HPC Programming Language Chapel: Base Language Overview

Jemmy Hu

SHARCNET/Compute Canada

Dec. 18, 2019

HPC Languages?

DARPA's High Productivity Computing Systems (HPCS) program

- Chapel, by Cray Inc.
- X10, designed specifically for parallel computing using the partitioned global address space (PGAS) model, IBM
- Fortress, by Sun Microsystems

- CAF - Coarray Fortran, an extension of Fortran 95/2003 for parallel processing, implemented in some Fortran compilers such as G95, Open64, Intel
- UPC - Unified Parallel C, an extension of the C programming language designed for high-performance computing on large-scale parallel machines
- Cilk

What is Chapel?

Chapel: A productive parallel programming language

- developed by *Cray Inc.*
- portable & scalable
- open-source & collaborative

Designed around a high-level abstraction and multi-resolution philosophy.
This means that users can incrementally add more detail to their original code prototype.

Goals:

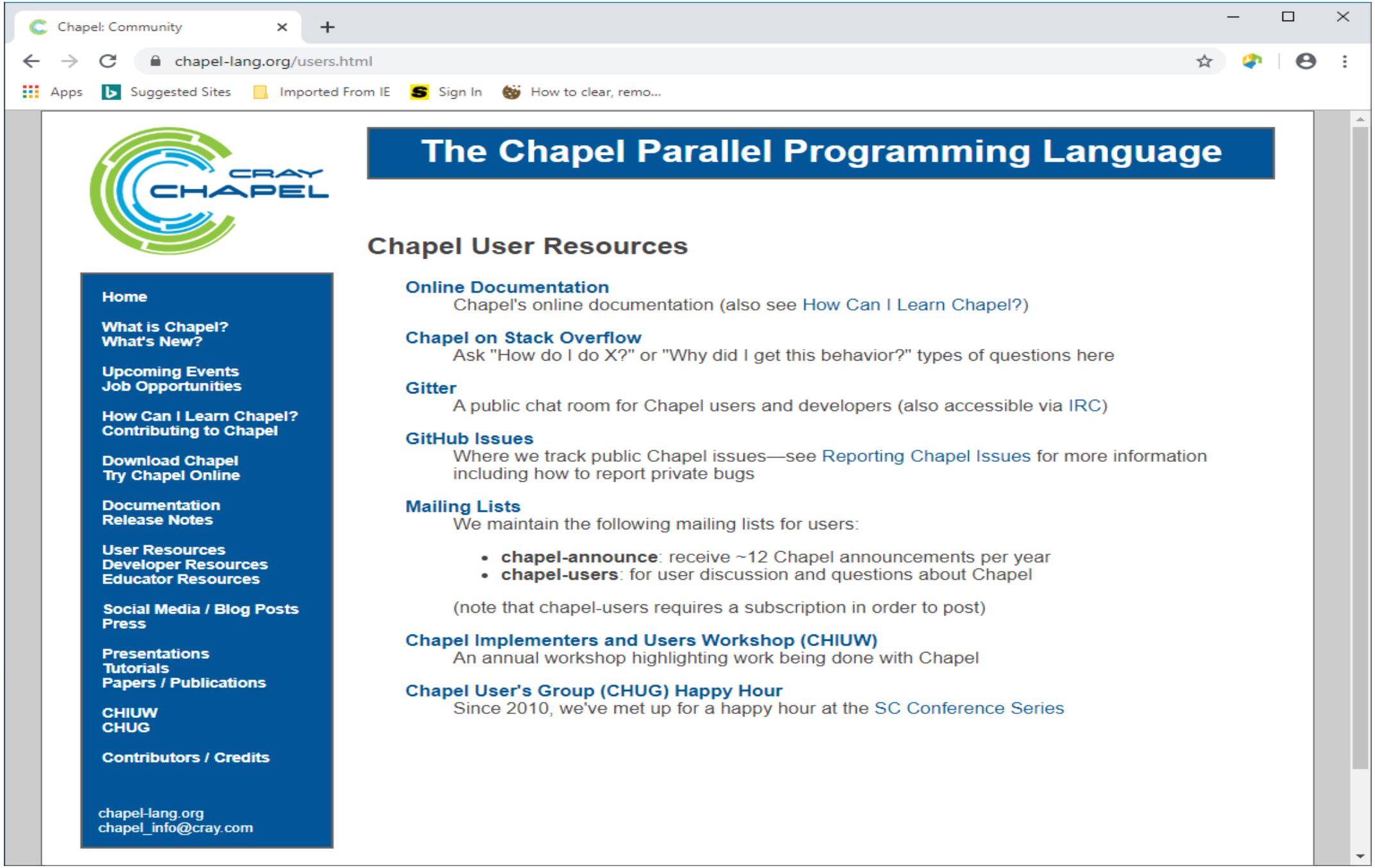
- Support general parallel programming
  - “any parallel algorithm on any parallel hardware”
- Make parallel programming at scale far more productive

Chapel and Productivity

Chapel aims to be as…

- …programmable as Python
- …fast as Fortran
- …scalable as MPI, SHMEM, or UPC
- …portable as C
- …flexible as C++
- …fun as [your favorite programming language]

Write code with the simplicity and readability of scripting languages such as Python and MATLAB, but achieving performance comparable to compiled language like C/C++ and Fortran with the traditional parallel libraries such as MPI and OpenMP


Chapel: Community
chapel-lang.org/users.html
Apps
Suggested Sites
Imported From IE
Sign In
How to clear, remo...
CRAY CHAPEL
The Chapel Parallel Programming Language
Home
What is Chapel?
What's New?
Upcoming Events
Job Opportunities
How Can I Learn Chapel?
Contributing to Chapel
Download Chapel
Try Chapel Online
Documentation
Release Notes
User Resources
Developer Resources
Educator Resources
Social Media / Blog Posts
Press
Presentations
Tutorials
Papers / Publications
CHIUW
CHUG
Contributors / Credits
chapel-lang.org
chapel_info@cray.com
Chapel User Resources
Online Documentation
Chapel's online documentation (also see How Can I Learn Chapel?)
Chapel on Stack Overflow
Ask "How do I do X?" or "Why did I get this behavior?" types of questions here
Gitter
A public chat room for Chapel users and developers (also accessible via IRC)
GitHub Issues
Where we track public Chapel issues—see Reporting Chapel Issues for more information including how to report private bugs
Mailing Lists
We maintain the following mailing lists for users:
chapel-announce: receive ~12 Chapel announcements per year
chapel-users: for user discussion and questions about Chapel
(note that chapel-users requires a subscription in order to post)
Chapel Implementers and Users Workshop (CHIUW)
An annual workshop highlighting work being done with Chapel
Chapel User's Group (CHUG) Happy Hour
Since 2010, we've met up for a happy hour at the SC Conference Series

Chapel compiler

Chapel is a compiled language, Chapel source code must be compiled to generate a binary or executable to be run on the computer.

Chapel source code must be written in text files with the extension .chpl.
Chapel compiler command is chpl.

chpl -o hello hello.chpl

chpl --fast -o hello hello.chpl

--fast indicates the compiler to optimise the binary to run as fast as possible in the given architecture.

```
//Chapel hellow.chpl

writeln('Hello World from Chapel!');

writeln('If we can see this, everything works!');
```

```
[jemmyhu@gra-login1 chapel]$ chpl -o hello hello.chpl
[jemmyhu@gra-login1 chapel]$ ./hello
Hello World from Chapel!
If we can see this, everything works!
```

Chapel on CC clusters

On Compute Canada clusters Cedar and Graham we have two versions of Chapel:

one is a single-locale (single-node) Chapel,
chapel-single/1.15.0

the other is a multi-locale (multi-node) Chapel,
chapel-slurm-gasnetrun_ibv/1.15.0

module spider chapel

Run Chapel on a cluster

Interactive job, e.g.,
salloc --time=0:30:0 --ntasks=1 --mem-per-cpu=100 --account=def-guest
./hello

For production jobs, submit a batch script to the queue
sbatch hello.sh

```
[jemmyhu@gra-login1 chapel]$ module load nixpkgs/16.09  gcc/5.4.0

[jemmyhu@gra-login1 chapel]$ module load chapel-single/1.15.0

[jemmyhu@gra-login1 chapel]$ which chpl
/cvmfs/soft.computecanada.ca/easybuild/software/2017/avx2/Compiler/gcc5.4/chapel-single/1.15.0/bin/linux64/chpl

[jemmyhu@gra-login1 chapel]$ chpl -o hello hello.chpl

[jemmyhu@gra-login1 chapel]$ ./hello

Hello World from Chapel!
If we can see this, everything works!
```

Basic syntax

```
// Comments are C-family style

// one line comment
/*
 multi-line comment
*/

// Basic printing

write("Hello, ");
writeln("World!");

// write and writeln can take a list of things to print.
// Each thing is printed right next to the others, so include your spacing!
writeln("There are ", 3, " commas (\",\") in this line of code");

// Different output channels:
stdout.writeln("This goes to standard output, just like plain writeln() does");
stderr.writeln("This goes to standard error");
```

Variables: name, type, value

```
//don't have to be explicitly typed as long as the compiler can figure out the type that it will hold.
var myVar = 10;

// There are a number of basic types.
var myInt: int = -1000; // Signed ints
var myUint: uint = 1234; // Unsigned ints
var myImag: imag = 5.0i; // Imaginary numbers
var myCplx: complex = 10 + 9i; // Complex numbers
myCplx = myInt + myImag; // Another way to form complex numbers
var myBool: bool = false; // Booleans
var myStr: string = "Some string..."; // Strings

// Typecasting
var intFromReal = myReal : int;
var intFromReal2: int = myReal : int;

// Type aliasing
type RGBColor = 3*chroma; // Type representing a full color
var black: RGBColor = (0,0,0);
var white: RGBColor = (255, 255, 255);
```

const, param, config

```
// A const is a constant, and cannot be changed after set in runtime.
const almostPi: real = 22.0/7.0;

// A param is a constant whose value must be known statically at
// compile-time.
param compileTimeConst: int = 16;

// The config modifier allows values to be set at the command line.
// Set with --varCmdLineArg=Value or --varCmdLineArg Value at runtime.
config var varCmdLineArg: int = -123;
config const constCmdLineArg: int = 777;

// config param can be set at compile-time.
// Set with --set paramCmdLineArg=value at compile-time.
config param paramCmdLineArg: bool = false;
writeln(varCmdLineArg, ", ", constCmdLineArg, ", ", paramCmdLineArg);
```

```
config var a=10;

if a % 3 == 0 {
  writeln(a, " is even divisible by 3.");
} else if a % 3 == 1 {
  writeln(a, " is divided by 3 with a remainder of 1.");
} else {
  writeln(a, " is divided by 3 with a remainder of 2.");
}
```

```
[jemmyhu@gra-login1 Base]$ ./if
10 is divided by 3 with a remainder of 1.
[jemmyhu@gra-login1 Base]$ ./if --a=9
9 is even divisible by 3.
[jemmyhu@gra-login1 Base]$ ./if --a=11
11 is divided by 3 with a remainder of 2.
[jemmyhu@gra-login1 Base]$ ./if --a=12
12 is even divisible by 3.
```

Operators

```
// Math operators:
var a: int, thisInt = 1234, thatInt = 5678;
a = thisInt + thatInt;  // Addition
a = thisInt * thatInt;  // Multiplication

// Logical operators:
var b: bool, thisBool = false, thatBool = true;
b = thisBool && thatBool; // Logical and
b = thisBool || thatBool; // Logical or

// Relational operators:
b = thisInt > thatInt;          // Greater-than

// Bitwise operators:
a = thisInt << 10;     // Left-bit-shift by 10 bits;
a = thatInt >> 5;      // Right-bit-shift by 5 bits;

// Compound assignment operators:
a += thisInt;          // Addition-equals (a = a + thisInt;)
a *= thatInt;          // Times-equals (a = a * thatInt;)
```

```
// Swap operator:
var old_this = thisInt;
var old_that = thatInt;
thisInt <=> thatInt; // Swap the values of thisInt and thatInt
writeln((old_this == thatInt) && (old_that == thisInt));
```

```
// Unlike other C family languages, there are no
// pre/post-increment/decrement operators, such as:
//
// ++j,  --j,  j++,  j--
```

Conditions, Control Flow

```
// if - then - else works just like any other C-family language.
if -1 < 1 then
  writeln("Continuing to believe reality");
else
  writeln("Send mathematician, something is wrong");

// You can use parentheses if you prefer.
if (10 > 100) {
  writeln("Universe broken. Please reboot universe.");
}

config var a=10;

if a % 3 == 0 {
  writeln(a, " is even divisible by 3.");
} else if a % 3 == 1 {
  writeln(a, " is divided by 3 with a remainder of 1.");
} else {
  writeln(a, " is divided by 3 with a remainder of 2.");
}
```

```
// while and do-while loops also behave like their C counterparts.
  var j: int = 1;
  var jSum: int = 0;

  while (j <= 1000) {
    jSum += j;
    j += 1;
  }
  writeln(jSum);

  do {
    jSum += j;
    j += 1;
  } while (j <= 10000);
  writeln(jSum);
```

for loops

// for loops are much like those in Python in that they iterate over a range.
// Ranges (like the 1..10 expression below) are a first-class object in Chapel,
// and as such can be stored in variables.

```
for i in 1..10 do write(i, ", ");
  writeln();

var iSum: int = 0;
  for i in 1..1000 {
    iSum += i;
  }
  writeln(iSum);

for x in 1..10 {
    for y in 1..10 {
       write((x,y), "\t");
    }
    writeln();
}
```

```
[jemmyhu@gra-login1 Base]$ chpl -o for-loop for-loop.chpl
[jemmyhu@gra-login1 Base]$ ./for-loop
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
500500
(1, 1)  (1, 2)  (1, 3)  (1, 4)  (1, 5)  (1, 6)  (1, 7)  (1, 8)  (1, 9)  (1, 10)
(2, 1)  (2, 2)  (2, 3)  (2, 4)  (2, 5)  (2, 6)  (2, 7)  (2, 8)  (2, 9)  (2, 10)
(3, 1)  (3, 2)  (3, 3)  (3, 4)  (3, 5)  (3, 6)  (3, 7)  (3, 8)  (3, 9)  (3, 10)
(4, 1)  (4, 2)  (4, 3)  (4, 4)  (4, 5)  (4, 6)  (4, 7)  (4, 8)  (4, 9)  (4, 10)
(5, 1)  (5, 2)  (5, 3)  (5, 4)  (5, 5)  (5, 6)  (5, 7)  (5, 8)  (5, 9)  (5, 10)
(6, 1)  (6, 2)  (6, 3)  (6, 4)  (6, 5)  (6, 6)  (6, 7)  (6, 8)  (6, 9)  (6, 10)
(7, 1)  (7, 2)  (7, 3)  (7, 4)  (7, 5)  (7, 6)  (7, 7)  (7, 8)  (7, 9)  (7, 10)
(8, 1)  (8, 2)  (8, 3)  (8, 4)  (8, 5)  (8, 6)  (8, 7)  (8, 8)  (8, 9)  (8, 10)
(9, 1)  (9, 2)  (9, 3)  (9, 4)  (9, 5)  (9, 6)  (9, 7)  (9, 8)  (9, 9)  (9, 10)
(10, 1) (10, 2) (10, 3) (10, 4) (10, 5) (10, 6) (10, 7) (10, 8) (10, 9) (10, 10)
```

Ranges

```
// For-loops and arrays both use ranges and domains to define an index set that
// can be iterated over. Ranges are single dimensional integer indices.
// They are first-class citizen types, and can be assigned into variables.
var range1to10: range = 1..10;  // 1, 2, 3, ..., 10
var range2to11 = 2..11; // 2, 3, 4, ..., 11
var rangeThisToThat: range = thisInt..thatInt; // using variables
var rangeEmpty: range = 100..-100; // this is valid but contains no indices

// Ranges can be unbounded.
var range1toInf: range(boundedType=BoundedRangeType.boundedLow) = 1.. ; // 1, 2, 3, 4, 5, ...
var rangeNegInfTo1 = ..1; // ..., -4, -3, -2, -1, 0, 1

// Ranges can be strided (and reversed) using the by operator.
var range2to10by2: range(stridable=true) = 2..10 by 2; // 2, 4, 6, 8, 10
var reverse2to10by2 = 2..10 by -2; // 10, 8, 6, 4, 2

// The end point of a range can be determined using the count (#) operator.
var rangeCount: range = -5..#12; // range from -5 to 6
```

Domains

```
//domains can be multi-dimensional and represent indices of different types.

// Rectangular domains are defined using the same range syntax,
// but they are required to be bounded (unlike ranges).
var domain1to10: domain(1) = {1..10};       // 1D domain from 1..10;
var twoDimensions: domain(2) = {-2..2,0..2}; // 2D domain over product of ranges
var thirdDim: range = 1..16;
var threeDims: domain(3) = {thirdDim, 1..10, 5..10}; // using a range variable

// Domains can also be resized
var resizedDom = {1..10};
writeln("before, resizedDom = ", resizedDom);
resizedDom = {-10..#10};
writeln("after, resizedDom = ", resizedDom);

// Indices can be iterated over as tuples.
for idx in twoDimensions do
  write(idx, ", ");
writeln();
```

```
[jemmyhu@gra-login3 Base]$ chpl -o domain_1 domain_1.chpl
[jemmyhu@gra-login3 Base]$ ./domain_1
before, resizedDom = {1..10}
after, resizedDom = {-10..-1}
(-2, 0), (-2, 1), (-2, 2), (-1, 0), (-1, 1), (-1, 2), (0, 0), (0, 1), (0, 2),
(1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2),
```

```
// Associative domains act like sets.
var stringSet: domain(string); // empty set of strings
stringSet += "a";
stringSet += "b";
stringSet += "c";
stringSet += "a"; // Redundant add "a"
stringSet -= "c"; // Remove "c"
writeln(stringSet.sorted());

// Associative domains can also have a literal syntax
var intSet = {1, 2, 4, 5, 100};

// Both ranges and domains can be sliced to produce a range or domain with the
// intersection of indices.
var rangeA = 1.. ; // range from 1 to infinity
var rangeB =  ..5; // range from negative infinity to 5
var rangeC = rangeA[rangeB]; // resulting range is 1..5
writeln((rangeA, rangeB, rangeC));

var domainA = {1..10, 5..20};
var domainB = {-5..5, 1..10};
var domainC = domainA[domainB];
writeln((domainA, domainB, domainC));
```

```
[jemmyhu@gra-login3 Base]$ chpl -o domains domains.chpl

[jemmyhu@gra-login3 Base]$ ./domains

a b
(1.., ..5, 1..5)
({1..10, 5..20}, {-5..5, 1..10}, {1..5, 5..10})
```

Arrays

```
// Arrays are similar to those of other languages. Their sizes are defined using domains that represent their indices.
var intArray: [1..10] int;
var intArray2: [{1..10}] int; // equivalent

// They can be accessed using either brackets or parentheses
for i in 1..10 do
  intArray[i] = -i;
writeln(intArray);

// We cannot access intArray[0] because it exists outside of the index set, {1..10}, we defined it to have.
// intArray[11] is illegal for the same reason.
var realDomain: domain(2) = {1..5,1..7};
var realArray: [realDomain] real;
var realArray2: [1..5,1..7] real;   // equivalent
var realArray3: [{1..5,1..7}] real; // equivalent

for i in 1..5 {
  for j in realDomain.dim(2) {    // Only use the 2nd dimension of the domain
    realArray[i,j] = -1.61803 * i + 0.5 * j;  // Access using index list
    var idx: 2*int = (i,j);                   // Note: 'index' is a keyword
    realArray[idx] = - realArray[(i,j)];      // Index using tuples
  }
}
// Arrays have domains as members, and can be iterated over as normal.
for idx in realArray.domain {  // Again, idx is a 2*int tuple
  realArray[idx] = 1 / realArray[idx[1], idx[2]]; // Access by tuple and list
}
writeln(realArray);
```

```
[jemmyhu@gra-login3 Base]$ chpl -o arrays arrays.chpl
[jemmyhu@gra-login3 Base]$ ./arrays
-1 -2 -3 -4 -5 -6 -7 -8 -9 -10
0.89443 1.61804 8.47242 -2.61801 -1.13383 -0.723605 -0.531358
0.365489 0.447215 0.576017 0.809022 1.35858 4.23621 -3.78874
0.229669 0.259465 0.298143 0.350374 0.424793 0.539348 0.738503
0.167445 0.182745 0.201121 0.223608 0.251755 0.288008 0.33646
0.13175 0.141041 0.151742 0.1642 0.178886 0.196458 0.217858
```

```
// Associative arrays (dictionaries) can be created using associative domains.
var dictDomain: domain(string) = { "one", "two" };
var dict: [dictDomain] int = ["one" => 1, "two" => 2];
dict["three"] = 3; // Adds 'three' to 'dictDomain' implicitly
for key in dictDomain.sorted() do
  writeln(dict[key]);

// Arrays can be assigned to each other in a few different ways.
var thisArray : [0..5] int = [0,1,2,3,4,5];
var thatArray : [0..5] int;

// First, simply assign one to the other.
thatArray = thisArray;
thatArray[1] = -1;
writeln((thisArray, thatArray));

// Assign a slice from one array to a slice (of the same size) in the other.
thatArray[4..5] = thisArray[1..2];
writeln((thisArray, thatArray));

// Operations can also be promoted to work on arrays. 'thisPlusThat' is also an array.
var thisPlusThat = thisArray + thatArray;
writeln(thisPlusThat);

// Moving on, arrays and loops can also be expressions, where the loop body expression is the result of each iteration.
var arrayFromLoop = for i in 1..10 do i;
writeln(arrayFromLoop);

// An expression can result in nothing, such as when filtering with an if-expression.
var evensOrFives = for i in 1..10 do if (i % 2 == 0 || i % 5 == 0) then i;
writeln(arrayFromLoop);
```

```
[jemmyhu@gra-login3 Base]$ chpl -o array_2 array_2.chpl
[jemmyhu@gra-login3 Base]$ ./array_2
1
3
2
(0 1 2 3 4 5, 0 -1 2 3 4 5)
(0 1 2 3 4 5, 0 -1 2 3 1 2)
0 0 4 6 5 7
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
```

Procedures

```
// Chapel procedures have similar syntax functions in other languages.
proc fibonacci(n : int) : int {
  if n <= 1 then return n;
  return fibonacci(n-1) + fibonacci(n-2);
}

// Input parameters can be untyped to create a generic procedure.
proc doublePrint(thing): void {
  write(thing, " ", thing, "\n");
}

// The return type can be inferred, as long as the compiler can figure it out.
proc addThree(n) {
  return n + 3;
}
doublePrint(addThree(fibonacci(20)));

// It is also possible to take a variable number of parameters.
proc maxOf(x ...?k) {
  // x refers to a tuple of one type, with k elements
  var maximum = x[1];
  for i in 2..k do maximum = if maximum < x[i] then x[i] else maximum;
  return maximum;
}
writeln(maxOf(1, -10, 189, -9071982, 5, 17, 20001, 42));
```

```
[jemmyhu@gra-login3 Base]$ chpl -o procedures procedures.chpl

[jemmyhu@gra-login3 Base]$ ./procedure
6768 6768
20001
```

Iterators

```
// Iterators are sisters to the procedure, and almost everything about procedures also applies to iterators.
// However, instead of returning a single value, iterators may yield multiple values to a loop.

iter oddsThenEvens(N: int): int {
  for i in 1..N by 2 do
    yield i; // yield values instead of returning.
  for i in 2..N by 2 do
    yield i;
}

for i in oddsThenEvens(10) do write(i, ", ");
writeln();

// Iterators can also yield conditionally, the result of which can be nothing
iter canBeNothing(N): int {
  for i in 1..N {
    if N <= i { // Always false, but last index
      yield i;     // Yield last one
    }
  }
}

for i in canBeNothing(10) {
  writeln("Woa there! yielded ", i);
}
```

```
[jemmyhu@gra-login2 Base]$ chpl -o iter iter.chpl
[jemmyhu@gra-login2 Base]$ ./iter
1, 3, 5, 7, 9, 2, 4, 6, 8, 10,
Woa there! yielded 10
```

Classes

```
// Classes are similar to those in C++ and Java, allocated on the heap.
class MyClass {

// Member variables
 var memberInt : int;

// Explicitly defined initializer.
// We also get the compiler-generated initializer, with one argument per field.
proc init(val : real) {
    this.memberInt = ceil(val): int;
  }

// Class methods.
  proc setMemberInt(val: int) {
    this.memberInt = val;
  }

  proc getMemberInt(): int{
    return this.memberInt;
  }

} // end MyClass
```

Modules

```
// Modules are Chapel's way of managing name spaces.
// The files containing these modules do not need to be named after the modules
// (as in Java), but files implicitly name modules.

module OurModule {
// We can use modules inside of other modules. Time is one of the standard modules.
  use Time;

// It is possible to create arbitrarily deep module nests.
// i.e. submodules of OurModule
  module ChildModule {
    proc foo() {
      writeln("ChildModule.foo()");
    }
  }

  module SiblingModule {
    proc foo() {
      writeln("SiblingModule.foo()");
    }
  }
} // end OurModule
```

```
// Using OurModule also uses all the modules it uses.
// Since OurModule uses Time, we also use Time.
use OurModule;

//We can explicitly call foo() through
SiblingModule.foo();
OurModule.ChildModule.foo();
```

Sample code – pi.chpl

```
//Sample serial Chapel code to calculate Pi

const pi = 3.14159265358979323846;

config const n = 1000;

var h, sum = 0.0, i: int;

h = 1.0 / n;

for i in 1..n {

  var x = h * ( i - 0.5 );

  sum += 4.0 / ( 1.0 + x**2);

}

sum *= h;

writef("%.12n   %.6n\n", sum, abs(sum-pi));
```

- **Numerical integration**

$$\int_0^1 \frac{4}{1+x^2} dx = \pi$$

- **Discretization:**

$\Delta = 1/N$: `step = 1/NBIN`

$x_i = (i+0.5)\Delta$ $(i = 0,\ldots,N-1)$

$$\sum_{i=0}^{N-1} \frac{4}{1+x_i^2} \Delta \cong \pi$$

```
[jemmyhu@gra-login3 Base]$ chpl -o pi pi.chpl

[jemmyhu@gra-login3 Base]$ ./pi
3.14159273692    8.33333e-08

[jemmyhu@gra-login3 Base]$ ./pi --n=100000
3.1415926536    8.36842e-12

[jemmyhu@gra-login3 Base]$ ./pi --n=10000000
3.14159265359    6.21725e-14
```

Sample code – Fibonacci_1.chpl

```
/*
* Chapel procedures have similar syntax functions in other languages.
* Fibonacci with procedure
*/

proc fibonacci(n : int) : int {
  if n <= 1 then return n;
  return fibonacci(n-1) + fibonacci(n-2);
}

config const n = 10;

writeln(fibonacci(n));
```

```
[jemmyhu@gra-login2 Base]$ chpl -o fibonacci_1
fibonacci_1.chpl
[jemmyhu@gra-login2 Base]$ ./fibonacci_1
55
[jemmyhu@gra-login2 Base]$ ./fibonacci_1 --n=15
610
```

Sample code – Fibonacci_2.chpl

```
/*
* Chapel iterator can yield multiple values to a loop
* It allows code defining interations to be separate from the loop
* Fibonacci with iterator
*/

iter fibonacci(n){
  var current = 0;
  var next = 1;

  for i in 1..n {
    yield current;
    current += next;
    current <=> next;
  }
}

config const n = 10;

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

```
[jemmyhu@gra-login2 Base]$ chpl -o fibonacci_2
fibonacci_2.chpl
[jemmyhu@gra-login2 Base]$ ./fibonacci_2
0
1
1
2
3
5
8
13
21
34
```

Sample code – Fibonacci_3.chpl

```
/*Chapel iterator can yield multiple values to a loop
* It allows code defining interations to be separate from the loop
* Fibonacci with iterator */

iter fibonacci(n){
  var current = 0;
  var next = 1;

  for i in 1..n {
    yield current;
    current += next;
    current <=> next;
  }
}

config const n = 10;

//Zippered iteration
for (i,f) in zip(0..#n, fibonacci(n)) do
  writeln("fibonacci #", i, " is ", f);
```

```
[jemmyhu@gra-login2 Base]$ chpl -o fibonacci_3
fibonacci_3.chpl
[jemmyhu@gra-login2 Base]$ ./fibonacci_3
fibonacci #0 is 0
fibonacci #1 is 1
fibonacci #2 is 1
fibonacci #3 is 2
fibonacci #4 is 3
fibonacci #5 is 5
fibonacci #6 is 8
fibonacci #7 is 13
fibonacci #8 is 21
fibonacci #9 is 34
```

References

https://chapel-lang.org/

https://learnxinyminutes.com/docs/chapel/

https://hpc-carpentry.github.io/hpc-chapel/01-intro/

https://chapel-lang.org/tmp/ACCU2017/