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

why do our languages fall short in this region?

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

⇒ Areas for Improvement
Programming Models
Base Languages
Interoperability
### 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**

```plaintext
var n: integer = 1000;
var a, b, c: [1..n] float;
forall i in (1..n) {
    c(i) = a(i) + b(i);
}
```

**SPMD**

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

\[
\text{global-view} \quad \text{fragmented} \\
\begin{array}{c}
\text{(}\begin{array}{c}
\text{0} \\
\text{1} \\
\text{2} \\
\text{3}
\end{array}\text{)} / 2 \\
\text{4} \\
\text{5} \\
\text{6} \\
\text{7}
\end{array}
\]

= \begin{array}{c}
\text{(}\begin{array}{c}
\text{0} \\
\text{1} \\
\text{2} \\
\text{3}
\end{array}\text{)} / 2 \\
\text{4} \\
\text{5} \\
\text{6} \\
\text{7}
\end{array}
\]

Programming Model Examples

**Chapel (9)**

```chapel
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;
}
```

**Chapel (10)**

```chapel
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;
}
```

**Chapel (11)**

```chapel
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;
}
```

**Chapel (12)**

```chapel
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;
}
```

Programming Model Examples

**SPMD**

```chapel
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**

```chapel
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;
}
```

Programming Model Examples

**SPMD**

```chapel
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**

```chapel
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;
}
```


**Chapel 3D NAS MG Stencil**

```chapel
param coeff1: 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) { 
  var w: [coeff1] float = [0.5, 0.25, 0.125, 0.0625];
  var w3d: [domain(3)] 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
**Chapel (17)**

**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

---

**Programming Model Examples**

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

  ```
  #! global-view
  for i in range(lo, hi):
    pivot = computePivot(lo, hi, data)
    partition(lo, pivot, hi, data)
    if iHaveParent:
      send(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
  for i in range(lo, hi):
    pivot = computePivot(lo, hi, data)
    partition(lo, pivot, hi, data)
    if iHaveParent:
      send(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)
  ```

---

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

```c
#include "mpi.h"

int main(int argc, char** argv) {
  int i, lo, hi, pivot, nElements;
  int* data; //array of nElements elements

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nElements);
  lo = i % nElements;
  hi = (i + 1) % nElements;

  pivot = computePivot(lo, hi, data);
  partition(lo, pivot, hi, data);
  if (i % nElements == 0) {
    send(MPI_COMM_WORLD, lo, hi, data);
  } else {
    recv(MPI_COMM_WORLD, lo, hi, data);
  }

  MPI_Finalize();
}
```
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;
forall i in [1..n] float:
  b(i) = (a(i-1) + a(i+1))/2;

Locality-aware global-view: data parallel
var b: [1..n] float;
forall i in [1..n] float:
  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);
end

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);
end

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

Outline

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

```chapel
var m: integer = 4;
var n: integer = 8;
var D: domain(2) = [1..m, 1..n];
```

Domain Uses

- Declaring arrays:
  ```chapel
  var A, B: [D] float;
  ```
- Sub-array references:
  ```chapel
  A(DInner) = B(DInner);
  ```
- Sequential iteration:
  ```chapel
  for (i,j) in DInner { …A(i,j)… }
  ```
- Parallel iteration:
  ```chapel
  forall (ij) in DInner { …A(i,j)… }
  ```
- Array reallocation:
  ```chapel
  D = [1..2*m, 1..2*n];
  ```
**Other Arithmetic Domains**

```chapel
var D2: domain(2) = (1,1)..(m,n);
var StridedD: domain(D) = D by (2,3);
var indexList: seq(index(D)) = ...;
var SparseD: sparse domain(D) = indexList;
```

**The Domain/Index Hierarchy**

```
domain(2) D D2
    Dinner
    StridedD
    SparseD

domain(1) domain(3) ...
    D
    D2
```

**Indefinite Domains**

```chapel
var People: domain(string);
var Ages: [People] integer;
var Birthdate: [People] string;
Age("john") = 60;
Birthdate("john") = "12/11/1943";
forall person in People {
    if (Birthdate(person) == today) {
        Ages(person) += 1;
    }
}
```

```
forall ij in Dinner { ...A(ij)... } |
| implicitly declared as:forall i in Dinner { ...A(i)... } |
| No bounds check needed since index(Dinner) ∈ D == domain(A) |
```
Opaque Domains

```chapel
var Vertices: domain(opaque);
for i in [1..5] {
    Vertices.new();
}
var AV, BV: [Vertices] float;
```

Task Parallelism

- co-begins: indicate statements that may run in parallel:
  ```chapel
  computePivot(lo, hi, data);
  cobegin {
    cobegin {
      ComputeTaskA(…);
      Quicksort(lo, pivot, data);
      ComputeTaskB(…);
    }
  }
  ```

- atomic sections: support atomic transactions
  ```chapel
  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
  ```bash
  prompt> myChapelProg -nl=8
  ```
- Chapel programs provided with built-in locale array:
  ```chapel
  const Locales: [1..numLocales] locale;
  ```
- Users may use this to create their own locale arrays:
  ```chapel
  var CompGrid: [1..GridRows, 1..GridCols] locale = …;
  ```
  ```chapel
  var TaskALocs: [1..numTaskALocs] locale = …;
  ```
  ```chapel
  var TaskBLocs: [1..numTaskBLocs] locale = …;
  ```

```bash
A B C D E F G H
```

```chapel
A B C D E F G H
```
Data Distribution

- Domains may be distributed across locales
  
  ```chapel
  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

Computation Distribution

- "on" keyword binds computation to locale(s):
  
  ```chapel
cobegin {
  on TaskALocs do ComputeTaskA(...);
  on TaskBLocs do ComputeTaskB(...);
}
```  

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

  ```chapel
define D = ...;
foreach i, j in D {
  on B(i/2, j*2) do A[i, j] = foo(B(i/2, j*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

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

- Type Query Variables

  ```chapel
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

  ```chapel
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 Pievyak
  - 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
Compact, High-Level Code…

...need not perform poorly

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

Yet it does build on them...

See also PPoPP'05 results from Rice University for HPF.