CHAPEL BASICS, PART II

Chapel Team, edited by Michelle Strout

April 8, 2025

HANDS ON: HOW TO DO THE HANDS ON

01-hello.chpl

Example codes for Chapel tutorial slides

- https://github.com/UofA-CSc-372-Spring-2025/CSc372Spring2025-CourseMaterials/tree/main/Sandboxes/ChapelTutorialExamples

Using a container on your laptop

- First, install docker for your machine and start it up (see the README.md for more info)
- Then, use the chapel-gasnet docker container

```
docker pull docker.io/chapel/chapel-gasnet    # takes about 5 minutes
cd CSc372Spring2025-CourseMaterials/Sandboxes/ChapelTutorialExamples/
docker run --rm -it -v "$PWD":/workspace chapel/chapel-gasnet
root@589405d07f6a:/opt/chapel# cd /workspace
root@xxxxxxxxx:/myapp# chpl 01-hello.chpl
root@xxxxxxxxx:/myapp# ./01-hello -nl 1
```

PLAN

- Announcements
  - Everyone should have their Final Project assignments (go over final project expectations)
  - SA7 is due Friday April 11th
- Last time
  - TopHat Questions
  - ICA10 Quiz
  - Chapel Programming Basics in the context of an Nbody simulation, part I
- Today
  - Chapel programming basics in the context of an Nbody simulation, part II

SA7 EDITS FROM YESTERDAY

- See piazza post, screen shot below for reference

Actions

SA7 Chapel edits

The SA7 assignment writeup, https://github.com/UofA-CSc-372-Spring-2025/CSc372Spring2025-CourseMaterials/blob/main/SmallAssignmentWriteups/sa7-chapel.md, and initial files, https://github.com/UofA-CSc-372-Spring-2025/sa7-chapel-start, have undergone some modifications.

- The array of files is being tested in sa7-student-tests.chpl using a set instead of just comparing the array values since sometimes the order didn't match.
- The rgb to gray conversion and the sobel edge detector descriptions have been clarified in the writeup. They gray value needs to be put in all of the rgb channels, before and after edge detection.
- There is are some tests for the values in the histogram now.
- The Gradescope autograder has been set up as of Monday April 7th at 8:15pm. Already submitted assignments have been regraded. The grading tests are somewhat different that the tests provided in sa7-student-tests.chpl.

OUTLINE: OVERVIEW OF PROGRAMMING IN CHAPEL

- Main( ) Procedure
- Ranges and basic control flow
- Procedures and iterators
- Where might we parallelize the n-body computation? (Hands On)

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

Activity: Using the table at https://chapel-lang.org/docs/modules/standard/IO/FormattedIO.html, format the energy values in three different ways.

```
…

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
```

Activity: Experiment with 02-range-operators.chpl. What are the two different ways the pound sign (#) are being used?

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();
}
```

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

$$m_1 \mathbf{a}_1 = \frac{G m_1 m_2}{r_{12}^3}(\mathbf{r}_2 - \mathbf{r}_1) \quad \text{Sun-Earth}$$

$$m_2 \mathbf{a}_2 = \frac{G m_1 m_2}{r_{21}^3}(\mathbf{r}_1 - \mathbf{r}_2) \quad \text{Earth-Sun}$$

```
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;
}
```

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

Question: Why might it be useful for the compiler to have this information?

- (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])
```

Question: Why would we want to pass something by reference if we didn't plan on modifying it?

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 formal variable to get its own value of the actual argument

```
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 +=

See https://chapel-lang.org/docs/technotes/reduceIntents.html

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;
  }
}
```

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:

- Discord: https://discord.com/invite/xu2xg45yqH
- Stack Overflow: https://stackoverflow.com/questions/tagged/chapel
- GitHub Issues: https://github.com/chapel-lang/chapel/issues