Chapel: Task Parallelism

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**
  
  ```chapel
  begin-stmt:
    begin stmt
  ```

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

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

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

• Syntax

```chapel
class sync-type:
  sync type

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

• Semantics

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

• Examples: Critical sections and futures

```chapel
var future$ : sync real;
begin
  future$ = compute();
  computeSomethingElse();
  useComputeResults(future$);
```

Chapel: Task Parallelism
Sync-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**: readFE, **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
Structured Task Invocation: Cobegin

- **Syntax**
  
  \[
  \text{cobegin-stmt:} = \text{cobegin} \{ \text{stmt-list} \}
  \]

- **Semantics**
  
  - Invokes a concurrent task for each listed \textit{stmt}
  
  - Control waits to continue – implicit join

- **Example**
  
  \[
  \text{cobegin} \{ \\
  \text{consumer}(1); \\
  \text{consumer}(2); \\
  \text{producer}(); \\
  \}
  \]
Cobegin is Unnecessary

Any cobegin statement

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

can be rewritten in terms of begin statements

```var
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.
A "Cobegin" Loop: Coforall

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

- **Semantics**
  - Loops over *iterator-expr* invoking concurrent tasks
  - Control waits to continue – implicit join

- **Example**
  ```chapel
begin producer();
coforall i in 1..numConsumers {
    consumer(i);
}
```

- **Note**: Like cobegin, coforall is unnecessary
Usage of Begin, Cobegin, and Coforall

- Use begin when
  - Creating tasks with unbounded lifetimes
  - Load balancing needs to be dynamic
  - Cobegin is insufficient to structure the tasks
- Use cobegin when
  - Invoking a fixed # of tasks
  - The tasks have bounded lifetimes
  - Load balancing is not an issue
- Use coforall when
  - Cobegin applies but the # of tasks is dynamic
Structuring Sub-Tasks: Sync-Statements

• **Syntax**

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

• **Semantics**
  
  • Executes \textit{stmt}
  
  • Waits on all \textit{dynamically-encountered} begins

• **Example**

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

Where the cobegin statement is static,

```chapel
cobegin {
    call1();
    call2();
}
```

the sync statement is dynamic.

```chapel
sync {
    begin call1();
    begin call2();
}
```

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**

  ```
  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 `atomic` statement
  - Races and memory consistency
- Implementation Notes and Examples
Atomic Transactions (Unimplemented)

- **Syntax**

  ```plaintext
  atomic-statements:
  atomic stmt
  ```

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

- **Example**

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

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

**Example**

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

**Expected Outputs**

- 11
- 01
- 00
- What about 10?
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 pThreads
  - Quick sort example
• **CHPL_THREADS**: Environment variable for threading
  • Default for most platforms is `pthreads`
  • Current alternatives include `none` and `mta`
• **maxThreads**: Config variable for limiting concurrency
  • Use `--maxThreads=#` to use at most # threads
  • Use `--maxThreads=0` to use a system limit
• Current task-to-thread scheduling policy
  • Once a thread gets a task, it runs to completion
  • If an execution runs out of thread, it may deadlock
  • In the future, blocked threads will run other tasks
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 QuickSort in Chapel
(Array Size: 2**21, Machine: 2 dual-core Opterons)

Value of “thresh”

Execution Time (Seconds)
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 pThreads
  - Quick sort example