Chapel: Task Parallelism
Task Parallelism Terminology

**Task:** a unit of parallel work in a Chapel program
- all Chapel parallelism is implemented using tasks

**Thread:** a system-level concept for executing tasks
- not exposed in the language
- sometimes exposed in the implementation
"Hello World" in Chapel: Two Parallel Versions

- **Multicore**

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

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

- **Multi-node**

```chapel
coforall loc in Locales do
    on loc do
        writeln("Hello, world! ",
                "from node ", loc.id, " of ", numLocales);
```
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**
  
  ```chapel
  begin-stmt:
  begin stmt
  ```

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

- **Example**
  ```chapel
  begin writeln("hello world");
  writeln("good bye");
  ```

- **Possible output**
  
  hello world
  good bye
  hello world
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**
  
  ```chapel
class single-type:
  
  single type
  ```

- **Semantics**
  - Similar to sync variable, but stays *full* once written

- **Example: Multiple Consumers of a future**
  
  ```chapel
var future$: single real;

begin future$ = compute();
begin computeSomethingElse(future$);
begin computeSomethingEls(future$);
  ```
Synchronization Type Methods

- **readFE()**: block until full, leave empty, return value
- **readFF()**: block until full, leave full, return value
- **readXX()**: 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

- **readFF()**: block until full, leave empty, return value
- **readFF()**: block until full, leave full, return value
- **readXX()**: 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: **readFF**, write: **writeEF**
Outline

- Primitive Task-Parallel Constructs
- Structured Task-Parallel Constructs
  - The `cobegin` statement
  - The `coforall` loop
  - The `sync` statement
  - The `serial` statement
- Atomic Transactions and Memory Consistency
- Implementation Notes and Examples
Block-Structured Task Creation: Cobegin

- **Syntax**

  ```chapel
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**

  ```chapel
cobegin {
  consumer(1);
  consumer(2);
  producer();
}
```
Any cobegin statement...

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

...can be rewritten in terms of begin statements...

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

...but cobegin is supported as an important common case and to enable compiler optimizations.
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 `iterable-expr`
  - Parent task waits for all iteration tasks to complete

- **Example**
  
  ```chapel
  begin producer();
  coforall i in 1..numConsumers {
    consumer(i);
  }
  ```
Like Cobegin, Coforall is not Strictly Necessary

```chapel
coforall i in 1..n do stmt();

var count$: sync int = 0, flag$: sync bool = true;
for i in 1..n {
    const count = count$;
    if count == 0 then flag$;
    count$ = count + 1;
    begin {
        stmt();
        const count = count$;
        if count == 1 then flag$ = true;
        count$ = count - 1;
    }
}
flag$;
```
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 $1 \leq \#tasks \leq \#iters$
  - # tasks typically controlled by variables or arguments
  - use when a loop should be executed in parallel...
  - ...but can legally be executed serially
  - use when desired # tasks $<<$ # of iterations

- **Coforall loops**: executed using a task per iteration
  - Use when the loop iterations *must* be executed in parallel
  - Use when 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
  - 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
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)…;
    }
}
```
Structuring Sub-Tasks: Sync-Statements

**Syntax**

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

**Semantics**

- Executes \textit{stmt}
- Waits for all \textit{dynamically-scoped} begins to complete

**Example**

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

```python
def search(N: TreeNode) { 
  if (N != \textbf{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**

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

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

- **Example**

  ```
  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
- Atomic Transactions and Memory Consistency
  - The `atomic` statement
  - Races and memory consistency
- Implementation Notes and Examples
Atomic Transactions (unsupported work-in-progress)

- **Syntax**
  
  \[
  \text{atomic-statement:} \\
  \text{atomic stmt}
  \]

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

- **Example**

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

  atomic {
    newNode.next = node;
    newNode.prev = node.prev;
    node.prev.next = newNode;
    node.prev = newNode;
  }
  ```
Races and Memory Consistency

**Example**

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

**Could the output be 10? Or 42?**

<table>
<thead>
<tr>
<th>Task 1</th>
<th>Task 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>chapel</code></td>
<td><code>chapel</code></td>
</tr>
<tr>
<td>var x = 1; y = 1;</td>
<td>write(y); // 0</td>
</tr>
<tr>
<td></td>
<td>write(x); // 0</td>
</tr>
<tr>
<td></td>
<td>write(y); // 0</td>
</tr>
<tr>
<td></td>
<td>write(x); // 1</td>
</tr>
<tr>
<td><code>chapel</code></td>
<td><code>chapel</code></td>
</tr>
<tr>
<td>x = 1;</td>
<td>write(y); // 1</td>
</tr>
<tr>
<td>y = 1;</td>
<td>write(x); // 1</td>
</tr>
<tr>
<td></td>
<td><code>chapel</code></td>
</tr>
<tr>
<td>x = 1;</td>
<td><code>chapel</code></td>
</tr>
<tr>
<td>y = 1;</td>
<td><code>chapel</code></td>
</tr>
<tr>
<td></td>
<td>write(y); // 1</td>
</tr>
<tr>
<td></td>
<td>write(x); // 1</td>
</tr>
</tbody>
</table>
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
Outline

- Primitive Task-Parallel Constructs
- Structured Task-Parallel Constructs
- Atomic Transactions and Memory Consistency
- Implementation Notes and Examples
Using the Current Version of Chapel

- **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 may deadlock
  - Cobegin/coforall parent threads help with child tasks
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);
        }
    }
}
Performance of QuickSort in Chapel
(Array Size: $2^{21}$, Machine: 2 dual-core Opterons)

- **Execution Time (Seconds)**

- **Value of “thresh”**

  - maxThreads=0
  - maxThreads=4
Most features working very well

Tasking advances would be helpful
  - ability for threads to set blocked tasks aside
  - lighter-weight tasking (joint work with BSC, Sandia)
  - work-stealing, load-balancing

atomic statements unimplemented in release
Future Directions

- Task teams
  - to provide a means of “coloring” different tasks
    - for the purposes of specifying policies or semantics
  - to support team-based collective operations
    - barriers, reductions, eurekas
- Task-private variables and task-reduction variables
- Work-stealing and/or load-balancing tasking layers
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
- Implementation Notes and Examples