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
Building Next Generation Applications & Libraries: 15 Strategies to Consider

1. Prepare for disruptive change
2. Design to the new scalability parameters: thread count and vector lengths
3. Encapsulate all parallelizable functionality into stateless (sequential) functions
4. Organize for vectorization
5. Decide on struct of arrays or array of structs
6. Prefer computation to data storage
7. Consider lower precision storage or computation, or both
8. Consider higher precision storage or computation, or both
9. Exploit data regularity
10. Separate physics indexing from storage indexing
11. Separate definition of array contents from filling of array data structures
12. Create library interfaces, even if you only call your own libraries
13. Look for untapped resources of parallelism
14. Make template meta-programming your friend
15. Build resilience into your software
Three Parallel Computing Design Points

- Terascale Laptop: Uninode-Manycore
- Petascale Deskside: Multinode-Manycore
- Exascale Center: Manynode-Manycore

Common Element
Emerging Architecture Programming Challenges: Overview
Factoring 1K to 1B-Way Parallelism

- Why 1K to 1B?
  - Clock rate: $O(1\text{GHz}) \rightarrow O(10^9)$ ops/sec sequential
  - Terascale: $10^{12}$ ops/sec $\rightarrow O(10^3)$ simultaneous ops
    - 1K parallel intra-node.
  - Petascale: $10^{15}$ ops/sec $\rightarrow O(10^6)$ simultaneous ops
    - 1K-10K parallel intra-node.
    - 100-1K parallel inter-node.
  - Exascale: $10^{18}$ ops/sec $\rightarrow O(10^9)$ simultaneous ops
    - 1K-10K parallel intra-node.
    - 100K-1M parallel inter-node.

- Current nodes:
  - SPARC64™ VIIIfx: 128GF (at 2.2GHz). “K” machine
  - NVIDIA Fermi: 500GF (at 1.1GHz). Tianhe-1A.
  - Sunway BlueLight MPP:
    - 1PF with 8,700 ShenWei SW1600 proc ~ 115GF/proc
Data Movement: Locality

• Locality always important:
  – Caches: CPU
  – L1$ vs L2$ vs DRAM: Order of magnitude latency.

• Newer concern:
  – NUMA affinity.
  – Initial data placement important (unless FLOP rich).
  – Example:
    • 4-socket AMD with dual six-core per socket (48 cores).
    • BW of owner-compute: 120 GB/s.
    • BW of neighbor-compute: 30 GB/s.

• GPUs: Different but related concerns.
Memory Size

- Current “healthy” memory/core:
  - 512 MB/core (e.g. MD computations).
  - 2 GB/core (e.g. Implicit CFD).
- Future:
  - 512 MB/core “luxurious”.


Resilience

• Individual component reliability:
  – Tuned for “acceptable” failure rate.

• Aggregate reliability:
  – Function of all components not failing.
  – May decline.

• Size of data sets may limit usage of standard checkpoint/restart.
Summary of Algorithms Challenge

• Realize node parallelism of $O(1K-10K)$.
• Do so
  – Within a more complicated memory system and
  – With reduced relative memory capacity and
  – With decreasing reliability.
New Trends and Responses

- Increasing data parallelism:
  - Design for vectorization and increasing vector lengths.
  - SIMT a bit more general, but fits under here.
- Increasing core count:
  - Expose task level parallelism.
  - Express task using DAG or similar constructs.
- Reduced memory size:
  - Express algorithms as multi-precision.
  - Compute data vs. store.
  - Data compression techniques.
- Memory architecture complexity:
  - Localize allocation_INITIALIZATION.
  - Favor algorithms with higher compute/communication ratio.
- Resilience: Distinguish what must be reliably computed.
Designing for Trends

• Long-term success must include design for change.
• Algorithms we develop today must adapt to future changes.

• Lesson from Distributed Memory (SPMD):
  – What was the trend? Increasing processor count.
  – Domain decomposition algs matched trend.
    • Design algorithm for $p$ domains.
    • Design software for expanded modeling within a domain.
Summary of Emerging Architecture Challenges & Responses

• Concurrency:
  – The new scalability parameter, includes vectorization.
  – Sample effort: ShyLU

• Data motion:
  – Don’t do it, but if you must, don’t do much.
  – Sample effort: CA-GMRES

• Data generation/use:
  – Generate and retain only that which is necessary.
  – Exploit underlying data patterns.
  – Example: Functor-based data initialization

• Resilience:
  – Formulate resilient algorithms.
  – Distinguish between high-reliability vs. bulk.
  – Example: FTGMRES.
Prepare for Disruptive Change
(And some ways to learn from the past)
Lessons Learned from MPI and SPMD

• SPMD pattern (implemented via MPI) is unmitigated success.

• Approach for future applications:
  – Study why SPMD/MPI was successful.
  – Study new parallel landscape.
  – Try to cultivate a similar approach.
MPI Impresssions
MPI: It Hurts So Good

- Observations
  - "assembly language" of parallel computing
  - lowest common denominator
    - portable across architectures
  - upfront effort required
    - system
    - even libraries

- Conclusions

So What Would Life Be Like Without MPI?

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

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

MPI Reality

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

How much MPI-specific code?
dft_fill_wjdc.c
MPI-specific code
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 sequential 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 replace 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 vectorizable computational kernels run on each core

- Message Passing
- Threading
- Network of computational nodes
- Computational node with manycore CPUs and/or GPGPU

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

• Today: MPI-only (SPMD) apps
  – Single parallel construct.
  – Simultaneous execution.
  – Parallelism of even the messiest serial code.

• 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

Parallel Filter
- Compute stiffnesses & loads

Several Serial Filters in series
- Assemble rows of stiffness into global matrix

Each assembly filter assembles certain rows from a stiffness, then passes it on to the next assembly filter.

Element-stiffness matrices computed in parallel.

Global Matrix

Source: Alan Williams
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

Global Matrix

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

Element-stiffness matrices computed in parallel
### 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.
Summary: FE/FV/FD and parallel node patterns

• 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 deeper parallel algorithm knowledge.
  – Useful in solvers.
Patterns & Frameworks in Other Contexts

• MapReduce:
  – Plug-n-Play data processing framework - 80% Google cycles.
• Pregel: Graph framework (other 20%).
• Ratio of domain to parallel computing experts: 1000s:1.
Programming Today for Tomorrow’s Machines
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.
FE/FV/FD Parallel Programming Tomorrow

```plaintext
pipeline <i,j,k> {
    filter(addPhysicsLayer1<i,j,k>);
    ...
    filter(addPhysicsLayern<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.
    - Sensitive to temporal locality.
  - Filter execution scheduling.
  - Storage association.
- Better assignment of responsibility (in general).
Preparing for Disruptive Change

• Getting onto the new commodity curves requires ubiquitous change:
  – Scalability in thread count, vectorization.
  – Design for sequential physics expression.

• Not unlike move from serial/vector to MPI:
  – ID parallel patterns.
  – Build scalable framework with minimal physics.
  – Mine previous application for functionality.

• Don’t be tempted by incremental approaches:
  – Quick payoff now, but ultimately limiting.
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 `dot()` and `norm1()`
  – **Parallel for loop** is the intersection of `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);

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

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

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

<table>
<thead>
<tr>
<th>Template (\langle\text{class } WDP\rangle)</th>
<th>Template (\langle\text{class } WDP\rangle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{void Node::parallel_for}(int \texttt{beg}, int \texttt{end}, \texttt{WDP workdata});</td>
<td>\texttt{WDP::ReductionType Node::parallel_reduce}(int \texttt{beg}, int \texttt{end}, \texttt{WDP workdata});</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Template (\langle\text{class } T\rangle)</th>
<th>Template (\langle\text{class } T\rangle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{struct AxpyOp { \texttt{const T * x;}}</td>
<td>\texttt{typedef T ReductionType; \texttt{struct DotOp {}}</td>
</tr>
<tr>
<td>\texttt{T * y;}}</td>
<td>\texttt{const T * x, * y;}}</td>
</tr>
<tr>
<td>\texttt{T alpha, beta;}}</td>
<td>\texttt{T identity(); { return (T)0; }}</td>
</tr>
<tr>
<td>\texttt{void execute(int i) { y[i] = alpha<em>x[i] + beta</em>y[i]; }}</td>
<td>\texttt{T generate(int i) { return x[i]*y[i]; }}</td>
</tr>
<tr>
<td>\texttt{}}</td>
<td>\texttt{T reduce(T x, T y) { return x + y; }}</td>
</tr>
<tr>
<td>\texttt{}}</td>
<td>\texttt{}}</td>
</tr>
</tbody>
</table>

\texttt{AxpyOp<double> op; \quad op.x = \ldots; \quad op.alpha = \ldots; \quad op.y = \ldots; \quad op.beta = \ldots; \quad node.parallel_for\langle AxpyOp<double> \rangle (0, length, op);}

\texttt{DotOp<float> op; \quad op.x = \ldots; \quad op.y = \ldots; \quad float dot; \quad dot = node.parallel\_reduce\langle DotOp<float> \rangle (0, length, op);}
Compile-time Polymorphism

Kokkos functor (e.g., AxpyOp)

+SerialNode

+TpiNode

+ThrustNode

+FutureNode

Serial Kernel

pthread Kernel

Thrust Kernel

Future Kernel
What’s the Big Deal about Vector-Vector Operations?

Examples from OOQP (Gertz, Wright)

\( y_i \leftarrow y_i + \alpha x_i z_i \), \( i = 1 \ldots n \)

\[
\begin{cases}
  y_{\min} - y_i & \text{if } y_i < y_{\min} \\
  y_{\max} - y_i & \text{if } y_i > y_{\max} \\
  0 & \text{if } y_{\min} \leq y_i \leq y_{\max}
\end{cases}
\]

\( y_i \leftarrow \frac{y_i}{x_i} \), \( i = 1 \ldots n \)

\( \alpha \leftarrow \{ \max \alpha : x + \alpha d \geq \beta \} \)

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)

\[
\begin{cases}
  x_i^L + \frac{x_i^U - x_i^L}{2} & \text{if } \bar{x}_i^L > \bar{x}_i^U \\
  \bar{x}_i^L & \text{if } x_i < \bar{x}_i^L \\
  \bar{x}_i^U & \text{if } x_i > \bar{x}_i^U
\end{cases}
\]

\( i = 1 \ldots n \)

Currently in MOOCHO: > 40 vector operations!
Tpetra RTI Components

- Set of stand-alone non-member methods:
  - unary_transform\langle UOP\rangle(\text{Vector} &v, \text{UOP} \text{ op})
  - binary_transform\langle BOP\rangle(\text{Vector} &v1, \text{const Vector} &v2, \text{BOP} \text{ op})
  - reduce\langle G\rangle(\text{const Vector} &v1, \text{const Vector} &v2, \text{G op}_\text{glob})
  - binary_pre_transform_reduce\langle G\rangle(\text{Vector} &v1, \text{const Vector} &v2, \text{G op}_\text{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\langle..., Kernel\rangle : Tpetra::Operator\langle...angle
  - class BinaryOp\langle..., BinaryOp\rangle : Tpetra::Operator\langle...angle
Tpetra RTI Example

// 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);
auto 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)
float result = 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:
    
    ```
    MultiVector<scalar=double, local_ordinal=int, global_ordinal=local_ordinal> …
    CrsMatrix<scalar=double, local_ordinal=int, global_ordinal=local_ordinal> …
    ```
  - Examples:
    
    ```
    MultiVector<double, int, long int> V;
    CrsMatrix<float> 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>

<table>
<thead>
<tr>
<th>float</th>
<th>double</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>18 s</td>
<td>26 s</td>
<td>1.42x</td>
</tr>
</tbody>
</table>

Arbitrary precision solves using Tpetra and Belos linear solver package.
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.

FP Accuracy Analysis: FloatShadowDouble Datatype
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 {

/** rief 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;
Teuchos::ScalarTraits<Scalar>::magnitudeType normz, residual = 0.0;

// power iteration
for (int iter = 0; iter < niters; ++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 == niters ) {
        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

<table>
<thead>
<tr>
<th>Description</th>
<th>Iters</th>
<th>FLOPS</th>
<th>Recursive Residual Error</th>
<th>Solution Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Correct Calcs</td>
<td>35</td>
<td>343M</td>
<td>4.6e-15</td>
<td>1.0e-6</td>
</tr>
<tr>
<td>Iter=2, y[1] += 1.0</td>
<td>35</td>
<td>343M</td>
<td>6.7e-15</td>
<td>3.7e+3</td>
</tr>
<tr>
<td>SpMV incorrect Ortho subspace</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q[1][1] += 1.0</td>
<td>N/C</td>
<td>N/A</td>
<td>7.7e-02</td>
<td>5.9e+5</td>
</tr>
<tr>
<td>Non-ortho subspace</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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)
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
Rank 0

App
Rank 1

App
Rank 2

App
Rank 3

Lib
Rank 0

Lib
Rank 1

Lib
Rank 2

Lib
Rank 3

Mem
Rank 0

Mem
Rank 1

Mem
Rank 2

Mem
Rank 3

Multicore: “PNAS” Layout

Lib
Rank 0
Thread 0
Thread 1
Thread 2
Thread 3

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.

```c
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.
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.
# 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>S. Collis, C. Ober</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shock &amp; Multiphysics</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>(Alegra)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A. Robinson, C. Ober</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiphysics</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>(Charon)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J. Shadid</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K. Pierson</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:
  – Substantial graph data reuse.

\[ \text{pattern}(A^i) = \text{pattern}(A^j) \]
An Alternative to Data Passing
Additional Challenge: Math/CS Interface

• Data motion directive: Don’t do it.
• Question:
  – How do we exchange data between app & lib in a manycore world?
  – More generally:
    • How do we reconcile a mathematical view and CS view?
    • Example: *Mathematical* vector vs. *STL* vector.
• Simple example: Odd/Even Merge Sort
• System: 48-core AMD system
  – 4 sockets.
  – Dual 6-core units per socket.
  – 8 NUMA regions.
OEMSort (2 half-steps)

```c
bool sort(int length, int * array) {
    bool isSorted = true;
    for (int j=0; j<2; ++j) {
        for (int i=0; i< (length-j)/2; ++i) {
            int v0 = array[2*i+j];
            int v1 = array[2*i+1+j];
            if (v0>v1) {
                array[2*i+j] = v1;
                array[2*i+1+j] = v0;
                isSorted = false; // Had to swap at least one value
            }
        }
    }
    return isSorted;
}
```
int main(int argc, char *argv[]) {
    int length = atoi(argv[1]);
    int * array = new int[length];  // Allocate array for sorting
    // Initialize the unsorted array, putting values into opposite order
    for (int i=0; i<length; ++i) array[i] = length – i;

    while(!sort(length, array));
    cout << "Sorted array[0:" << print_length <<"] = "; for (int i=0; i<print_length; ++i) cout << array[i] << " " ; cout << endl;

    return 0 ;
}
First attempt at threading
BTW: This is the common approach today

```c
bool sort(int length, int * array) {
    bool isSorted = true;
    for (int j=0; j<2; ++j) {
        #pragma omp parallel
        {
            bool thread_isSorted = true;
        #pragma omp for
            for (int i=0; i< (length-j)/2; ++i) {
                int v0 = array[2*i+j];
                int v1 = array[2*i+1+j];
                if (v0>v1) {
                    array[2*i+j] = v1;
                    array[2*i+1+j] = v0;
                    thread_isSorted = false;
                }
            }
        }  // omp for
    }  // omp parallel
    return isSorted;
}
```

What’s wrong with this?
OEMSort main() is Still Serial!

```c
int main(int argc, char *argv[]) {
    int length = atoi(argv[1]);
    int * array = new int[length]; // Allocate array for sorting
    // Initialize the unsorted array, putting values into opposite order
    for (int i=0; i<length; ++i) array[i] = length – i;

    while(!sort(length, array));
    cout << "Sorted array[0:" << print_length <<"] = ";
    for (int i=0; i<print_length; ++i) cout << array[i] << " ";
    cout << endl;

    return 0 ;
}
```
int main(int argc, char *argv[]) {
    int length = atoi(argv[1]);
    int * array = new int[length];  // Allocate array for sorting
    // Initialize the unsorted array, putting values into opposite order
    #pragma omp parallel for
    for (int i=0; i<length; ++i) array[i] = length – i;

    while(!sort(length, array));
        cout << "Sorted array[0:" << print_length << "] = "; for (int i=0; i<print_length; ++i) cout << array[i] << " "; cout << endl;

        return 0 ;
}
int numCycles = 1; // <= Added feature allows CA-type approaches
int numGhost = numCycles*2;
int * exchangeBuffer = new int[2*numGhost*numThreads];
#pragma omp parallel
{
    int threadNum = omp_get_thread_num();
    int myStart, myStop, numLeftGhost, numRightGhost;
    computeStartStop(threadNum, numThreads, length, numGhost, myStart, myStop, numLeftGhost, numRightGhost, debug);

    int myPaddedLength = myStop-myStart+numLeftGhost+numRightGhost;
    int * myPaddedArray = new int[myPaddedLength];
    int myLength = myStop-myStart;
    int * myArray = myPaddedArray+numLeftGhost;
    int * ptr = myArray;
    for (int i=myStart; i<myStop; ++i) *ptr++ = array[i];
}
class ArrayInitializer {
private:
    int offset;

public:
    ArrayInitializer(int initOffset) : offset(initOffset) {
    }

    int operator () (int i) const {
        return offset - i;
    }
};
OEMSort main() Version 4
Use functor to create thread-private data

ArrayInitializer arrayFunctor(length);
int numCycles = 128; // <= Do 128 steps before global sync.
int numGhost = numCycles*2;
int * exchangeBuffer = new int[2*numGhost*numThreads];
#pragma omp parallel
{
    int threadNum = omp_get_thread_num();
    int myStart, myStop, numLeftGhost, numRightGhost;
    computeStartStop(threadNum, numThreads, length, numGhost, myStart, myStop, numLeftGhost, numRightGhost, debug);

    int myPaddedLength = myStop-myStart+numLeftGhost+numRightGhost;
    int * myPaddedArray = new int[myPaddedLength];
    int myLength = myStop-myStart;
    int * myArray = myPaddedArray+numLeftGhost;
    int * ptr = myArray;
    for (int i=myStart; i<myStop; ++i) *ptr++ = arrayFunctor(i);
# OEM Sort Results

Length = 10M, 1K Steps

48-core AMD Node (4-socket, dual hexcore per socket)

<table>
<thead>
<tr>
<th>Version</th>
<th>Time (sec)</th>
<th>Data Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>29.44</td>
<td>40 MB</td>
</tr>
<tr>
<td>Simple OMP (Common today)</td>
<td>16.09</td>
<td>40 MB</td>
</tr>
<tr>
<td>Conditioned OMP</td>
<td>2.23</td>
<td>80 MB</td>
</tr>
<tr>
<td>Thread Private</td>
<td>0.83</td>
<td>80 MB</td>
</tr>
<tr>
<td>Thread Private (1 level redundancy)</td>
<td>0.73</td>
<td>80+eps MB</td>
</tr>
<tr>
<td>Thread Private with Functor</td>
<td>0.83</td>
<td>40 MB</td>
</tr>
<tr>
<td>Thread Private (1 level redundancy &amp; Functor)</td>
<td>0.73</td>
<td>40+eps MB</td>
</tr>
</tbody>
</table>
Building Next Generation Applications & Libraries: 15 Strategies to Consider

1. Prepare for disruptive change
2. Design to the new scalability parameters: thread count and vector lengths
3. Encapsulate all parallelizable functionality into stateless (sequential) functions
4. Organize for vectorization
5. Decide on struct of arrays or array of structs
6. Prefer computation to data storage
7. Consider lower precision storage or computation, or both
8. Consider higher precision storage or computation, or both
9. Exploit data regularity
10. Separate physics indexing from storage indexing
11. Separate definition of array contents from filling of array data structures
12. Create library interfaces, even if you only call your own libraries
13. Look for untapped resources of parallelism
14. Make template meta-programming your friend
15. Build resilience into your software