# 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

```
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
  - Synchronization types
- Structured Task-Parallel Constructs
- Miscellaneous Task-Parallel Constructs

# 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
```

# 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 future$: sync real;

begin future$ = compute();
computeSomethingElse();
useComputedResults(future$);
```

```
var lock$: sync bool;

lock$ = true;
critical();
var lockval = lock$;
```

# Example: Bounded Buffer Producer/Consumer

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

begin 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]…;
   }
}
```

# Single Variables

- Syntax

```
single-type:
  single type
```

- Semantics
  - Similar to sync variable, but stays *full* once written
- Example: Multiple Consumers of a future

```
var future$: single real;

begin future$ = compute();
begin computeSomethingElse(future$);
begin computeSomethingElse(future$);
```

# 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

- ~~`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: **`readFF`**, write: **`writeEF`**

# Atomic Variables

- Syntax

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

- Semantics
  - Supports operations on variable atomically w.r.t. other tasks
  - Based on C/C++ atomic operations
- Example: Trivial barrier

```
var count: atomic int, done: atomic bool;

proc barrier(numTasks) {
  const myCount = count.fetchAdd(1);
  if (myCount < numTasks) then
    done.waitFor(true);
  else
    done.testAndSet();
}
```

# Atomic Methods

- `read():t` return current value
- `write(v:t)` store *v* as current value
- `exchange(v:t):t` store *v*, returning previous value
- `compareExchange(old:t,new:t):bool`
  store *new* iff previous value was *old*; returns true on success
- `waitFor(v:t)` wait until the stored value is *v*
- `add(v:t)` add *v* to the value atomically
- `fetchAdd(v:t)` same, and return sum

(*sub*, *or*, *and*, *xor* also supported similarly)

- `testAndSet()` like *exchange(true)* for atomic bool
- `clear()` like *write(false)* for atomic bool

# Comparison of Synchronization Types

## sync/single:

- Best for producer/consumer style synchronization
- Imply a memory fence w.r.t. other loads/stores
- Use single for write-once values

## atomic:

- Best for uncoordinated accesses to shared state

# Outline

- Primitive Task-Parallel Constructs
- Structured Task-Parallel Constructs
  - The **cobegin** statement
  - The **coforall** loop
  - Relations between task- and data-parallel concepts
- Miscellaneous 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

```
coforall-loop:
  coforall index-expr in iteratable-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 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

# 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 << #iters

- 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 you want # tasks == # of iterations
- use when each iteration has substantial work

# Forall Loops: Lingering Questions

```
forall a in A do
  writeln("Here is an element of A: ", a);
```

- How many tasks will be used?
- How are iterations mapped to the tasks?

```
forall (a, i) in zip(A, 1..n) do
  a = i / 10.0;
```

Forall-loops may be zippered, like for-loops
- Corresponding iterations must match up
- But how does this work?

# Leader-Follower Iterators: Definition

- Chapel defines all zippered forall loops in terms of leader-follower iterators:
  - *leader iterators:* create parallelism, assign iterations to tasks
  - *follower iterators:* serially execute work generated by leader

- Given…

```
forall (a,b,c) in zip(A,B,C) do
  a = b + alpha * c;
```

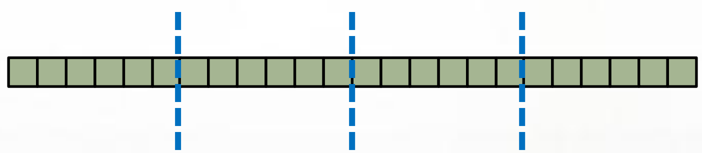
…A is defined to be the *leader*

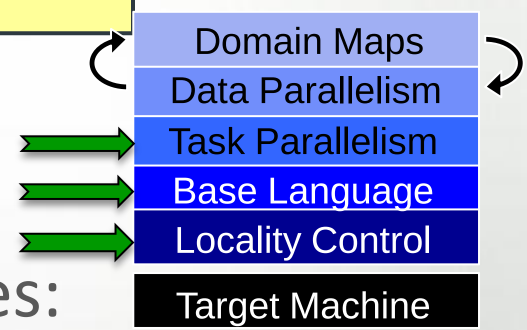
…*A, B,* and *C* are all defined to be *followers*

# Writing Leaders and Followers

Leader iterators are defined using task parallelism:

```
iter BlockArr.lead() {
  const numTasks = here.numCores();
  coforall tid in numTasks do
    yield computeMyChunk(tid, numTasks);
}
```






Follower iterators simply use serial features:

```
iter BlockArr.follow(work) {
  for i in work do
    yield accessElement(i);
}
```

# For More Information on Leader-Follower Iterators

**PGAS 2011:** *User-Defined Parallel Zippered Iterators in Chapel,* Chamberlain, Choi, Deitz, Navarro; October 2011

**Chapel release:**

- $CHPL_HOME/examples/primers/leaderfollower.chpl
- See the *AdvancedIters* module, described in the “Standard Modules” section of the language specification for some interesting leader-follower iterators:
  - OpenMP-style dynamic schedules
  - work-stealing iterators

# Outline

- Primitive Task-Parallel Constructs
- Structured Task-Parallel Constructs
- Miscellaneous Task-Parallel Constructs
  - serial statement
  - sync statement
  - release notes

# Limiting Concurrency: Serial

- Syntax

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

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

```
proc 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);
```

# QuickSort in Chapel

```
proc 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);
    }
  }
}
```

# Joining Sub-Tasks: Sync-Statements

- Syntax

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

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

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

```
proc search(N: TreeNode) {
  if (N != nil) {
    begin search(N.left);
    begin search(N.right);
  }
}
sync { search(root); }
```

# Sync-Statements and Program Termination

Where the cobegin statement is static…

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

…the sync statement is dynamic.

```
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:

```
sync main();
```

# Using the Current Version of Chapel

- Concurrency limiter: **numThreadsPerLocale**
  - Use **--numThreadsPerLocale=<i>** for at most *i* threads
  - Use **--numThreadsPerLocale=0** for a system limit *(default)*
- Default 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
  - (other tasking layers can be selected and differ in approach)
    - see $CHPL_HOME/README.tasks for details
- Help with deadlock detection
  - Running with -b and -t flags can help debug deadlocks
    - see $CHPL_HOME/doc/README.executing for details

# Status: Task Parallel Features

- All features working well

# Future Directions

- Using lighter-weight tasks by default
- Task teams: a means of “coloring” tasks by role
  - 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?

- **begin**, **cobegin**, **coforall**
- **sync**, **single atomic** variables
- **sync**, **serial** statements