Chapel: Task Parallelism
Task Parallelism Terminology

**Task:** a unit of parallel work in a Chapel program
- all Chapel parallelism is implemented using tasks
- `main()` is the only task when execution begins

**Thread:** a system-level concept that executes tasks
- not exposed in the language
- occasionally exposed in the implementation
"Hello World" in Chapel: a Task-Parallel Version

- Multicore Hello World

```chapel
config const numTasks = here.numCores;

coforall tid in 0..#numTasks do
  writeln("Hello, world! ",
           "from task ", tid, " of ", numTasks);
```
Outline

- Primitive Task-Parallel Constructs
- Structured Task-Parallel Constructs
Unstructured Task Creation: Begin

- **Syntax**
  
  ```
  begin-stmt:
  begin stmt
  ```

- **Semantics**
  
  - Creates a task to execute `stmt`
  - Original (“parent”) task continues without waiting

- **Example**
  
  ```
  begin writeln(“hello world”);
  writeln(“good bye”);
  ```

- **Possible output**
  
  ```
  hello world
  good bye
  ```
Synchronization Variables

- **Syntax**

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

- **Semantics**

  - Stores *full/empty* state along with normal value
  - Defaults to *full* if initialized, *empty* otherwise
  - Default read blocks until *full*, leaves *empty*
  - Default write blocks until *empty*, leaves *full*

- **Examples: Critical sections and futures**

  ```chapel
  var lock$: sync bool;
  lock$ = true;
critical();
  var lockval = lock$;
  ```

  ```chapel
  var future$: sync real;
  begin future$ = compute();
  computeSomethingElse();
  useComputedResults(future$);
  ```
Synchronization Type Methods

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

- **Defaults**: read: **readFE**, write: **writeEF**
Primitive Task-Parallel Constructs

Structured Task-Parallel Constructs
Block-Structured Task Creation: Cobegin

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

- **Semantics**
  - Creates a task for each statement in `stmt-list`
  - Parent task waits for `stmt-list` tasks to complete

- **Example**
  
  ```
  cobegin {
  consumer(1);
  consumer(2);
  producer();
  }
  // wait here for both consumers and producer to return
  ```
Loop-Structured Task Invocation: Coforall

- **Syntax**

  ```chapel
coforall-loop:
  coforall index-expr in iterable-expr { stmt-list }
  ```

- **Semantics**
  - Create a task for each iteration in `iteratable-expr`
  - Parent task waits for all iteration tasks to complete

- **Example**

  ```chapel
begin producer();
coforall i in 1..numConsumers {
  consumer(i);
}
// wait here for all consumers to return
  ```
Comparison of Loops: For, Forall, and Coforall

- **For loops**: executed using one task
  - use when a loop must be executed serially
  - or when one task is sufficient for performance

- **Forall loops**: typically executed using $1 < \#tasks \ll \#iters$
  - use when a loop *should* be executed in parallel...
  - ...but *can* legally be executed serially
  - use when desired \# tasks $\ll \# of iterations$

- **Coforall loops**: executed using a task per iteration
  - use when the loop iterations *must* be executed in parallel
  - use when you want \# tasks $= \# of iterations$
  - use when each iteration has substantial work
Bounded Buffer Producer/Consumer Example

```chapel
var buff$: [0..#buffersize] sync real;

cobegin {
    producer();
    consumer();
}

proc producer() {
    var i = 0;
    for … {
        i = (i+1) % buffersize;
        buff$(i) = …;
    }
}

proc consumer() {
    var i = 0;
    while … {
        i= (i+1) % buffersize;
        ...buff$(i)...;
    }
}
```

Chapel: Task Parallelism
Status: Task Parallel Features

- Most features working very well
- Ongoing task scheduling improvements (w/ Sandia, BSC):
  - ability for threads to set blocked tasks aside
  - lighter-weight tasking

**see talk by Kyle Wheeler on Tuesday afternoon**
Future Directions

- Task teams: provide a means of “coloring” tasks
  - for code isolation
  - to support task-based collective operations
    - barriers, reductions, eurekas
  - for the purposes of specifying execution policies
- Task-private variables and task-reduction variables
- Work-stealing and/or load-balancing tasking layers
Questions?