# Introduction to Chapel

Jingke Li

Portland State University



## Chapel

A new high-level parallel programming language. Started out as a Cray project with DARPA funding, now is open-source.

- Support multiple parallel programming paradigms
- Modern, comprehensive, and programmer friendly
- Not fully developed, performance is not yet competitive

*Topics (for now):*

- Basic language features
- Data parallel features
- Task parallel features

*Disclaimer:* Some materials are adapted from Cray's tutorials on Chapel.

## The "Hello, World!" Program

- *Ver 1.* Quick, scripting-language style.

```
writeln("Hello, world!");
```

- *Ver 2.* With module declaration.

```
module Hello {
  writeln("Hello, world!");
}
```

Module-level code is executed before program starts.

*Note:* These two versions are exactly equivalent. Chapel compiler uses an inference engine to fill in the missing parts in Ver 1 to convert it to Ver 2.



## The "Hello, World!" Program (cont.)

- *Ver 3.* Full, structured programming style.

```
module Hello {
  proc main() {
    writeln("Hello, world!");
  }
}
```

This version is slightly different from the previous versions — the "main" procedure is executed when program begins running.

## Data Types

- Primitive Types:

  `bool, int, uint, real, image, complex, string`

- Type's bit width can be explicitly specified:

  `int(16), int(32), real(32)`

- Type casts are allowed, but with a different syntax from C:

```
5:int(8)          // store value as int(8) rather than int
"54":int          // convert string to an int
249:complex(64)   // convert int to complex(64)
```

- Every primitive type has a default value.



## Variables and Constants

- Compile-time constant: `param`
- Execution-time constant: `const`
- Execution-time variable: `var`

```
param pi: real = 3.14159;
const debug = true;       // inferred to be bool
var count: int;           // initialized to 0
```

Chapel compiler provides initial value if user does not provide one. It also infers type if the type info is missing.

*Note:* Any of these items can be declared with `config` to make them configurable.

## The "Hello, World!" Program (again)

- *Ver 1a.* With a configurable message.

```
config const message = "Hello, world!";
writeln(message);
```

```
linux> ./hello2
Hello, world!
linux> ./hello2 --message="Hi!"
Hi!
```

However, you'll get a surprising result if you type

```
linux> ./hello2 --message="Hi!!"
```

— because `"!!"` is interpreted as a shell-command.



## Tuples and Arrays

- Tuples provide lightweight grouping of values:

```
var coord: (int, int, int) = (1, 2, 3);
var coord2: 3*int = coord;
var (i, j, k) = coord;
var triple: (int, string, real) = (7, "eight", 9.0);
```

- Arrays have flexible syntax forms:

```
var A[1..3] int = [5,6,7];
var B: [1..3, 1..5] real;
var C: [1..3][1..5] real;
var i = 1, j = 2;
var ij = (i,j);
B[ij] = 1.0;
B(ij) = 2.0;
B[i,j] = 3.0;
B(i,j) = 4.0;
```

## For Loops

- Flexible Syntax — with or without the keyword `do`; different forms of index domain:

```
var A: [1..3] string = ["April", "May", "June"];
for i in 1..3 do
  write(A(i));
for a in A {
  a += ", 2014";
}
write(A);
```

- Compressed nested-loop form:

```
var B: [0..9] real;
for (b,i,j) in zip(B, 1..10, 2..20 by 2) do
  b = j + i/10.0;
writeln(B);
// 2.1 4.2 6.3 8.4 10.5 12.6 14.7 16.8 18.9 21.0
```



## Iterators

- Generalized form of loops; requires multi-threading support.

```
iter fibonacci(n) {
  var current = 0, next = 1;
  for 1..n {
    yield current;     // return current fib number
    current += next;   // create a new fib number
    current <=> next;  // swap the two values
  }
}

for f in fibonacci(7)
  do writeln(f);
```

Chapel supports iterators → modern language.

## Procedures

- Can set parameter's default value

```
proc writeCoord(x: real = 0.0, y: real = 0.0) {
  writeln((x,y));
}
```

- Can supply arguments out of order (like python ☺)

```
writeCoord(2.0);          // (2.0, 0.0)
writeCoord(y=2.0);        // (0.0, 2.0)
writeCoord(y=2.0, 3.0); // (3.0, 2.0)
```

- I/O procedures have flexible parameter forms (look it up).

```
write(expr-list)    // prints an expr list
read(expr-list)     // reads into a param list
read(type-list)     // returns a tuple
```



Very similar to C++ structs and classes

## Records and Classes

One big difference -> records are value based, classes are reference based (unlike in C++)

Both may contain fields and variables.

*The difference:* records are value-based, classes are reference-based. (Main difference)

- Methods without arguments need not use parenthesis

```
class circle { var radius: real; }
proc circle.circumference {
  return 2 * pi * radius;
}
var c1 = new circle(radius=1.0);
writeln(c1.circumference);
```

- Methods can be defined for any type

```
proc int.square() {
  return this**2;
}
writeln(5.square());
```

→ This is cool.



Any function that is applicable to integer value can use this form (Syntactic Sugar) - compiler will convert it back to regular method.

Chapel tried to separate shape and size of an array from the values.
Values are first class ~~sg~~citizens. (can use lambdas)

## Chapel's Data-Parallel Features

Similar to Fortran 90/95, but more user friendly.

- *Array Operations:* (Arrays are very important in Chapel)

use aggregate to init a →

```
var a,b,c: [{1..5}] int; // declaring three int array vars
a = [i in {1..5}] i;     // a = [1,2,3,4,5]
b = 1;                   // b = [1,1,1,1,1]
c = a + b;               // c = [2,3,4,5,6]  (element wise)
writeln(c);
```

- *Forall Loops:*

```
var a: [{1..5}] int = 2; // Can execute in any order
forall i in {1..5} do    // i.e order is not specified.
  a[i] += 1;             //
writeln(a);
```

Loop body instances are intended to be executed in parallel; but they must also be executable sequentially.

Fortran doesn't allow to do that.



## Domains

Index sets are first class objects!

In Chapel, index sets can be manipulated separately from arrays — they are called *domains*. They can be assigned to variables, passed around, and so forth. (In other words, domains are "first-class" objects.)

```
var D = {1..5};          // Declare a domain
var a,b,c: [D] int;      // declaring three int array vars
a = [i in D] i;          // a = [1,2,3,4,5]
b = 1;                   // b = [1,1,1,1,1]
c = a + b;               // c = [2,3,4,5,6]
writeln(c);
```

doesn't have type!

Values on →

```
D = {2..4};              // Now, redefine domain D
writeln(c);              // What happens?
```

shrink the index range



Ways to map distributed memory models

But when you change the base all arrays will be affected (data is not erased)

If domain is longer than an array then default values are used to fill in gaps.

Use more unified sintax for data parallel systems and message passing system.

## Domains (cont.)

Domain serves a fundamental role for data parallelism in Chapel. (Also for message passing Parallelism)

- Arrays are defined in terms of domains. (previous examples)
- Forall are also defined over domains:

```
Var D = {1..5};
var a: [D] int = 2;
forall i in D do
  a[i] += 1;
writeln(a);
```

This allows to

- Domain serves as based for aligning multiple arrays.
- Domain can be distributed over locales to easily transform a data-parallel program into a distributed program.

Two array coallocated in memory? In other languages needs some sort of allignment, But not in Chapel.



## "Hello, world!" (Revisit)

- *Data Parallel Version:*

```
config const numMessages = 100;
forall msg in 1..numMessages do
  writeln("Hello, world! (from iteration ",
          msg, " of ", numMessages, ")");
```

this will run on any systems sequential or not.

Auto partitions for available number of threads

- *Distributed Data Parallel Version:*

```
use CyclicDist;
config const numMessages = 100;
const MessageSpace = {1..numMessages}  // Domain(no type)
  dmapped Cyclic(startIdx=1);
forall msg in MessageSpace do
  writeln("Hello, world! (from iteration ",
          msg, " of ", numMessages, " owned by locale ",
          here.id, " of ", numLocales, ")");
```

You can query thread info



When Compiler sees {} it realises that this is domain
→ Partition domain to pieces
Swap of Domain allows to switch from Data Parallelism to message passing

Very few things are invented by Chapel
Domain → Some regearders
Reduce → Fortran.
Reduce allows to build a tree to save some time (requires multiple threads).

## Reduce and Scan

- Reduce:

```
total = + reduce A;
bigDiff = max reduce [i in Inner] abs(A[i]-B[i]);
(minVal, minLoc) = minloc reduce zip(A, D);
```

- Scan: // instead of producing single result it

```
var A, B, C: [1..5] int;
A = 1;                  // A: 1 1 1 1 1
B = + scan A;           // B: 1 2 3 4 5
B[3] = -B[3];           // B: 1 2 -3 4 5
C = min scan B;         // C: 1 1 -3 -3 -3
```

// produces all partial results as well.



## Chapel's Task-Parallel Features (Addition to Pthreads and Open MP).

*Three Constructs:* (low)

odd

- begin — for creating a dynamic task with an unstructured lifetime
- cobegin — for creating a related set of heterogeneous tasks
- coforall — for creating a fixed or dynamic # of homogeneous tasks

?

For `cobegin` and `coforall`, the created tasks will joint back to the parent.

coforall - data parallel.

? Begin B for created a single thread (like a task).
new process created and detached.



CoBegin → Create new thread and join them after.
forall - no user control about number of threads created.
no? • coforall - creates single thread for each iteration (High overhead)
Coforall → Use OpenMP for directives. Iterations are distributed among threads.

Examples.

So far manageable and easy to understand!

## "Hello, world!" (Again)

- *Task Parallel Version 1:*

doesn't write goodbye, only greeting

```
begin writeln("Hello, world!");
writeln("Good bye!");
```

→ new thread executed
→ executed by this thread

- *Task Parallel Version 2:*

```
cobegin {
  writeln("Hello, world! (1)");
  writeln("Hello, world! (2)");
} and Joined after.
```

One thread created does each thread inside

- *Task Parallel Version 3:*

```
config const numTasks = here.numCores;
coforall tid in 0..#numTasks do
  writeln("Hello, world! (from task " + tid
          + " of " + numTasks + ")");
```

Each of the iterations will be assigned to a separate thread



## Task Synchronization

Makes it a lot harder!

*Four Mechanisms:*

- Sync variables — equivalent to locks (this is the exact purpose)
- Single variables — similar to signals
- Atomic variables — providing atomic accesses (like a critical section)
- Sync statements — similar to waits

## Sync Variables

→ can be any variable of any type with a state attached (the state is {full, empty}.

- Stores full/empty state along with normal value (default to full at initialization)
- A read (by default) blocks until full, leaves empty
- A write (by default) blocks until empty, leaves full

→ can be changed

// like a built in queue

```
var lock$: sync bool;

lock$ = true;
critical();
var lockval = lock$;
```

$ signifies a convention that indicates that the variable is used for sync.

// state will be full

// release the lock by reading // silly artificial read.

*Note:* As a suggested convention, synchronization variables are named with a $ at the end (although it's not required).



## Example: Consumer/Producer with Bounded Buffer

Illustration for sync variable.

queue is a lock itself.

```
var buff$: [0..#buffersize] sync real;

begin producer();
consumer();

proc producer() {
  var i = 0;
  for ... {
    i = (i+1) % buffersize;
    buff$[i] = ...;
  }
}
proc consumer() {
  var i = 0;
  while ... {
    i = (i+1) % buffersize;
    ...buff$[i]...;
  }
}
```

Do round robin writing.

If an element hasn't been read it blocks.

Only each element is synchronized not the entire array.

Otherwise it becomes too strong!

## Single Variables

Can be written only once, and read multiple times.

They serve as a flag.

Used when all threads are waiting for a condition

- Similar to sync variable, but stays full once written

```
var signal$: single real;

begin signal$ = master();
begin worker(signal$);
begin worker(signal$);
```

// reading until signal is filled - blocking

*Note:* The default behavior of synchronization and single variables can be changed by user.

This can be changed



## Atomic Variables

What kind of operations (Must specify which operations included into atomic execution)

- Supports operations on variable atomically (with respect to other tasks)

```
var count: atomic int, done: atomic bool;
proc barrier(numTasks) {
  const myCount = count.fetchAdd(1);
  if (myCount < numTasks) then
    done.waitFor(true);
  else
    done.testAndSet();
}
```

*Atomic Methods:* (These are guaranteed to be atomic!)

| | |
|---|---|
| `read():t` | `compareExchange(old:tnew:t):bool` |
| `write(v:t)` | `waitFor(v:t)` |
| `exchange(v:t):t` | `add(v:t)` |
| `testAndSet()` | `fetchAdd(v:t)` |
| `clear()` | |

## Sync Statements // Like a separate Join.

for example when used with begin() -> "begin" becomes almost "cobegin()".

- Waits for all dynamically-scoped begins to complete

Sync Scope

```
sync {
  for i in 1..numConsumers {
    begin consumer(i);
  }
  producer();
}
```

You can define the entire program as a scope
By doing this you are basically saying to wait for all threads to Join.



## Chapel's Other Features

To be Covered in future:

(To be covered at a future time.)

Support for message-passing programming:

- Domains
- Domain maps
- Locales