Parallel Chapel

2/29/16
Simple example program

```plaintext
var total : int = 0;
for i in 1..10000 do total += i;
```
Task creation with begin

```plaintext
var lowTotal : int = 0;
var highTotal : int = 0;

begin ref(lowTotal) {
    for i in 1..5000 do lowTotal += i;
}
begin ref(highTotal) {
    for i in 5001..10000 do highTotal += i;
}

var total = lowTotal + highTotal;
```
Task creation with begin

```plaintext
var lowTotal : int = 0;
var highTotal : int = 0;

begin ref(lowTotal) {
    for i in 1..5000 do lowTotal += i;
}
begin ref(highTotal) {
    for i in 5001..10000 do highTotal += i;
}

var total = lowTotal + highTotal;

Incorrect: race condition
```
Correct implementation w/ begin

```plaintext
var lowTotal : int = 0;
var highTotal : int = 0;
sync {
    begin ref(lowTotal) {
        for i in 1..5000 do lowTotal += i;
    }
    begin ref(highTotal) {
        for i in 5001..10000 do highTotal += i;
    }
}
var total = lowTotal + highTotal;
```
Why do we need the ref part in begin
begin ref(lowTotal) { ... } ?

• Tell compiler that lowTotal should be passed by reference into the begin block
• Called a task intent; use w/ begin, cobegin, coforall

• Consider the following:
  var i = 0;
  while(i < 10) {
    begin f(i);
    i += 1;
  }
  Probably want f(0), f(1), f(2), ..., f(9)
Which of the following could be printed by the code below?

```plaintext
begin write(“1”);
write(“2”);
begin write(“3”);
```

A. 132
B. 213
C. 231
D. 321
E. More than one of the above
Which of the following could be printed by the code below?

```plaintext
begin write("1");
write("2");
begin write("3");
A. 132
B. 213
C. 231
D. 321
E. More than one of the above (B or C)
```
Which of the following could be printed by the code below?

```c
sync {
    begin write("1");
    write("2");
}
begin write("3");
```

A. 132  
B. 213  
C. 231  
D. 321  
E. More than one of the above
Which of the following could be printed by the code below?

code:
```
sync {
    begin write("1");
    write("2");
}
begin write("3");
```

A. 132
B. 213
C. 231
D. 321
E. More than one of the above
A common pattern and some syntactic sugar

• These are equivalent:

```plaintext
cobegin {
    begin statement1;
    begin statement2;
    ...
}
```

```plaintext
cobegin {
    statement1;
    statement2;
    ...
}
```

A common pattern and some syntactic sugar

• These are equivalent:

```
sync {
begin statement1;
begin statement2;
...
}
cobegin {
    statement1;
    statement2;
    ...
}
```

(Not strictly syntactic sugar: this isn’t the most efficient implementation and uses a different mechanism so begins inside a cobegin can escape.)
Which of the following could be printed by the code below?

cobegin {
    write("1");
    write("2");
}
write("3");
A. 132
B. 213
C. 231
D. 321
E. More than one of the above
Which of the following could be printed by the code below?

cobegin {
    write("1");
    write("2");
}
write("3");
A. 132
B. 213
C. 231
D. 321
E. More than one of the above
What dependency graph does this code realize?

A();
cobegin {
    B();
    C();
} 
D();
What dependency graph does this code realize?

A();
cobegin {
    B();
    C();
} D();
What code realizes the given dependency graph?
What code realizes the given dependency graph?
Can every DAG be realized by code using sync and begin?

A. Yes  
B. I think so  
C. I don’t think so  
D. No  
E. What’s a DAG again?
Can every DAG be realized by code using sync and begin?

A. Yes
B. I think so
C. I don’t think so
D. No
E. What’s a DAG again?
Parallel Loops

• Two kinds of parallel loops:
  
  \[ \text{forall } i \text{ in } 1..10 \text{ do statement; } \]  //omit do w/ braces
  
  \[ \text{coforall } i \text{ in } 1..10 \text{ do statement; } \]

• forall creates 1 task per processing unit

• coforall creates 1 per loop iteration
  
  • Used when each iteration requires lots of work and/or they must be done concurrently
(Broken) Parallel summing

var total : int = 0;
forall i in 1..100 {
    total += i;
}

Sync variables

• sync variables have value and empty/full state
  – store ≤ 1 value and block operations can’t proceed

• Can be used as lock:

```plaintext
var lock : sync int;
lock = 1; //acquires lock
...
var temp = lock; //releases the lock
```
Fixing the race

var total : sync int = 0;
forall i in 1..100 {
    total += i;
}

More sugar: forall shortened

```rust
var total : sync int = 0;
forall i in 1..100 {
    total += i;
}
[i in 1..100] total += i;
```
Reductions
Reductions in Chapel

• Express reduction operation in single line:
  
  var s = + reduce A;    //A is array, s gets sum

• Supports +, *, ^ (xor), &&, ||, max, min, ...

• minloc and maxloc return a tuple with value and its index:
  
  var (val, loc) = minloc reduce A;
Reduction example

- Can also use reduce on function plus a range
- Ex: Approximate $\pi/2$ using $\int_{-1}^{1} \sqrt{1-x^2} \, dx$ :

```javascript
config const numRect = 10000000;
const width = 2.0 / numRect; // rectangle width
const baseX = -1 - width/2;
const halfPI = + reduce [i in 1..numRect]
    (width * sqrt(1.0 - (baseX + i*width)**2));
```
Recall: Defining reductions

- Tally: Intermediate state of computation
- Combine: Combine 2 tallies
- Reduce-gen: Generate result from tally
- Init: Create “empty” tally
- Accumulate: Add single value to tally
Defining a custom reduction in Chapel

• Create object to represent intermediate state

• Must support
  – accumulate: adds a single element to the state
  – combine: adds another intermediate state
  – generate: converts state object into final output
Classes in Chapel

class Circle {
    var radius : real;
    proc area() : real {
        return 3.14 * radius * radius;
    }
}

var c1, c2 : Circle;  //creates 2 Circle references
var c1, c2 : Circle;  //creates 2 Circle references
c1 = new Circle(10);  /* uses system-supplied constructor
c1 = new Circle(10);  /* uses system-supplied constructor
to create a Circle object
    and makes c1 refer to it */
to create a Circle object
    and makes c1 refer to it */
c2 = c1;  //makes c2 refer to the same object
c2 = c1;  //makes c2 refer to the same object
delete c1;  //memory must be manually freed
delete c1;  //memory must be manually freed
Example “custom” reduction

class MyMin : ReduceScanOp {
  // finds min element (equiv. to built-in “min”)
  type eltType; // type of elements
  var soFar : eltType = max(eltType); // minimum so far

  proc accumulate(val : eltType) {
    if(val < soFar) { soFar = val; }
  }

  proc combine(other : MyMin) {
    if(other.soFar < soFar) { soFar = other.soFar; }
  }

  proc generate() { return soFar; }
}

var theMin = MyMin reduce A;
Scans

- Instead of just getting overall value, also compute value for every prefix

```
<table>
<thead>
<tr>
<th>A</th>
<th>2</th>
<th>1</th>
<th>4</th>
<th>3</th>
<th>1</th>
<th>3</th>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>14</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>
```

```r
var sum = + scan A;
```
use Time;  //include Time library

var timer = new Timer();  //create Timer object

timer.start();
//do something...
timer.stop();

timer.elapsed()  //returns (real-valued) number of seconds
timer.clear();  //get ready to use it again!