Hewlett Packard Enterprise

ONE-DAY CHAPEL TUTORIAL SESSION 2: CHAPEL BASICS

Chapel Team

October 16, 2023

ONE DAY CHAPEL TUTORIAL

- 9-10:30: Getting started using Chapel for parallel programming
- 10:30-10:45: break
- 10:45-12:15: Chapel basics in the context of the n-body example code
- 12:15-1:15: lunch
- 1:15-2:45: Distributed and shared-memory parallelism especially w/arrays (data parallelism)
- 2:45-3:00: break
- 3:00-4:30: More parallelism including for asynchronous parallelism (task parallelism)
- 4:30-5:00: Wrap-up including gathering further questions from attendees

OUTLINE: CHAPEL BASICS

- Running Example: n-body computation (Hands On)
- Variables, Constants, and Operators
- Records and Classes
- Tuples
- Arrays
- Writing out Tuples, Records, and Arrays (Hands On)
- Main( ) Procedure
- Ranges and basic control flow
- Procedures and iterators
- Where might we parallelize the n-body computation? (Hands On)

RUNNING EXAMPLE: N-BODY COMPUTATION (HANDS ON)

N-BODY IN CHAPEL (WHERE N == 5)

- A serial computation

- From the Computer Language Benchmarks Game
  - Chapel implementation in release under examples/benchmarks/shootout/nbody.chpl

- Computes the influence of 5 bodies on one another
  - The Sun, Jupiter, Saturn, Uranus, Neptune

- Executes for a user-specifiable number of timesteps

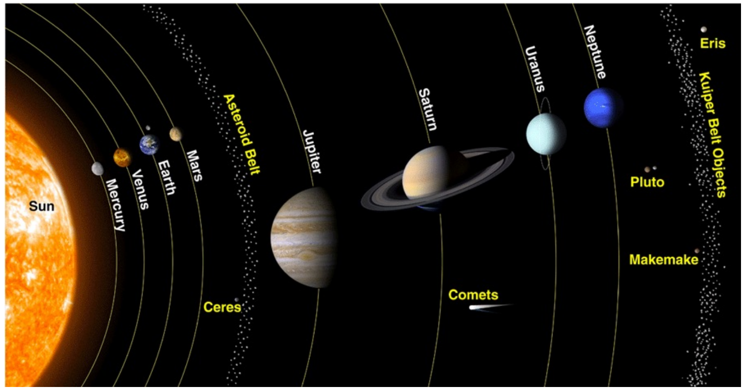

Image source: http://spaceplace.nasa.gov/review/ice-dwarf/solar-system-lrg.png

HANDS ON: COMPILING AND RUNNING N-BODY

nbody.chpl

Things to try

```
chpl nbody.chpl
time ./nbody -nl 1
time ./nbody -nl 1 -n=100000

chpl --fast nbody.chpl
time ./nbody -nl 1
time ./nbody -nl 1 -n=100000
```

```
// number of timesteps to simulate
config const n = 10000;
...
```

Key concepts

- *nix 'time' command is an easy way to see how long a program takes to run
- Compile with '--fast' to have 'chpl' compiler generate faster code

VARIABLES, CONSTANTS, AND OPERATORS

5-BODY IN CHAPEL: VARIABLE AND RECORD DECLARATIONS

nbody.chpl

```
const pi = 3.141592653589793,
      solarMass = 4 * pi**2,
      daysPerYear = 365.24;

config const numsteps = 10000;

record body {
  var pos: 3*real;
  var v: 3*real;
  var mass: real;
}


…
```

Variable declarations

VARIABLES, CONSTANTS, AND PARAMETERS

Basic syntax

```
declaration:
  var   identifier [: type] [= init-expr];
  const identifier [: type] [= init-expr];
  param identifier [: type] [= init-expr];
```

Examples

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

Meaning

- var/const: execution-time variable/constant
- param: compile-time constant
- No init-expr ⇒ initial value is the type's default
- No type ⇒ type is taken from init-expr

PRIMITIVE TYPES

Syntax

Type	Description	Default Value	Currently-Supported Bit Widths	Default Bit Width
bool	logical value	false		impl. dep.
int	signed integer	0	8, 16, 32, 64	64
uint	unsigned integer	0	8, 16, 32, 64	64
real	real floating point	0.0	32, 64	64
imag	imaginary floating point	0.0i	32, 64	64
complex	complex floating points	0.0 + 0.0i	64, 128	128
string	character string	""	N/A	N/A

```
primitive-type:
  type-name [( bit-width )]
```

Examples

```
int(16)    // 16-bit int
real(32)   // 32-bit real
uint       // 64-bit uint
```

CHAPEL'S STATIC TYPE INFERENCE

nbody.chpl

```
const pi = 3.14,                 // pi is a real
      coord = 1.2 + 3.4i,        // coord is a complex…
      coord2 = pi*coord,         // …as is coord2
      name = "brad",             // name is a string
      verbose = false;           // verbose is boolean

proc addem(x, y) {               // addem() has generic arguments
  return x + y;                  //   and an inferred return type
}

var sum = addem(1, pi),                    // sum is a real
    fullname = addem(name, "ford");        // fullname is a string

writeln((sum, fullname));
```

```
(4.14, bradford)
```

BASIC OPERATORS AND PRECEDENCE

<table>
<tr><th>Operator</th><th>Description</th><th>Associativity</th><th>Overloadable</th></tr>
<tr><td>:</td><td>cast</td><td>left</td><td>yes</td></tr>
<tr><td>**</td><td>exponentiation</td><td>right</td><td>yes</td></tr>
<tr><td>! ~</td><td>logical and bitwise negation</td><td>right</td><td>yes</td></tr>
<tr><td>* / %</td><td>multiplication, division and modulus</td><td>left</td><td>yes</td></tr>
<tr><td>(unary) + -</td><td>positive identity and negation</td><td>right</td><td>yes</td></tr>
<tr><td><< >></td><td>shift left and shift right</td><td>left</td><td>yes</td></tr>
<tr><td>&</td><td>bitwise/logical and</td><td>left</td><td>yes</td></tr>
<tr><td>^</td><td>bitwise/logical xor</td><td>left</td><td>yes</td></tr>
<tr><td>|</td><td>bitwise/logical or</td><td>left</td><td>yes</td></tr>
<tr><td>+ -</td><td>addition and subtraction</td><td>left</td><td>yes</td></tr>
<tr><td><= >= < ></td><td>ordered comparison</td><td>left</td><td>yes</td></tr>
<tr><td>== !=</td><td>equality comparison</td><td>left</td><td>yes</td></tr>
<tr><td>&&</td><td>short-circuiting logical and</td><td>left</td><td>via isTrue</td></tr>
<tr><td>||</td><td>short-circuiting logical or</td><td>left</td><td>via isTrue</td></tr>
</table>

5-BODY IN CHAPEL: DECLARATIONS

nbody.chpl

```
const pi = 3.141592653589793,
      solarMass = 4 * pi**2,
      daysPerYear = 365.24;

config const numsteps = 10000;

record body {
  var pos: 3*real;
  var v: 3*real;
  var mass: real;
}


…
```

Variable declarations

5-BODY IN CHAPEL: DECLARATIONS

nbody.chpl

```
const pi = 3.141592653589793,
      solarMass = 4 * pi**2,
      daysPerYear = 365.24;

config const numsteps = 10000;

record body {
  var pos: 3*real;
  var v: 3*real;
  var mass: real;
}


…
```

Configuration Variable

5-BODY IN CHAPEL: DECLARATIONS

nbody.chpl

```
const pi = 3.141592653589793,
      solarMass = 4 * pi**2,
      daysPerYear = 365.24;

config const numsteps = 10000;

record body {
  var pos: 3*real;
  var v: 3*real;
  var mass: real;
}

…
```

Configuration Variable

```
$ ./nbody --numsteps=100
```

CONFIGS

02-configs.chpl

```
param intSize = 32;
type elementType = real(32);
const epsilon = 0.01:elementType;
var start = 1:int(intSize);
```

CONFIGS

02-configs.chpl

```
config param intSize = 32;
config type elementType = real(32);
config const epsilon = 0.01:elementType;
config var start = 1:int(intSize);
```

```
$ chpl 02-configs.chpl -sintSize=64 -selementType=real
$ ./02-configs-start=2 -nl 1 --epsilon=0.00001
```

5-BODY IN CHAPEL: DECLARATIONS

nbody.chpl

```
const pi = 3.141592653589793,
      solarMass = 4 * pi**2,
      daysPerYear = 365.24;

config const numsteps = 10000;

record body {
  var pos: 3*real;
  var v: 3*real;
  var mass: real;
}


…
```

Configuration Variable

5-BODY IN CHAPEL: DECLARATIONS

nbody.chpl

```
const pi = 3.141592653589793,
      solarMass = 4 * pi**2,
      daysPerYear = 365.24;

config const numsteps = 10000;

record body {
  var pos: 3*real;
  var v: 3*real;
  var mass: real;
}


…
```

Record declaration

RECORDS AND CLASSES

RECORDS AND CLASSES

02-records-and-classes.chpl

Chapel's object types

- Contain variable definitions (fields)
- Contain procedure & iterator definitions (methods)
- Records: value-based (e.g., assignment copies fields)
- Classes: reference-based (e.g., assignment aliases object)

Example

```
use Math;
record circle {
  var radius: real;
  proc area() {
    return pi*radius**2;
  }
}
```

```
var c1 = new circle(radius=1.0);
var c2 = c1;      // copies c1
c1.radius = 5.0;
writeln(c2.radius);   // prints 1.0
```

RECORDS AND CLASSES

02-records-and-classes.chpl

Chapel’s object types

- Contain variable definitions (fields)
- Contain procedure & iterator definitions (methods)
- Records: value-based (e.g., assignment copies fields)
- Classes: reference-based (e.g., assignment aliases object)

Example

```
use Math;
class Circle {
  var radius: real;
  proc area() {
    return pi*radius**2;
  }
}
```

```
// c1 is a nilable class
var c1: Circle? = new shared Circle(radius=1.0);
var c2 = c1;              // aliases c1's circle
c1!.radius = 5.0;
writeln(c2!.radius);  // prints 5.0
```

CLASSES VS. RECORDS

Classes

- heap-allocated
  - Variables point to objects
  - Support mem. mgmt. policies
- ‘reference' semantics
  - compiler will only copy pointers
- support inheritance
- support dynamic dispatch
- identity matters most
- similar to Java classes

Records

- allocated in-place
  - Variables are the objects
  - Always freed at end of scope
- 'value' semantics
  - compiler may introduce copies
- no inheritance
- no dynamic dispatch
- value matters most
- similar to C++ structs
  - (sans pointers)

5-BODY IN CHAPEL: DECLARATIONS

nbody.chpl

```
const pi = 3.141592653589793,
      solarMass = 4 * pi**2,
      daysPerYear = 365.24;

config const numsteps = 10000;

record body {
  var pos: 3*real;
  var v: 3*real;
  var mass: real;
}

…
```

Tuple type

OUTLINE: CHAPEL BASICS

- Running Example: n-body computation (Hands On)
- Variables, Constants, and Operators
- Records and Classes
- Tuples
- Arrays
- Writing out Tuples, Records, and Arrays (Hands On)
- Main( ) Procedure
- Ranges and basic control flow
- Procedures and iterators
- Where might we parallelize the n-body computation? (Hands On)

TUPLES (HANDS ON)

TUPLES

02-tuples.chpl

Use

- support lightweight grouping of values
  - e.g., passing/returning multiple procedure arguments at once
  - short vectors
  - multidimensional array indices
- support heterogeneous data types

Examples

```
var coord: (int, int, int) = (1, 2, 3);
var coordCopy: 3*int = coord;
var (i1, i2, i3) = coord;
var triple: (int, string, real) = (7, "eight", 9.0);
```

5-BODY IN CHAPEL: DECLARATIONS

nbody.chpl

```
const pi = 3.141592653589793,
      solarMass = 4 * pi**2,
      daysPerYear = 365.24;

config const numsteps = 10000;

record body {
  var pos: 3*real;
  var v: 3*real;
  var mass: real;
}

…
```

Variable declarations

Configuration Variable

Record declaration

Tuple type

5-BODY IN CHAPEL: THE BODIES

nbody.chpl

```
var bodies =
    [  /* sun */
       new body(mass = solarMass),

       /* jupiter */
       new body(pos = ( 4.84143144246472090e+00,
                       -1.16032004402742839e+00,
                       -1.03622044471123109e-01),
                  v = ( 1.66007664274403694e-03 * daysPerYear,
                        7.69901118419740425e-03 * daysPerYear,
                       -6.90460016972063023e-05 * daysPerYear),
               mass =   9.54791938424326609e-04 * solarMass),

       /* saturn */
       new body(…),

       /* uranus */
       new body(…),

       /* neptune */
       new body(…)
   ];
```

5-BODY IN CHAPEL: THE BODIES

nbody.chpl

```
var bodies =
    [  /* sun */
       new body(mass = solarMass),

       /* jupiter */
       new body(pos = ( 4.84143144246472090e+00,
                       -1.16032004402742839e+00,
                       -1.03622044471123109e-01),
                  v = ( 1.66007664274403694e-03 * daysPerYear,
                        7.69901118419740425e-03 * daysPerYear,
                       -6.90460016972063023e-05 * daysPerYear),
               mass =   9.54791938424326609e-04 * solarMass),

       /* saturn */
       new body(…),

       /* uranus */
       new body(…),

       /* neptune */
       new body(…)
    ];
```

Create a record object

5-BODY IN CHAPEL: THE BODIES

nbody.chpl

```
var bodies =
    [  /* sun */
       new body(mass = solarMass),

       /* jupiter */
       new body(pos = ( 4.84143144246472090e+00,
                       -1.16032004402742839e+00,
                       -1.03622044471123109e-01),
                  v = ( 1.66007664274403694e-03 * daysPerYear,
                        7.69901118419740425e-03 * daysPerYear,
                       -6.90460016972063023e-05 * daysPerYear),
               mass =   9.54791938424326609e-04 * solarMass),

       /* saturn */
       new body(…),

       /* uranus */
       new body(…),

       /* neptune */
       new body(…)
   ];
```

Tuple values

5-BODY IN CHAPEL: THE BODIES

nbody.chpl

```
var bodies =
    [  /* sun */
       new body(mass = solarMass),

       /* jupiter */
       new body(pos = ( 4.84143144246472090e+00,
                       -1.16032004402742839e+00,
                       -1.03622044471123109e-01),
                  v = ( 1.66007664274403694e-03 * daysPerYear,
                        7.69901118419740425e-03 * daysPerYear,
                       -6.90460016972063023e-05 * daysPerYear),
               mass =   9.54791938424326609e-04 * solarMass),

       /* saturn */
       new body(…),

       /* uranus */
       new body(…),

       /* neptune */
       new body(…)
    ];
```

Array value

ARRAYS

ARRAY TYPES

02-array-examples.chpl

Syntax

```
array-type:
  [ domain-expr ] elt-type
array-value:
  [elt1, elt2, elt3, … eltn]
```

Meaning

- array-type: stores an element of elt-type for each index
- array-value: represent the array with these values

Examples

```
var A: [1..3] int,              // A stores 0, 0, 0
    B = [5, 3, 9],              // B stores 5, 3, 9
    C: [1..m, 1..n] real,       // 2D m by n array of reals
    D: [1..m][1..n] real;       // array of arrays of reals
```

More on arrays in data parallelism section later…

5-BODY IN CHAPEL: THE BODIES

nbody.chpl

```
var bodies =
    [  /* sun */
       new body(mass = solarMass),

       /* jupiter */
       new body(pos = ( 4.84143144246472090e+00,
                       -1.16032004402742839e+00,
                       -1.03622044471123109e-01),
                  v = ( 1.66007664274403694e-03 * daysPerYear,
                        7.69901118419740425e-03 * daysPerYear,
                       -6.90460016972063023e-05 * daysPerYear),
               mass =   9.54791938424326609e-04 * solarMass),

       /* saturn */
       new body(…),

       /* uranus */
       new body(…),

       /* neptune */
       new body(…)
    ];
```

Create a record object

Tuple values

Array value

HANDS ON: WRITING TUPLES, RECORDS, AND ARRAYS

nbody.chpl

Put a 'writeln("bodies = ", bodies);' into program

```
chpl nbody.chpl
./nbody -nl 1
bodies =(pos = (0.0, 0.0, 0.0), vel = (0.0, 0.0, 0.0),
mass = 39.4784) (pos = (4.84143, -1.16032, -0.103622), vel
= (0.606326, 2.81199, -0.0252184), mass = 0.0376937) (pos
= (8.34337, 4.1248, -0.403523), vel = (-1.01077, 1.82566,
0.00841576), mass = 0.0112863) (pos = (12.8944, -15.1112,
-0.223308), vel = (1.08279, 0.868713, -0.0108326), mass =
0.00172372) (pos = (15.3797, -25.9193, 0.179259), vel =
(0.979091, 0.594699, -0.034756), mass = 0.00203369)
-0.169075164
-0.169016441
```

MAIN( ) PROCEDURE

5-BODY IN CHAPEL: MAIN( )

nbody.chpl

```
…

proc main() {
  initSun();

  writef("%.9r\n", energy());
  for 1..numsteps do
    advance(0.01);
  writef("%.9r\n", energy());
}

…
```

5-BODY IN CHAPEL: MAIN( )

nbody.chpl

Procedure Definition

```
…

proc main() {
  initSun();

  writef("%.9r\n", energy());
  for 1..numsteps do
    advance(0.01);
  writef("%.9r\n", energy());
}

…
```

5-BODY IN CHAPEL: MAIN( )

nbody.chpl

```
…

proc main() {
  initSun();

  writef("%.9r\n", energy());
  for 1..numsteps do
    advance(0.01);
  writef("%.9r\n", energy());
}

…
```

Procedure Call

5-BODY IN CHAPEL: MAIN( )

nbody.chpl

```
…

proc main() {
  initSun();

  writef("%.9r\n", energy());
  for 1..numsteps do
    advance(0.01);
  writef("%.9r\n", energy());
}

…
```

Formatted I/O

5-BODY IN CHAPEL: MAIN( )

nbody.chpl

```
…

proc main() {
  initSun();

  writef("%.9r\n", energy());
  for 1..numsteps do
    advance(0.01);
  writef("%.9r\n", energy());
}

…
```

Range Value

RANGES: INTEGER SEQUENCES

RANGE VALUES: INTEGER SEQUENCES

Syntax

```
range-expr:
  [low] .. [high]
```

Definition

- Regular sequence of integers

  low <= high: low, low+1, low+2, ..., high

  low > high: degenerate (an empty range)

  low or high unspecified: unbounded in that direction

Examples

```
1..6            // 1, 2, 3, 4, 5, 6
6..1            // empty
3..             // 3, 4, 5, 6, 7, …
```

RANGE OPERATORS

02-range-operators.chpl

```
const r = 1..10;

printVals(r);
printVals(r # 3);
printVals(r by 2);
printVals(r by -2);
printVals(r by 2 # 3);
printVals(r # 3 by 2);
printVals(0.. #n);

proc printVals(r) {
  for i in r do
    write(i, " ");
  writeln();
}
```

```
1 2 3 4 5 6 7 8 9 10
1 2 3
1 3 5 7 9
10 8 6 4 2
1 3 5
1 3
0 1 2 3 4 … n-1
```

5-BODY IN CHAPEL: MAIN( )

nbody.chpl

```
…

proc main() {
  initSun();

  writef("%.9r\n", energy());
  for 1..numsteps do
    advance(0.01);
  writef("%.9r\n", energy());
}

…
```

Serial for loop

BASIC SERIAL CONTROL FLOW

FOR LOOPS

02-for-loops.chpl

Syntax

```
for-loop:
  for [index-expr in] iteratable-expr { stmt-list }
```

Meaning

- Executes loop body serially, once per loop iteration
- Declares new variables for identifiers in *index-expr*
  - type and const-ness determined by *iteratable-expr*
  - *iteratable-expr* could be a range, array, iterator, iterable object, ...

Examples

```
var A: [1..3] string = [" DO", " RE", " MI"];

for i in 1..3 { write(A[i]); }              // DO RE MI
for a in A { a += "LA"; } write(A);     // DOLA RELA MILA
```

CONTROL FLOW: OTHER FORMS

- Conditional statements

```
if cond { computeA(); } else { computeB(); }
```

- While loops

```
while cond {
  compute();
}
```

- For loops

```
for indices in iteratable-expr {
  compute();
}
```

- Select statements

```
select key {
  when value1 { compute1(); }
  when value2 { compute2(); }
  otherwise   { compute3(); }
}
```

CONTROL FLOW: BRACES VS. KEYWORDS

Control flow statements specify bodies using curly brackets (compound statements)

- Conditional statements

```
if cond { computeA(); } else { computeB(); }
```

- While loops

```
while cond {
  compute();
}
```

- For loops

```
for indices in iteratable-expr {
  compute();
}
```

- Select statements

```
select key {
  when value1 { compute1(); }
  when value2 { compute2(); }
  otherwise   { compute3(); }
}
```

CONTROL FLOW: BRACES VS. KEYWORDS

They also support keyword-based forms for single-statement cases

- Conditional statements

```
if cond then computeA(); else computeB();
```

- While loops

```
while cond do
  compute();
```

- For loops

```
for indices in iteratable-expr do
  compute();
```

- Select statements

```
select key {
  when value1 do compute1();
  when value2 do compute2();
  otherwise   do compute3();
}
```

CONTROL FLOW: BRACES VS. KEYWORDS

Of course, since compound statements are single statements, the two forms can be mixed…

- Conditional statements

```
if cond then { computeA(); } else { computeB(); }
```

- While loops

```
while cond do {
  compute();
}
```

- For loops

```
for indices in iteratable-expr do {
  compute();
}
```

- Select statements

```
select key {
  when value1 do { compute1(); }
  when value2 do { compute2(); }
  otherwise   do { compute3(); }
}
```

PROCEDURES AND ITERATORS

5-BODY IN CHAPEL: MAIN( )

nbody.chpl

```
…

proc main() {
  initSun();

  writef("%.9r\n", energy());
  for 1..numsteps do
    advance(0.01);
  writef("%.9r\n", energy());
}

…
```

Procedure Definition

Procedure Call

Formatted I/O
*not covered here

Serial for loop

Range Value

5-BODY IN CHAPEL: ADVANCE( )

nbody.chpl

```
advance(0.01);
…
proc advance(dt) {
  for i in 1..numbodies {
    for j in i+1..numbodies {
      const dpos = bodies[i].pos - bodies[j].pos,
            mag = dt / sqrt(sumOfSquares(dpos))**3;

      bodies[i].v -= dpos * bodies[j].mass * mag;
      bodies[j].v += dpos * bodies[i].mass * mag;
    }
  }

  for b in bodies do
    b.pos += dt * b.v;
}
```

5-BODY IN CHAPEL: ADVANCE( )

nbody.chpl

```
advance(0.01);
…
proc advance(dt) {
  for i in 1..numbodies {
    for j in i+1..numbodies {
      const dpos = bodies[i].pos - bodies[j].pos,
            mag = dt / sqrt(sumOfSquares(dpos))**3;

      bodies[i].v -= dpos * bodies[j].mass * mag;
      bodies[j].v += dpos * bodies[i].mass * mag;
    }
  }

  for b in bodies do
    b.pos += dt * b.v;
}
```

$$m_1\mathbf{a}_1 = \frac{Gm_1m_2}{r_{12}^3}(\mathbf{r}_2 - \mathbf{r}_1) \quad \text{Sun-Earth}$$

$$m_2\mathbf{a}_2 = \frac{Gm_1m_2}{r_{21}^3}(\mathbf{r}_1 - \mathbf{r}_2) \quad \text{Earth-Sun}$$

5-BODY IN CHAPEL: ADVANCE( )

nbody.chpl

```
advance(0.01);
…
proc advance(dt) {
  for i in 1..numbodies {
    for j in i+1..numbodies {
      const dpos = bodies[i].pos - bodies[j].pos,
            mag = dt / sqrt(sumOfSquares(dpos))**3;

      bodies[i].v -= dpos * bodies[j].mass * mag;
      bodies[j].v += dpos * bodies[i].mass * mag;
    }
  }

  for b in bodies do
    b.pos += dt * b.v;
}
```

Procedure call

Procedure definition

PROCEDURES, BY EXAMPLE

02-procedures-and-arg-intents.chpl

- Example to compute the area of a circle

```
proc area(radius: real): real {
  return 3.14 * radius**2;
}

writeln(area(2.0));   // 12.56
```

```
proc area(radius) {
  return 3.14 * radius**2;
}
```

Argument and return types can be omitted

- Example of argument default values, naming

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

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)
```

ARGUMENT INTENTS

Arguments can optionally be given intents

- (blank): varies with type; follows principle of least surprise
  - most types: `const in` or `const ref`
  - sync/single vars, atomics: `ref`
- `ref`: formal is a reference back to the actual
- `const [ref | in]`: disallows modification of the formal
- `param`/`type`: actual must be a param/type
- `in`: initializes formal using actual; permits formal to be modified
- `out`: copies formal into actual at procedure return
- `inout`: does both of the above

ARGUMENT INTENTS, BY EXAMPLE

02-procedures-and-arg-intents.chpl

- For some types, argument intents are needed so as to avoid inadvertent races

```
proc foo(x: real, y: [] real) {
  // x = 1.2;    // illegal: scalars are passed 'const in' by default
  // y = 3.4;    // illegal: 'ref' by default for arrays is deprecated
}

var r: real,
    A: [1..3] real;

foo(r, A);

writeln((r, A));
```

ARGUMENT INTENTS, BY EXAMPLE

02-procedures-and-arg-intents.chpl

- Arguments can optionally be given intents.
- 'ref' intent means the actual being passed in will be modified

```
proc foo(ref x: real, ref y: [] real) {
  x = 1.2;    // OK: actual is modified
  y = 3.4;    // OK: actual is modified
}

var r: real,
    A: [1..3] real;

foo(r, A);

writeln((r, A));    // writes (1.2, [3.4, 3.4, 3.4])
```

ARGUMENT INTENTS, BY EXAMPLE

02-procedures-and-arg-intents.chpl

- Can't pass a 'const' to a 'ref' intent

```
proc foo(ref x: real, ref y: [] real) {
  x = 1.2;    // OK: actual is modified
  y = 3.4;  // OK: actual is modified
}

const r: real,
      A: [1..3] real;

// foo(r, A);    // illegal, can't pass a constant to a 'ref' intent

writeln((r, A));    // writes (0.0, [0.0, 0.0, 0.0])
```

ARGUMENT INTENTS, BY EXAMPLE

02-procedures-and-arg-intents.chpl

- Can pass a 'const' to a 'const ref' intent
- However, can't write to a formal coming in as 'const' intent

```
proc foo(const ref x: real, const ref y: [] real) {
  // x = 1.2;    // illegal: can’t modify constant arguments
  // y = 3.4;  // illegal: can’t modify constant arguments
}

const r: real,
      A: [1..3] real;

foo(r, A);    // OK to create constant references to constants

writeln((r, A));    // writes (0.0, [0.0, 0.0, 0.0])
```

ARGUMENT INTENTS, BY EXAMPLE

02-procedures-and-arg-intents.chpl

- Can't pass 'const' and 'var' into 'param' intents

```
proc foo(param x: real, type t) {
  …
  …
}

const r: real,
      A: [1..3] real;

// foo(r, A);    // illegal: can't pass vars and consts to params and types

writeln((r, A));    // writes (0.0, [0.0, 0.0, 0.0])
```

ARGUMENT INTENTS, BY EXAMPLE

02-procedures-and-arg-intents.chpl

- Can pass a literal, param, or a type into 'param' intent

```
proc foo(param x: real, type t) {
  …
  …
}

const r: real,
      A: [1..3] real;

foo(1.2, r.type);    // OK: passing a literal/param and a type

writeln((r, A));     // writes (0.0, [0.0, 0.0, 0.0])
```

ARGUMENT INTENTS, BY EXAMPLE

02-procedures-and-arg-intents.chpl

- 'in' intents cause the actual argument value to be copied into the formal

```
proc foo(in x: real, in y: [] real) {
  x = 1.2;    // OK: local copy is modified
  y = 3.4;    // OK: local copy is modified
}

var r: real,
    A: [1..3] real;

foo(r, A);

writeln((r, A));    // writes (0.0, [0.0, 0.0, 0.0])
```

ARGUMENT INTENTS, BY EXAMPLE

02-procedures-and-arg-intents.chpl

- 'out' intents cause the formal value to be copied into actual argument upon return from procedure

```
proc foo(out x: real, out y: [] real) {
  x = 1.2;    // OK: local copy is modified
  y = [3.4,3.4,3.4];    // OK: local copy is modified
}

var r: real,
    A: [1..3] real;

foo(r, A);

writeln((r, A));    // writes (1.2, [3.4, 3.4, 3.4])
```

ARGUMENT INTENTS, BY EXAMPLE

02-procedures-and-arg-intents.chpl

- 'inout' intent is a combination of 'in' and 'out' intent

```
proc foo(inout x: real, inout y: [] real) {
   x = 1.2;    // OK: local copy is modified
   y = 3.4;    // OK: local copy is modified
}

var r: real,
    A: [1..3] real;

foo(r, A);

writeln((r, A));    // writes (1.2, [3.4, 3.4, 3.4])
```

5-BODY IN CHAPEL: ADVANCE( )

nbody.chpl

```
proc advance(dt) {
  for i in 1..numbodies {
    for j in i+1..numbodies {
      const dpos = bodies[i].pos - bodies[j].pos,
            mag = dt / sqrt(sumOfSquares(dpos))**3;

      bodies[i].v -= dpos * bodies[j].mass * mag;
      bodies[j].v += dpos * bodies[i].mass * mag;
    }
  }

  for b in bodies do
    b.pos += dt * b.v;
}
```

5-BODY IN CHAPEL: ALTERNATIVE USING ITERATORS

nbody.chpl

Use of iterator

```
proc advance(dt) {
  for (i,j) in triangle(numbodies) {
    const dpos = bodies[i].pos - bodies[j].pos,
           mag = dt / sqrt(sumOfSquares(dpos))**3;
…
  }
…
}
…

iter triangle(n) {
  for i in 1..n do
    for j in i+1..n do
      yield (i,j);
}
```

Definition of iterator

5-BODY IN CHAPEL: ADVANCE( ) USING ITERATORS

nbody.chpl

```
proc advance(dt) {
  for (i,j) in triangle(numbodies) {




    const dpos = bodies[i].pos - bodies[j].pos,
          mag = dt / sqrt(sumOfSquares(dpos))**3;

    bodies[i].v -= dpos * bodies[j].mass * mag;
    bodies[j].v += dpos * bodies[i].mass * mag;
  }

  for b in bodies do
    b.pos += dt * b.v;
}
```

HANDS ON: WHERE MIGHT WE CONSIDER PARALLELIZING N-BODY

nbody.chpl

Look at 'nbody.chpl' and identify…

- 'for' loops that can be parallelized
- 'for' loops that need to stay serial to keep meaning
- 'for' loops that are "mostly" parallel but have something like +=

Can be parallelized

Inherently serial loop

Can be parallelized but have to avoid races when adding into velocity field

```
for b in bodies do
  b.pos += dt * b.v;

for 1..numsteps do
  advance(0.01);

for i in 1..numbodies {
  for j in i+1..numbodies {
    const dpos = bodies[i].pos - bodies[j].pos,
          mag = dt / sqrt(sumOfSquares(dpos))**3;
    bodies[i].v -= dpos * bodies[j].mass * mag;
    bodies[j].v += dpos * bodies[i].mass * mag;
  }
}
```

OUTLINE: CHAPEL BASICS

- Running Example: n-body computation (Hands On)
- Variables, Constants, and Operators
- Records and Classes
- Tuples
- Arrays
- Writing out Tuples, Records, and Arrays (Hands On)
- Main( ) Procedure
- Ranges and basic control flow
- Procedures and iterators
- Where might we parallelize the n-body computation? (Hands On)

LEARNING OBJECTIVES FOR TODAY'S CHAPEL TUTORIAL

- Familiarity with the Chapel execution model including how to run codes in parallel on a single node, across nodes, and both

- Learn Chapel concepts by compiling and running provided code examples
  - ✓Serial code using map/dictionary, (k-mer counting from bioinformatics)
  - ✓Parallelism and locality in Chapel
  - ✓Distributed parallelism and 1D arrays, (processing files in parallel)
  - ✓Chapel basics in the context of an n-body code
  - Distributed parallelism and 2D arrays, (heat diffusion problem)
  - How to parallelize histogram
  - Using CommDiagnostics for counting remote reads and writes
  - Chapel and Arkouda best practices including avoiding races and performance gotchas

- Where to get help and how you can participate in the Chapel community

ONE DAY CHAPEL TUTORIAL

- 9-10:30: Getting started using Chapel for parallel programming
- 10:30-10:45: break
- 10:45-12:15: Chapel basics in the context of the n-body example code
- 12:15-1:15: lunch
- 1:15-2:45: Distributed and shared-memory parallelism especially w/arrays (data parallelism)
- 2:45-3:00: break
- 3:00-4:30: More parallelism including for asynchronous parallelism (task parallelism)
- 4:30-5:00: Wrap-up including gathering further questions from attendees

CHAPEL RESOURCES

Chapel homepage: https://chapel-lang.org

- (points to all other resources)

Social Media:

- Twitter: @ChapelLanguage
- Facebook: @ChapelLanguage
- YouTube: http://www.youtube.com/c/ChapelParallelProgrammingLanguage

Community Discussion / Support:

- Discourse: https://chapel.discourse.group/
- Gitter: https://gitter.im/chapel-lang/chapel
- Stack Overflow: https://stackoverflow.com/questions/tagged/chapel
- GitHub Issues: https://github.com/chapel-lang/chapel/issues