Chapel: Task Parallelism

Outline

- Primitive Task-Parallel Constructs
  - The **begin** statement
  - The **sync** types
- Structured Task-Parallel Constructs
- Atomic Transactions and Memory Consistency

Unstructured Task Creation: Begin

- Syntax

```
begin-stmt:
  begin stmt
```

- Semantics
  - Creates a concurrent task to execute *stmt*
  - Control continues immediately (no join)
- Example

```
begin writeln("hello world");
writeln("good bye");
```

- Possible output

```
hello world
good bye
```

```
good bye
hello world
```

Another Begin Example

- Combine begin and on

```
begin on Locale(1) do
  writeln("Hi from ", here.id);
writeln("Bye from ",here.id);
```

- Possible output

```
Hi from 1
Bye from 0
```

```
Bye from 0
Hi from 1
```

Synchronization via Sync-Types

- Syntax

```
sync-type:
  sync type
```

- Semantics
  - Variable has a value and state (full or empty)
  - Default read blocks until written (until full)
  - Default write blocks until read (until empty)
- Examples: Critical sections

```
var lock$: sync bool; // state is empty

lock$ = true; // state is full
critical();
lock$;         // state is empty
```

Sync-Type Methods

- `readFE():t` wait until full, leave empty, return value
- `readFF():t` wait until full, leave full, return value
- `readXX():t` non-blocking, return value
- `writeEF(v:t)` wait until empty, leave full, set value to `v`
- `writeFF(v:t)` wait until full, leave full, set value to `v`
- `writeXF(v:t)` non-blocking, leave full, set value to `v`
- `reset()` non-blocking, leave empty, reset value
- `isFull: bool` non-blocking, return true if full else false

- Defaults – read: **`readFE`**, write: **`writeEF`**

Primitive Task Parallelism Examples

- examples/primers/taskParallel.chpl
- examples/programs/prodCons.chpl

Outline

- Primitive Task-Parallel Constructs
- Structured Task-Parallel Constructs
  - The **cobegin** statement
  - The **coforall** loop
  - The **sync** statement
- Atomic Transactions and Memory Consistency
- Implementation Notes and Examples

Block-Structured Task Invocation: Cobegin

- Syntax

```
cobegin-stmt:
  cobegin { stmt-list }
```

- Semantics
  - Invokes a concurrent task for each listed *stmt*
  - Control waits to continue – implicit join
- Example

```
cobegin {
  consumer(1);
  consumer(2);
  producer();
}
```

Cobegin is Unnecessary

Any cobegin statement

```
cobegin {
  stmt1();
  stmt2();
  stmt3();
}
```

can be rewritten in terms of begin statements

```
var s1$, s2$, s3$: sync bool;
begin { stmt1(); s1$ = true; }
begin { stmt2(); s2$ = true; }
begin { stmt3(); s3$ = true; }
s1$; s2$; s3$;
```

but the compiler may miss out on optimizations.

Loop-Structured Task Invocation: Coforall

- Syntax

```
coforall-loop:
  coforall index-expr in iteratable-expr { stmt }
```

- Semantics
  - Loop over *iteratable-expr* invoking concurrent tasks
  - Control waits to continue – implicit join
- Example

```
begin producer();
coforall i in 1..numConsumers {
  consumer(i);
}
```

Usage of Begin, Cobegin, and Coforall

- Use begin when
  - Creating tasks with unbounded lifetimes
  - Load balancing requires dynamic task creation
  - Cobegin and coforall are insufficient for task structuring
- Use cobegin when
  - Invoking a fixed # of tasks (potentially heterogeneous)
  - The tasks have bounded lifetimes
- Use coforall when
  - Invoking a fixed or dynamic # of homogeneous task
  - The tasks have bounded lifetimes

Usage of For, Forall, and Coforall

- Use for when
  - A loop must be executed serially
  - One task is sufficient for performance
- Use forall when
  - The loop can be executed in parallel
  - The loop can be executed serially
  - Degree of concurrency << # of iterations
- Use coforall when
  - The loop must be executed in parallel
    (And not just for performance reasons!)
  - Each iteration has substantial work

Structuring Sub-Tasks: Sync-Statements

- Syntax

```
sync-statement:
  sync stmt
```

- Semantics
  - Executes *stmt*
  - Waits on all *dynamically-encountered* begins
- Example

```
sync {
  for i in 1..numConsumers {
    begin consumer(i);
  }
  producer();
}
```

Program Termination and Sync-Statements

Where the cobegin statement is static,

```
cobegin {
  functionWithBegin();
  functionWithoutBegin();
}
```

the sync statement is dynamic.

```
sync {
  begin functionWithBegin();
  begin functionWithoutBegin();
}
```

Program termination is defined by an implicit sync.

```
sync main();
```

Structured Task Parallelism Examples

- examples/primers/taskParallel.chpl

Outline

- Primitive Task-Parallel Constructs
- Structured Task-Parallel Constructs
- Atomic Transactions and Memory Consistency
  - The **atomic** statement
  - Races and memory consistency
- Implementation Notes and Examples

Atomic Transactions (Unimplemented)

- Syntax

```
atomic-statement:
  atomic stmt
```

- Semantics
  - Executes stmt so it appears as a single operation
  - No other task sees a partial result
- Example

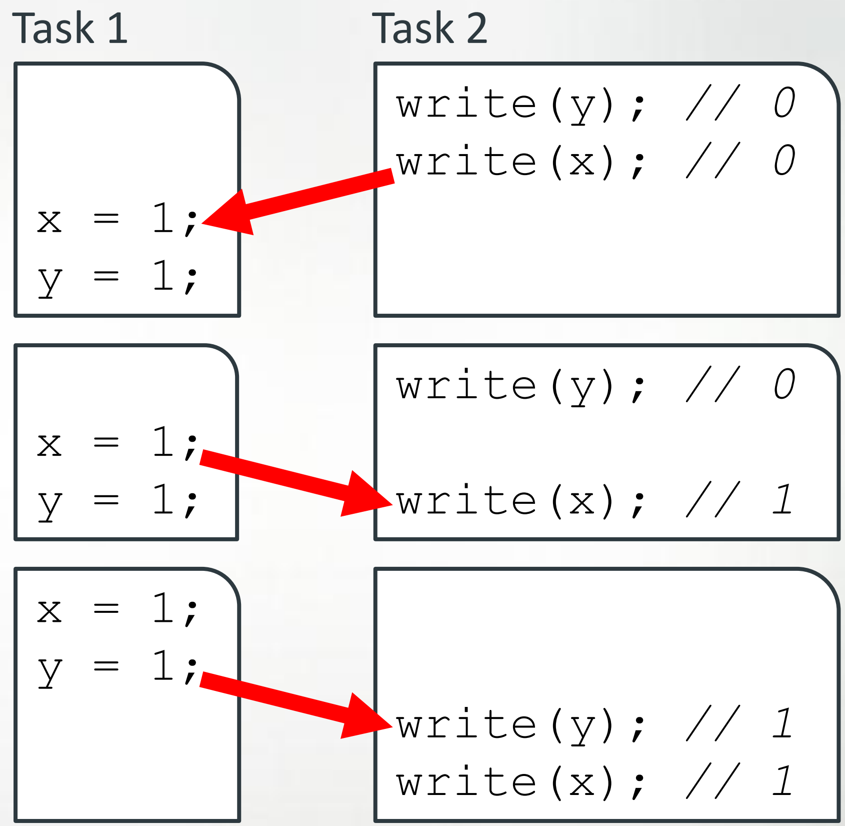
```
atomic A(i) = A(i) + 1;
```

```
atomic {
  newNode.next = node;
  newNode.prev = node.prev;
  node.prev.next = newNode;
  node.prev = newNode;
}
```

Races and Memory Consistency

- Example

```
var x = 0, y = 0;
cobegin {
  {
    x = 1;
    y = 1;
  }
  {
    write(y);
    write(x);
  }
}
```



- Could the output be 10? Or 42?

Data-Race-Free Programs (The Fine Print)

A program without races is sequentially consistent.

> A multi-processing system has sequential consistency if "*the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*" – Leslie Lamport

The behavior of a program with races is undefined.

Synchronization is achieved in two ways:

- By reading or writing sync (or single) variables
- By executing atomic statements

Future Directions

- Task teams
- Suspendable tasks
- Work stealing, load balancing
- Eurekas
- Task-private variables

Questions?

- Primitive Task-Parallel Constructs
  - The **begin** statement
  - The **sync** types
- Structured Task-Parallel Constructs
  - The **cobegin** statement
  - The **coforall** loop
  - The **sync** statement
- Atomic Transactions and Memory Consistency
  - The **atomic** statement
  - Races and memory consistency