An Introduction to Chapel

Cray Cascade's High-Productivity Language

compiled for Mary Hall, February 2006

Brad Chamberlain

Cray Inc.

Context of this work

HPCS = High Productivity Computing Systems
(a DARPA program)

Overall Goal: Increase productivity for High-End Computing (HEC) community by the year 2010

Productivity = Programmability
+ Performance
+ Portability
+ Robustness

Result must be…
…revolutionary, not evolutionary
…marketable to users beyond the program sponsors

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

Parallel Language Wishlist

1) a global view of computation
2) general support for parallelism
   – data- and task-parallel; nested parallelism
3) clean separation of algorithm and implementation
4) broad-market language features
   – OOP, GC, latent types, overloading, generic functions/types, …
5) data abstractions
   – sparse arrays, hash tables, sets, graphs, …
   – distributed as well as local versions of these
6) good performance
7) execution model transparency
8) portability to various architectures
9) interoperability with existing codes

Parallel Language Evaluation

current parallel languages

wishlist	MPI, SHMEM	UPC, CAF	Titanium	OpenMP
global view				➽
general parallelism				~
separation of alg. & impl.			~	
broad-market features			➽	
data abstractions			~	
good performance	➽	➽	➽	➽
execution transparency	➽	➽	➽	~
portability	➽	➽	➽	~
interoperability	~	~		~

Why do our languages fall short in this region?

OpenMP uses a shared-memory model; others use SPMD models

Titanium based on Java; others on C, C++, Fortran

⇒ <u>Areas for Improvement</u>
Programming Models
Base Languages
Interoperability

Outline

- Chapel Motivation & Foundations
  - ✓ Parallel Language Wishlist
  - ➢ Programming Models and Productivity
- Chapel Overview
- Wrap-up

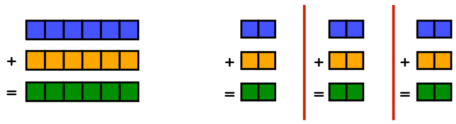
Parallel Programming Models

- *Fragmented Models:*
  - Programmer writes code on a task-by-task basis
    - breaks distributed data structures into per-task chunks
    - breaks work into per-task iterations/control flow
- *Single Program Multiple Data (SPMD) Model:*
  - Programmer writes one program, runs multiple instances of it
    - code parameterized by instance #
    - the most commonly-used example of the fragmented model
- *Global-view Models:*
  - Programmer need not decompose everything task-by-task
    - burden of decomposition shifts to compiler/runtime
    - user may guide this process via language constructs
- *Locality-aware Models:*
  - Programmer understands how data, control are mapped to the machine

Programming Model Examples

- **Data parallel example:** "Add 1000-element vectors"

global-view | fragmented



Programming Model Examples

- **Data parallel example:** "Add 1000-element vectors"

global-view

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

forall i in (1..n) {
  c(i) = a(i) + b(i);
}
```

SPMD

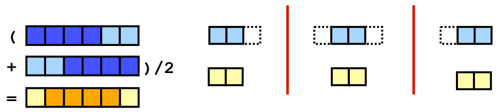
```
var n: integer = 1000;
var locN: integer = n/numProcs;
var a, b, c: [1..locN] float;

forall i in (1..locN) {
  c(i) = a(i) + b(i);
}
```

Assumes *numProcs* divides *n*; a more general version would require additional effort

Programming Model Examples

- **Data parallel example 2:** “Apply 3-pt stencil to vector”

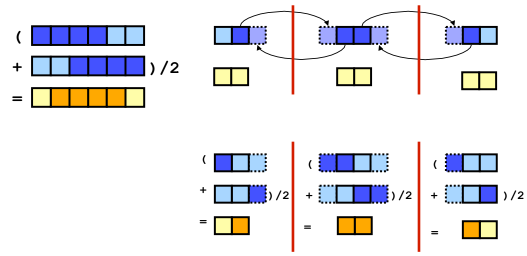
global-view　　　　fragmented



Programming Model Examples

- **Data parallel example 2:** “Apply 3-pt stencil to vector”

global-view　　　　fragmented



Programming Model Examples

- **Data parallel example 2:** “Apply 3-pt stencil to vector”

global-view

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

forall i in (2..n-1) {
  b(i) = (a(i-1) + a(i+1))/2;
}
```

SPMD

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

if (iHaveRightNeighbor) {
  send(right, a(locN));
  recv(right, a(locN+1));
}
if (iHaveLeftNeighbor)
  send(left, a(1));
  recv(left, a(0));
}
forall i in (1..locN) {
  b(i) = (a(i-1) + a(i+1))/2;
}
```


Programming Model Examples

- **Data parallel example 2:** “Apply 3-pt stencil to vector”

global-view

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

forall i in (2..n-1) {
  b(i) = (a(i-1) + a(i+1))/2;
}
```

SPMD

```
var n: integer = 1000;
var locN: integer = n/numProcs;
var a, b: [0..locN+1] float;
var innerLo: integer = 1;
var innerHi: integer = locN;

if (iHaveRightNeighbor) {
  send(right, a(locN));
  recv(right, a(locN+1));
} else {
  innerHi = locN-1;
}
if (iHaveLeftNeighbor)
  send(left, a(1));
  recv(left, a(0));
} else {
  innerLo = 2;
}
forall i in (innerLo..innerHi) {
  b(i) = (a(i-1) + a(i+1))/2;
}
```

Programming Model Examples

- **Data parallel example 2:** “Apply 3-pt stencil to vector”

SPMD (pseudocode + MPI)

```
var n: integer = 1000, locN: integer = n/numProcs;
var a, b: [0..locN+1] float;
var innerLo: integer = 1, innerHi: integer = locN;
var numProcs, myPE: integer;
var retval: integer;
var status: MPI_Status;

MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
MPI_Comm_rank(MPI_COMM_WORLD, &myPE);
if (myPE < numProcs-1) {
  retval = MPI_Send(&(a(locN)), 1, MPI_FLOAT, myPE+1, 0, MPI_COMM_WORLD);
  if (retval != MPI_SUCCESS) { handleError(retval); }
  retval = MPI_Recv(&(a(locN+1)), 1, MPI_FLOAT, myPE+1, 1, MPI_COMM_WORLD, &status);
  if (retval != MPI_SUCCESS) { handleError(retval); }
} else
  innerHi = locN-1;
if (myPE > 0)
  retval = MPI_Send(&(a(1)), 1, MPI_FLOAT, myPE-1, 1, MPI_COMM_WORLD);
  if (retval != MPI_SUCCESS) { handleError(retval); }
  retval = MPI_Recv(&(a(0)), 1, MPI_FLOAT, myPE-1, 0, MPI_COMM_WORLD, &status);
  if (retval != MPI_SUCCESS) { handleError(retval); }
} else
  innerLo = 2;
forall i in (innerLo..innerHi) {
  b(i) = (a(i-1) + a(i+1))/2;
}
```

Communication becomes geometrically more complex for higher-dimensional arrays

Fortran+MPI Communication for 3D Stencil in NAS MG

Chapel 3D NAS MG Stencil

```
param coeff: domain(1) = [0..3]; // for 4 unique weight values
param Stencil: domain(3) = [-1..1, -1..1, -1..1]; // 27-points

function rprj3(S, R) {
  param w: [coeff] float = (/0.5, 0.25, 0.125, 0.0625/);
  param w3d: [(i,j,k) in Stencil] float
         = w((i!=0) + (j!=0) + (k!=0));
  const SD = S.Domain,
        Rstr = R.stride;

  S = [ijk in SD] sum reduce [off in Stencil]
                            (w3d(off) * R(ijk + Rstr*off));
}
```


Chapel 3D NAS MG Stencil

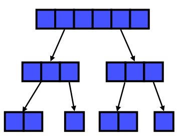
```
param coeff: domain(1) = [0..3]; // for 4 unique weight values
param Stencil: domain(3) = [-1..1, -1..1, -1..1]; // 27-points

function rprj3(S, R) {
  param w: [coeff] float = (/0.5, 0.25, 0.125, 0.0625/);
  param w3d: [(i,j,k) in Stencil] float
         = w((i!=0) + (j!=0) + (k!=0));
  const SD = S.Domain,
        Rstr = R.stride;

  S = [ijk in SD] sum reduce [off in Stencil]
                            (w3d(off) * R(ijk + Rstr*off));
}
```

- Global-view model supports computation better:
  - more concise
  - more general (no constraints on problem size, locale set size)
  - performance need not be sacrificed

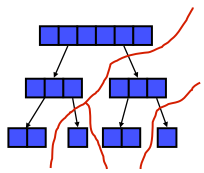
Programming Model Examples

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

global-view



fragmented



Programming Model Examples

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

global-view

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

SPMD

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


Fragmented/SPMD Languages

- Fragmented & SPMD programming models…
  - …obfuscate algorithms by interspersing per-task management details in-line with the computation
    - local bounds, per-task data structures
    - communication, synchronization
  - …provide only a very blunt means of expressing parallelism, distributed data structures
    - run multiple programs simultaneously
    - have each program allocate a piece of the data structure
  - …are our main parallel programmability limiter today
  - …tend to be simpler to implement than global-view languages
    - at minimum, only need a good node compiler
  - …can take the credit for the majority of our parallel application successes to date

Global-View Languages

- Single-processor languages are trivially global-view
  - Matlab, Java, Python, Perl, C, C++, Fortran, …
- Parallel global-view languages have been developed…
  - High Performance Fortran (HPF)
  - ZPL
  - Sisal
  - NESL
  - Cilk
  - Cray MTA extensions to C/Fortran
  - …
- …yet most have not achieved widespread adoption
  - reasons why are as varied as the languages themselves
- Chapel has been designed…
  - …to support global-view programming
  - …with experience from preceding global-view languages

Locality-Aware Languages

- Fragmented/SPMD languages are trivially locality-aware
- A global-view language may also be locality-aware

abstract global-view: data parallel

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

forall i in (2..n-1) {
  b(i) = (a(i-1) + a(i+1))/2;
}
```

locality-aware global-view: data parallel

```
var n: integer = 1000;
var D: domain(1) distributed Block = 1..n;
var InnerD: subdomain(D) = 2..n-1;
var a, b: [D] float;

forall i in InnerD {
  b(i) = (a(i-1) + a(i+1))/2;
}
```

abstract global-view: task parallel

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

locality-aware global-view: task parallel

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


Outline

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

1) Multithreaded Parallel Programming

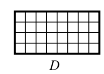
- Virtualization of threads
  - *i.e.*, no fork/join
- Abstractions for data and task parallelism
  - *data:* domains, arrays, iterators, …
  - *task:* cobegins, atomic transactions, sync variables, …
- Composition of parallelism
- Global view of computation, data structures

Data Parallelism: Domains

- *domain:* an index set
  - specifies size and shape of "arrays"
  - supports sequential and parallel iteration
  - potentially decomposed across locales
- 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

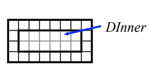
A Simple Domain Declaration

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

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



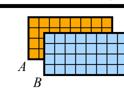
A Simple Domain Declaration

```
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];
```

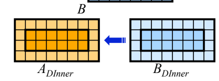


Domain Uses

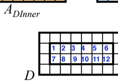
- Declaring arrays:
  ```
  var A, B: [D] float;
  ```



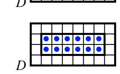
- Sub-array references:
  ```
  A(DInner) = B(DInner);
  ```



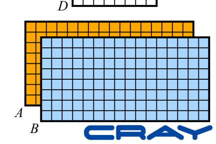
- Sequential iteration:
  ```
  for (i,j) in DInner { …A(i,j)… }
  ```
  or:
  ```
  for ij in DInner { …A(ij)… }
  ```



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

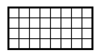


- Array reallocation:
  ```
  D = [1..2*m, 1..2*n];
  ```



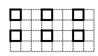
Other Arithmetic Domains

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



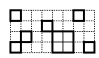
D2

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



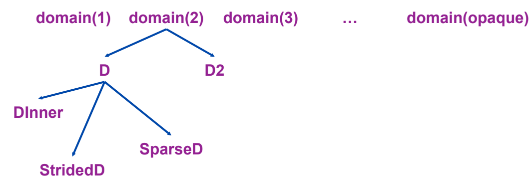
StridedD

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

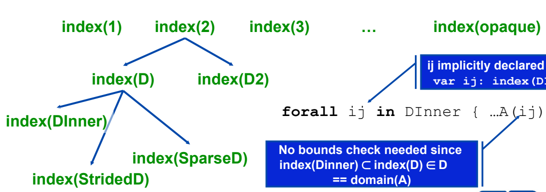


SparseD

The Domain/Index Hierarchy



The Domain/Index Hierarchy



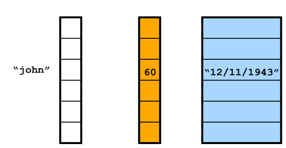
ij implicitly declared as:
`var ij: index(DInner);`

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

No bounds check needed since index(Dinner) ⊂ index(D) ∈ D == domain(A)

Indefinite Domains

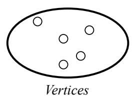
```
var People: domain(string);
var Age: [People] integer;
var Birthdate: [People] string;

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



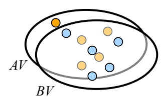
People *Age* *Birthday*

Opaque Domains

```
var Vertices: domain(opaque);

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



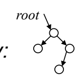

```
var AV, BV: [Vertices] float;
```



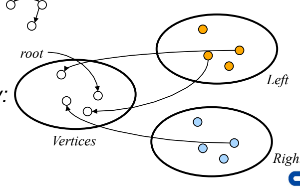
Opaque Domains

```
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:

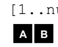
```
const Locales: [1..numLocales] locale;
```

- Users may use this to create their own locale arrays:

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

CompGrid

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



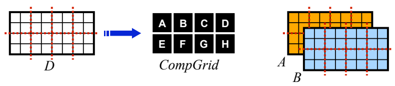
TaskALocs

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

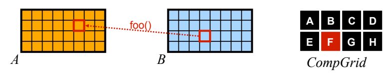
Computation Distribution

- "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 Latent Types

- 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;
}
```

- These concepts result in statically typed code

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

Outline

- ✓ Chapel Motivation & Foundations
  - ✓ Parallel Language Wishlist
  - ✓ Programming Models and Productivity
- ✓ Chapel Overview
- ➢ Wrap-up

Chapel Challenges

- User Acceptance
  - True of any new language
  - Skeptical audience
- Commodity Architecture Implementation
  - 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 many others as well…

What's next?

- HPCS phase III
  - proposals due this spring
  - 1 or 2 vendors expected to be funded for phase III
  - July 2006 – December 2010
- HPCS Language Effort forking off
  - all 3 phase II language teams eligible for phase III
  - High Productivity Language Systems (HPLS) team
    - language experts/enthusiasts from national labs, academia
    - to study, evaluate the vendor languages, report to DARPA
    - July 2006 – December 2007
  - DARPA hopes…
    …that a language consortium will emerge from this effort
    …to involve mainstream computing vendors as well
    …to avoid repeating mistakes of the past (Ada, HPF, …)

Chapel Contributors

- Cray Inc.
  - Brad Chamberlain
  - David Callahan
  - Steven Deitz
  - John Plevyak
  - Shannon Hoffswell
  - Mackale Joyner
- Caltech/JPL:
  - Hans Zima
  - Roxana Diaconescu
  - Mark James

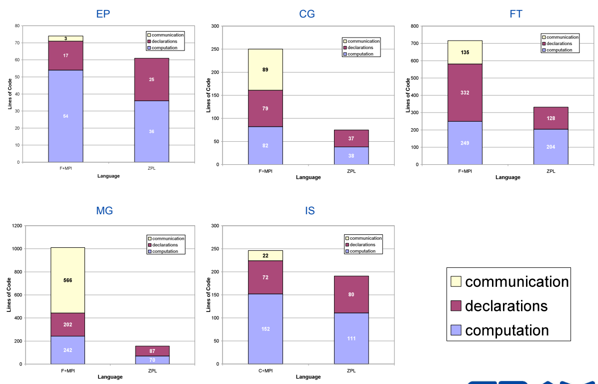
Summary

- Chapel is being designed to…
  …enhance programmer productivity
  …address a wide range of workflows
- Via high-level, extensible abstractions for…
  …global-view 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
  - User feedback desired

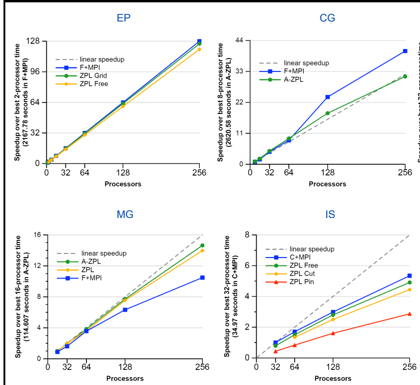
Backup Slides

NPB in ZPL

Compact, High-Level Code…



…need not perform poorly



See also PPoPP'05 results from Rice University for HPF

Chapel is not ZPL or HPF:
- user-defined distributions
- task parallelism
- more flexible array operations
- productivity features

Yet it does build on them…