You can read all about Chapel at Cray's official Chapel website. In short, Chapel is an open-source, high-productivity, parallel-programming language in development at Cray Inc., and is designed to run on multi-core PCs as well as multi-kilocore supercomputers.

More information and support can be found at the bottom of this document.

```
// 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 each other, 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 don't have to be explicitly typed as long as
// the compiler can figure out the type that it will hold.
var myVar = 10; // 10 is an int, so myVar is implicitly an int
myVar = -10;
var mySecondVar = myVar;
// var anError; // this would be a compile-time error.

// We can (and should) explicitly type things
var myThirdVar: real;
var myFourthVar: real = -1.234;
myThirdVar = myFourthVar;

// There are a number of basic types.
var myInt: int = -1000; // Signed ints
var myUint: uint = 1234; // Unsigned ints
var myReal: real = 9.876; // Floating point numbers
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

// Some types can have sizes
var my8Int: int(8) = 10; // 8 bit (one byte) sized int;
var my64Real: real(64) = 1.516; // 64 bit (8 bytes) sized real

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

// consts are constants, they cannot be changed after set in runtime.
const almostPi: real = 22.0/7.0;
```

```
// params are constants whose value must be known statically at compile-time
// Their value cannot be changed.
param compileTimeConst: int = 16;

// The config modifier allows values to be set at the command line
// and is much easier than the usual getOpts debacle
// config vars and consts can be changed through the command line at run time
config var varCmdLineArg: int = -123;
config const constCmdLineArg: int = 777;
// Set with --VarName=Value or --VarName Value at run time

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

// refs operate much like a reference in C++
var actual = 10;
ref refToActual = actual; // refToActual refers to actual
writeln( actual, " == ", refToActual ); // prints the same value
actual = -123; // modify actual (which refToActual refers to)
writeln( actual, " == ", refToActual ); // prints the same value
refToActual = 99999999; // modify what refToActual refers to (which is actual)
writeln( actual, " == ", refToActual ); // prints the same value

// Math operators
var a: int, thisInt = 1234, thatInt = 5678;
a = thisInt + thatInt;  // Addition
a = thisInt * thatInt;  // Multiplication
a = thisInt - thatInt;  // Subtraction
a = thisInt / thatInt;  // Division
a = thisInt ** thatInt; // Exponentiation
a = thisInt % thatInt;  // Remainder (modulo)

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

// Relational Operators
b = thisInt > thatInt;           // Greater-than
b = thisInt >= thatInt;          // Greater-than-or-equal-to
b = thisInt < a && a <= thatInt; // Less-than, and, less-than-or-equal-to
b = thisInt != thatInt;          // Not-equal-to
b = thisInt == thatInt;          // Equal-to

// Bitwise operations
a = thisInt << 10;     // Left-bit-shift by 10 bits;
a = thatInt >> 5;      // Right-bit-shift by 5 bits;
a = ~thisInt;          // Bitwise-negation
a = thisInt ^ thatInt; // Bitwise exclusive-or

// Compound assignment operations
```

```
a += thisInt;          // Addition-equals ( a = a + thisInt;)
a *= thatInt;          // Times-equals ( a = a * thatInt; )
b &&= thatBool;        // Logical-and-equals ( b = b && thatBool; )
a <<= 3;               // Left-bit-shift-equals ( a = a << 10; )
// and many, many more.
// Unlike other C family languages there are no
// pre/post-increment/decrement operators like
//  ++j, --j, j++, j--

// 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) );

// Operator overloads can also be defined, as we'll see with procedures

// Tuples can be of the same type
var sameTup: 2*int = (10,-1);
var sameTup2 = (11, -6);
// or different types
var diffTup: (int,real,complex) = (5, 1.928, myCplx);
var diffTupe2 = ( 7, 5.64, 6.0+1.5i );

// Accessed using array bracket notation
// However, tuples are all 1-indexed
writeln( "(", sameTup[1], ",", sameTup[2], ")" );
writeln( diffTup );

// Tuples can also be written into.
diffTup[1] = -1;

// Can expand tuple values into their own variables
var (tupInt, tupReal, tupCplx) = diffTup;
writeln( diffTup == (tupInt, tupReal, tupCplx) );

// Useful for writing a list of variables ( as is common in debugging)
writeln( (a,b,thisInt,thatInt,thisBool,thatBool) );

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

// If-then-else works just like any other C-family language
if 10 < 100 then
  writeln( "All is well" );

if -1 < 1 then
  writeln( "Continuing to believe reality" );
else
  writeln( "Send mathematician, something's wrong" );
```

```
if ( 10 > 100 ) {
  writeln( "Universe broken. Please reboot universe." );
}

if ( a % 2 == 0 ) {
  writeln( a, " is even." );
} else {
  writeln( a, " is odd." );
}

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( b, " is divided by 3 with a remainder of 2." );
}

// Ternary: if-then-else in a statement
var maximum = if ( thisInt < thatInt ) then thatInt else thisInt;

// Select statements are much like switch statements in other languages
// However, Select statements don't cascade like in C or Java
var inputOption = "anOption";
select( inputOption ){
  when "anOption" do writeln( "Chose 'anOption'" );
  when "otherOption" {
    writeln( "Chose 'otherOption'" );
    writeln( "Which has a body" );
  }
  otherwise {
    writeln( "Any other Input" );
    writeln( "the otherwise case doesn't need a do if the body is one line" );
  }
}

// While and Do-While loops are basically the same in every language.
var j: int = 1;
var jSum: int = 0;
while( j <= 1000 ){
  jSum += j;
  j += 1;
}
writeln( jSum );

// Do-While loop
do{
  jSum += j;
  j += 1;
}while( j <= 10000 );
writeln( jSum );

// For loops are much like those in python in that they iterate over a range.
// Ranges themselves are types, and can be stuffed into variables
// (more about that later)
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( );
}

// Ranges and Domains
// For-loops and arrays both use ranges and domains to
// define an index set that can be iterated over.
// Ranges are single dimensional
// Domains can be multi-dimensional and can
// represent indices of different types as well.
// 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, ...
// Note: the range(boundedType= ... ) is only
// necessary if we explicitly type the variable

var rangeNegInfto1 = ..1; // ..., -4, -3, -2, -1, 0, 1

// Ranges can be strided using the 'by' operator.
var range2to10by2: range(stridable=true) = 2..10 by 2; // 2, 4, 6, 8, 10
// Note: the range(stridable=true) is only
// necessary if we explicitly type the variable

// Use by to create a reverse range
var reverse2to10by2 = 10..2 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

// Can mix operators
var rangeCountBy: range(stridable=true) = -5..#12 by 2; // -5, -3, -1, 1, 3, 5
writeln( rangeCountBy );

// Can query properties of the range
// Print the first index, last index, number of indices,
// stride, and ask if 2 is include in the range
writeln( ( rangeCountBy.first, rangeCountBy.last, rangeCountBy.length,
           rangeCountBy.stride, rangeCountBy.member( 2 ) ) );

for i in rangeCountBy{
  write( i, if i == rangeCountBy.last then "\n" else ", " );
}

// Rectangular domains are defined using the same range syntax
// However 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

// Can iterate over the indices as tuples
for idx in twoDimensions do
  write( idx , ", ");
writeln( );

// or can deconstruct the tuple
for (x,y) in twoDimensions {
  write( "(", x, ", ", y, ")", ", " );
}
writeln( );

// 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 );

// 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) );

// Array 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

// Accessed using bracket notation
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 that we can iterate over
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 );

// Can also iterate over the values of an array
var rSum: real = 0;
for value in realArray {
  rSum += value; // Read a value
  value = rSum;  // Write a value
}
writeln( rSum, "\n", realArray );

// Using associative domains we can create associative arrays (dictionaries)
var dictDomain: domain(string) = { "one", "two" };
var dict: [dictDomain] int = [ "one" => 1, "two" => 2 ];
dict["three"] = 3;
for key in dictDomain do writeln( dict[key] );

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

// Simply assign one to the other.
// This copies thisArray into thatArray, instead of just creating a reference.
// Modifying thisArray does not also modify thatArray.
thatArray = thisArray;
thatArray[1] = -1;
writeln( (thisArray, thatArray) );

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

```
// Operation can also be promoted to work on arrays.
var thisPlusThat = thisArray + thatArray;
writeln( thisPlusThat );

// Arrays and loops can also be expressions, where loop
// body's 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 );

// Or could be written with a bracket notation
// Note: this syntax uses the 'forall' parallel concept discussed later.
var evensOrFivesAgain = [ i in 1..10 ] if (i % 2 == 0 || i % 5 == 0) then i;

// Or over the values of the array
arrayFromLoop = [ value in arrayFromLoop ] value + 1;

// Note: this notation can get somewhat tricky. For example:
// evensOrFives = [ i in 1..10 ] if (i % 2 == 0 || i % 5 == 0) then i;
// would break.
// The reasons for this are explained in depth when discussing zipped iterators.

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

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

// Return type can be inferred (as long as the compiler can figure it out)
proc addThree( n ) {
  return n + 3;
}

doublePrint( addThree( fibonacci( 20 ) ) );

// Can also take 'unlimited' 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 ) );
```

```
// The ? operator is called the query operator, and is used to take
// undetermined values (like tuple or array sizes, and generic types).

// Taking arrays as parameters.
// The query operator is used to determine the domain of A.
// This is important to define the return type (if you wanted to)
proc invertArray( A: [?D] int ): [D] int{
  for a in A do a = -a;
  return A;
}

writeln( invertArray( intArray ) );

// Procedures can have default parameter values, and
// the parameters can be named in the call, even out of order
proc defaultsProc( x: int, y: real = 1.2634 ): (int,real){
  return (x,y);
}

writeln( defaultsProc( 10 ) );
writeln( defaultsProc( x=11 ) );
writeln( defaultsProc( x=12, y=5.432 ) );
writeln( defaultsProc( y=9.876, x=13 ) );

// Intent modifiers on the arguments convey how
// those arguments are passed to the procedure
// in: copy arg in, but not out
// out: copy arg out, but not in
// inout: copy arg in, copy arg out
// ref: pass arg by reference
proc intentsProc( in inarg, out outarg, inout inoutarg, ref refarg ){
  writeln( "Inside Before: ", (inarg, outarg, inoutarg, refarg) );
  inarg = inarg + 100;
  outarg = outarg + 100;
  inoutarg = inoutarg + 100;
  refarg = refarg + 100;
  writeln( "Inside After: ", (inarg, outarg, inoutarg, refarg) );
}

var inVar: int = 1;
var outVar: int = 2;
var inoutVar: int = 3;
var refVar: int = 4;
writeln( "Outside Before: ", (inVar, outVar, inoutVar, refVar) );
intentsProc( inVar, outVar, inoutVar, refVar );
writeln( "Outside After: ", (inVar, outVar, inoutVar, refVar) );

// Similarly we can define intents on the return type
// refElement returns a reference to an element of array
proc refElement( array : [?D] ?T, idx ) ref : T {
  return array[ idx ]; // returns a reference to
}
```

```
var myChangingArray : [1..5] int = [1,2,3,4,5];
writeln( myChangingArray );
// Store reference to element in ref variable
ref refToElem = refElement( myChangingArray, 5 );
writeln( refToElem );
refToElem = -2; // modify reference which modifies actual value in array
writeln( refToElem );
writeln( myChangingArray );
// This makes more practical sense for class methods where references to
// elements in a data-structure are returned via a method or iterator

// We can query the type of arguments to generic procedures
// Here we define a procedure that takes two arguments of
// the same type, yet we don't define what that type is.
proc genericProc( arg1 : ?valueType, arg2 : valueType ): void {
  select( valueType ){
    when int do writeln( arg1, " and ", arg2, " are ints" );
    when real do writeln( arg1, " and ", arg2, " are reals" );
    otherwise writeln( arg1, " and ", arg2, " are somethings!" );
  }
}

genericProc( 1, 2 );
genericProc( 1.2, 2.3 );
genericProc( 1.0+2.0i, 3.0+4.0i );

// We can also enforce a form of polymorphism with the 'where' clause
// This allows the compiler to decide which function to use.
// Note: that means that all information needs to be known at compile-time.
// The param modifier on the arg is used to enforce this constraint.
proc whereProc( param N : int ): void
 where ( N > 0 ) {
  writeln( "N is greater than 0" );
}

proc whereProc( param N : int ): void
 where ( N < 0 ) {
  writeln( "N is less than 0" );
}

whereProc( 10 );
whereProc( -1 );
// whereProc( 0 ) would result in a compiler error because there
// are no functions that satisfy the where clause's condition.
// We could have defined a whereProc without a where clause that would then have
// served as a catch all for all the other cases (of which there is only one).

// Operator definitions are through procedures as well.
// We can define the unary operators:
// + - ! ~
// and the binary operators:
// + - * / % ** == <= >= < > << >> & | ˆ by
// += -= *= /= %= **= &= |= ˆ= <<= >>= <=>
// Boolean exclusive or operator
proc ^( left : bool, right : bool ): bool {
  return (left || right) && !( left && right );
}

writeln( true  ^ true  );
writeln( false ^ true  );
writeln( true  ^ false );
writeln( false ^ false );

// Define a * operator on any two types that returns a tuple of those types
proc *( left : ?ltype, right : ?rtype): ( ltype, rtype ){
  return (left, right );
}

writeln( 1 * "a" ); // Uses our * operator
writeln( 1 * 2 );   // Uses the default * operator

/*
Note: You could break everything if you get careless with your overloads.
This here will break everything. Don't do it.
proc +( left: int, right: int ): int{
  return left - right;
}
*/

// Iterators are a sisters to the procedure, and almost
// everything about procedures also applies to iterators
// However, instead of returning a single value,
// iterators yield many values to a loop.
// This is useful when a complicated set or order of iterations is needed but
// allows the code defining the iterations to be separate from the loop body.
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 absolutelyNothing( N ): int {
  for i in 1..N {
    if ( N < i ) { // Always false
      yield i;     // Yield statement never happens
    }
  }
}

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

```
// We can zipper together two or more iterators (who have the same number
// of iterations)  using zip() to create a single zipped iterator, where each
// iteration of the zipped iterator yields a tuple of one value yielded
// from each iterator.
                                 // Ranges have implicit iterators
for (positive, negative) in zip( 1..5, -5..-1) do
  writeln( (positive, negative) );

// Zipper iteration is quite important in the assignment of arrays,
// slices of arrays, and array/loop expressions.
var fromThatArray : [1..#5] int = [1,2,3,4,5];
var toThisArray : [100..#5] int;

// The operation
toThisArray = fromThatArray;
// is produced through
for (i,j) in zip( toThisArray.domain, fromThatArray.domain) {
  toThisArray[ i ] = fromThatArray[ j ];
}

toThisArray = [ j in -100..#5 ] j;
writeln( toThisArray );
// is produced through
for (i, j) in zip( toThisArray.domain, -100..#5 ){
  toThisArray[i] = j;
}
writeln( toThisArray );

// This is all very important in understanding why the statement
// var iterArray : [1..10] int = [ i in 1..10 ] if ( i % 2 == 1 ) then j;
// exhibits a runtime error.
// Even though the domain of the array and the loop-expression are
// the same size, the body of the expression can be thought of as an iterator.
// Because iterators can yield nothing, that iterator yields a different number
// of things than the domain of the array or loop, which is not allowed.

// Classes are similar to those in C++ and Java.
// They currently lack privatization
class MyClass {
  // Member variables
  var memberInt : int;
  var memberBool : bool = true;

  // Classes have default constructors that don't need to be coded (see below)
  // Our explicitly defined constructor
  proc MyClass( val : real ){
    this.memberInt = ceil( val ): int;
  }

  // Our explicitly defined destructor
  proc ~MyClass( ){
    writeln( "MyClass Destructor called ", (this.memberInt, this.memberBool) );
  }
  // Class methods
  proc setMemberInt( val: int ){
    this.memberInt = val;
  }

  proc setMemberBool( val: bool ){
    this.memberBool = val;
  }

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

  proc getMemberBool( ): bool {
    return this.memberBool;
  }

}

// Construct using default constructor, using default values
var myObject = new MyClass( 10 );
    myObject = new MyClass( memberInt = 10 ); // Equivalent
writeln( myObject.getMemberInt( ) );
// ... using our values
var myDiffObject = new MyClass( -1, true );
    myDiffObject = new MyClass( memberInt = -1,
                                memberBool = true ); // Equivalent
writeln( myDiffObject );

// Construct using written constructor
var myOtherObject = new MyClass( 1.95 );
    myOtherObject = new MyClass( val = 1.95 ); // Equivalent
writeln( myOtherObject.getMemberInt( ) );

// We can define an operator on our class as well but
// the definition has to be outside the class definition
proc +( A : MyClass, B : MyClass) : MyClass {
  return new MyClass( memberInt = A.getMemberInt( ) + B.getMemberInt( ),
                      memberBool = A.getMemberBool( ) || B.getMemberBool( ) );
}

var plusObject = myObject + myDiffObject;
writeln( plusObject );

// Destruction
delete myObject;
delete myDiffObject;
delete myOtherObject;
delete plusObject;

// Classes can inherit from one or more parent classes
class MyChildClass : MyClass {
  var memberComplex: complex;
```

```
}

// Generic Classes
class GenericClass {
  type classType;
  var classDomain: domain(1);
  var classArray: [classDomain] classType;

  // Explicit constructor
  proc GenericClass( type classType, elements : int ){
    this.classDomain = {1..#elements};
  }

  // Copy constructor
  // Note: We still have to put the type as an argument, but we can
  // default to the type of the other object using the query (?) operator
  // Further, we can take advantage of this to allow our copy constructor
  // to copy classes of different types and cast on the fly
  proc GenericClass( other : GenericClass(?otherType),
                     type classType = otherType ) {
    this.classDomain = other.classDomain;
    // Copy and cast
    for idx in this.classDomain do this[ idx ] = other[ idx ] : classType;
  }

  // Define bracket notation on a GenericClass
  // object so it can behave like a normal array
  // i.e. objVar[ i ] or objVar( i )
  proc this( i : int ) ref : classType {
    return this.classArray[ i ];
  }

  // Define an implicit iterator for the class
  // to yield values from the array to a loop
  // i.e. for i in objVar do ....
  iter these( ) ref : classType {
    for i in this.classDomain do
      yield this[i];
  }

}

var realList = new GenericClass( real, 10 );
// We can assign to the member array of the object using the bracket
// notation that we defined ( proc this( i: int ){ ... }  )
for i in realList.classDomain do realList[i] = i + 1.0;
// We can iterate over the values in our list with the iterator
// we defined ( iter these( ){ ... } )
for value in realList do write( value, ", " );
writeln( );

// Make a copy of realList using the copy constructor
var copyList = new GenericClass( realList );
for value in copyList do write( value, ", " );
```

```
writeln( );

// Make a copy of realList and change the type, also using the copy constructor
var copyNewTypeList = new GenericClass( realList, int );
for value in copyNewTypeList do write( value, ", " );
writeln( );

// 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.
// In this case, this file implicitly names the 'learnchapel' module

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

  // We'll use this procedure in the parallelism section.
  proc countdown( seconds: int ){
    for i in 1..seconds by -1 {
      writeln( i );
      sleep( 1 );
    }
  }

  // Submodules of OurModule
  // It is possible to create arbitrarily deep module nests.
  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;

// At this point we have not used ChildModule or SiblingModule so their symbols
// (i.e. foo ) are not available to us.
// However, the module names are, and we can explicitly call foo() through them.
SiblingModule.foo();         // Calls SiblingModule.foo()

// Super explicit naming.
OurModule.ChildModule.foo(); // Calls ChildModule.foo()

use ChildModule;
foo();   // Less explicit call on ChildModule.foo()
```

```
// We can declare a main procedure
// Note: all the code above main still gets executed.
proc main(){

  // Parallelism
  // In other languages, parallelism is typically done with
  // complicated libraries and strange class structure hierarchies.
  // Chapel has it baked right into the language.

  // A begin statement will spin the body of that statement off
  // into one new task.
  // A sync statement will ensure that the progress of the main
  // task will not progress until the children have synced back up.
  sync {
    begin { // Start of new task's body
      var a = 0;
      for i in 1..1000 do a += 1;
      writeln( "Done: ", a);
    } // End of new tasks body
    writeln( "spun off a task!");
  }
  writeln( "Back together" );

  proc printFibb( n: int ){
    writeln( "fibonacci(",n,") = ", fibonacci( n ) );
  }

  // A cobegin statement will spin each statement of the body into one new task
  cobegin {
    printFibb( 20 ); // new task
    printFibb( 10 ); // new task
    printFibb( 5 );  // new task
    {
      // This is a nested statement body and thus is a single statement
      // to the parent statement and is executed by a single task
      writeln( "this gets" );
      writeln( "executed as" );
      writeln( "a whole" );
    }
  }
  // Notice here that the prints from each statement may happen in any order.

  // Coforall loop will create a new task for EACH iteration
  var num_tasks = 10; // Number of tasks we want
  coforall taskID in 1..#num_tasks {
    writeln( "Hello from task# ", taskID );
  }
  // Again we see that prints happen in any order.
  // NOTE! coforall should be used only for creating tasks!
  // Using it to iterating over a structure is very a bad idea!

  // forall loops are another parallel loop, but only create a smaller number
  // of tasks, specifically --dataParTasksPerLocale=number of task
  forall i in 1..100 {
```

```
  write( i, ", ");
}
writeln( );
// Here we see that there are sections that are in order, followed by
// a section that would not follow ( e.g. 1, 2, 3, 7, 8, 9, 4, 5, 6, ).
// This is because each task is taking on a chunk of the range 1..10
// (1..3, 4..6, or 7..9) doing that chunk serially, but each task happens
// in parallel.
// Your results may depend on your machine and configuration

// For both the forall and coforall loops, the execution of the
// parent task will not continue until all the children sync up.

// forall loops are particularly useful for parallel iteration over arrays.
// Lets run an experiment to see how much faster a parallel loop is
use Time; // Import the Time module to use Timer objects
var timer: Timer;
var myBigArray: [{1..4000,1..4000}] real; // Large array we will write into

// Serial Experiment
timer.start( ); // Start timer
for (x,y) in myBigArray.domain { // Serial iteration
  myBigArray[x,y] = (x:real) / (y:real);
}
timer.stop( ); // Stop timer
writeln( "Serial: ", timer.elapsed( ) ); // Print elapsed time
timer.clear( ); // Clear timer for parallel loop

// Parallel Experiment
timer.start( ); // start timer
forall (x,y) in myBigArray.domain { // Parallel iteration
  myBigArray[x,y] = (x:real) / (y:real);
}
timer.stop( ); // Stop timer
writeln( "Parallel: ", timer.elapsed( ) ); // Print elapsed time
timer.clear( );
// You may have noticed that (depending on how many cores you have)
// that the parallel loop went faster than the serial loop

// The bracket style loop-expression described
// much earlier implicitly uses a forall loop.
[ val in myBigArray ] val = 1 / val; // Parallel operation

// Atomic variables, common to many languages, are ones whose operations
// occur uninterrupted. Multiple threads can both modify atomic variables
// and can know that their values are safe.
// Chapel atomic variables can be of type bool, int, uint, and real.
var uranium: atomic int;
uranium.write( 238 );      // atomically write a variable
writeln( uranium.read() ); // atomically read a variable

// operations are described as functions, you could define your own operators.
uranium.sub( 3 ); // atomically subtract a variable
writeln( uranium.read() );
```

```
var replaceWith = 239;
var was = uranium.exchange( replaceWith );
writeln( "uranium was ", was, " but is now ", replaceWith );

var isEqualTo = 235;
if ( uranium.compareExchange( isEqualTo, replaceWith ) ) {
  writeln( "uranium was equal to ", isEqualTo,
           " so replaced value with ", replaceWith );
} else {
  writeln( "uranium was not equal to ", isEqualTo,
           " so value stays the same...  whatever it was" );
}

sync {
  begin { // Reader task
    writeln( "Reader: waiting for uranium to be ", isEqualTo );
    uranium.waitFor( isEqualTo );
    writeln( "Reader: uranium was set (by someone) to ", isEqualTo );
  }

  begin { // Writer task
    writeln( "Writer: will set uranium to the value ", isEqualTo, " in..." );
    countdown( 3 );
    uranium.write( isEqualTo );
  }
}

// sync vars have two states: empty and full.
// If you read an empty variable or write a full variable, you are waited
// until the variable is full or empty again
var someSyncVar$: sync int; // varName$ is a convention not a law.
sync {
  begin { // Reader task
    writeln( "Reader: waiting to read." );
    var read_sync = someSyncVar$;
    writeln( "Reader: value is ", read_sync );
  }

  begin { // Writer task
    writeln( "Writer: will write in..." );
    countdown( 3 );
    someSyncVar$ = 123;
  }
}

// single vars can only be written once. A read on an unwritten single results
// in a wait, but when the variable has a value it can be read indefinitely
var someSingleVar$: single int; // varName$ is a convention not a law.
sync {
  begin { // Reader task
    writeln( "Reader: waiting to read." );
    for i in 1..5 {
      var read_single = someSingleVar$;
      writeln( "Reader: iteration ", i,", and the value is ", read_single );
    }
  }

  begin { // Writer task
    writeln( "Writer: will write in..." );
    countdown( 3 );
    someSingleVar$ = 5; // first and only write ever.
  }
}

// Heres an example of using atomics and a synch variable to create a
// count-down mutex (also known as a multiplexer)
var count: atomic int; // our counter
var lock$: sync bool;   // the mutex lock

count.write( 2 );       // Only let two tasks in at a time.
lock$.writeXF( true );  // Set lock$ to full (unlocked)
// Note: The value doesnt actually matter, just the state
// (full:unlocked / empty:locked)
// Also, writeXF() fills (F) the sync var regardless of its state (X)

coforall task in 1..#5 { // Generate tasks
  // Create a barrier
  do{
    lock$;                 // Read lock$ (wait)
  }while ( count.read() < 1 ); // Keep waiting until a spot opens up

  count.sub(1);          // decrement the counter
  lock$.writeXF( true ); // Set lock$ to full (signal)

  // Actual 'work'
  writeln( "Task #", task, " doing work." );
  sleep( 2 );

  count.add( 1 );        // Increment the counter
  lock$.writeXF( true ); // Set lock$ to full (signal)
}

// we can define the operations + * & | ^ && || min max minloc maxloc
// over an entire array using scans and reductions
// Reductions apply the operation over the entire array and
// result in a single value
var listOfValues: [1..10] int = [15,57,354,36,45,15,456,8,678,2];
var sumOfValues = + reduce listOfValues;
var maxValue = max reduce listOfValues; // 'max' give just max value

// 'maxloc' gives max value and index of the max value
// Note: We have to zip the array and domain together with the zip iterator
var (theMaxValue, idxOfMax) = maxloc reduce zip(listOfValues,
                                                listOfValues.domain);

writeln( (sumOfValues, maxValue, idxOfMax, listOfValues[ idxOfMax ] ) );
```

```
  // Scans apply the operation incrementally and return an array of the
  // value of the operation at that index as it progressed through the
  // array from array.domain.low to array.domain.high
  var runningSumOfValues = + scan listOfValues;
  var maxScan = max scan listOfValues;
  writeln( runningSumOfValues );
  writeln( maxScan );
} // end main()
```

Who is this tutorial for?

This tutorial is for people who want to learn the ropes of chapel without having to hear about what fiber mixture the ropes are, or how they were braided, or how the braid configurations differ between one another. It won't teach you how to develop amazingly performant code, and it's not exhaustive. Refer to the language specification and the module documentation for more details.

Occasionally check back here and on the Chapel site to see if more topics have been added or more tutorials created.

What this tutorial is lacking:

- Exposition of the standard modules
- Multiple Locales (distributed memory system)
- Records
- Parallel iterators

Your input, questions, and discoveries are important to the developers!

The Chapel language is still in-development (version 1.12.0), so there are occasional hiccups with performance and language features. The more information you give the Chapel development team about issues you encounter or features you would like to see, the better the language becomes. Feel free to email the team and other developers through the sourceforge email lists.

If you're really interested in the development of the compiler or contributing to the project, check out the master Github repository. It is under the Apache 2.0 License.

Installing the Compiler

Chapel can be built and installed on your average 'nix machine (and cygwin). Download the latest release version and it's as easy as

1. `tar -xvf chapel-1.12.0.tar.gz`
2. `cd chapel-1.12.0`
3. `make`
4. `source util/setchplenv.bash # or .sh or .csh or .fish`

You will need to `source util/setchplenv.EXT` from within the Chapel directory (`$CHPL_HOME`) every time your terminal starts so it's suggested that you drop that command in a script that will get executed on startup (like .bashrc).

Chapel is easily installed with Brew for OS X

1. `brew update`
2. `brew install chapel`

Compiling Code

Builds like other compilers:

```
chpl myFile.chpl -o myExe
```

Notable arguments:

- `--fast`: enables a number of optimizations and disables array bounds checks. Should only enable when application is stable.
- `--set <Symbol Name>=<Value>`: set config param `<Symbol Name>` to `<Value>` at compile-time.
- `--main-module <Module Name>`: use the main() procedure found in the module `<Module Name>` as the executable's main.
- `--module-dir <Directory>`: includes `<Directory>` in the module search path.