The Cascade High Productivity Language

Brad Chamberlain
David Callahan
Hans Zima*
Chapel Team, Cascade Project
Cray Inc., *CalTech/JPL

Chapel's Context

HPCS = High Productivity Computing Systems
(a DARPA program)

Overall Goal: Increase productivity for HEC community by the year 2010

Productivity = Programmability
+ Performance
+ Portability
+ Robustness

Result must be…

…revolutionary not evolutionary

…marketable to people other than program sponsors

Phase II Competitors (7/03-7/06): Cray, IBM, and Sun

Why develop a new language?

- We believe current parallel languages are inadequate:
  - tend to require fragmentation of data, control
  - tend to support a single parallel model (data or task)
  - fail to support composition of parallelism
  - few data abstractions (sparse arrays, graphs)
  - poor support for generic programming
  - fail to cleanly isolate computation from changes to…

    …virtual processor topology

    …data decomposition

    …communication details

    …choice of data structure

    …memory layout

What is Chapel?

- *Chapel:* Cascade High-Productivity Language
- Overall goal: Solve the parallel programming problem
  - simplify the creation of parallel programs
  - support their evolution to extreme-performance, production-grade codes
- Motivating Language Technologies:
  1) multithreaded parallel programming
  2) locality-aware programming
  3) object-oriented programming
  4) generic programming and type inference

1) Multithreaded Parallel Programming

- Global view of computation, data structures
- Abstractions for data and task parallelism
  - data: domains, foralls
  - task: cobegins, synch/future variables
- Composition of parallelism
- Virtualization of threads

Global-view: Definition

- “Must programmer code on a per-processor basis?”
- **Data parallel example:** “Add 1000 x 1000 matrices”

global-view

```
var n: integer = 1000;
var a, b, c: [1..n, 1..n] float;

forall ij in [1..n, 1..n]
  c(ij) = a(ij) + b(ij);
```

fragmented

```
var n: integer = 1000;
var locX: integer = n/numProcRows;
var locY: integer = n/numProcCols;
var a, b, c: [1..locX, 1..locY] float;

forall ij in [1..locX, 1..locY]
  c(ij) = a(ij) + b(ij);
```

- **Task parallel example:** “Run Quicksort”

global-view

```
computePivot(lo, hi, data);
cobegin {
  Quicksort(lo, pivot, data);
  Quicksort(pivot, hi, data);
}
```

fragmented

```
if (iHaveParent)
  recv(parent, lo, hi, data);
computePivot(lo, hi, data);
if (iHaveChild)
  send(child, lo, pivot, data);
else
  LocalSort(lo, pivot, data);
LocalSort(pivot, hi, data);
if (iHaveChild)
  recv(child, lo, pivot, data);
if (iHaveParent)
  send(parent, lo, hi, data);
```

Global-view: Impact

- Fragmented languages…
  …obfuscate algorithms by interspersing per-processor management details in-line with the computation
  …require programmers to code with SPMD model in mind
- Global-view languages abstract the processors from the computation

global-view languages
OpenMP
HPF
ZPL
Sisal
MTA C/Fortran
Matlab
Chapel

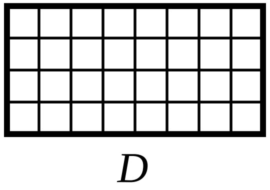
fragmented languages
MPI
SHMEM
Co-Array Fortran
UPC
Titanium

Data Parallelism: Domains

- *domain:* an index set
  - potentially decomposed across locales
  - specifies size and shape of data structures
  - supports sequential and parallel iteration
- Two main classes:
  - *arithmetic:* indices are Cartesian tuples
    - rectilinear, multidimensional
    - optionally strided and/or sparse
    - possibly “triangular” or “bounded” varieties?
  - *opaque:* indices are anonymous
    - supports sets, graph-based computations
- Fundamental Chapel concept for data parallelism
- Similar to ZPL’s *region* concept

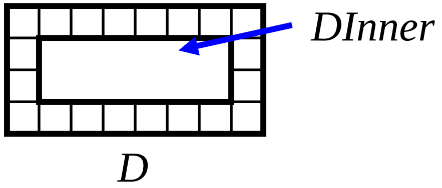
A Simple Domain Declaration

```
var m: integer = 4;
var n: integer = 8;

var D: domain(2) = [1..m, 1..n];
```



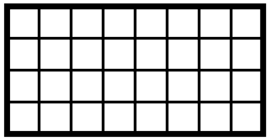
A Simple Domain Declaration

```
var m: integer = 4;
var n: integer = 8;

var D: domain(2) = [1..m, 1..n];
var DInner: domain(D) = [2..m-1, 2..n+1];
```



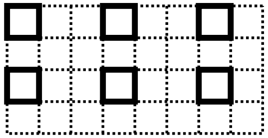
Other Arithmetic Domains

```
var D2: domain(2) = (1,1)..(m,n);
```



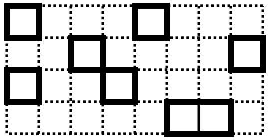
D2

```
var StridedD: domain(D) = D by (2,3);
```



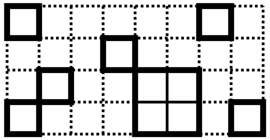
StridedD

```
function foo(ind: index(D)): boolean { … }
var SparseD: domain(D) = [ij:D] where foo(ij);
```



SparseD

```
var indArray: [1..numInds] index(D) = …;
var SparseD2: domain(D) = D where indArray;
```

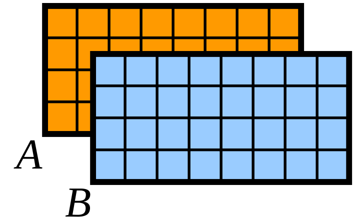


SparseD2

Domain Uses

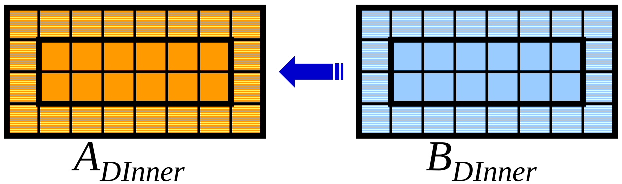
- Declaring arrays:

```
var A, B: [D] float;
```



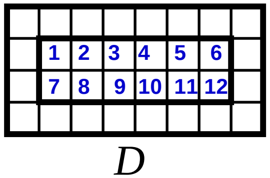
- Sub-array references:

```
A(DInner) = B(DInner);
```



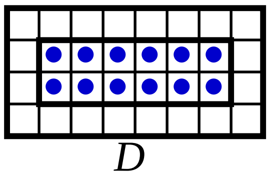
- Sequential iteration:

```
    for (i,j) in DInner { …A(i,j)… }
or: for ij in DInner { …A(ij)… }
```



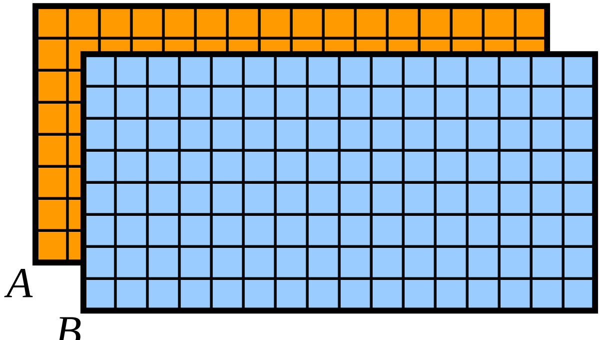
- Parallel iteration:

```
    forall ij in DInner { …A(ij)… }
or: [ij:DInner] …A(ij)…
```



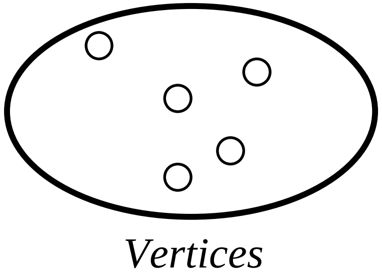
- Array reallocation:

```
D = [1..2*m, 1..2*n];
```

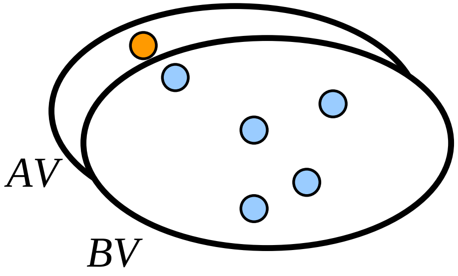


Opaque Domains

```
var Vertices: domain(opaque);

for i in (1..5) {
  Vertices.newIndex();
}
```



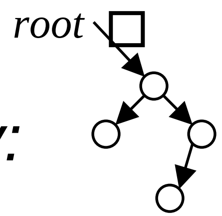

```
var AV, BV: [Vertices] float;
```



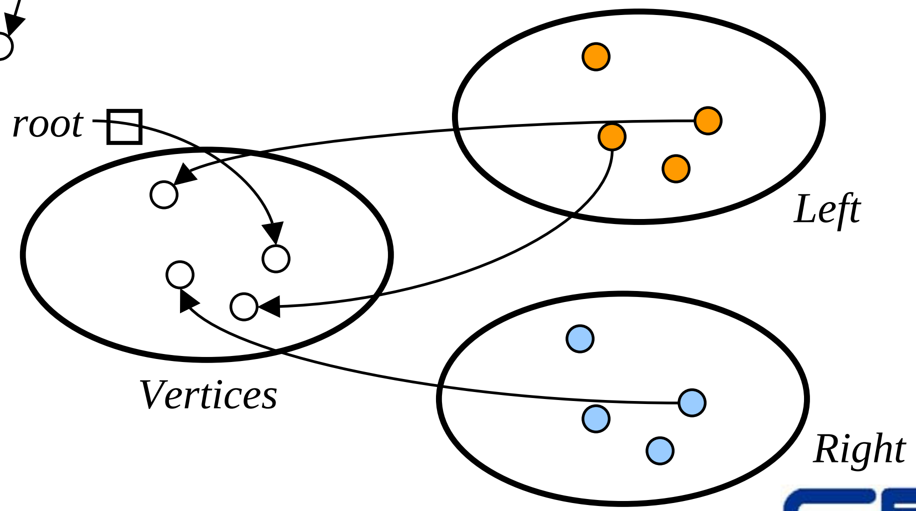
Opaque Domains II

```
var Vertices: domain(opaque);
var left, right: [Vertices] index(Vertices);
var root: index(Vertices);

root = Vertices.newIndex();
left(root) = Vertices.newIndex();
right(root) = Vertices.newIndex();
left(right(root)) = Vertices.newIndex();
```

conceptually:



more precisely:



Task Parallelism

- co-begin indicates statements that may run in parallel:

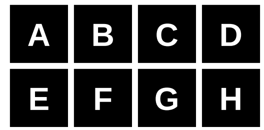
```
computePivot(lo, hi, data);
cobegin {
  Quicksort(lo, pivot, data);
  Quicksort(pivot, hi, data);
}

cobegin {
  ComputeTaskA(…);
  ComputeTaskB(…);
}
```

- synch and future variables as on the Cray MTA

2) Locality-aware Programming

- *locale:* machine unit of storage and processing

```
var CompGrid: [1..GridRows, 1..GridCols] locale = …;
```



CompGrid

```
var TaskALocs: [1..numTaskALocs] locale = …;
var TaskBLocs: [1..numTaskBLocs] locale = …;
```

TaskALocs

TaskBLocs

- domains may be distributed across locales

```
var D: domain(2) distributed(block(2)) to CompGrid = …;
```

- “on” keyword binds computation to locale(s)

```
 cobegin {
   on TaskALocs: ComputeTaskA(…);
   on TaskBLocs: ComputeTaskB(…);
 }
```

3) Object-oriented Programming

- OOP can help manage program complexity
  - separates common interfaces from specific implementations
  - facilitates reuse
- Classes and objects are provided in Chapel, but their use is typically not required
- Advanced language features expressed using classes
  - user-defined reductions, distributions, etc.

4) Generic Programming and Type Inference

- Type Parameters

```
function copyN(data: [..] type t; n: integer): [1..n] t {
  var newcopy: [1..n] t;
  forall i in (1..n)
    newcopy(i) = data(i);
  return newcopy;
}
```

Type of *data* named but unspecified

Type can be used elsewhere

- Latent Types

```
function inc(val) {
  var tmp = val;
  val = tmp + 1;
}
```

Types of *val* and *tmp* elided

- Variables are statically-typed

Other Chapel Features

- Tuples and sequences
- Anonymous functions, closures, currying
- Support for user-defined…
  …iterators
  …reductions and parallel prefix operations
  …data distributions
  …data layout specifications
    - row/column-major order, block-recursive, Morton order...
    - different sparse representations
- Garbage Collection

Chapel Implementation

- Current Implementation (Phase II)
  - source-to-source compilation
    Chapel $\rightarrow$ C
    + communication library (ARMCI, GASnet, ???)
    + threading library
  - targeting commodity architectures
    - desktop workstations, clusters
  - goal: proof-of-concept, experimentation, development
  - open-source effort
- Ultimate Implementation (Phase III)
  - target Cascade
  - likely stick to source-to-source compilation in near-term
  - replace explicit comm. and threading with compiler pragmas
- Mid-range Implementations? (Phase ???)
  - X1/X1e?
  - MTA-2?

Summary

- Chapel is being designed to…
  …enhance programmer productivity
  …address a wide range of workflows

- Via high-level, extensible abstractions for…
  …multithreaded parallel programming
  …locality-aware programming
  …object-oriented programming
  …generic programming and type inference