Scientific discovery, analysis and prediction made possible through high performance computing.
Introduction to the Chapel Programming Language

Lecturer: Bob Torgerson
February 22, 2013
University of Alaska Fairbanks
Disclaimer

Much of the information used in this presentation was heavily borrowed from the Chapel presentation slides available at:

http://chapel.cray.com/

Chapel is a work in progress, so the accuracy and validity of this presentation is subject to changes in the code base beyond Chapel version 1.6.
Contents

• Introduction
• Cray Chapel
• Chapel Basic Syntax
• Chapel Parallelism
• Examples
Introduction

• HPC has been an important aspect in research for many decades
  – Performing calculations that would be otherwise impossible in a feasible amount of time

• Transistor count & by relation computational power has continued to increase
  – Can your problem utilize the increased potential performance?

• The PetaFLOP barrier was broken in 2008
  – What do we have to do to reach 1 ExaFLOP?
Introduction

• Achieving a high amount of performance is no small task
  – Requires knowledge of low level language and use of parallel libraries

• Often writing code optimized for the current architecture
  – Bad when moving to a new architecture

• What happens when you want to run a small subset of your problem?
  – Hard coded parallelism always expects to be able to use the resources specified
Cray Chapel

- **Cray set out to create a language to help solve these problems**
  - Chapel is an emerging parallel programming language

- **Chapel has the overall goal of improving programmer productivity**
  - Allowing the same code to be used on high-end computer clusters or multicore desktop machines

- **Designed to be easy to learn for those who have used other popular languages**

- **Working to improve performance and use the GPU in the future**
Chapel Basic Syntax

• To start things off, we will do the obligatory first step to learning any language

```chapel
writeln(“Hello World!”);
```

• That’s all that it takes! That’s your first line of Chapel!
  - This actually does some of the house keeping for you
Chapel Basic Syntax: Comments & Variables

- **Comments**
  - /* standard C style multi-line */
  - // standard C++ style single-line

- **Identifiers:**
  - Composed of A-Z, a-z, _, $, 0-9
  - Cannot start with 0-9

- **Case-sensitive**

- **Primitive Types**
  - Bool, int, uint, real, imag, complex, string
  - Declared as: var <variable name> [::<type>] [= <init.val>]
    - Either the type or the initial value must be declared
Chapel Basic Syntax:
Type Casts & Constants

- Chapel is able to perform type casts against values & variables
  - Can also declare a type variable to be used for type casts

  ```
  type elementType = int(32);

  3:int(8)
  var degrees: real = 2
  "90":elementType
  ```

- Constants are declared by using the “const” declaration rather than “var”
Chapel Basic Syntax: Param & Config

- **Param** are compile-time constants
  - Must be modified before compiling the code to make changes to the value
- **Config** indicates that this value can be modified using command line flags
  - This can affect params & const variables without modifying the code as well

```plaintext
config param debug = False;
config const dt = 0.01;
config var max: real = 1
$ chpl program.chpl --sdebug=True
$ ./a.out --dt=0.1 --max=10 -nl 1
```
Chapel Basic Syntax: Statement Lists & Console I/O

• A list of statements are placed into a curly bracketed block
  – As in C, if (...) { <statement list> }

• Output is done with write or writeln
  – Both take arguments to print, but writeln appends a linefeed

• Input is done with read or readln
  – Both reads can take either a list of variables to populate with input from the console or a list of types to be read

```chapel
var first, last, address: string;
write("What is your name? ");
read(first);
last = read(string);
writeln("What is your address? ");
readln(address);
```
Chapel Basic Syntax: Tuples & Arrays

- Chapel has two basic data structures: tuples & arrays

- **Tuples can be homogeneous or heterogeneous**
  - Any number of expressions can go into the tuple

- **Arrays can only be homogenous**

```plaintext
var two_tuple: (int, string, real) = (2, "two", 2.0);
var cubes: 3*int = (8, 27, 64);

var arrA: [1..3] int = [1, 2, 3];
var arrB: [1..3][1..5] real; // 2-D array
```
Chapel Basic Syntax: Ranges

- Ranges are used to express a set of integers using a specified stride
  - Ranges are declared using the syntax `[low]..[high]
- There are a number of ways to modify the range of the values

```
1..10 by 2    // 1,3,5,7,9
1..10 by -1   // 10,9,8...
1..10 #4      // 1,2,3,4
1..10[5..]    // 5,6,7,8,9,10
```

```
1.. by 2      // 1,3,5,7,9,...
1.. by 2 align 2    // 2,4,...
1.. by 2 #3    // 1,3,5
1.. #3 by 2    // 1,3
```
Chapel Basic Syntax: For Loops

• For loops are used to loop over a range or iterable expression such as an array
  – The for loop executes the entire body of the block statement serially

• A new variable is declared for the identifier of the loop statement
  – The iterator in the loop e.g. i does not need to be declared before using it in the loop

```chapel
var arrA: [1..2] string = ["FOO", "BAR"];  
for i in 1..2 { write(arrA(i)); } // FOO BAR
for a in arrA { write(a); } // FOO BAR
```
Chapel Basic Syntax: Zipper Iteration

- Zippered iteration allows for multiple distinct indices to be used for directing flow
  - Zippers can use any iterable expression including arrays

```chapel
var arrA: [1..10] real;

for (a, i, j) in zip(arrA, 1..10, 3..30 by 3) do
  a = (j / 10.0) * i;
writeln(A); // 0.3, 1.2, 2.7, ..., 30
```
Chapel Basic Syntax: Control Flow

- Chapel includes the normal control flow statements of other programming languages
  - If, else, while, do { } while, and select statements
- I will not go into depth on these beyond syntax
  - Any book on imperative languages will include these concepts

```plaintext
if (x > 3) {
  // do something
}
else {
  // do something
}

while (x > 3) {
  // do something
}

select key {
  when value1 {...}
  when value2 {...}
  otherwise {...}
}

do {
  // do something
} while (x > 3)
```
Chapel Basic Syntax: Functions

- In Chapel, procedures i.e. functions are declared using the `proc` command
  - A procedure can have any number of arguments and they can have default values specified by the function
- A return type can be specified if function is meant to only return a single type
  - No return type is necessary or returning anything from the function
- All functions are first class
  - Functions can be passed as arguments to other functions

```chapel
proc area(height: real, width: real) : real
{
    return height * width;
}
proc print(value)
{
    writeln("Here is the value: ", value);
}
print(area(2.0, 4.0));
```

```chapel
proc myfunc(x: real = 2.0, y: real = 2.0)
{
    z = x ** (y*x);
    writeln(z);
}
myfunc(3.0, 4.0);
myfunc(1.0);
myfunc(y=3.0);
```
Chapel Basic Syntax: Modules

- **In Chapel, everything is encapsulated inside of a module**
  - Definition of a module allows for that module to be called by another module
    - For convenience, any code declared outside of a module is encapsulated in a module using the file’s name
  - Code inside of a module, outside of a function, are executed immediately upon “use”-ing that module

```chapel
module mod1 {
  use mod2;
  writeln("Starting module 1!");
  proc main() {
    writeln("Running module 1!");
  }
}

module mod2 {
  writeln("Starting module 2!");
}
```
Chapel Basic Syntax: Records and Classes

• Records and classes are two ways in Chapel to provide object-oriented programming
  – A new object is created from a class definition and all other objects that assign that object will gain aliases of class fields
    • Changes to the original object are reflected in the aliasing object
  – A new object created from a record will allow other objects of that type to copy all fields
    • Makes an individual copy of the original object (no changes shared)

class square {
  var width, height: real;
  proc area() { return height*width;}
}

var s1, s2: square;
s1 = new square(width=2.0, height=2.0);
s2 = s1;
s1.height = 3.0;
// s2 is affected, not if it was a record
Chapel Parallelism: Introduction

• There are two forms of parallelism available in Chapel:
  – Data parallelism
  – Task parallelism

• In data parallelism, the parallelism is derived from the data
  – For example, for all elements in an array, do …

• In task parallelism, the parallelism is derived from dividing required work into distinct computational chunks
  – In other languages, this is often referred to as “threads of execution”
Data Parallelism: Domains

- Chapel domains are a way of expressing an n-dimensional first-class index set
  - These domains can be used to specify new arrays or data structures that use these index sets
- Subdomains are also capable of being created from a domain
- We will also discuss optionally making a domain distributed across any number of nodes

```
config const m = 4, n = 4;
var D: domain(2) = {1..m, 1..n};
var Inside: subdomain(D) = {2..m-1, 2..n-1};

var arrA: [D] real;
var arrB: [1..4, 1..4] real; // arrA & arrB equiv.
var arrC: [Inside] real;
```
Data Parallelism: Forall Loops

- **For iterating over all of the values in an iterable structure**
  - Executes the loop’s body once per domain index in parallel
    - Only loops which are able to be executed by one task should use a forall loop
- **For those familiar with OpenMP, forall loops are very similar in design**
  - The loop variable e.g. i takes on the domain index value of the current index
- **Forall loops can also use zippered iteration**

```plaintext
const D = {1..10};
var arrA: [D] real;
for all i in D {
    arrA[i] = i;
}

const DD: domain(2) = {1..5, 1..5};
var arrB: [DD] int;
for all (b, (i, j)) in zip(arrB, DD) {
    b = (i - 1) * j + j;
}
```
Data Parallelism: More about Arrays

• Functions or operations that are expecting a single scalar are overloaded to use arrays or domains
  
  \[
  \sin(\text{arrA}) \approx \forall a \in \text{arrA} \, \text{do} \, \sin(a)
  \]
  
  \[
  2 \cdot \text{arrA} \approx \forall a \in \text{arrA} \, \text{do} \, 2 \cdot a
  \]

• Multiple arguments can also be assigned two arrays or domains
  – They use zippers to accomplish that task

  \[
  \text{pow}(\text{arrA}, \text{arrB}) \approx \forall (a, b) \in \text{zip(\text{arrA}, \text{arrB})} \, \text{do} \, \text{pow}(a, b)
  \]
Task Parallelism: Introduction

- **Everything in Chapel is implemented as a task**
  - When a module is started, the function main starts a task that runs until the program completes
    - Each task can break into multiple tasks

- **In fact, when calling forall, you are actually breaking the loop iterations into individual tasks**
  - The forall has an inherent barrier at the end of the forall loop
    - Allowing all of the tasks to catch up and end before the main loop continues onward

- **As we will see, we can use the task parallelism to be more specific when we want our code to become parallel**
Task Parallelism: Begin

- When we want a new task to start, we can use the command `begin`
  - Begin creates a new task, but it can not be determined in what order that task will be completed
  - The original parent task continues on without waiting for the spawned task to finish

```plaintext
begin writeln("Hello world!");
writeln("Goodbye world!");
```

Possible Outcome #1: Hello World! Goodbye World!
Possible Outcome #2: Goodbye World! Hello World!
Task Parallelism: Sync

- A variable can be set to be used to sync or act as a barrier for a task
  - A variable marked as a sync is considered full if it is initialized and empty otherwise
  - For tasks that must block until a set of conditions are met, this is the perfect way to allow for parallelism without getting non-deterministic results

```plaintext
var mySync$: sync real;
begin mySync$ = compute();
anotherFunction();
useComputedResults(mySync$);

sync {
    compute();
anotherFunction();
}
useComputedResults();
```
Task Parallelism: Cobegin

- Much like the begin statement, the cobegin statement starts a set of new tasks
  - Each new statement in the code block is given its own task
    - The cobegin places a barrier at the end of the code block so that the parent task must wait for all child tasks to finish

```plaintext
cobegin {
  functionOne();
  functionTwo();
  writeln("Running function one & two…");
}
writeln("Function one & two have finished!");
```
Task Parallelism: Coforall

- The coforall statement is nearly identical to the forall statement
  - However, for every iteration of the loop, a new task is created and a barrier is placed at the end of the loop
  - This is where the forall and coforall loops differ. The forall loop can continue with the parent task if not all of the cores are being consumed by the forall loop.

```javascript
coforall i in 1..10 {
    arrA[i] = eachLoopFunction(i);
    eachLoopFunction2(i);
    writeln(eachLoopFunction3(i));
}
```
Task Parallelism: To Loop or Not to Loop?

• For loops are executed using one task
  – Use when the loop MUST be done in serial
  – When performance is not gained by running in parallel

• Forall loops are used when number of tasks will be much less than the number of iterations
  – Use when the loop CAN be run in parallel, but does not have to be

• Coforall loops are used when the number of tasks equals the number of iterations
  – Use when the loop MUST be executed in parallel
  – When the loop has a substantial amount of work to justify creating a task for every loop iteration
Chapel Parallelism: Locales

• Thus far, all of the parallelism provided has taken advantage of a single node
  – Locales is Chapel’s way of defining multiple different nodes on which the job can execute

• Each locale is its own compute area
  – Has its own processors & memory

• A locale has access to all of its local variables
  – Each locale can access any other locale’s variables also, but that comes at the cost of passing a message to gather that information
    • Fortunately (or unfortunately) that task is done for you, and can lead to confusing performance if data locality is not considered
Chapel Parallelism: Locales

- **Before a Chapel binary is run, a number of locales MUST be specified**
  - This gives the binary an idea of how it can spread out onto a given environment
    - The number of nodes requested in a PBS script will almost always be the number of locales to indicate when starting the binary
- **This number of locales is placed into an array containing the Locales**
  - This array can be iterated on and used to access each of the locales from your code
  - Additionally, the number of locales and a domain of the locales is created for each run

```chapel
writeln(“Number of locales: “, numLocales);
var arrA: [LocaleSpace] real;
for loc in Locales do writeln(loc.id)
```
Chapel Parallelism: Locales

- **Locales have a number of built-in operations**
  - `locale.id` // Assigned number for locale
  - `locale.name` // Hostname of locale
  - `locale.numCores` // Number of cores at the locale

```plaintext
writeln("Locale 1 ID: ", Locales[1].id);
writeln("Locale 2 Name: ", Locales[2].name);
writeln("Number of cores in Locale 3: ", Locales[3].numCores);
```
Chapel Parallelism: Locales

- **To perform work on a given locale, you use the “on” keyword**
  - Specifying a locale and using the keyword on will execute the following code block on a remote locale
- **This is a powerful and simple way of getting large amounts of parallelism out of your code**
  - No additional work to setup communication between the nodes is necessary
    - The task will immediately begin working on the specified locale.

```chapel
on Locales[1] do writeln("Locale 1’s ID = ", here.id);
on Locales[2] do writeln("Now I’m on Locale 2!");
writeln("Back on Locale 0, the starting node.");
```

```chapel
coforall loc in Locales {
    on loc {
        writeln("Written once per locale!");
    }
}
```
Chapel Parallelism: Locales

• Finally, a locale can use all of the processors available to it for additional parallelism
  – Once a Locale, specifying a forall or coforall loop will execute the loop body on the cores available to it
  • Begin and cobegin will also utilize the available cores of a locale

```chapel
coforall loc in Locales {
    on loc {
        forall i in 1..loc.numCores {
            writeln("Greetings from core ", i, " of ", numCores, " on locale ", loc.id);
        }
    }
}
```
Chapel Parallelism: Domain Maps

• As previously mentioned, domains have the ability to be specified to be multi-locale
  – This is done by mapping the (sub)domain’s indices and associated array element to the number of locales that are specified

• This was touted heavily as Chapel’s “secret sauce”
  – Allows for almost all of the parallelism to be done with little to no interaction on the programmer’s side
    • As simple as applying an additional code segment and letting the compiler do the rest
Chapel Parallelism: Domain Maps

• **Domain mapping requires that a specified library be used at the beginning of your program**
  - “use BlockDist;”

• **Two arguments must be supplied to the Block domain map declaration**
  - The boundingBox & the target locales for the domain

```chapel
use BlockDist;
config const numIters = 100000;
const WorkSpace = {1..numIters} dmapped Block(boundingBox={1..100000}, targetLocales=Locales);

forall i in WorkSpace do
    writeln("Hello world! From iteration ", i, " of ", numIters, " on locale ", here.id, " of ", numLocales);
```
Examples