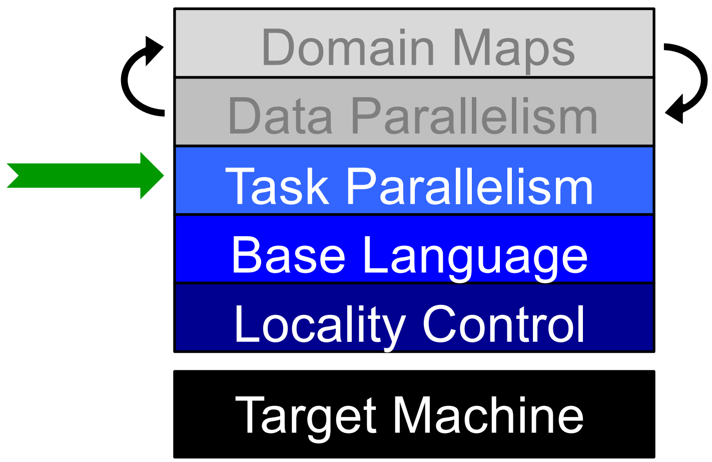
Task Parallelism

Task Parallelism



Defining our Terms

Task: a unit of computation that can/should execute in parallel with other tasks

Thread: a system resource that executes tasks

- not exposed in the language
- occasionally exposed in the implementation

Task Parallelism: a style of parallel programming in which parallelism is driven by programmer-specified tasks

(in contrast with):

Data Parallelism: a style of parallel programming in which parallelism is driven by computations over collections of data elements or their indices

Task Parallelism: Begin Statements

```
// create a fire-and-forget task for a statement
begin writeln("hello world");
writeln("goodbye");
```

Possible outputs:

```
hello world
goodbye
```

```
goodbye
hello world
```

Task Parallelism: Cobegin Statements

```
// create a task per child statement
cobegin {
  producer(1);
  producer(2);
  consumer(1);
}   // implicit join of the three tasks here
```

Cobegins/Serial by Example: QuickSort

```
proc quickSort(arr: [?D],
               depth = 0,
               thresh = log2(here.maxTaskPar),
               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, depth+1, thresh, low, pivotLoc-1);
      quickSort(arr, depth+1, thresh, pivotLoc+1, high);
    }
  }
}
```

Cobegins/Serial by Example: QuickSort

```
proc quickSort(arr: [?D],
               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 (here.runningTasks > here.maxTaskPar) do
      cobegin {
        quickSort(arr, low, pivotLoc-1);
        quickSort(arr, pivotLoc+1, high);
      }
  }
}
```

Task Parallelism: Coforall Loops

```
// create a task per iteration
coforall t in 0..#numTasks {
  writeln("Hello from task ", t, " of ", numTasks);
} // implicit join of the numTasks tasks here

writeln("All tasks done");
```

Sample output:

```
Hello from task 2 of 4
Hello from task 0 of 4
Hello from task 3 of 4
Hello from task 1 of 4
All tasks done
```

Comparison of Begin, Cobegin, and Coforall

begin:

- Use to create a dynamic task with an unstructured lifetime
- "fire and forget" (or at least "leave running for awhile")

cobegin:

- Use to create a related set of heterogeneous tasks
  …or a small, fixed 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

Task Parallelism: Data-Driven Synchronization

- **sync variables:** store full-empty state along with value
  - by default, reads/writes block until full/empty, leave in opposite state

- **atomic variables:** support atomic operations
  - e.g., compare-and-swap; atomic sum, multiply, etc.
  - similar to C/C++

Bounded Buffer Producer/Consumer Example

```
begin producer();
consumer();

// 'sync' types store full/empty state along with value
var buff$: [0..#buffersize] sync real;

proc producer() {
  var i = 0;
  for … {
    i = (i+1) % buffersize;
    buff$[i] = …;    // wait for empty, write, leave full
} }

proc consumer() {
  var i = 0;
  while … {
    i = (i+1) % buffersize;
    …buff$[i]…;     // wait for full, read, leave empty
} }
```

Synchronization Variables

- **Syntax**

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

- **Semantics**
  - Stores *full/empty* state along with normal value
  - Initially *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();
res = computeSomethingElse();
useComputedResults(future$, res);
```

Synchronization Variable 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 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$);
begin computeSomethingElse(future$);
```

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**

```
atomic-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 - 1) 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, returning pre-sum value

  (*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
  - "this task should block until something happens"
  - use single for write-once values

atomic:

- Best for uncoordinated accesses to shared state
  - "these tasks are unlikely to interfere with each other, at least for very long…"

Task Intents

- **Tells how to "pass" variables from outer scopes to tasks**
  - Similar to argument intents in syntax and philosophy
    - also adds a "reduce intent", similar to OpenMP
  - Design principles:
    - "principle of least surprise"
    - avoid simple race conditions
    - avoid copies of (potentially) expensive data structures
    - support coordination via sync/atomic variables
- **Congruent to forall intents, but for task-parallel constructs**

Task Intent Examples

```
var sum: real;
coforall i in 1..n do              // default task intent of scalars is 'const in'
  sum += computeMyResult(i);       // so this is illegal: (and avoids a race)

var sum: real;
coforall i in 1..n with (ref sum) do   // override default task intent
  sum += computeMyResult(i);           // we've now requested a race

var sum: real;
coforall i in 1..n with (+ reduce sum) do   // override default intent
  sum += computeMyResult(i);   // per-task sums will be reduced on task exit

var sum: atomic real;
coforall i in 1..n do                 // default task intent of atomics is 'ref'
  sum.add(computeMyResult(i));        // so this is legal, meaningful, and safe
```

Task-Private Variables

- **Task-parallel features support task-private variables easily**

```
coforall i in 1..numTasks {
  var mySum: real;   // each task gets its own copy of mySum
  for j in 1..n do
    mySum += A[i][j];
}
```

- **Forall loops need special support for task-private variables**

```
var oneSingleVariable: real;
forall i in 1..n {
  var onePerIteration: real;
}
```

Task-Private Variables

- **Task-parallel features support task-private variables easily**

```
coforall i in 1..numTasks {
  var mySum: real;    // each task gets its own copy of mySum
  for j in 1..n do
    mySum += A[i][j];
}
```

- **Forall loops need special support for task-private variables**

```
var oneSingleVariable: real;
forall i in 1..n with (var onePerTask: real) {
  var onePerIteration: real;
}
```

Task-Private Variables

- **Task-parallel features support task-private variables easily**

```
coforall i in 1..numTasks {
  var mySum: real;    // each task gets its own copy of mySum
  for j in 1..n do
    mySum += A[i][j];
}
```

- **Forall loops need special support for task-private variables**

```
var oneSingleVariable: real;
forall i in 1..n with (var onePerTask = 3.14) {
  var onePerIteration: real;
}
```

Task-Private Variables

- **Task-parallel features support task-private variables easily**

```
coforall i in 1..numTasks {
  var mySum: real;    // each task gets its own copy of mySum
  for j in 1..n do
    mySum += A[i][j];
}
```

- **Forall loops need special support for task-private variables**

```
var oneSingleVariable: real;
forall i in 1..n with (ref myLocArr = A[localInds]) {
  var onePerIteration: real;
}
```

Joining Sub-Tasks: Sync-Statements

- **Syntax**

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

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

```
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); }
```

Questions about Task Parallelism in Chapel?

Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, and URIKA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.