Programming Models for Sustained Petaflops

David E. Bernholdt, Wael R. Elwasif, Robert J. Harrison, and Aniruddha G. Shet

Oak Ridge National Laboratory

Oak Ridge National Laboratory is managed by UT-Battelle, LLC for the US Dept. of Energy under contract DE-AC-05-00OR22725.

Today: Approaching a Peak Petaflop

- Hardware characteristics (~100-300 Teraflops peak)
  - 10,000-100,000 processors
  - 2-4 cores per processor
  - Homogeneous processor environment
- Application characteristics
  - Scaling up problem size/resolution
  - Increasing physical fidelity/model complexity
  - Early explorations of coupled simulation
- Dominant programming model
  - Sequential language (Fortran)
  - 2-sided messaging library (MPI)
  - threads (OpenMP)
  - Age: ~30 years

Tomorrow: Sustained Petaflops and Beyond

- Hardware characteristics (10-100 petaflops peak)
  - 100,000-1,000,000 processors
  - 100-1,000 cores per processor
  - Heterogeneous processor environment *may be* common
    - *Example:* LANL Roadrunner: Opteron, PowerPC, Cell
    - Also GP-GPUs, integrated GPUs, FPGAs, etc.
- Application characteristics
  - Scaling up problem size/resolution leveling off
  - Increasing physical fidelity/model complexity
  - Serious coupled simulation
  - Serious algorithmic scaling challenges
  - Increase in adaptive representations, irregular computations
  - Increase in community-wide code sharing
- Dominant programming model
  - ???

Fortran+MPI+OpenMP Forever?

- *I hope not!*
  - *The "assembly language" of parallel programming*
- Pushes too much complexity onto the programmer
  - Explicit/local management of complex distributed data structures
  - Coordination among O(10^6-10^8) communication endpoints, more threads
- Parallelism bolted on to sequential language makes program expression, comprehension, tuning, etc. harder
- Fortran is still relatively low level
  - Expresses basic computations, but not domain abstractions
  - F2003 better, but not widely used (implemented?) yet
- MPI emphasizes the mechanics of data movement, obscures scientific problem and abstractions
- OpenMP is mostly about loop-level parallelism, but much more is needed

The Nature of Scientific Programming has Changed

- It used to be that straightforward **FOR**mula **TRAN**slation was sufficient
- Now, it involves the manipulation of complex, hierarchical abstractions in environments requiring huge levels of concurrency
- The change has been incremental, and the tools are still just doing formula translation
  - Boiled frog analogy
- Programming models must make a leap!

What's the Alternative?

- Higher-level core language
- Integrated concurrency
- Global view of data
- PGAS (Partitioned Global Address Space)?
  - Co-Array Fortran (CAF)
  - Unified Parallel C (UPC)
  - Titanium
- HPCS (High Productivity Computing Systems)
  - Chapel (Cray)
  - Fortress (Sun)
  - X10 (IBM)
- Domain-specific languages
  - What to build upon?

PGAS is Not Enough

- Simplest possible extension to {Fortran,C,Java} to provide basic parallelism
- Base languages are not high enough level
- Each language has a variety of problems and limitations
  - CAF doesn't really provide a global *view* of data
  - UPC only understands distributed arrays in 1d, hard to optimize
  - Titanium is a dialect of Java
- A step in a positive direction
  - But not a large enough step
  - Possibly useful in a transitional sense (more later)

HPCS Languages: Core Features

- Rich array data types
- Strongly typed
- Object oriented model
  - Distinction between reference and value types
- Generic programming
- Strongly library oriented
- Extensible language model (more later)

Productivity Features

- Index sets/regions for arrays, etc.
  - “Array language” (Chapel, X10)
- Safe(r) and more powerful language constructs
  - Atomic sections vs locks
  - Sync variables and futures
  - Clocks (X10)
- Type inference
- Leverage IDE environments
- Units and dimensions (Fortress)
- Component management, testing, contracts (Fortress)
- Math/science-based presentation (Fortress)

Concurrency

- Not SPMD!
  - Initially single thread of control, parallelism through language constructs
- True global view of memory, one-sided access model
- Support for both task and data parallelism
- “Threads” grouped by “memory locality”
  - Explicitly two level (Chapel, X10), or hierarchical (Fortress)
- Rich distributed array capability
  - Programmer-provided distribution details
- Parallel loops
- “Generator” concept used widely for loops, distributions
- Futures
  - Local and remote

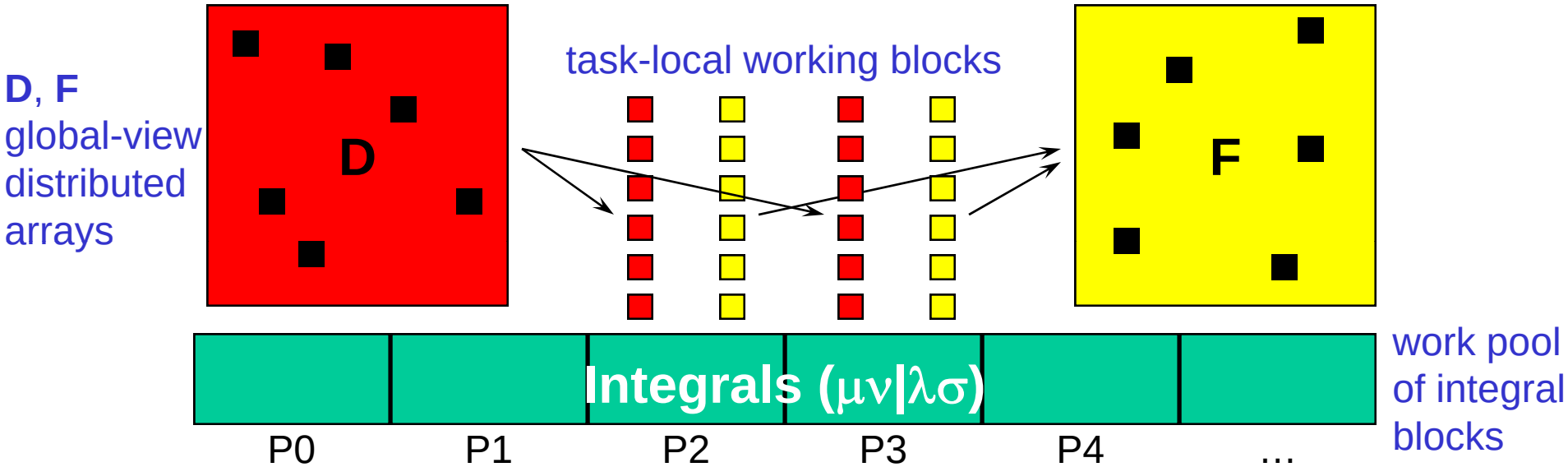
Fock Matrix Construction (Quantum Chemistry)

$$\mathbf{F}_{\mu\nu} \leftarrow \mathbf{D}_{\lambda\sigma} \{ 2\,(\mu\nu|\lambda\sigma) - (\mu\lambda|\nu\sigma) \}$$

- Indices μ, ν, λ, σ represent basis functions (*N*)
- **F** is Fock matrix, **D** is density matrix
  - Held in core
- $(\mu\nu|\lambda\sigma)$ are "two-electron repulsion integrals"
  - Due to permutational symmetries, only $O(N^4/8)$ unique
  - Can be evaluated on the fly
- In integral-driven algorithm, each integral contracts with six different **D** elements, contributing to six different **F** elements
- Challenge: irregularity
  - Integrals evaluated in blocks of varying size (1-10,000+ integrals)
  - Average 500 FLOPs *per integral*, but wide variation

Scalable Fock Build Algorithm

Hierarchically blocked, dynamically load balanced, integral-driven



- Idea first implemented by Furlani and King (1995) using MPI
  - Efficient implementation (up to 16 CPUs) required heroic effort
  - Privately, approach not considered viable in general
- Inspired development of Global Array Toolkit (PNNL)
  - Library-based implementation of PGAS concepts
- NWChem implementation (1995) using GA scales to 1000s of CPUs
  - Simple to code

Fock Build Code Snippets (1)

- Defining a distributed array (X10)

```
final region Dregion = [1:n, 1:n];
final dist Ddist = dist.factory.block(Dregion);
final double [.] F = new double [Ddist];
```

- Local working copy of **D** block (Chapel)

```
const ij_points = [ilo..ihi, jlo..jhi];
const Dij = D(ij_points);
```

- Atomic update of global **F** from working block (X10)

```
final double value [.] Fij_val = Fij.toValueArray();
ateach(point [i,j] : Ddist | [ilo:ihi, jlo:jhi])
   atomic F[i,j] += Fij_val[i,j];
```

Fock Build Code Snippets (2)

- Transposition of distributed matrix (Chapel)

```
cobegin {
   [(i,j) in Ddist] JT(i,j) = J(j,i);
   [(i,j) in Ddist] KT(i,j) = K(j,i);
  }
```

- Transposition of distributed matrix (Fortress)

```
( JT, KT ) = ( J.t(), K.t() )
```

- Parallel four-fold loop with symmetries (Fortress)

```
for iat<-1#natom,
    jat<-1#iat, kat<-1#iat,
    lat<-1#(if (kat=iat) then jat else kat
  end) do
```

Integral Evaluation Loops (1)

- Work pool managed by language runtime (Fortress)

```
for iat<-1#natom, jat<-1#iat, kat<-1#iat,
      lat<-1#(if (kat=iat) then jat else kat end) do
   buildjk_atom4 blockIndices( (*many arguments*) )
end
```

- Work pool managed by language runtime (Chapel)

```
iterator allQuartets() {
  forall iat in 1..natom {
     forall jat in 1..iat {
        forall kat in 1..iat {
           const lattop
              = if (kat==iat) then jat else kat;
           forall lat in 1..lattop {
              yield blockIndices(/*many arguments*/);
} } } }
forall bI in allQuartets() do buildjk_atom4(bI);
```

Integral Evaluation Loops (2)

- User-managed work pool w/ atomic read & increment (X10)

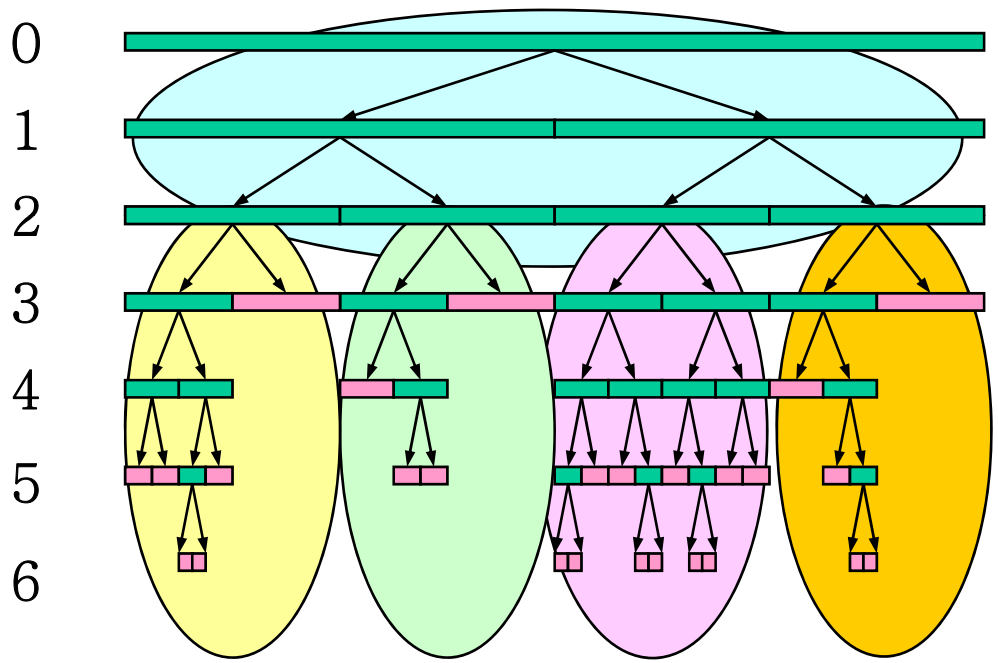
```
/* Launch a task on each place (~node) */
finish ateach(point [p] :
                dist.factory.unique(place.places)) {
   int myG, L = 0;
   /* Get my assignment for the next task */
   future<int> F = future (place.FIRST_PLACE)
                          {read_and_increment_G()};
   myG = F.force();
   /* Begin four-fold loop over iat,jat,kat,lat */
      if (L == myG) { /* If this is my task */
         /* Request new next task */
         F = future (place.FIRST_PLACE)
                          {read_and_increment_G()};
         buildjk_atom4(/* many arguments*/ );
         myG = F.force(); /* Get next task */ }
         ++L; /* Count my progress through loop */
   /* End iat, jat, kat, lat loops */
```

Integral Evaluation Loops (3)

- User-managed work pool w/ sync variables (Chapel)

```
// Simple one-element work pool
var task : sync blockIndices;
cobegin {
   // Fill work pool with tasks
   forall bI in allQuartets() do task = bI;
   // On each processor, consume work from pool
   begin on ( /*all locales (~nodes)*/ ) {
      bI = task; // Get next task from pool
      while ( /*bI is valid task*/ ) {
         // Evaluate block, get next task
      } // End while
      // Received sentinel, no more work to do
      // Put sentinel value back in pool
      // Increment count of completed consumers
   } // End begin on locales
} // End cobegin
// Wait for all consumers to complete
```

MADNESS Example – I



- Adaptive mesh in 1-6+ dimensions
- Very dynamic refinement
- Distribution by subtrees == spatial decomposition
- Sequential code very compactly written using recursion
- Initial parallel code using MPI is about 10x larger

MADNESS Example – II

Cilk-like multithreaded – all of the HPLS solutions look like this

```
template <typename T> void Function::_refine (OctTreeTPtr tree) {
    FOREACH_CHILD(OctTreeTPtr, tree, _project(child););
    TensorT* t = coeff(tree);
    TensorT d = filter(*t);
    if (d.normf() > truncate_tol(data->thresh,tree->n())) {
        unset_coeff(tree);
        FOREACH_CHILD(OctTreeTPtr, tree, spawn _refine(child););
} }
```

The complexity of the MPI code mostly arises from using an SPMD model to maintain a consistent distributed data structure.

The explicitly distributed code must work on active and inactive nodes – not necessary with global-view and remote activity creation.

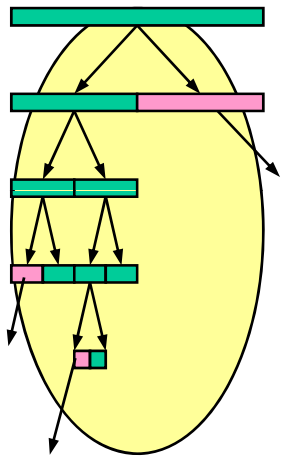
Explicit MPI

```
template <typename T>    void Function<T>::_refine(OctTreeTPtr& tree) {
    if (tree->islocalsubtreeparent() && isremote(tree)) {
        FOREACH_CHILD(OctTreeTPtr, tree, bool dorefine;
                comm()->Recv(dorefine, tree->rank(), REFINE_TAG);
                if (dorefine) {
                    set_active(child); set_active(tree);  _project(child);
                }
                _refine(child););
    } else {
        TensorT* t = coeff(tree);
        if (t) {
            TensorT d = filter(*t);
            d(data->cdata->s0) = 0.0;
            bool dorefine = (d.normf() > truncate_tol(data->thresh,tree->n()));
            if (dorefine) unset_coeff(tree);
            FOREACH_REMOTE_CHILD(OctTreeTPtr, tree,
                        comm()->Send(dorefine, child->rank(), REFINE_TAG);
                        set_active(child););
            FORIJK(OctTreeTPtr child = tree->child(i,j,k);
                if (!child && dorefine) child = tree->insert_local_child(i,j,k);
                if ( child && islocal(child)) {
                    if (dorefine) { _project(child);  set_active(child);  }
                    _refine(child);
                });
    } else {
            FOREACH_REMOTE_CHILD(OctTreeTPtr, tree,
                    comm()->Send(false, child->rank(), REFINE_TAG););
            FOREACH_LOCAL_CHILD(OctTreeTPtr, tree, _refine(child););
} } }
```

MADNESS Example – III

- X10, 1-D equivalent (ran in parallel, in one place)

```
void refine(final int n, final int l, final int nmax) {
        left = new Tree(this,2.0*l);
        right =new Tree(this, 2.0*l+1);
        final nullable Tree ll = left, rr=right;
        if (n < (nmax-1)) {
                async {ll.refine(n+1,2*l,nmax);}
                async { rr.refine(n+1,2*l+1,nmax);}
        }
        if (n < nmax) data = null;
}
```



Issues for all of the languages…

- How to control (or relinquish control) of the initial data placement?
  - One or the other is cumbersome in all of the languages
- How to express dynamic load balancing between localities?
  - Only Fortress seems to provide this

Building More Complex Abstractions

- Object orientation, generic programming, and programmer-written data distributions
- All three languages also intend to offer an unprecedented flexibility to extend the language
  - Libraries
  - Compiler optimizations and specializations
  - Language syntax (Fortress, X10)
- Extremely powerful tools to support high-level domain-specific abstractions

Tensor Contraction Engine (TCE)

- High-level domain-specific language for a class of problems in quantum chemistry/physics based on contraction of large multi-dimensional tensors
- Specialized optimizing compiler
  - Produces F77+GA code, linked to runtime libs

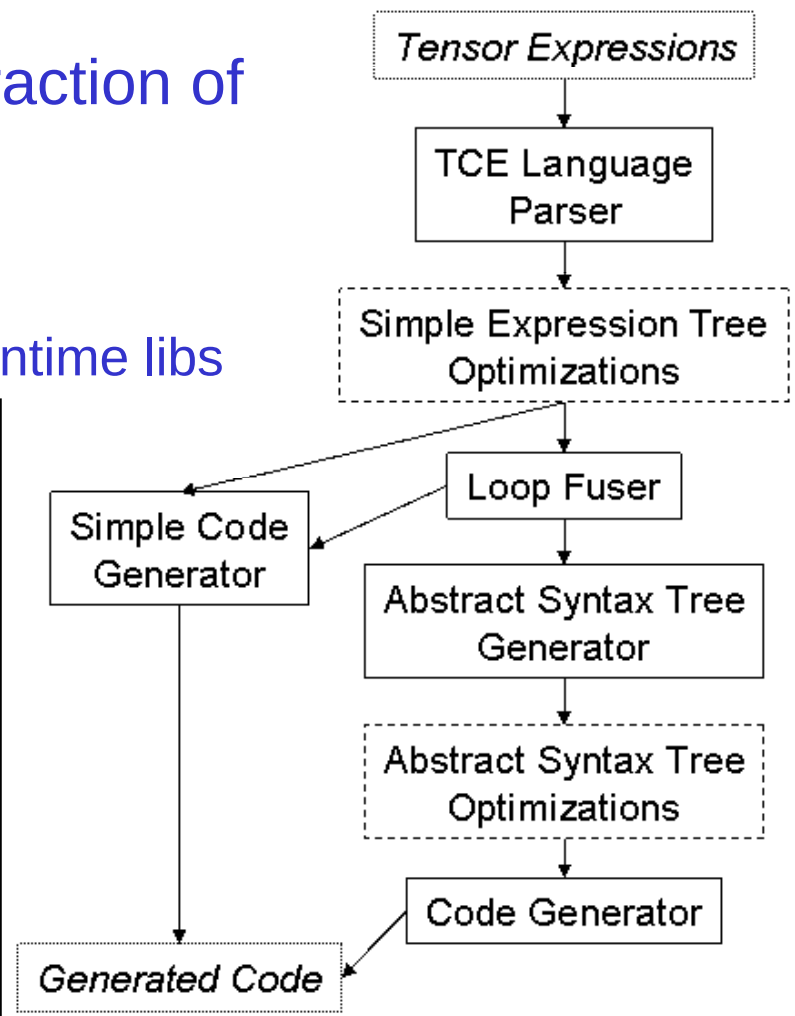
```
range V = 3000;
range O = 100;

index a,b,c,d,e,f : V;
index i,j,k,l : O;

mlimit = 100GB;

procedure P(in A[V,V,O,O], in B[V,V,V,O],
            in C[V,V,O,O], in D[V,V,V,O],
            out S[V,V,O,O])=
begin
  S[a,b,i,j] == sum[ A[a,c,i,k] * B[b,e,f,l]
                   * C[d,f,j,k] * D[c,d,e,l],
                     {c,d,e,f,k,l}];
end
```

$$S_{abij} = \sum_{cefkl} A_{acik} B_{befl} C_{dfjk} D_{cdel}$$



TCE Equivalent in HPCS Languages

- Object model, distributed array language capable of expressing complex tensor data structures
  - Reduces need for separate language
- User-written array distributions implement tensors
  - Reduces need for separate runtime
- Ability to extend compiler
  - Reduces need for separate compiler

Maybe it's Not So Far Off?

Simple TCE input

```
range V = 3000;
range O = 100;

index a,b,c,d,e,f : V;
index i,j,k,l : O;

mlimit = 100GB;

procedure P(in A[V,V,O,O], in B[V,V,V,O],
            in C[V,V,O,O], in D[V,V,V,O],
            out S[V,V,O,O])=
begin
  S[a,b,i,j] == sum[ A[a,c,i,k] * B[b,e,f,l]
                   * C[d,f,j,k] * D[c,d,e,l],
                     {c,d,e,f,k,l}];
end
```

Chapel version
by Brad Chamberlain, Cray
(working code!)

```
config const V = 3000,
             O = 100;
const DV = 1..V,
      DO = 1..O;
const DVVOO = [DV, DV, DO, DO],
      DVVVO = [DV, DV, DV, DO];
var A, C, S: [DVVOO] real,
       B, D: [DVVVO] real;
forall (a, b, i, j) in DVVOO do
   S(a,b,i,j) = + reduce [(c,d,e,f,k,l) in [DV,DV,DV,DV,DO,DO]]
                    (A(a,c,i,k) * B(b,e,f,l) * C(d,f,j,k) * D(c,d,e,l));
);
```

But Computational Scientists Don't Like Big Leaps!

- Much work to be done before these languages are ready for users
- Most areas have a few people who would willingly experiment with early versions
- Must define interoperability mechanisms w/ existing approaches
  - Language interop (a la Babel)
  - Programming model interop
    - X10 group developing C library of key X10 concepts
  - Data model interop
- Develop transitional approaches
  - User education
    - HPCS concepts unfamiliar to the majority
  - Code migration
    - For those who want a multistage migration path

Possible Transitional Approaches

- PGAS languages
  - Introduces global-view concepts, integrated parallelism
  - Moves users away from thinking about messages
  - Co-Array features on fast-track for next Fortran standard
- Bundle-Exchange-Compute (BEC) model (Wen, Sandia)
  - User indicates desire to share data, and when data must be present
  - Library manages data layout, movement, etc.
  - Available as library with or without (small) language extension
  - Shown useful for algorithms with random fine-grained data sharing (PRAM)
- Others?
- More work needed!

Conclusions

- Current Fortran+MPI+OpenMP approach will not get us to sustained petaflops *sustainably*
- Need to make a revolutionary leap in approach
- HPCS languages offer the kinds of features we need
  - Perhaps not *the* solution, but definitely the right direction!
- Must provide an evolutionary path to join the revolution
- All of this will take time (and $$$), so we had better get started
- Remember the frog!