R&D in Trilinos for Emerging Parallel Systems

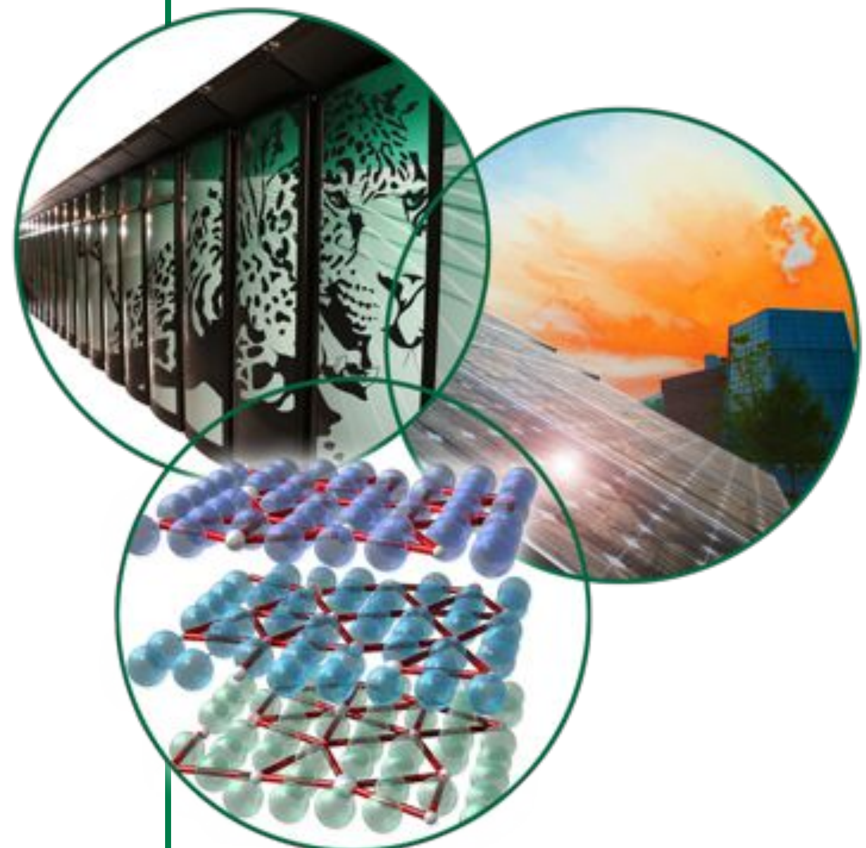
Christopher Baker

Computational Engineering and Energy Studies

Oak Ridge National Laboratory, USA

Accelerating Computational Science Symposium 2012 (ACSS 2012)

March 29-30, 2012, Washington, D.C.



U.S. DEPARTMENT OF ENERGY

OAK RIDGE NATIONAL LABORATORY

MANAGED BY UT-BATTELLE FOR THE DEPARTMENT OF ENERGY

Related Developers

- Oak Ridge National Laboratory
  – **Chris Baker**
  – **Ross Bartlett**
- Sandia National Laboratories
  – **Mike Heroux**
  – **Mark Hoemmen**
  – **Alan Williams**
  – **Carter Edwards**
- École Polytechnique Fédérale de Lausanne
  – **Radu Popescu**

Challenges of Heterogeneous Many-Core

- MPI-only not enough
  - Need to port: it doesn't work for accelerators.
  - Inefficient: it misses a lot of shared-memory benefits.
- MPI+ can entail significant work
  - We want to minimize the number of code bases.
  - We want to minimize the effort to add a new code base.
- Programming language issues
  - Many APIs require a particular language.
  - Developers resent being told what language to use.
- Lib/User interface issues
  - Extending the library should not introduce serial bottlenecks.
  - Shouldn't require users to be shared-memory API experts.

Algorithm R&D Directions

- Current focus on MPI+X, where X is any/all reasonable industry standard.
  - Distributed memories ➔ distributed memory programming
- New efforts on efficient kernels and problem setup
- Support for embedded UQ and optimization
- Krylov solvers for emerging problems (e.g., UQ):
  - Interacting subspace methods for simultaneous/sequenced systems (incl. block and recycling methods)
  - Communication avoiding methods for single RHS systems
  - Numerically fault-resilient solvers, e.g., FT-GMRES
- Mixed/multi-precision solvers and preconditioners

Software R&D Directions

- Templated C++ code
  - Templating data allows more efficient use of cache and bandwidth.
  - Templating data expands capability (e.g., integer limit, `complex`)
- Generic shared memory parallel node
  - Template metaprogramming shared memory parallel node API
  - Static translation layer to, e.g., TBB, Thrust, OpenMP
- Hybrid programming model
  - Hybrid programming skeletons to support most common patterns
  - Expose models for high-productivity, performance-portable apps
- Non-intrusive modification of structures and algorithms
  - Expose the shared-memory parallel node API to apps
  - Static polymorphism to support node-optimized kernels

Example: A Benefit of Generic Kernels

- Tpetra distributed linear algebra library provides a set of methods for executing user kernels on vectors, e.g.:
  - `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)`
- Fine level for expressiveness, coarser levels for convenience.

```
// single-prec dot() with double-prec accumulator via custom kernel
result = reduce( *x, *y,  myDotProductKernel<float,double>() );
// Or a composite adaptor and STL functors
result = reduce( *x, *y, reductionGlob<ZeroOp<double>>(
                                       std::multiplies<float>(),
                                       std::plus<double>()) );
// Or using inline functors via C++11 lambda functions
result = reduce( *x, *y, reductionGlob<ZeroOp<double>>(
                                 [](float x, float y)  {return x*y;} ,
                                 [](double a, double b){return a+b;} );
// Or using a convenience macro to generate all of that
result = REDUCE2( x, y,  x*y, ZeroOp<float>,  std::plus<double>() );
```

Example: Inline, Templated MPI+ CG

- The API supports rapid prototyping of algorithms
  - Fun game: Find the MPI or threading!

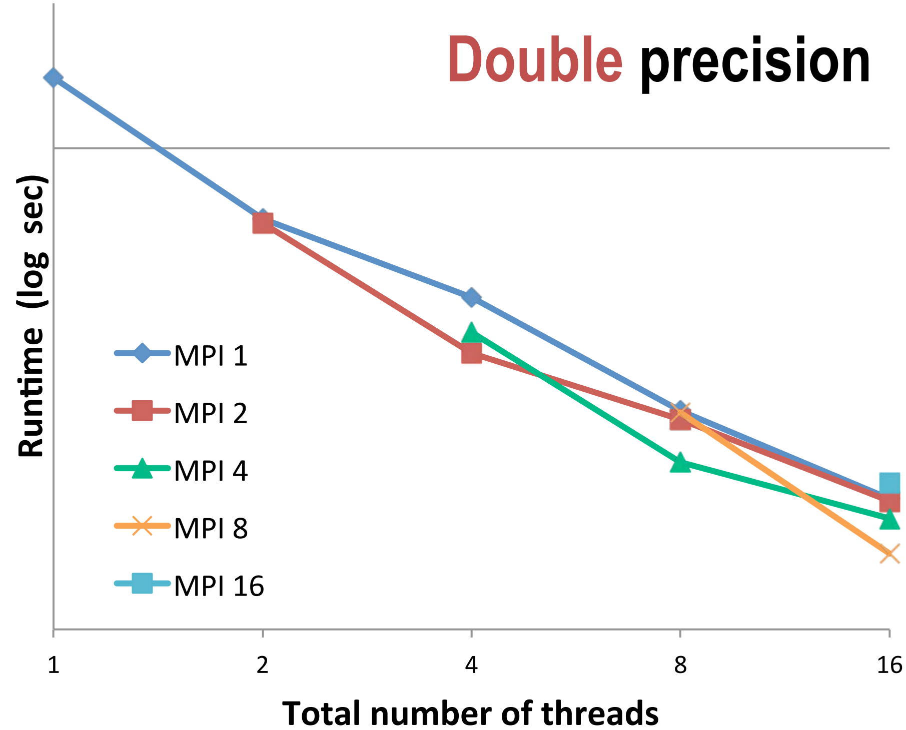
```
for (k=0; k<numIters; ++k) {
  A->apply( *p, *Ap );                              // Ap = A*p
  S pAp = REDUCE2(
             p, Ap,
             p*Ap, ZeroOp<S>, plus<S>() );          // p'*Ap
  const S alpha = rr / pAp;                         // alpha = r′*r/p′*Ap
  BINARY_TRANSFORM( x, p,
                    x + alpha*p );                  // x = x + alpha*p
  S rrold = rr;
  rr = BINARY_PRETRANSFORM_REDUCE(
             r, Ap,                                 // fused kernels
             r - alpha*Ap,                          //   r - alpha*Ap
             r*r, ZeroOp<S>, plus<S>() );           //   sum r'*r
  const S beta = rr / rrold;                        // beta = r′*r/old(r′*r)
  BINARY_TRANSFORM( p, r,
                    r + beta*p);                    // p = z + beta*p
}
```

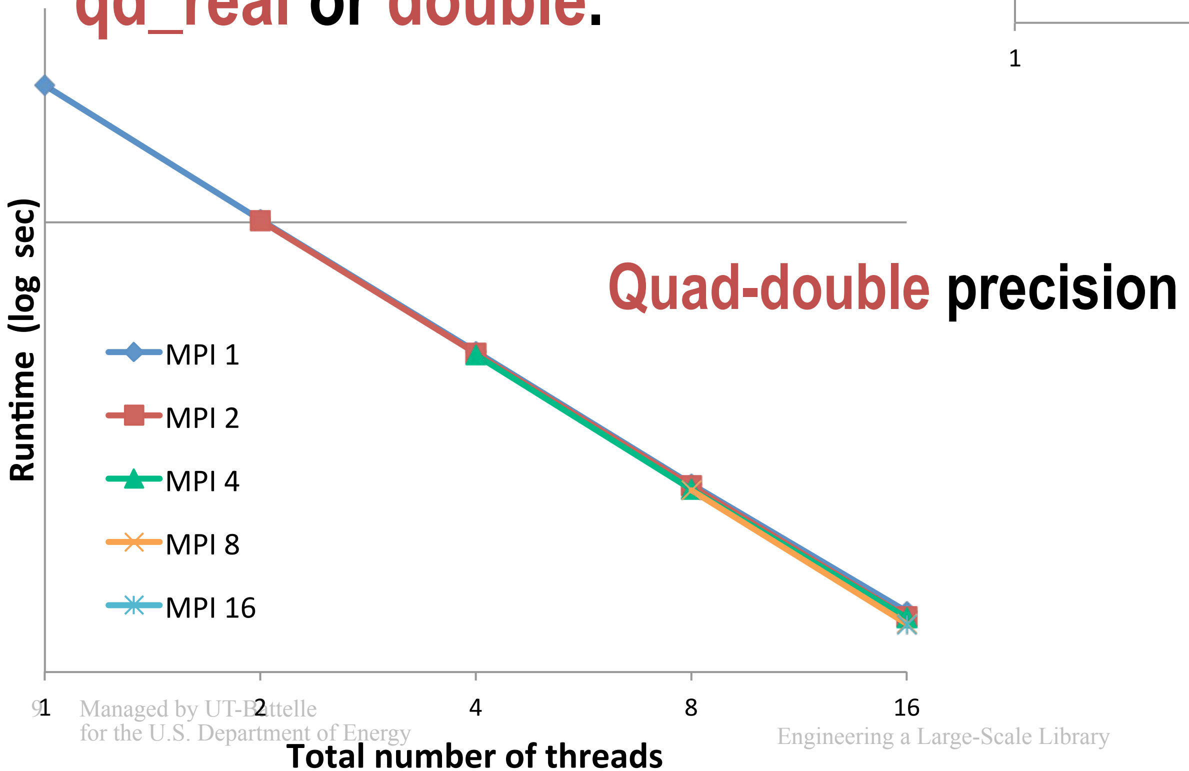
Example: Recursive Multi-Prec. FPCG

```
for (k=0; k<numIters; ++k) {
  A->apply(*p,*Ap);                                    // Ap = A*p
  T pAp = REDUCE2( p, Ap,
                   p*Ap, ZeroOp<T>, plus<T>());        // p'*Ap
  const T alpha = zr / pAp;
  BINARY_TRANSFORM( x,    p,    x + alpha*p  );        // x = x + alpha*p
  BINARY_TRANSFORM( rold, r,    r );                   // rold = r
  T rr = BINARY_PRETRANSFORM_REDUCE(
                    r, Ap,                             // fused:
                    r - alpha*Ap,                      // r - alpha*Ap
                    r*r, ZeroOp<T>, plus<T>() );       // sum r'*r
  recursiveFPCG<TS::next,LO,GO,Node>(out,db_T2);       // recurse
  auto plusTT = make_pair_op<T,T>(plus<T>());
  pair<T,T> both = REDUCE3( z, r, rold,                // fused:
                 make_pair( z*r, z*rold ),             // z'*r, z'*r_old
                     ZeroPTT, plusTT );
  const T beta = (both.first - both.second) / zr;
  zr = both.first;
  BINARY_TRANSFORM( p, z,    z + beta*p );             // p = z + beta*p
}
```

Example: Simple CG

- MPI+TBB parallel node
- #threads = #mpi x #tbb
- Single codebase, solver instantiated on either qd_real or double.





Example: Recursive Multi-Prec. FPCG

- Problem: Oberwolfach/gyro, `N=17K`, `NNZ=1M`
- Single solver code-base, templated on `qd_real`/`dd_real`/`double`

```
TBBNode initializing with numThreads == 2
TBBNode initializing with numThreads == 2
Running test with Node==Kokkos::TBBNode on rank 0/2
Beginning recursiveFPCG<qd_real>
   Beginning recursiveFPCG<dd_real>
     |res|/|res_0|: 1.269903e-14
     |res|/|res_0|: 3.196573e-24
     |res|/|res_0|: 6.208795e-35
   Convergence detected!
   Leaving recursiveFPCG<dd_real> after 2 iterations
|res|/|res_0|: 2.704682e-32
   Beginning recursiveFPCG<dd_real>
     |res|/|res_0|: 4.531185e-09
     |res|/|res_0|: 6.341084e-20
     |res|/|res_0|: 8.326745e-31
   Convergence detected!
   Leaving recursiveFPCG<dd_real> after 2 itera
|res|/|res_0|: 3.661388e-58
Leaving recursiveFPCG<qd_real> after 2 iterations.
```

2 iters. of `qd_real`, 4 iters. of `dd_real`, 99.9% of time spent in `double` iters.

Solved to nearly 60 digits

Example: Problems with Generic Kernels

- Generic kernels are not always successful:
  - e.g., CRS mat-vec on GPUs is typically sub-optimal
- Different sparse mat-vec kernels use different data structure.
- We want vendors/researchers to substitute their own kernels.
- One solution treats the kernel as a first-class object.
  - Template param. dictating data structure and mat-vec kernel
- Another specializes a class for a unique platform, non-intrusively, e.g., `CrsMatrix< double, XK6Node >`
- These mirror the solutions undertaken by many others:
  - static polymorphism via #ifdefs
  - runtime polymorphism, often object-oriented

Closing Comments

- What about issues reliability and resilience?
  - How much can we handle via analytically robust algorithms?
- What is the proper balance of generic kernels and architecture specific kernels?
- We are current focused on leveraging generic programming and abstract interfaces for flexible implementations and easy composition for larger problems.
- The goal is to maximize programmer efficiency (both library and app) without significant performance sacrifices.