Building the Next Generation of Parallel Applications

Michael A. Heroux
Scalable Algorithms Department
Sandia National Laboratories, USA
A Brief Personal Computing History

1988 - 1997

CMIC$ DO ALL VECTOR IF (N .GT. 800)
CMIC$1  SHARED(BETA, N, Y, Z)
CMIC$2  PRIVATE(I)
CDIR$ IVDEP

    do 15 i = 1, n
        z(i) = beta * y(i)
    15    continue
endif

1993 - 2008

#include <mpi.h>
int main(int argc, char *argv[]) {
    // Initialize MPI
    MPI_Init(&argc,&argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

```c++
#include <mpi.h>
#include <omp.h>
int main(int argc, char *argv[]) {
    // Initialize MPI
    MPI_Init(&argc,&argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    ...

    #pragma omp parallel
    {
        double localasum = 0.0;
        #pragma omp for
        for (int j=0; j< MyLength_; j++) localasum += std::abs(from[j]);
        #pragma omp critical
        asum += localasum;
    }
}
```

```c++
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>

thrust::device_vector<int> vd(10, 1);
thrust::host_vector<int> vh(10,1);
```
Quiz (True or False)

1. MPI-only has the best parallel performance.
2. Future parallel applications will not have MPI_Init().
3. All future programmers will need to write parallel code.
4. Use of “markup”, e.g., OpenMP pragmas, is the least intrusive approach to parallelizing a code.
5. DRY is not possible across CPUs and GPUs
6. GPUs are a harbinger of CPU things to come.
7. Checkpoint/Restart will be sufficient for scalable resilience.
8. Resilience will be built into algorithms.
9. MPI-only and MPI+X can coexist in the same application.
10. Kernels will be different in the future.
Basic Exascale 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.

One outcome: Greatly increased interest in OpenMP
Implications

- MPI-Only is not sufficient, except … much of the time.
- Near-to-medium term:
  - MPI+[OMP|TBB|Pthreads|CUDA|OCL|MPI]
  - Long term, too?
- 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: 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.
MPI Impresssions
MPI: It Hurts So Good

- Observations
  - "assembly language" of parallel
  - lowest common denominator
    - portable across applications
    - upfront effort
      - system
      - environment

- Conclusion

So What Would Life Be Like Without MPI?

Tim Stitts, CSCS
SOS14 Talk
March 2010

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

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 Reality
dft_fill_wjdc.c
MPI-specific
code
MFIX: Multiphase Flows with Interphase eXchanges (https://www.mfix.org/)

source_pp_g.f

• 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
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

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.

• Next-generation 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:
  – No loop-carried dependence.
  – Rich loops.

• Parallel reduce:
  – 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.
Other construct: Thread team

- Multiple threads.
- Fast barrier.
- Shared, fast access memory pool.
- Example: Nvidia SM
- X86 more vague, emerging more clearly in future.
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.

• Thread team:
  – Complicated.
  – Requires true parallel algorithm knowledge.
  – Useful in solvers.
Preconditioners for Scalable Multicore Systems

• Observe: Iteration count increases with number of subdomains.
• With scalable threaded triangular solves
  – Solve triangular system on larger subdomains.
  – Reduce number of subdomains.
• Goal:
  – Better kernel scaling (threads vs. MPI processes).
  – Better convergence, More robust.
• Note: App (-solver) scales very well in MPI-only mode.
• Exascale Potential: Tiled, pipelined implementation.

Level Set Triangular Solver

Triangular Solve:
- Critical Kernel
  - MG Smoothers
  - Incomplete IC/ILU
- Naturally Sequential
- Building on classic algorithms:
  - Level Sched:
    - circa 1990.
    - Vectorization.
  - New: Generalized.

\[ \tilde{L} = P L P^T = \begin{bmatrix}
D_1 & D_2 & D_3 \\
A_{2,1} & A_{3,2} & A_{3,3} \\
\vdots & \vdots & \vdots \\
A_{l,1} & A_{l,2} & A_{l,3} & \ldots & D_l
\end{bmatrix} \]

\[
\tilde{x}_1 = D_1^{-1} \tilde{y}_1 \\
\tilde{x}_2 = D_2^{-1} (\tilde{y}_2 - A_{2,1} \tilde{x}_1) \\
\vdots \\
\tilde{x}_l = D_l^{-1} (\tilde{y}_l - A_{l,1} \tilde{x}_1 - \ldots - A_{l,l-1} \tilde{x}_{l-1})
\]

Permutated System
Triangular Solve Results

Passive (PB) vs. Active (AB) Barriers: Critical for Performance

Nehalem

Istanbul

AB + No Thread Affinity (NTA) vs. AB + Thread Affinity (TA) : Also Helpful

Level sets: Trilinos/Isorropia

Core Kernel Timings: Trilinos/Kokkos.
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.
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.
Transition: MPI-only to 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
  - Rank 1
  - Rank 2
  - Rank 3

- **Lib**
  - Rank 0
  - Rank 1
  - Rank 2
  - Rank 3

- **Mem**
  - Rank 0
  - Rank 1
  - Rank 2
  - 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
Resilient Algorithms
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 2008, 2010)
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.
Faults: Hard vs. Soft

• Hard:
  – Program flow interrupted.
  – Majority of faults.
  – Presently handled by (global) checkpoint/restart.
  – Numerous papers on alternatives.

• Soft:
  – Program flow continues.
  – Minor perturbations in data state:
    • Incorrect address lookup (but still in user scope).
    • Incorrect FP value.
Algorithm-Based (Hard) Fault Tolerance

• Numerous approaches.
• Most common strategies:
  – Meta data:
    • Embed meta data into user-defined data structures.
    • Manage fault detection, recovery manually.
  – Algorithm results validation:
    • Use known algorithm properties.
    • Validate computed to known (e.g., residual check).
• Note: A lack of app awareness.
Madame President,

although there was some rough weather, our fault tolerant linear solver worked and I have returned with our portion of the linear solution.

Thank you, but we lost nonlinear state and cannot use your results.

Common Approach to FT (Diplomacy Analogy)

We have linearized our portion of the nonlinear problem and would like you to negotiate a global linear solution with the other processors.

Yes, Madame President. I will return with our portion of the global linear solution, ASAP.

Madame President, although there was some rough weather, our fault tolerant linear solver worked and I have returned with our portion of the linear solution.

Thank you, we recovered nonlinear state, the linear solution is expensive. We can use your results.
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: Dim 21K, Nz 923K.
• ILUT/GMRES
• Correct computation 35 Iters: 343M FLOPS

Two examples of a single bad floating point op

<table>
<thead>
<tr>
<th>Description</th>
<th>Iterations</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>
One possible approach is transactional computation

- Database transactions: atomic
- Transactional memory: atomic memory operation
- Transactional computation:
  - Designated sensitive computation region (orthogonalization step in GMRES)
  - Guarantee accurate computation or notify user.
Needs to be coupled with SW-enabled guaranteed data regions

• User-designated reliable data region
• Extra protection to improve reliable data storage and transfer
• Examples
  – Original input data (needed for verification)
  – Linear solver: $A, x, b$
  – Orthogonal vectors for GMRES
• OpenMP pragma-enabled?
Goal

- Algorithms well-conditioned wrt soft failure.

Now:
  - Single soft error produces erroneous results.

Goal:
  - Correct results always.
  - Cost increase proportional to number of soft errors.

Note: These are just two approaches to ABFT.
Software Development and Delivery
Compile-time Polymorphism
Templates and Sanity upon a shifting foundation

Software delivery:
• Essential Activity

How can we:
• Implement mixed precision algorithms?
• Implement generic fine-grain parallelism?
• Support hybrid CPU/GPU computations?
• Support extended precision?
• Explore redundant computations?
• Prepare for both exascale “swim lanes”?

C++ templates only sane way:
• Moving to completely templated Trilinos libraries.
• Other important benefits.
• A usable stack exists now in Trilinos.

Template Benefits:
– Compile time polymorphism.
– True generic programming.
– No runtime performance hit.
– Strong typing for mixed precision.
– Support for extended precision.
– Many more…

Template Drawbacks:
– Huge compile-time performance hit:
  • But good use of multicore :)
  • Eliminated for common data types.
– Complex notation:
  • Esp. for Fortran & C programmers).
  • Can insulate to some extent.

“Are C++ templates safe? No, but they are good.”
## Solver Software Stack

**Phase I packages: SPMD, int/double**

<table>
<thead>
<tr>
<th>Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unconstrained:</td>
</tr>
<tr>
<td>Constrained:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bifurcation Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given nonlinear operator $F(x, u) \in \mathbb{R}^{n+m}$</td>
</tr>
<tr>
<td>For $F(x, u) = 0$ find space $u \in U \ni \frac{\partial F}{\partial x}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transient Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAEs/ODEs:</td>
</tr>
<tr>
<td>Solve $f(\dot{x}(t), x(t), t) = 0$</td>
</tr>
<tr>
<td>$t \in [0, T], x(0) = x_0, \dot{x}(0) = x'_0$</td>
</tr>
<tr>
<td>for $x(t) \in \mathbb{R}^n, t \in [0, T]$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nonlinear Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given nonlinear operator $F(x) \in \mathbb{R}^m \to \mathbb{R}^n$</td>
</tr>
<tr>
<td>Solve $F(x) = 0, x \in \mathbb{R}^n$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Linear Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Equations:</td>
</tr>
<tr>
<td>Given Linear Ops (Matrices) $A, B \in \mathbb{R}^{m \times n}$</td>
</tr>
<tr>
<td>Solve $Ax = b$ for $x \in \mathbb{R}^n$</td>
</tr>
</tbody>
</table>

| Eigen Problems: |
| Solve $A\nu = \lambda B\nu$ for (all) $\nu \in \mathbb{R}^n, \lambda \in \mathbb{R}$ |

| Distributed Linear Algebra |
| Matrix/Graph Equations: |
| Compute $y = Ax; A = A(G); A \in \mathbb{R}^{m \times n}, G \in \mathbb{S}^{m \times n}$ |

| Vector Problems: |
| Compute $y = \alpha x + \beta w; \alpha = \langle x, y \rangle; x, y \in \mathbb{R}^n$ |

**Phase II packages: Templated**

### Sensitivities (Automatic Differentiation: Sacado)

- MOOCHO
- LOCA
- Rythmos
- NOX
- Anasazi
- Ifpack, ML, etc...
- AztecOO
- Epetra
- Teuchos

---

57
### Solver Software Stack

<table>
<thead>
<tr>
<th>Phase I packages</th>
<th>Phase II packages</th>
<th>Phase III packages: Manycore*, templated</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Optimization</strong></td>
<td><strong>Bifurcation Analysis</strong></td>
<td><strong>Transients Problems</strong></td>
</tr>
<tr>
<td>Unconstrained:</td>
<td>Find ( u \in \mathbb{R}^n ) that minimizes ( g(u) )</td>
<td>Solve ( f(x(t), x(t), t) = 0 )</td>
</tr>
<tr>
<td>Constrained:</td>
<td>Find ( x \in \mathbb{R}^m ) and ( u \in \mathbb{R}^n ) that minimizes ( g(x, u) ) s.t. ( f(x, u) = 0 )</td>
<td>( t \in [0, T] ), ( x(0) = x_0 ), ( \dot{x}(0) = x'_0 ) for ( x(t) \in \mathbb{R}^n ), ( t \in [0, T] )</td>
</tr>
<tr>
<td></td>
<td>Given nonlinear operator ( F(x, u) \in \mathbb{R}^{n+m} )</td>
<td>Given nonlinear operator ( F(x) \in \mathbb{R}^m \rightarrow \mathbb{R}^n )</td>
</tr>
<tr>
<td></td>
<td>For ( F(x, u) = 0 ) find space ( u \in U \epsilon \frac{\partial F}{\partial x} )</td>
<td>Solve ( F(x) = 0 ) ( x \in \mathbb{R}^n )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Linear Problems</th>
<th>Linear Equations:</th>
<th>Eigen Problems:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Distributed Linear Algebra</strong></td>
<td><strong>Matrix/Graph Equations:</strong></td>
<td><strong>Vector Problems:</strong></td>
</tr>
<tr>
<td>Compute ( y = Ax; A = A(G); A \in \mathbb{R}^{m \times n}, G \in \mathbb{S}^{m \times n} )</td>
<td>Compute ( y = \alpha x + \beta w; \alpha = \langle x, y \rangle; x, y \in \mathbb{R}^n )</td>
<td></td>
</tr>
<tr>
<td><strong>Linear Equations:</strong></td>
<td><strong>Eigen Problems:</strong></td>
<td></td>
</tr>
<tr>
<td>Solve ( Ax = b ) for ( x \in \mathbb{R}^n )</td>
<td>Solve ( Av = \lambda Bv ) for (all) ( v \in \mathbb{R}^n ), ( \lambda \in \mathbb{C} )</td>
<td></td>
</tr>
</tbody>
</table>

| MOOCHO | LOCA T-LOCA | Rythmos | NOX T-NOX | Anasazi | AztecOO | Belos* | T-Ifpack*, T-ML*, etc | Epetra | Tpetra* | Kokkos | Teuchos |
Trilinos/Kokkos Node API
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
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:

  ```cpp
  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:

```
 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()}

\begin{verbatim}
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);

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

AxpyOp<double> op;
op.x = ...; op.alpha = ...;
op.y = ...; op.beta = ...
node.parallel_for< AxpyOp<double> > (0, length, op);

template <class T>
struct DotOp {
    typedef T ReductionType;
    const T * x, * y;
    T identity() { return (T)0; }
    T generate(int i) { return x[i]*y[i]; }
    T reduce(T x, T y) { return x + y; }
};

DotOp<float> op;
op.x = ...; op.y = ...
float dot;
dot = node.parallel_reduce< DotOp<float> > (0, length, op);
\end{verbatim}
Hybrid CPU/GPU Computing
Hybrid Timings (Tpetra)

- Tests of a simple iterations:
  - power method: one sparse mat-vec, two vector operations
  - conjugate gradient: one sparse mat-vec, five vector operations

- DNVS/x104 from UF Sparse Matrix Collection (100K rows, 9M entries)

- NCCS/ORNL Lens node includes:
  - one NVIDIA Tesla C1060
  - one NVIDIA 8800 GTX
  - Four AMD quad-core CPUs

- Results are very tentative!
  - suboptimal GPU traffic
  - bad format/kernel for GPU
  - bad data placement for threads

<table>
<thead>
<tr>
<th>Node</th>
<th>PM (mflop/s)</th>
<th>CG (mflop/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single thread</td>
<td>140</td>
<td>614</td>
</tr>
<tr>
<td>8800 GPU</td>
<td>1,172</td>
<td>1,222</td>
</tr>
<tr>
<td>Tesla GPU</td>
<td>1,475</td>
<td>1,531</td>
</tr>
<tr>
<td>Tesla + 8800</td>
<td>981</td>
<td>1,025</td>
</tr>
<tr>
<td>16 threads</td>
<td>816</td>
<td>1,376</td>
</tr>
<tr>
<td><strong>1 node</strong> 15 threads + Tesla</td>
<td>867</td>
<td>1,731</td>
</tr>
<tr>
<td><strong>2 nodes</strong> 15 threads + Tesla</td>
<td><strong>1,677</strong></td>
<td><strong>2,102</strong></td>
</tr>
</tbody>
</table>
New Core Linear Algebra Needs
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>\textit{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) \textit{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) \textit{J. Shadid}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{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.

\[ \text{pattern}(A^i) = \text{pattern}(A^j) \]
Summary

• App targets will change:
  – Advanced modeling and simulation: Gives a better answer.
  – Kernel set changes.

• 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.

• 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.
Quiz (True or False)

1. MPI-only has the best parallel performance.
2. Future parallel applications will not have MPI_Init().
3. All future programmers will need to write parallel code.
4. Use of “markup”, e.g., OpenMP pragmas, is the least intrusive approach to parallelizing a code.
5. DRY is not possible across CPUs and GPUs
6. GPUs are a harbinger of CPU things to come.
7. Checkpoint/Restart will be sufficient for scalable resilience.
8. Resilience will be built into algorithms.
9. MPI-only and MPI+X can coexist in the same application.
10. Kernels will be different in the future.