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
Multicore Hello World

```
config const numTasks = here.numCores;

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

- Primitive Task-Parallel Constructs
  - The `begin` statement
  - The `sync` types
- Structured Task-Parallel Constructs
- Implementation Notes and Examples
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

  good bye  
  hello world

Chapel: Task Parallelism
Synchronization Variables

- **Syntax**

  ```
  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**

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

  ```
  var future$: sync real;
  begin future$ = compute();
  computeSomethingElse();
  useComputedResults(future$);
  ```
Single Variables

• **Syntax**

\[
\text{single-type:} \\
\text{single type}
\]

• **Semantics**

• Similar to sync variable, but stays *full* once written

• **Example: Multiple Consumers of a future**

\[
\text{var future}: \text{single real}; \\
\text{begin future} = \text{compute}(); \\
\text{begin computeSomethingElse(future$)}; \\
\text{begin computeSomethingElse(future$)};
\]
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`
Single Type Methods

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

- \textbf{Defaults}: read: \texttt{readFF}, write: \texttt{writeEF}
A Note on Memory Consistency in Chapel

Memory Consistency Models:
• define how reads and writes to variables shared between tasks can occur
• a crucial topic, but one that makes peoples’ eyes glaze over

Chapel’s current memory consistency model, in brief:
• accesses to normal variables are relaxed
• accesses to sync/single variables are sequentially consistent and serve as memory fences for normal variables

We’re very open to expert feedback in this area
Outline

- Primitive Task-Parallel Constructs
- Structured Task-Parallel Constructs
  - The **cobegin** statement
  - The **coforall** loop
  - The **sync** statement
  - The **serial** statement
- Implementation Notes and Examples
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**

  ```
  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**

  ```
  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 $1 < \text{#tasks} \ll \text{#iters}$
  - 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
Comparison of Begin, Cobegin, and Coforall

- **begin:**
  - Use to create a dynamic task with an unstructured lifetime
  - “fire and forget”

- **cobegin:**
  - Use to create a related set of heterogeneous tasks
  - ...or a small, finite set of homogenous tasks
  - The parent task depends on the completion of the tasks

- **coforall:**
  - Use to create a fixed or dynamic # of homogenous tasks
  - The parent task depends on the completion of the tasks

**Note:** All these concepts can be composed arbitrarily
Atomic Transactions (work-in-progress with U. Notre Dame)

- **Syntax**
  
  ```plaintext
  atomic-statement:
  atomic stmt
  ```

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

- **Example**
  ```plaintext
  atomic A(i) += 1;
  ```
  ```plaintext
  atomic {
  newNode.next = node;
  newNode.prev = node.prev;
  node.prev.next = newNode;
  node.prev = newNode;
  }
  ```
Structuring Sub-Tasks: Sync-Statements

- **Syntax**
  
  \[
  \textit{sync-statement:} \quad \text{sync stmt}
  \]

- **Semantics**
  - Executes *stmt*
  - Waits for all *dynamically-scoped* begins to complete

- **Example**

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

```chapel
def search(N: TreeNode) {
    if (N != nil) {
        begin search(N.left);
        begin search(N.right);
    }
}
}
sync { search(root); }
```
Where the cobegin statement is static...

```chapel
cobegin {
    functionWithBegin();
    functionWithoutBegin();
} // waits on these two tasks, but not any others
```

...the sync statement is dynamic.

```chapel
sync {
    begin functionWithBegin();
    begin functionWithoutBegin();
} // waits on these tasks and any other descendents
```

Program termination is defined by an implicit sync on the main() procedure:

```chapel
sync main();
```
### Limiting Concurrency: Serial

**Syntax**

```plaintext
serial-statement:
  serial expr { stmt }
```

**Semantics**

- Evaluates `expr` and then executes `stmt`
- Suppresses any dynamically-encountered concurrency

**Example**

```plaintext
def search(N: TreeNode, depth = 0) {
  if (N != nil) then
    serial (depth > 4) do cobegin {
      search(N.left, depth+1);
      search(N.right, depth+1);
    }
  }
search(root);
```
Outline

- Primitive Task-Parallel Constructs
- Structured Task-Parallel Constructs
- Implementation Notes and Examples
Bounded Buffer Producer/Consumer Example

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

cobegin {
    producer();
    consumer();
}

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

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

Chapel: Task Parallelism
**Concurrency limiter: `maxThreadsPerLocale`**
- Use `--maxThreadsPerLocale=<i>` for at most `i` threads
- Use `--maxThreadsPerLocale=0` for a system limit *(default)*

**Current task scheduling policy**
- Once a thread starts running a task, it runs to completion
  - If an execution runs out of threads, it could deadlock
- Cobegin/coforall parent threads help with child tasks

**Help with deadlock detection**
- Running with `-b` and `-t` flags can help debug deadlocks
- For more information, see *Flags for tracking tasks* section of `$CHPL_HOME/doc/README.executing`
QuickSort in Chapel

```chapel
def quickSort(arr: [?D],
    thresh = log2(here.numCores()),
    depth = 0,
    low: int = D.low,
    high: int = D.high) {

    if high - low < 8 {
        bubbleSort(arr, low, high);
    } else {
        const pivotVal = findPivot(arr, low, high);
        const pivotLoc = partition(arr, low, high, pivotVal);
        serial (depth >= thresh) do cobegin {
            quickSort(arr, thresh, depth+1, low, pivotLoc-1);
            quickSort(arr, thresh, depth+1, pivotLoc+1, high);
        }
    }
}
```
Status: Task Parallel Features

- Most features working very well
- Ongoing task scheduling improvements (w/ BSC and Sandia):
  - ability for threads to set blocked tasks aside
  - lighter-weight tasking
- atomic statements a work-in-progress (w/ Notre Dame)
Future Directions

- **Task teams**
  - to provide a means of “coloring” different tasks
    - for code isolation
    - for the purposes of specifying execution policies
  - to support task-based collective operations
    - barriers, reductions, eurekas
- **Task-private variables and task-reduction variables**
- **Work-stealing and/or load-balancing tasking layers**
• Primitive Task-Parallel Constructs
  • The `begin` statement
  • The `sync` types

• Structured Task-Parallel Constructs
  • The `cobegin` statement
  • The `coforall` loop
  • The `sync` statement

• Implementation Notes and Examples