Chapel: Features

Brad Chamberlain
Cray Inc.

CSEP 524
May 20, 2010

Outline

➢ Language Overview
  ➢ Base Language
  ❑ Task Parallelism
  ❑ Data Parallelism
  ❑ Locality
  ❑ Distributions
Base Language: Design

- 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 Features

- **Rich compile-time language**
  - parameter values (compile-time constants)
  - folded conditionals, unrolled for loops, tuple expansions
  - 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)
  ```sh
cfg n = 100; // override with ./a.out --n=100000
  ```
- **Tuples**
- **Iterators** (in the CLU, Ruby sense, not C++/Java-style)
- **Declaration Syntax**: more like Pascal/Modula/Scala than C
Task Parallelism: Task Creation

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

```plaintext
begin DoThisTask();
    WhileContinuing();
    TheOriginalThread();
```

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

```plaintext
sync {
    begin treeSearch(root);
}
```

```plaintext
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:

```plaintext
computePivot(lo, hi, data);

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

```plaintext
cobegin {
    computeTaskA(...);
    computeTaskB(...);
    computeTaskC(...);
} // implicit join
```

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

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

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

```plaintext
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:** writeable once only

```plaintext
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

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

Producer/Consumer example

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

cobegin {
    producer();
    consumer();
}

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

def consumer() {
    var i = 0;
    while {
        i = (i+1) % buffsize;
        ...buff$(i)...;
    }
}
QuickSort in Chapel

```chapel
def quickSort(arr: [], thresh: int, low: int = arr.domain.low, high: int = arr.domain.high) {
if high - low < 8 {
    bubbleSort(arr, low, high);
} else {
    const pivotVal = findPivot(arr, low, high);
    const pivotLoc = partition(arr, low, high, pivotVal);
    serial thresh <= 0 do cobegin {
        quickSort(arr, thresh-1, low, pivotLoc-1);
        quickSort(arr, thresh-1, pivotLoc+1, high);
    }
}
}
```

Data Parallelism: Domains

**domain**: a first-class index set

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

*domain*: a first-class index set

\[
\begin{align*}
\text{var } & m = 4, \ n = 8; \\
\text{var } & D: \text{domain}(2) = [1..m, 1..n]; \\
\text{var } & \text{Inner: subdomain}(D) = [2..m-1, 2..n-1];
\end{align*}
\]

Domains: Some Uses

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

- Iteration (sequential or parallel):
  \[
  \begin{align*}
  & \text{for } i\ j \text{ in } \text{Inner} \{ \ldots \} \\
  & \text{or: for all } i\ j \text{ in } \text{Inner} \{ \ldots \} \\
  & \text{or: } \ldots
  \end{align*}
  \]

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

- Array reallocation:
  \[
  D = [1..2^m, 1..2^n];
  \]
Forall vs. For vs. Coforall

for loops:
• Use the current task to execute the loop serially

coforall loops:
• Execute the loop using a distinct task per iteration
• Can have synchronization between iterations

forall loops:
• Use some number of tasks between these two extremes
• Must be legally executable by a single task
• How many tasks are used in practice?

Data Parallelism Throttles

--dataParTasksPerLocale=#
• Specify # of tasks to execute forall loops
• Default: number of cores (in current implementation)

--dataParIgnoreRunningTasks=[true|false]
• If false, reduce # of forall tasks by # of running tasks
• Default: true (in current implementation)

--dataParMinGranularity=#
• reduce # of tasks if any task has fewer iterations
• Default: 1 (in current implementation)
Data Parallelism: Domain Types

Chapel supports several domain types...

```chapel
define var OceanSpace = [0..#lat, 0..#long],
      AirSpace = OceanSpace by (2,4),
      IceSpace = sparse subdomain(OceanSpace) = genCaps();
```

```
var OceanSpace = [0..#lat, 0..#long],
      AirSpace = OceanSpace by (2,4),
      IceSpace = sparse subdomain(OceanSpace) = genCaps();
```

```
define var Ocean: [OceanSpace] real,
      Air: [AirSpace] real,
      IceCaps: [IceSpace] real;
```

```
var Ocean: [OceanSpace] real,
      Air: [AirSpace] real,
      IceCaps: [IceSpace] real;
```

```
var Weight: [Vertices] real,
      Age: [People] int;
```

```
var Weight: [Vertices] real,
      Age: [People] int;
```
Data Parallelism: Domain Uses

...to iterate over index sets...

\[
\text{forall } \ ij \ in \ \text{AirSpace} \ do \\
\text{Ocean}(ij) \ += \ \text{IceCaps}(ij);
\]

\[
\text{forall } \ v \ in \ \text{Vertices} \ do \\
\text{Weight}(v) \ = \ \text{numEdges}(v);
\]

\[
\text{forall } \ p \ in \ \text{People} \ do \\
\text{Age}(p) \ += \ 1;
\]

Data Parallelism: Domain Uses

...to slice arrays...

\[
\text{Ocean}[\text{AirSpace}] \ += \ \text{IceCaps}[\text{AirSpace}];
\]

...\text{Vertices}[\text{Interior}]... 

...\text{People}[\text{Interns}]...
Data Parallelism: Domain Uses

...and to reallocate arrays

AirSpace = OceanSpace by (2,2);
IceSpace += genEquator();

Locale: 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:
  
  prompt> myChapelProg -nl=8 # run using 8 locales

  ![Locales Diagram]

- 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. Conceptually...
    - locale #0 starts running program’s entry point (main())
    - other locales wait for work to arrive

Locale Variables

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

```chapel
config 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:

```python
var TaskALocs = Locales[..numTaskALocs];  # L0 L1
var TaskBLocs = Locales[numTaskALocs+1..];  # L2 L3 L4 L5 L6 L7

var CompGrid = Locales.reshape([1..gridRows,  # L0 L1 L2 L3
                                 1..gridCols]);  # L4 L5 L6 L7
```

Locale Methods

- The locale type supports built-in methods:

  ```python
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
...
```

- Locale queries can also be made:

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

_on clauses:_ indicate where statements should execute:

Either by naming locales explicitly...

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

...or in a data-driven manner:

```
const pivot = computePivot(lo, hi, data);
cobegin {
  on data[lo] do Quicksort(lo, pivot, data);
  on data[hi] do Quicksort(pivot+1, hi, data);
}
```

They can also control where data is allocated:

```
var person: Employee;
on Locales(1) do person = new Employee("Brad");
on Locales(2) do var ref2ToPerson = person;
```

Chapel Distributions

_Distributions:_ “Recipes for parallel, distributed arrays”

- help the compiler map from the computation’s global view...

...down to the _fragmented_, per-processor implementation
Domain Distribution

Domains may be distributed across locales

\[
\text{var } D: \text{domain}(2) \text{ dmapped Block(CompGrid, ...) = ...;}
\]

A distribution defines...
...ownership of the domain’s indices (and its arrays’ elements)
...default work ownership for operations on the domains/arrays
  - e.g., forall loops or promoted operations
...memory layout/representation of array elements/domain indices
...implementation of operations on its domains and arrays
  - e.g., accessors, iterators, communication patterns, ...

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: Goals & Research

- Advanced users can write their own distributions
  - specified in Chapel using lower-level language features
- Chapel will provide a standard library of distributions
  - written using the same user-defined distribution mechanism

*(Draft paper describing user-defined distribution strategy available by request)*
The Block Distribution

The Block Distribution maps the indices of a domain in a dense fashion across the target Locales according to the `boundingBox` argument.

```javascript
const Dist = new dmap(new Block(boundingBox=[1..4, 1..8]));
var Dom: domain(2) dmapped Dist = [1..4, 1..8];
```

The Cyclic Distribution

The Cyclic Distribution maps the indices of a domain in a round-robin fashion across the target Locales according to the `startIdx` argument.

```javascript
const Dist = new dmap(new Cyclic(startIdx=(1,1)));
var Dom: domain(2) dmapped Dist = [1..4, 1..8];
```
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)