.
- Real question: How do we program the node?
Preparing for Manycore
Refactoring for Manycore

- Regardless of node-level programming model:
  - Isolate all computation to stateless functions.
  - Formulate functions so that work granularity can vary.

- Fortran/C:
  - Natural approach.
  - Still requires some change for variable granularity.

- C++:
  - Separate data organization from functions.
  - Can still have computational methods.
Beyond the Forward Problem
Advanced Modeling and Simulation Capabilities: Stability, Uncertainty and Optimization

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

SPDEs:

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

- Size of a single forward problem
## 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: $A^i = A_0 + \Delta A^i$
  - Preconditioners for related systems.

- Data structures/communication: $\text{pattern}(A^i) = \text{pattern}(A^j)$
  - Substantial graph data reuse.
Solvers for Scalable Multicore
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: \( Ax=b \)

**Diagram:**
- **App** (Rank 0, 1, 2, 3)
  - Passes matrix and vector values to **Lib** (Rank 0, 1, 2, 3)
  - All ranks store \( A, x, b \) data in memory visible to rank 0
- **Lib** (Rank 0, 1, 2, 3)
  - Library solves \( Ax=b \) using shared memory algorithms on the node.
- **Mem** (Rank 0, 1, 2, 3)

**Multicore: "PNAS" Layout**
- Sandia National Laboratories
Hybrid Parallelism Opportunities

- **Selective Shared Memory Use:**
  - App: 4096 MPI tasks.
  - Solver: 256 MPI tasks, 16-way threading.

- **Robustness:**
  - 117 iterations (not 153).
  - Eliminates green region.

- **Speed: Threading (carefully used):**
  - Same asymptotic speed as MPI.
  - Faster ramp up: 2X or more.
  - Better load imbalance tolerance.

---

**Bottom line:** Hybrid parallelism promises better:
- Robustness,
- Strong scaling and
- Load balancing.

* Thread Results: H. Carter Edwards*
Hybrid Parallelism: Shared Memory Algorithms

Critical kernel for many scalable preconditioners.

Key Idea: Use sparsity as resource for parallelism.

Solve $Ly = x$. 

$L = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
\ell_{21} & 1 & 0 & 0 & 0 \\
\ell_{31} & 0 & 1 & 0 & 0 \\
0 & 0 & \ell_{43} & 1 & 0 \\
\ell_{51} & 0 & \ell_{53} & 0 & 1 \\
\end{bmatrix}$
Portable Multicore/Manycore Programming
## APIs for Heterogeneous Nodes

(A mess, but some light)

<table>
<thead>
<tr>
<th>Processor</th>
<th>API</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA</td>
<td>CUDA</td>
</tr>
<tr>
<td>AMD/ATI</td>
<td>Brook+</td>
</tr>
<tr>
<td>STI Cell</td>
<td>ALF</td>
</tr>
<tr>
<td>Intel Larrabee</td>
<td>Ct</td>
</tr>
<tr>
<td>Most/All?</td>
<td>Sequoia</td>
</tr>
<tr>
<td>Most</td>
<td>RapidMind (Proprietary)</td>
</tr>
<tr>
<td>Apple/All</td>
<td>OpenCL</td>
</tr>
</tbody>
</table>

Commonality: Fine-grain functional programming.
Our Response: A Library Node Abstraction Layer
Kokkos Node Package

- Trilinos/Kokkos: Trilinos compute node package.
- Generic Node object defines:
  - Memory structures for parallel buffers
  - Parallel computation routines (e.g., `parallel_for`, `parallel_reduce`)
- Kokkos also employs this API to provide local linear algebra for use in Tpetra distributed objects.

Example:
```
Kokkos::LocalCrsMatrix<int,double,NODE> lclA;
lclA.submitEntries(...);  // fill the matrix
Kokkos::SparseMatVec<int,double,NODE> multOp(lclA);
Kokkos::LocalMultiVector<int,double,NODE> lclX(...), lclY(...);
multOp.apply(lclX,lclY);  // apply the matrix operator
```
Node Memory Architecture

- Node defines abstract memory structure.
- Parallel compute buffer is a region of memory suitable for use by parallel kernels.
  - `typename Node::buffer<T>::buffer_t`
- Methods for interacting with buffers:
  - `Node::allocBuffer<T>(int size)`
  - `Node::copyToBuffer<T>(int size, T *src, buffer<T>::buffer_t buf)`
  - `T * Node::viewBuffer<T>(buffer<T>::buffer_T buf)`
  - `Node::releaseView<T>(T *viewptr)`
  - ...
- Necessary abstraction for attached processors w/ distinct memory.
- Node-specific allocation useful for “typical” nodes as well:
  - allows optimal placement for NUMA architectures (AMD)
  - allows node-visible allocation for MPI/SMP hybrid approaches
Node Compute Architecture

- Node currently provides two parallel operations:
  - `parallel_for`
  - `parallel_reduce`
- Encapsulate necessary work/data into work-data struct.
- Template meta-programming does the rest.

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

template <class T, class NODE>
struct AxpyOp{
    NODE::buffer<const T>::buffer_t y;
    NODE::buffer<T>::buffer_t y;
    T alpha, beta;
    void execute(int i)
    { y[i] = alpha*x[i] + beta*y[i]; }
};

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

template <class T, class NODE>
struct DotOp {
    typedef T ReductionType;
    NODE::buffer<const T>::buffer_t x, y;
    T generate(int i) { return x[i]*y[i]; }    
    T reduce(T x, T y){ return x + y; }        
};
```

Sample Code Comparison: dot()

MPI-only:

double dot(int lcl_len,
    double *x,
    double *y)
{
    double lcl = 0.0, gbl;
    for (int i=0; i<lcl_len; ++i)
        lcl += x[i]*y[i];
    MPI_ALLREDUCE(lcl,gbl,…);
    return gbl;
}

Tpetra/Kokkos:

template <class ST, class NODE>
ST Tpetra::Vector<ST>::dot(
    Comm comm,
    Kokkos::LocalVector<ST,NODE> x,
    Kokkos::LocalVector<ST,NODE> y) {
    Scalar lcl, gbl;
    const int n = x.length();
    DotOp<ST,NODE> wdp( x.data(), y.data() );
    Node node = x.getNode();
    lcl = node.parallel_reduce<DotOp>(0,n,wdp);
    reduceAll<ST>(comm,SUM,lcl,&gbl);
    return gbl;
}

- For appropriate choices of Node and Comm, both implementations are equivalent.
- Right hand example is limited only by the available implementations of these classes:
  - can determine whether library was compiled with support for GPU, MPI, etc.
  - can compose different nodes for heterogeneous platforms
Reactions to the Past Two Days

- **Architectures:**
  - Treating everything as manycore (single API).
  - Abstract NodeAPI with specializations.

- **Compiler-based approaches:**
  - Source to source: Good, but easy access helpful.
  - Other approaches: Difficult uphill climb.
  - History of similar efforts: HPF, UPC, CAF, Ti, …
  - Exception: CUDA, but special case.

- **Library based approaches:**
  - OpenCL: Will definitely use (for portability).
  - OSKI: Already use with success, but can improve.
  - PLASMA: Great, will definitely use.
  - Concern: Attention to data layout.

- **Scope of Autotuning Usage:**
  - Node level.
  - Luxury of long runs, repeated usage.
3rd party kernels in Trilinos

- Sparse Graph/Matrix consumers: Access data via abstract layers.
- Default implementations provided (CrsMatrix derives from RowMatrix).
- Optional Adapters: OskiMatrix, HypreMatrix, PetscMatrix.
Desirable Autotuning Sparse Kernel features

- Assemble from matrix fragments: No global object input.
  - Eliminates need for user copy and assembly.
- Option to work from graph only.
  - Most optimizations are structure based.
- Support `getRowView/releaseRowView` functions.
  - Preconditioners need access to matrix coefficients.
  - Eliminates need for persistent second copy.
- Support value update for given pattern.
  - Most real apps solve many problems with same structure.
- Support pattern update (with potential performance hit).
  - Supports graph/mesh adaptivity.
- Record optimization strategy for future use.
- Improve performance, don’t hurt it.
- Integrate easily into existing software environment.
OSKI status on these features

- Assemble from matrix fragments: No global object input.
  - No.
- Option to work from graph only.
  - No.
- Support getRowView/releaseRowView functions.
  - No.
- Support value update for given pattern.
  - Yes.
- Support pattern update (with potential performance hit).
  - No.
- Record optimization strategy for future use.
  - Yes.
- Improve performance, don’t hurt it.
  - Yes.
- Integrate easily into existing software environment.
  - Sort of.
Conclusions

- Autotuning is very important (or I wouldn’t be here).
- App-aware autotuning is very important (also why I’m here).
- Compiler/tool-chain-augmentation approaches challenging:
  - Threshold to effective, portable use is often too high.
  - Many corpses throughout history.
  - Requires a distribution/support plan or will fail.
- Library-based approaches most attractive.
- Harness-based approach possible?
  - Example: CppUnit, Junit style approach?
- About inferior, but parallel algorithms…