# An Introduction to Chapel

## Cray Cascade's High-Productivity Language

AHPCRC/DARPA PGAS Conference
September 14, 2005

Brad Chamberlain
Cray Inc.

# HPCS in one slide

**HPCS** = High Productivity Computing Systems
(a DARPA program)

**Overall Goal:** Increase productivity for HEC community by the year 2010 (via HW, arch., OS, compilers, tools, …)

**Productivity =** Programmability
\+ Performance
\+ Portability
\+ Robustness

**Result must be…**
…revolutionary not evolutionary
…marketable to users other than program sponsors

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

# Why develop a new language?

- We believe current parallel languages are lacking
- In general, they…
  - …tend to require fragmentation of data and control
  - …fail to cleanly isolate computation from changes to…
    - …virtual processor topology
    - …data decomposition
    - …communication details
    - …choice of data structures, memory layout
  - …tend to support a single type of parallelism
    - data or task parallelism
  - …fail to support composition of parallelism
  - …have few data abstractions
    - distributed sparse arrays, graphs, hash tables, …
  - …lack broad-market language features (Java, Matlab, …)

# 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
  - emphasize generality

- Motivating Language Technologies:
  1) multithreaded parallel programming
  2) locality-aware programming
  3) object-oriented programming
  4) generic programming and type inference

# Outline

- Setting and Motivation
- Chapel Overview
- Challenges and Summary

# 1) Multithreaded Parallel Programming

- Virtualization of threads
  - e.g., no fork/join
- Abstractions for data and task parallelism
  - *data:* domains, arrays, iterators, …
  - *task:* cobegins, atomic transactions, sync variables, …
- Composition of parallelism
- Global view of computation, data structures

# 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: Definition

- "Must programmer code on a per-processor basis?"
- **Data parallel example:** "Apply 3-pt stencil to vector"

**global-view**

```
var n: integer = 1000;
var x, y: [1..n] float;

forall i in (2..n-1)
  y(i) = (x(i-1) + x(i+1))/2;
```

**fragmented**

```
var n: integer = 1000;
var locN: integer = n/numProcs;
var x, y: [0..locN+1] float;

if (iHaveRightNeighbor) {
  send(right, x(locN));
  recv(right, x(locN+1));
}
if (iHaveLeftNeighbor)
  send(left, x(1));
  recv(left, x(0));
}
```

```
forall i in (1..locN)
  y(i) = (x(i-1) + x(i+1))/2;
```

- **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

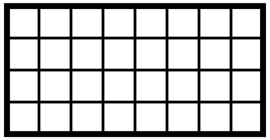
| fragmented languages | global-view languages |
| --- | --- |
| MPI | HPF |
| SHMEM | ZPL |
| Co-Array Fortran | Sisal |
| UPC | NESL |
| Titanium | MTA C/Fortran |
| | OpenMP (often) |
| | Matlab (trivially) |
| | Chapel |

# Data Parallelism: Domains

- *domain:* an index set
  - specifies size and shape of “arrays”
  - potentially decomposed across locales
  - supports sequential and parallel iteration
- Three main classes:
  - *arithmetic:* indices are Cartesian tuples
    - rectilinear, multidimensional
    - optionally strided and/or sparse
  - *indefinite:* indices serve as hash keys
    - supports hash tables, associative arrays, dictionaries
  - *opaque:* indices are anonymous
    - supports sets, graph-based computations
- Fundamental Chapel concept for data parallelism
- A generalization of ZPL’s *region* concept

# A Simple Domain Declaration

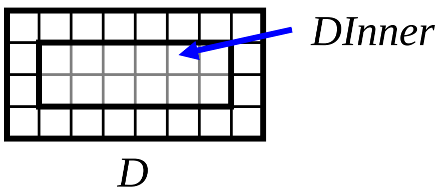
```
var m: integer = 4;
var n: integer = 8;

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



*D*

# A Simple Domain Declaration

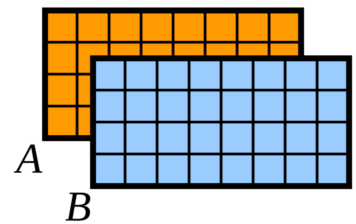
```
var m: integer = 4;
var n: integer = 8;

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



# Domain Uses

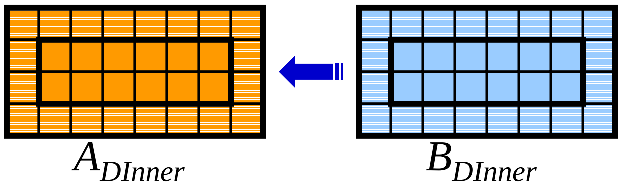
- Declaring arrays:

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



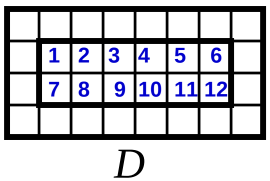

- Sub-array references:

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



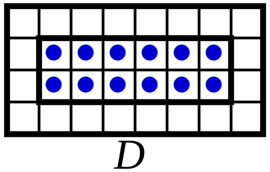

- Sequential iteration:

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



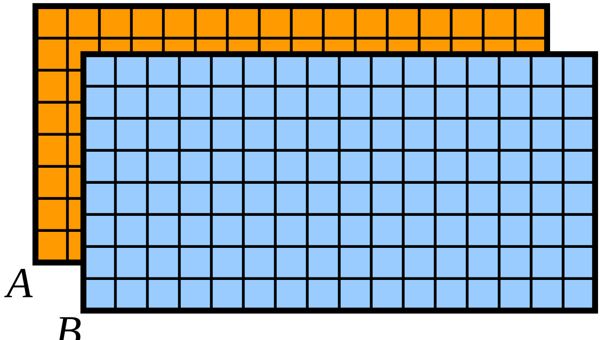

- Parallel iteration:

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



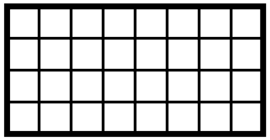

- Array reallocation:

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



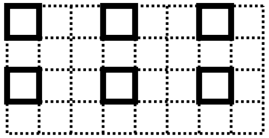
# Other Arithmetic Domains

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



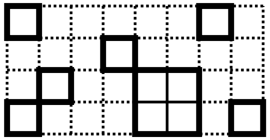
*D2*

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



*StridedD*

```
var indexList: seq(index(D)) = …;
var SparseD: sparse domain(D) = indexList;
```



*SparseD*

# The Domain/Index Hierarchy

domain(1) domain(2) domain(3) … domain(opaque)

domain(D) domain(D2)

domain(DInner)

domain(SparseD)

domain(StridedD)

# The Domain/Index Hierarchy

domain(1) domain(2) domain(3) … domain(opaque)

domain(D) domain(D2)

domain(DInner)

domain(SparseD)

domain(StridedD)

index(1) index(2) index(3) … index(opaque)

index(D) index(D2)

index(DInner)

index(SparseD)

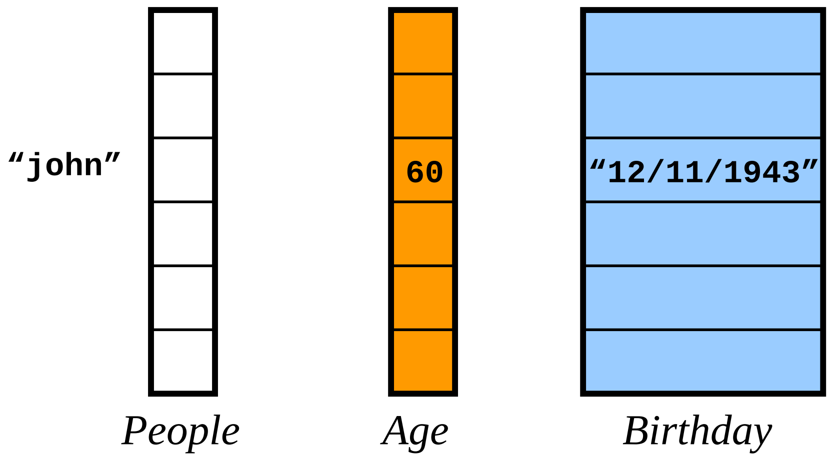
index(StridedD)

```
forall ij in DInner { …A(ij)… }
```

ij: index(DInner);

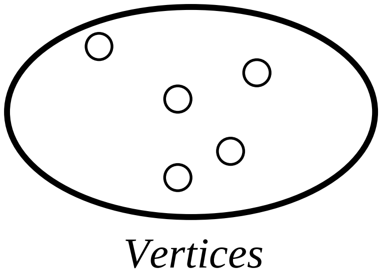
# Indefinite Domains

```
var People: domain(string);
var Age: [People] integer;
var Birthdate: [People] string;

Age("john") = 60;
Birthdate("john") = "12/11/1943";
…
forall person in People {
  if (Birthdate(person) == today) {
    Age(person) += 1;
  }
}
```

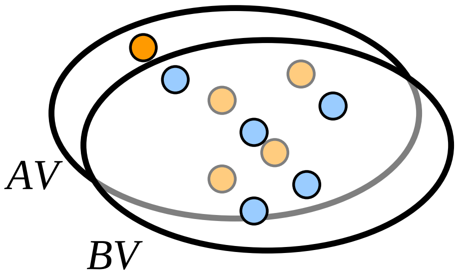


# Opaque Domains

```
var Vertices: domain(opaque);

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



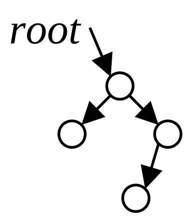

```
var AV, BV: [Vertices] float;
```



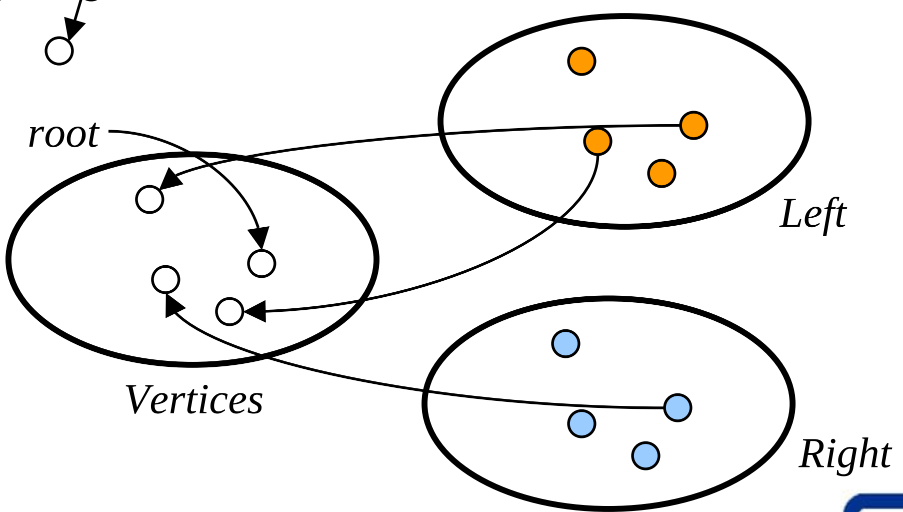
# Opaque Domains II

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

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

*conceptually:*




*more precisely:*



# Task Parallelism

- *co-begins:* indicate statements that may run in parallel:

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

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

- *atomic sections:* support atomic transactions

```
atomic {
  newnode.next = insertpt;
  newnode.prev = insertpt.prev;
  insertpt.prev.next = newnode;
  insertpt.prev = newnode;
}
```

- *sync and single-assignment variables:* synchronize tasks
  - similar to Cray MTA C/Fortran

# 2) Locality-aware Programming

- *locale:* machine unit of storage and processing
- programmer specifies number of locales on executable command-line

```
prompt> myChapelProg –nl=8
```

- Chapel programs provided with built-in locale array:

```
const Locales: [1..numLocales] locale;
```

- Users may define their own locale arrays:

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

| | | | |
|---|---|---|---|
| A | B | C | D |
| E | F | G | H |

*CompGrid*

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

| | |
|---|---|
| A | B |

*TaskALocs*

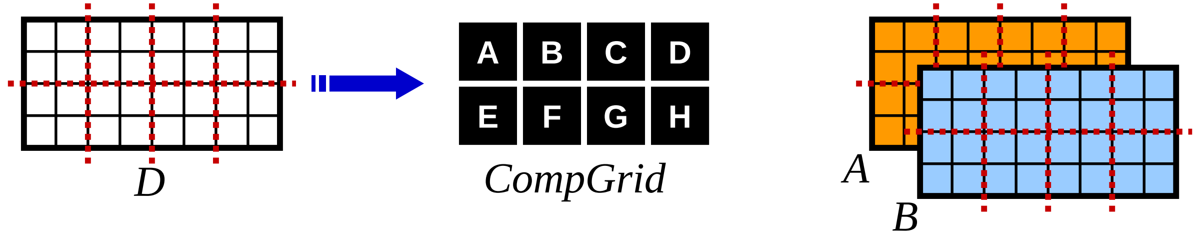
| | | | | | |
|---|---|---|---|---|---|
| C | D | E | F | G | H |

*TaskBLocs*

# Data Distribution

- domains may be distributed across locales

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




- Distributions specify…
  …mapping of indices to locales
  …per-locale storage layout of domain indices and array elements

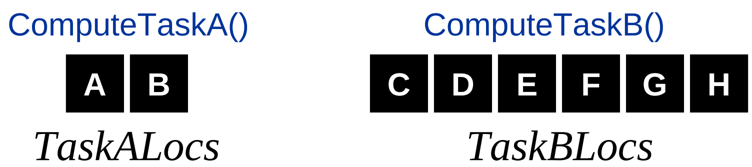
- Distributions implemented as a class hierarchy
  - Chapel provides a number of standard distributions
  - Users may also write their own

one of our biggest challenges

# Computation Distribution

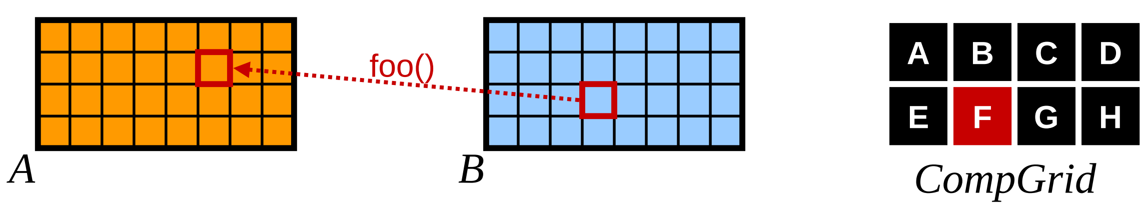
- "on" keyword binds computation to locale(s):

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




- "on" can also be used in a data-driven manner:

```
forall (i,j) in D {
  on B(j/2,i*2) do A(i,j) = foo(B(j/2,i*2));
}
```



# 3) Object-oriented Programming

- OOP can help manage program complexity
  - encapsulates related data and code
  - facilitates reuse
  - separates common interfaces from specific implementations
- Chapel supports traditional and value classes
  - traditional – pass, assign by reference
  - value – pass, assign by value/name
- OOP is typically not required (user's preference)
- Advanced language features expressed using classes
  - user-defined distributions, reductions, …

# 4) Generic Programming and Type Inference

- Type Variables and Parameters

```
class Stack {
  type t;
  var buffsize: integer = 128;
  var data: [1..buffsize] t;
  function top(): t { … };
}
```

- Type Query Variables

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

- Latent Types

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

- Chapel programs are statically-typed

# Other Chapel Features

- Tuple types, type unions, and typeselect statements
- Sequences, user-defined iterators
- Support for reductions and scans (parallel prefix)
  - including user-defined operations
- Default arguments, name-based argument passing
- Function and operator overloading
- Curried function calls
- Modules (for namespace management)
- Interoperability with other languages
- Garbage Collection

# Chapel Challenges

- User Acceptance
  - True of any new language
  - Skeptical audience
- Commodity Architecture Implementation (Phase III)
  - Chapel designed with idealized architecture in mind
  - Clusters are not ideal in many respects
  - Results in implementation and performance challenges
- Cascade Implementation
  - Efficient user-defined domain distributions
  - Type determination w/ OOP w/ overloading w/ …
  - Parallel Garbage Collection
- And others as well…

# 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
- Status:
  - draft language specification available at:
    http://chapel.cs.washington.edu
  - Open source implementation proceeding apace
  - Your feedback desired!