Chapel
the Cascade High Productivity Language

Brad Chamberlain
Cray Inc.

SC09: Tutorial M04 – 11/16/09
What is Chapel?

- A new parallel language being developed by Cray Inc.
- Part of Cray’s entry in DARPA’s HPCS program

**Main Goal:** Improve programmer productivity
- Improve the *programmability* of parallel computers
- Match or beat the *performance* of current programming models
- Provide better *portability* than current programming models
- Improve *robustness* of parallel codes

**Target architectures:**
- multicore desktop machines
- clusters of commodity processors
- Cray architectures
- systems from other vendors

- A work in progress
Chapel’s Setting: HPCS

**HPCS:** High *Productivity* Computing Systems (DARPA *et al.*)

- **Goal:** Raise productivity of high-end computing users by 10×
- **Productivity** = Performance
  + Programmability
  + Portability
  + Robustness

**Phase II:** Cray, IBM, Sun (July 2003 – June 2006)

- Evaluated the entire system architecture’s impact on productivity…
  - processors, memory, network, I/O, OS, runtime, compilers, tools, …
  - …and new languages:
    - Cray: Chapel
    - IBM: X10
    - Sun: Fortress

**Phase III:** Cray, IBM (July 2006 – 2010)

- Implement the systems and technologies resulting from phase II
- (Sun also continues work on Fortress, without HPCS funding)
Chapel: Motivating Themes

1) general parallel programming
2) global-view abstractions
3) multiresolution design
4) control of locality/affinity
5) reduce gap between mainstream & parallel languages
Outline

✓ Chapel Context

➢ Chapel Themes

☐ Language Overview

☐ Status, Community, Future Work
1) General Parallel Programming

- **General software parallelism**
  - *Algorithms*: should be able to express any that comes to mind
    - should never hit a limitation requiring the user to return to MPI
  - *Styles*: data-parallel, task-parallel, concurrent algorithms
    - as well as the ability to compose these naturally
  - *Levels*: module-level, function-level, loop-level, statement-level, …

- **General hardware parallelism**
  - *Types*: multicore, clusters, HPC systems
  - *Levels*: inter-machine, inter-node, inter-core, vectors, multithreading
2) Global-view vs. Fragmented

Problem: “Apply 3-pt stencil to vector”

\[
\text{global-view} = \left( \begin{array}{cccc}
\text{blue} & \text{blue} & \text{blue} & \text{blue} \\
\text{blue} & \text{blue} & \text{blue} & \text{blue} \\
\end{array} \right) / 2
\]

\[
\text{fragmented} = \begin{array}{cccc}
\text{yellow} & \text{yellow} & \text{yellow} & \text{yellow} \\
\text{yellow} & \text{yellow} & \text{yellow} & \text{yellow} \\
\end{array}
\]
2) Global-view vs. Fragmented

**Problem:** “Apply 3-pt stencil to vector”

- **global-view**
  - \( \frac{1}{2} \) + \( \frac{1}{2} \)
  - \( = \) 

- **fragmented**
  - \( \frac{1}{2} \) + \( \frac{1}{2} \)
  - \( = \) 

\[ \begin{align*}
\text{global-view} & \quad ( \frac{1}{2} \quad + \quad \frac{1}{2} ) / 2 \\
& \quad \frac{1}{2} \quad + \quad \frac{1}{2} \\
& \quad = \quad \frac{1}{2} \quad + \quad \frac{1}{2} \\
\end{align*} \]
2) Global-view vs. SPMD Code

**Problem:** “Apply 3-pt stencil to vector”

**global-view**

```chapel
def main() {
    var n: int = 1000;
    var a, b: [1..n] real;

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

**SPMD**

```python
def main() {
    var n: int = 1000;
    var locN: int = n/numProcs;
    var a, b: [0..locN+1] real;

    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;
    }
}
```
2) Global-view vs. SPMD Code

Problem: “Apply 3-pt stencil to vector”

Assumes `numProcs` divides `n`; a more general version would require additional effort
2) SPMD pseudo-code + MPI

Problem: “Apply 3-pt stencil to vector”

```
var n: int = 1000, locN: int = n/numProcs;
var a, b: [0..locN+1] real;
var innerLo: int = 1, innerHi: int = locN;
var numProcs, myPE: int;
var retval: int;
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) { handleErrorWithStatus(retval, status); }
} 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(1)), 1, MPI_FLOAT, myPE-1, 0, MPI_COMM_WORLD, &status);
    if (retval != MPI_SUCCESS) { handleErrorWithStatus(retval, status); }
} 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
2) $rprj3$ stencil from NAS MG

\[ = w_0 \]
\[ = w_1 \]
\[ = w_2 \]
\[ = w_3 \]
2) NAS MG $rprj3$ stencil in Fortran + MPI

```
subroutine com3(x,n1,n2,n3,kk)

use caf_intrinsics
implicit none

include 'globals.h'
integer axis, dir, n1, n2, n3, k, ierr

include 'cafnpb.h'

double precision u(n1, n2, n3)
integer i3, i2, i1, buff_len, buff_id

contains

side() then
  do i3 = 1, n3
    buff_len = buff_len + 1
  enddo

  buff(buff_len, buff_id) = u(i1, i2, i3)
  buff_len = buff_len + 1
  do i2 = 1, n2
    buff_len = buff_len + 1
    buff(buff_len, buff_id) = u(i1, i2, i3)
  enddo

  buff(buff_len, buff_id) = u(i1, i2, i3)
  buff_len = buff_len + 1
end subroutine com3
```

Chapel (13)
Our previous work in ZPL showed that compact, global-view codes like these can result in performance that matches or beats hand-coded Fortran+MPI while also supporting more runtime flexibility.
2) Classifying HPC Programming Notations

- **communication libraries:**
  - MPI, MPI-2
  - SHMEM, ARMCI, GASNet
  - data / control fragmented / fragmented/SPMD fragmented / SPMD

- **shared memory models:**
  - OpenMP, pthreads
  - global-view / global-view (trivially)

- **PGAS languages:**
  - Co-Array Fortran
  - UPC
  - Titanium
  - fragmented / SPMD
global-view / SPMD
fragmented / SPMD

- **HPCS languages:**
  - Chapel
  - X10 (IBM)
  - Fortress (Sun)
  - global-view / global-view
global-view / global-view
global-view / global-view
3) Multiresolution Languages: Motivation

Two typical camps of parallel language design:
low-level vs. high-level

“Why is everything so painful?”

“Why do my hands feel tied?”
3) Multiresolution Language Design

Our Approach: Structure the language in a layered manner, permitting it to be used at multiple levels as required/desired

- provide high-level features and automation for convenience
- provide the ability to drop down to lower, more manual levels
- use appropriate separation of concerns to keep these layers clean

Diagram:

- Distributions
- Data parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target Machine

language concepts
4) Ability to Tune for Locality/Affinity

- Large-scale systems tend to locate memory with processors
  - a good approach for building scalable parallel systems

- Remote accesses tend to be significantly more expensive than local

- Therefore, placement of data relative to computation matters for scalable performance
  ⇒ programmer should have control over placement of data, tasks

- As multicore chips grow in #cores, locality likely to become more important in mainstream parallel programming as well
  - GPUs/accelerators are another case where locality matters
5) Support for Modern Language Concepts

- students graduating with training in Java, Matlab, Perl, C#
- HPC community mired in Fortran, C (maybe C++) and MPI
- we’d like to narrow this gulf
  - leverage advances in modern language design
  - better utilize the skills of the entry-level workforce…
    …while not ostracizing traditional HPC programmers

- examples:
  - build on an imperative, block-structured language design
  - support object-oriented programming, but make its use optional
  - support for static type inference, generic programming to support…
    …exploratory programming as in scripting languages
  - code reuse
Outline

✓ Chapel Context
✓ Chapel Themes

➢ Language Overview
  ➢ Base Language
    ❑ Task Parallelism
    ❑ Data Parallelism
    ❑ Locality and Distributions

❑ Status, Community, Future Work
Block-structured, imperative programming

Intentionally not an extension to an existing language

Instead, select attractive features from others:

- **ZPL, HPF**: data parallelism, index sets, distributed arrays
  (see also APL, NESL, Fortran90)
- **Cray MTA C/Fortran**: task parallelism, lightweight synchronization
- **CLU**: iterators (see also Ruby, Python, C#)
- **ML**: latent types (see also Scala, Matlab, Perl, Python, C#)
- **Java, C#**: OOP, type safety
- **C++**: generic programming/templates (without adopting its syntax)
- **C, Modula, Ada**: syntax

Follow lead of C family of languages when useful
(C, Java, C#, Perl, …)
**Base Language: My Favorite Departures**

- **Rich compile-time language**
  - parameter values (compile-time constants)
  - folded conditionals, unrolled for loops, expanded tuples
  - type and parameter functions – evaluated at compile-time

- **Latent types**
  - ability to omit type specifications for convenience or code reuse
  - type specifications can be omitted from…
    - variables (inferred from initializers)
    - class members (inferred from constructors)
    - function arguments (inferred from callsite)
    - function return types (inferred from return statements)

- **Configuration variables** (and parameters)
  
  ```
  config const n = 100;  // override with ./a.out --n=1000000
  ```

- **Tuples**

- **Iterators** (in the CLU, Ruby sense)

- **Declaration Syntax** (more like Pascal/Modula than C)
Task Parallelism: Task Creation

** begin: creates a task for future evaluation **

begin DoThisTask();
WhileContinuing();
TheOriginalThread();

** sync: waits on all begins created within a dynamic scope **

sync {
  begin treeSearch(root);
}

def treeSearch(node) {
  if node == nil then return;
  begin treeSearch(node.right);
  begin treeSearch(node.left);
}
Task Parallelism: Structured Tasks

**cobegin**: creates a task per component statement:

```chapel
cobegin {
    computePivot(lo, hi, data);
    cobegin {
        Quicksort(lo, pivot, data);
        Quicksort(pivot, hi, data);
    } // implicit join here
}
```

**coforall**: creates a task per loop iteration

```chapel
coforall e in Edges {
    exploreEdge(e);
} // implicit join here
```
Task Parallelism: Task Coordination

**sync variables**: store full/empty state along with value

```chapel
var result$: sync real;  // result is initially empty
sync {
    begin ... = result$;  // block until full, leave empty
    begin result$ = ...;  // block until empty, leave full
}
result$.readXX();  // read value, leave state unchanged;
                    // other variations also supported
```

**single-assignment variables**: writable once only

```chapel
var result$: single real = begin f();  // result initially empty
    ...  // do some other things
total += result$;  // block until f() has completed
```

**atomic sections**: support transactions against memory

```chapel
atomic {
    newnode.next = insertpt;
    newnode.prev = insertpt.prev;
    insertpt.prev.next = newnode;
    insertpt.prev = newnode;
}
```
Producer/Consumer example

```chapel
var buff$: [0..buffersize-1] sync int;

cobegin {
    producer();
    consumer();
}

def producer() {
    var i = 0;
    for ... {
        i = (i+1) % buffersize;
        buff$(i) = ...;
    }
}

def consumer() {
    var i = 0;
    while {
        i = (i+1) % buffersize;
        ...buff$(i)...;
    }
}
```

Chapel (26)
Data Parallelism: Domains

domain: a first-class index set

```chapel
var m = 4, n = 8;

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

$D$
Data Parallelism: Domains

*domain*: a first-class index set

```
var m = 4, n = 8;
var D: domain(2) = [1..m, 1..n];
var Inner: subdomain(D) = [2..m-1, 2..n-1];
```
Domains: Some Uses

- **Declaring arrays:**
  \[
  \text{var } A, B: \text{ [D]} \text{ real;}
  \]

- **Iteration (sequential or parallel):**
  \[
  \text{for } \text{ij in Inner} \{ \ldots \} \\
  \text{or: forall } \text{ij in Inner} \{ \ldots \} \\
  \text{or: } \ldots
  \]

- **Array Slicing:**
  \[
  A[\text{Inner}] = B[\text{Inner}];
  \]

- **Array reallocation:**
  \[
  D = [1..2*m, 1..2*n];
  \]
Data Parallelism: Domain Types

- **dense**
- **strided**
- **sparse**

- **graphs**

- **associative**
  - “steve”
  - “lee”
  - “sung”
  - “david”
  - “jacob”
  - “albert”
  - “pete”
Data Parallelism: Domain Uses

All domain types can be used to declare arrays…
Data Parallelism: Domain Uses

...to iterate over index sets...

forall ij in StrDom {
    DnsArr(ij) += SpsArr(ij);
}

“steve”
“lee”
“sung”
“david”
“jacob”
“albert”
“pete”
Data Parallelism: Domain Uses

...to slice arrays...

\[ \text{DnsArr}[\text{StrDom}] += \text{SpsArr}[\text{StrDom}] ; \]
Data Parallelism: Domain Uses

...and to reallocate arrays

\[
\text{StrDom} = \text{DnsDom} \text{ by } (2,2); \\
\text{SpsDom} += \text{genEquator}();
\]

---

**Distributions**

**Data Parallelism**

**Task Parallelism**

**Base Language**

**Locality Control**

**Target Machine**

---

“steve”
“lee”
“sung”
“david”
“jacob”
“albert”
“pete”
Locality: Locales

locale: An abstract unit of the target architecture
- supports reasoning about locality
- has capacity for processing and storage
- two threads in a given locale have similar access to a given address
  - addresses in that locale are ~uniformly accessible
  - addresses in other locales are also accessible, but at a price
- locales are defined for a given architecture by a Chapel compiler
  - e.g., a multicore processor or SMP node could be a locale
Locales and Program Startup

- Chapel users specify # locales on executable command-line

  \[\text{prompt} > \text{myChapelProg} \ -\text{nl}=8\]  # run using 8 locales

- Chapel launcher bootstraps program execution:
  - obtains necessary machine resources
    - e.g., requests 8 nodes from the job scheduler
  - loads a copy of the executable onto the machine resources
  - starts running the program. \textit{Conceptually}…

    …locale #0 starts running program’s entry point (\texttt{main()})
    …other locales wait for work to arrive
Locale Variables

Built-in variables represent a program’s locale set:

```chapel
cfg const numLocales: int; // number of locales
const LocaleSpace = [0..numLocales-1], // locale indices
Locales: [LocaleSpace] locale; // locale values
```

- `numLocales`: 8
- `LocaleSpace`: [0, 1, 2, 3, 4, 5, 6, 7]
- `Locales`: [L0, L1, L2, L3, L4, L5, L6, L7]
Locale Views

Using standard array operations, users can create their own locale views:

```chapel
var TaskALocs = Locales[..numTaskALocs];
var TaskBLocs = Locales[numTaskALocs+1..];
var CompGrid = Locales.reshape( [1..gridRows, 1..gridCols]);
```
Locale Methods

- The locale type supports built-in methods:

```chapel
def locale.id: int;  // index in LocaleSpace
def locale.name: string;  // similar to uname -n
def locale.numCores: int;  // # of processor cores
def locale.physicalMemory(...): ...;  // amount of memory
...
```

- Locales can also be queried:

```
...myvar.locale...  // query the locale where myvar is stored
...here...  // query where the current task is running
```
Locality: Task Placement

**on clauses:** indicate where tasks should execute

Either by naming locales explicitly...

```chapel
cobegin {
  on TaskALocs do computeTaskA(...);
  on TaskBLocs do computeTaskB(...);
  on Locales(0) do computeTaskC(...);
}
```

...or in a data-driven manner:

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

**Distributions:** “Recipes for parallel, distributed arrays”
- help the compiler map from the computation’s global view…

\[ \alpha \cdot \text{MEMORY} \]

…down to the fragmented, per-processor implementation
Domain Distribution

Domains may be distributed across locales

```chapel
var D: domain(2) distributed Block on CompGrid = ...;
```

A distribution implies…

...ownership of the domain’s indices (and its arrays’ elements)
...the default work ownership for operations on the domains/arrays
- e.g., forall loops or promoted operations
Domain Distributions

- Any domain type may be distributed
- Distributions do not affect program semantics
  - only implementation details and therefore performance
Domain Distributions

- Any domain type may be distributed
- Distributions do not affect program semantics
  - only implementation details and therefore performance

- Distributions
  - Data Parallelism
  - Task Parallelism
  - Base Language
  - Locality Control
  - Target Machine
Distributions: Goals & Research

- Advanced users can write their own distributions
  - specified using lower-level language features

- Chapel will provide a standard library of distributions
  - written using the same user-defined distributions concepts

(Pre-print of paper describing user-defined distribution strategy available on request)
Other Features

- zippered and tensor flavors of iteration and promotion
- subdomains and index types to help reason about indices
- reductions and scans (standard or user-defined operators)
Outline

- Chapel Context
- Global-view Programming Models
- Language Overview

- Status, Future Work, Collaborations
The Chapel Team

- Interns
  - Jacob Nelson (‘09 – UW)
  - Albert Sidelnik (‘09 – UIUC)
  - Andy Stone (‘08 – Colorado St)
  - James Dinan (‘07 – Ohio State)
  - Robert Bocchino (‘06 – UIUC)
  - Mackale Joyner (‘05 – Rice)

- Alumni
  - David Callahan
  - Roxana Diaconescu
  - Samuel Figueroa
  - Shannon Hoffswell
  - Mary Beth Hribar
  - Mark James
  - John Plevyak
  - Wayne Wong
  - Hans Zima

Sung-Eun Choi, David Iten, Lee Prokowich, Steve Deitz, Brad Chamberlain
Chapel Work

- Chapel Team’s Focus:
  - specify Chapel syntax and semantics
  - implement open-source prototype compiler for Chapel
  - perform code studies of benchmarks, apps, and libraries in Chapel
  - do community outreach to inform and learn from users/researchers
  - support users of code releases
  - refine language based on all these activities
Chapel and the Community

- **Our philosophy:**
  - Help the parallel community understand what we are doing
  - Develop Chapel as an open-source project
  - Encourage external collaborations
  - Over time, turn language over to the community (if accepted)

- **Goals:**
  - to get feedback that will help make the language more useful
  - to support collaborative research efforts
  - to accelerate the implementation
  - to aid with adoption
Outreach: Active Collaborations

**Notre Dame/ORNL (Peter Kogge, Srinivas Sridharan, Jeff Vetter):**
Asynchronous STM over distributed memory

**UIUC (David Padua, Albert Sidelnik):**
Chapel for hybrid CPU-GPU computing

**OSU (Gagan Agrawal, Bin Ren):**
Data-intensive computing using Chapel’s user-defined reductions

**ORNL (David Bernholdt *et al.*; Steve Poole *et al.*):** Chapel code studies – Fock matrix computations, MADNESS, Sweep3D, coupled models, …

**Berkeley (Dan Bonachea *et al.*):** APGAS over GASNet; collectives

(Your name here?)
Collaboration Opportunities

- memory management policies/mechanisms
- dynamic load balancing: task throttling and stealing
- parallel I/O and checkpointing
- language interoperability
- application studies and performance optimizations
- index/subdomain semantics and optimizations
- targeting different back-ends (LLVM, MS CLR, …)
- runtime compilation
- library support
- tools
  - correctness debugging
  - performance debugging
  - IDE support
  - Chapel interpreter
  - visualizations, algorithm animations

(your ideas here…)
Chapel Release

- **Current release:** v1.0

- **How to get started:**
  1. Download from: [http://sourceforge.net/projects/chapel](http://sourceforge.net/projects/chapel)
  2. Unpack tar.gz file
  3. See top-level README
     - for quick-start instructions
     - for pointers to next steps with the release

- **Your feedback desired!**

- **Remember:** a work-in-progress
  - it’s likely that you will find problems with the implementation
  - this is still a good time to influence the language’s design
Implementation Status

- **Base language:** stable (a few gaps and bugs remain)
- **Task parallel:**
  - stable, multithreaded implementation of tasks, synchronization vars
  - atomic sections are an area of ongoing research with U. Notre Dame
- **Data parallel:**
  - stable multi-threaded data parallelism for dense domains/arrays
  - limited support for multi-threaded data parallelism other domain types
- **Locality:**
  - stable locale types and arrays
  - stable task parallelism across multiple locales
  - initial support for standard distributions
- **Performance:**
  - has received much attention in designing the language
  - yet scant implementation effort to date
Next Steps

- Expand our set of supported distributions
- Continue to improve performance
- Continue to add missing features
- Expand the set of codes that we are studying
- Expand the set of architectures that we are targeting
- Support the public release
- Continue to support collaborations and seek out new ones
Summary

Chapel strives to greatly improve Parallel Productivity through its support for:

...general parallel programming
...global-view abstractions
...control over locality
...multiresolution features
...modern language concepts and themes
For More Information

chapel_info@cray.com

http://chapel.cray.com

http://sourceforge.net/projects/chapel