High-Productivity Language Systems: Next-Generation Petascale Programming

Presented by
Sadaf R. Alam
Richard F. Barrett
David E. Bernholdt
Valmor F. de Almeida
Robert J. Harrison
Wael R. Elwasif
Jeffery A. Kuehn
Aniruddha G. Shet
Stephen W. Poole
Oak Ridge National Laboratory

Managed by UT-Battelle
for the Department of Energy
Revolutionary approach to large-scale parallel programming

• Million-way concurrency (and more) will be required on coming HPC systems

• The current “Fortran+MPI+OpenMP” model will not scale

• New languages from the DARPA HPCS program point the way toward the next-generation programming environment

• Emphasis on performance and productivity

• Not SPMD
  – Lightweight “threads,” LOTS of them
  – Different approaches to locality awareness/management

• High-level (sequential) language constructs
  – Rich array data types (part of the base languages)
  – Strongly typed object oriented base design
  – Extensible language model
  – Generic programming
Concurrent*: The next generation

- Single initial thread of control
  - Parallelism through language constructs
- True global view of memory, one-sided access model
- Support *task* and *data* parallelism
- “Threads” grouped by “memory locality”
- Extensible, rich distributed array capability
- Advanced concurrency constructs
  - Parallel loops
  - Generator-based looping and distributions
  - Local and remote futures
What about productivity?

- Index sets/regions for arrays
  - “Array language” (Chapel, X10)

- Safe(r) and more powerful language constructs
  - Atomic sections vs. locks
  - Sync variables and futures
  - Clocks (X10)

- Type inference

- Leverage advanced IDE capabilities

- Units and dimensions (Fortress)

- Component management, testing, contracts (Fortress)

- Math/science-based presentation (Fortress)
Dynamic concurrency in MADNESS

refine(node) {
    child = node.get_children();
    ...
    if (normf(node) < threshold) {
        // local heuristic to match physical parallelism
        if (n+1 < log2(maxThreads)) then
            // asynchronous node update
            on child.locale do begin sumC[child] = ..;
        else
            // serial node update
            sumC[child] = ..;
    }
    else {
        // local heuristic to match physical parallelism
        if (n+1 < log2(maxThreads)) then
            // asynchronous data-driven refinement
            on child.locale do begin refine(child);
        else
            // serial refinement
            refine(child);
    }
}

- Adaptive mesh in 1-6+ dimensions, very dynamic refinement
- Spatial decomposition by subtrees
- Distributed container enables nonprocess-centric computing
- Compact parallel code using task-parallelism and locality control
Fusion simulation: AORSA using Chapel

- Multiple spatial regions: Languages provide well-defined views

- Language and programming-model interoperability needed to interact with existing libraries

- Language constructs such as locales, regions, and places provide natural mapping to local view process abstraction—needed to access existing libraries
Parallel mesh sweeping in Chapel

```chapel
var setT = new FiniteElementPartition("mesh.dat");
const setTCardinality: int = setT.GetCardinality();
enum{blue, green, red};
foreach sweepDir in sweepDirections{
    var waveFrontMask = newVector(int, 0..setTCardinality, blue);
    waveFrontMask(0) = green
    while (waveFrontMask.floor(blue)) {
        foreach fe in setT.getMembers() {
            if (waveFrontMask(fe.getId()) == blue) {
                var setColor: bool = true
                foreach edge in fe.getEdges() {
                    const normal = fe.getEdgeNormal(edge);
                    if (normal ^ sweepDir < 0.0) {
                        const neMemberId: int = setT.getNeighborMemberId(fe, edge);
                        if (waveFrontMask(neMemberId) != green) { setColor = false; }
                    }
                    if (setColor == true) { waveFrontMask(fe.getId()) = red; }
                }
            }
        }
        foreach i in waveFrontMask { if (i != blue) { i = green; }}
    }
}
```

```chapel
class Vector {
    // Generic template class
    def getType() type { return T; }
    def getSize() { return rng.high - rng.low + 1; }
    def getRange() { return rng; }
    def getDomain() { return dom; }
    def resize(n: range = 1..0){rng = n; dom = [rng];}
    def clear(){this.resize();}
    def find(val: T): bool {
        return linearSearch(data, val(1));
    }
    def these() var {for i in dom {yield data(i);}}
    // Members
    type T;
    var rng : range(int);
    var value: T;
    var dom domain(1) distributed(Block) = [rng];
    var data: [dom] T = value;
}
```
Tradeoffs in HPLS language design

- Emphasis on parallel safety (X10) vs. expressivity (Chapel, Fortress)
  - Locality control and awareness
  - X10: Explicit placement and access
  - Chapel: User-controlled placement, transparent access
  - Fortress: Placement “guidance” only, local/remote access blurry (data may move!!!)
  - *What about mental performance models?*

- Programming language representation
  - Fortress: Allows mathlike representation
  - Chapel, X10: Traditional programming language front end
  - *How much do developers gain from mathematical representation?*

- Productivity/performance tradeoff
  - Different users have different “sweet spots”
Remaining challenges

• (Parallel) I/O model

• Interoperability with (existing) languages and programming models

• Better (preferably portable) performance models and scalable memory models
  – Especially for machines with 1M+ processors

• Other considerations:
  – Viable gradual adoption strategy
  – Building a complete development ecosystem
Contacts

Sadaf R. Alam
alamsr@ornl.gov

Richard F. Barrett
rbarrett@ornl.gov

David E. Bernholdt
bernholdtde@ornl.gov

Valmor F. de Almeida
dealmeidav@ornl.gov

Wael R. Elwasif
elwasifwr@ornl.gov

Robert J. Harrison
harrisonrj@ornl.gov

Jeffery A. Kuehn
kuehn@ornl.gov

Stephen W. Poole
spoole@ornl.gov

Aniruddha G. Shet
shetag@ornl.gov