Chapel: Task Parallelism

Sung-Eun Choi and Steve Deitz
Cray Inc.
Outline

- Primitive Task-Parallel Constructs
  - The `begin` statement
  - The `sync` types
- Structured Task-Parallel Constructs
- Atomic Transactions and Memory Consistency
- Implementation Notes and Examples
Unstructured Task Creation: Begin

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

- **Semantics**
  
  - Creates a concurrent task to execute `stmt`
  - Control continues immediately (no join)

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

- **Possible output**
  
  ```
  hello world
  good bye
  ```
Synchronization via Sync-Types

• Syntax

\[
\textit{sync-type:}
\]

\[
\textit{sync type}
\]

• Semantics

• Default read blocks until written (until “full”)
• Default write blocks until read (until “empty”)

• Examples: Critical sections and futures

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

```
var future$: sync real;
begi future$ = compute();
computeSomethingElse();
useComputeResults(future$);
```
Synchronization via Single-Types

• **Syntax**

```chapel
class single-type:
  single type
```

• **Semantics**
  • Default read blocks until written (until “full”)
  • Write once

• **Example: Multiple consumers**

```chapel
var future$: single real;
begin { computeTaskA(); useResult1(future$); }
begin { computeTaskB(); useResult2(future$); }  
future$ = computeResult();
```
Sync-Type Methods

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

- Defaults – read: **readFE**, write: **writeEF**
Single-Type Methods

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

- Defaults – read: **readFF**, write: **writeEF**
Outline

- Primitive Task-Parallel Constructs
- Structured Task-Parallel Constructs
  - The \texttt{cobegin} statement
  - The \texttt{coforall} loop
  - The \texttt{sync} statement
  - The \texttt{serial} statement
- Atomic Transactions and Memory Consistency
- Implementation Notes and Examples
Block-Structured Task Invocation: Cobegin

• **Syntax**

```plaintext
cobegin-stmt:
cobegin { stmt-list }
```

• **Semantics**
  - Invokes a concurrent task for each listed *stmt*
  - Control waits to continue – implicit join

• **Example**

```plaintext
cobegin {
    consumer(1);
    consumer(2);
    producer();
}
```
Cobegin is Unnecessary

Any cobegin statement

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

can be rewritten in terms of begin statements

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

but the compiler may miss out on optimizations.
Loop-Structured Task Invocation: Coforall

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

- **Semantics**
  - Loop over `iterable-expr` invoking concurrent tasks
  - Control waits to continue – implicit join

- **Example**

  ```chapel
begin producer();
coforall i in 1..numConsumers {
  consumer(i);
}
```
Like Cobegin, Coforall is also Unnecessary

```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$;
```
Usage of Begin, Cobegin, and Coforall

• Use begin when
  • Creating tasks with unbounded lifetimes
  • Load balancing requires dynamic task creation
  • Cobegin and coforall are insufficient for task structuring

• Use cobegin when
  • Invoking a fixed # of tasks (potentially heterogeneous)
  • The tasks have bounded lifetimes

• Use coforall when
  • Invoking a fixed or dynamic # of homogeneous task
  • The tasks have bounded lifetimes
Structuring Sub-Tasks: Sync-Statements

- **Syntax**
  
  ```
  sync-statement:
  sync stmt
  ```

- **Semantics**
  - Executes *stmt*
  - Waits on all *dynamically-encountered* begins

- **Example**

  ```
  sync {
    for i in 1..numConsumers {
      begin consumer(i);
    }
  producer();
  }
  ```
Program Termination and Sync-Statements

Where the cobegin statement is static,

```chapel
cobegin
    functionWithBegin();
    functionWithoutBegin();
}
```

the sync statement is dynamic.

```chapel
sync
    begin
        functionWithBegin();
        begin
            functionWithoutBegin();
    }
```

Program termination is defined by an implicit sync.

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

- Syntax

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

- Semantics
  - Evaluates `expr` and then executes `stmt`
  - Squelches dynamically-encountered concurrency

- Example

```chapel
def search(i: int) {
  // search node i
  serial i > 8 do cobegin {
    search(i*2);
    search(i*2+1);
  }
}
```
Outline

- Primitive Task-Parallel Constructs
- Structured Task-Parallel Constructs
- Atomic Transactions and Memory Consistency
  - The \texttt{atomic} statement
  - Races and memory consistency
- Implementation Notes and Examples
Atomic Transactions (Unimplemented)

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

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

- **Example**
  ```
  atomic A(i) = A(i) + 1;
  ```
  ```
  atomic {
    newNode.next = node;
    newNode.prev = node.prev.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?**

Task 1:

- \( x = 1; \)
- \( y = 1; \)
- write(y); // 0
- write(x); // 0

Task 2:

- write(y); // 0
- write(x); // 1
- write(y); // 1
- write(x); // 1
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 Chapel version 1.1
  - Quicksort example
• Concurrency limiter: `maxThreadsPerLocale`
  • Use `--maxThreadsPerLocale=<i>` for at most i threads
  • Use `--maxThreadsPerLocale=0` for a system limit (default)

• Current task-to-thread scheduling policy
  • Once a thread gets a task, it runs to completion
  • `Cobegin/coforall` parent threads execute subtasks
  • If an execution runs out of threads, it may deadlock
def quickSort(arr: [],
    thresh: int,
    low: int = arr.domain.low,
    high: int = arr.domain.high) {

    if high - low < 8 {
        bubbleSort(arr, low, high);
    } else {
        const pivotVal = findPivot(arr, low, high);
        const pivotLoc = partition(arr, low, high, pivotVal);
        serial thresh <= 0 do cobegin {
            quickSort(arr, thresh-1, low, pivotLoc-1);
            quickSort(arr, thresh-1, pivotLoc+1, high);
        }
    }
}
Performance of Multi-Threaded Chapel

Performance of QuickSort in Chapel
(Array Size: $2^{21}$, Machine: 2 dual-core Opterons)

Execution Time (Seconds)

Value of “thresh”

- maxThreads=0
- maxThreads=4
Future Directions

• Task teams
• Suspensable tasks
• Work stealing, load balancing
• Eurekas
• Task-private variables
Questions?

- Primitive Task-Parallel Constructs
  - The `begin` statement
  - The `sync` types
- Structured Task-Parallel Constructs
  - The `cobegin` statement
  - The `coforall` loop
  - The `sync` statement
  - The `serial` statement
- Atomic Transactions and Memory Consistency
  - The `atomic` statement
  - Races and memory consistency
- Implementation Notes and Examples
  - Using Chapel version 1.1
  - Quicksort example