Chapel

An Emerging Parallel Programming Language

If you wouldn't recommend your HPC programming model to a non-HPC friend, why aren't you demanding something better?

HPC has no languages that are as modern, productive, and enjoyable as Python / Matlab / Java / (your favorite language here). This is due less to technical challenges than to lack of community will, resources, effort, and patience.

Let's change that!

Chapel Characteristics

- General Parallelism
  - Any parallel algorithm
  - Any parallel hardware
- Separate Parallelism from Locality from Mechanism
  - What should run concurrently?
  - Where should tasks / data be placed?
  - Leave "how should it run?" to lower levels of software
- Designed for Performance
  - Features designed to compete with C/Fortran + MPI
- Modern Productivity Features
  - Type inference, iterators, rich array types, ranges, tuples, generic programming, optional OOP, …

Chapel's Implementation

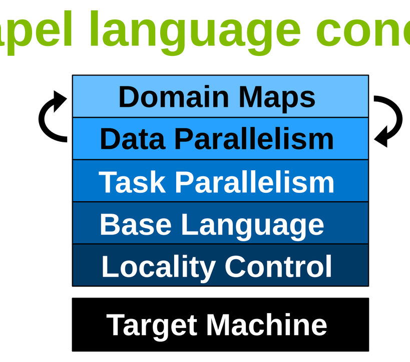
- Open Source
  - BSD License
  - Hosted and developed at SourceForge
- Scalable, Portable Design
  - From laptops to clusters, supercomputers, & the cloud
  - Requirements: C/C++, POSIX tools and threads
- Status
  - Version 1.8 released October 17, 2013
  - Core features are functional, ready to be used
  - Additional performance optimizations needed
- Community Effort
  - v1.8: 19 contributors from 8 organizations/5 countries

Multiresolution Design

Goal: Support multiple tiers of features

- higher levels for programmability, productivity
- lower levels for greater degrees of control

Chapel language concepts



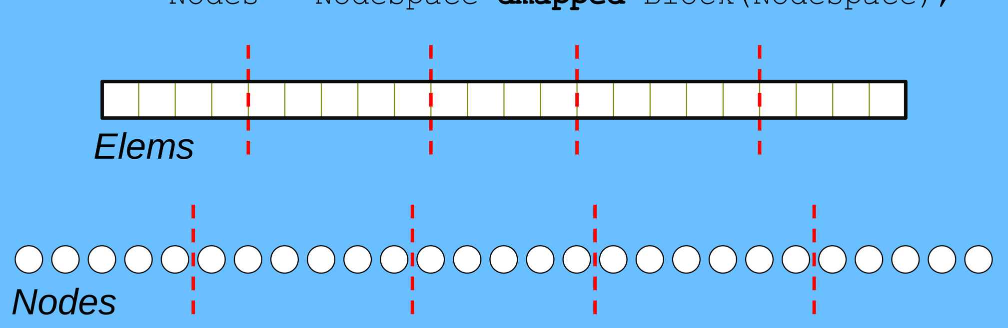
- build the higher levels in terms of the lower
- permit the user to intermix layers arbitrarily

Domain Maps

Role: map global-view domains/arrays to locales

```
const ElemSpace = {0..#numElems},
      NodeSpace = {0..#numNodes},
      Elems = ElemSpace dmapped Block(ElemSpace),
      Nodes = NodeSpace dmapped Block(NodeSpace);
```

local and distributed domains (first-class index sets)



Declarations from the Chapel version of LULESH

Base Language

Role: support productive serial programming

```
iter tiledRMO(D, tilesize) {
  const tile = {0..#tilesize, 0..#tilesize};

  for base in D by tilesize do
    for ij in D[tile + base] do
      yield ij;
}

for ij in tiledRMO({1..m, 1..n}, 2) do
  write(ij);
```

the base language includes a rich algebra for operating on domains

yield a value back to the callsite; then continue executing the iterator

This loop prints:
(1,1) (1,2) (2,1) (2,2) (1,3) (1,4) (2,3) (2,4)…
(3,1) (3,2) (4,1) (4,2) (3,3) (3,4) (4,3) (4,4)…

Tiled, row-major order iterator and a serial invocation of it

Data Parallelism

Role: drive parallelism via domains/arrays

```
// find the total velocity across atoms
vtot = + reduce forall (bin, c) in zip(Bins, Count) do
           + reduce forall a in bin[1..c] do a.v;

// compute the average velocity
vtot /= numAtoms;

// adjust atom velocities using the average
forall (bin, c) in zip(Bins, Count) do
  for a in bin[1..c] do
    a.v -= vtot;
```

forall loops and reductions are core data parallel operations

zippered iteration is used to traverse multiple collections simultaneously

Computations from the Chapel version of MiniMD

Task Parallelism

Role: express parallelism via concurrent tasks

```
var buff$: [0..#buffsize] sync real;

cobegin {
  producer();
  consumer();
}

proc producer() {                proc consumer() {
  var i = 0;                       var i = 0;
  for … {                            while … {
    i = (i+1) % buffsize;              i= (i+1) % buffsize;
    buff$[i] = …;                      …buff$[i]…;
  }                                  }
}                                  }
```

an array of synchronization variables: each stores a full/empty state plus value

concurrently begin two tasks, a producer and a consumer

writes to sync variables block until empty, leave full

reads block until full, leave empty

Bounded buffer producer-consumer in Chapel

Locality Control

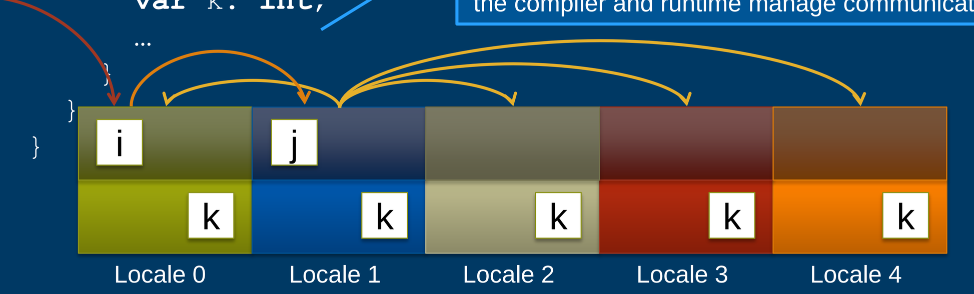
Role: permit tuning for locality and affinity

```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      var k: int;
      …
    }
  }
}
```

on-clauses migrate the current task to the specified locale (compute node)

variables are declared on the locale where the current task is running

within this scope, *i* and *j* may be referenced; the compiler and runtime manage communication



Sample mapping of tasks and their variables to locales

Next Steps

- Performance Optimizations
- Accelerator Support (GPUs, MIC)
- Improved Interoperability
- Feature Improvements
- Research Efforts
- Outreach

A Team Effort — Join Us!

CRAY · LTS Laboratory for Telecommunications Sciences · RICE · Sandia National Laboratories · Lawrence Livermore National Laboratory · Berkeley Lab Lawrence Berkeley National Laboratory · Argonne National Laboratory · Oak Ridge National Laboratory · Pacific Northwest National Laboratory · 東京大学 The University of Tokyo · Universidad de Málaga · IU · University of Maryland

CRAY CHAPEL