Chapel: Task Parallelism

Brad Chamberlain

Outline

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

- **Syntax**
  
  \[ \text{begin-stmt:} \]
  
  \[ \begin{align*}
    \text{begin} & \quad \text{stmt}
  \end{align*} \]

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

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

- **Possible output**
  
  ```chapel
  hello world
  good bye
  hello world
  good bye
  ```

Synchronization: `sync-types`

- **Syntax**
  
  \[ \text{sync-type:} \]
  
  \[ \begin{align*}
    \text{sync} & \quad \text{type}
  \end{align*} \]

- **Semantics**
  
  - Default read blocks until written (until “full”)
  - Default write blocks until read (until “empty”)

- **Example: A critical section**
  
  ```chapel
  var lock$ : sync bool; // lock$ is uninitialized so empty

  def foo() {
    lock$ = true; // wait until lock$ empty, leave full
    begin critical();
    lock$; // wait until lock$ full, leave empty
  }

  begin foo(); begin foo();
  ```
Synchronization: single-types

- Syntax

- Semantics
  - Default read blocks until written (until “full”)
  - Can only be written once

- Examples

```chapel
var future$: single real;
begin
  future$ = compute();
  computeSomethingElse();
  useComputeResult(future$);
```

Methods on sync t

- `readFE(): t` wait until full, leave empty, return value
- `readFF(): t` wait until full, leave full, return value
- `readXX(): t` return value (non-blocking)
- `writeEF(v: t)` wait until empty, leave full, sets value to v
- `writeFF(v: t)` wait until full, leave full, sets value to v
- `writeXF(v: t)` non-blocking, leave full, sets value to v
- `reset()` non-blocking, leave empty, resets value
- `isFull: bool` non-blocking, returns true iff full

- Default read: `readFE`
- Default write: `writeEF`
Methods on single t

- readFE(): t wait until full, leave empty, return value
- readFF(): t wait until full, leave full, return value
- readXX(): t return value (non-blocking)
- writeEF(v: t) wait until empty, leave full, sets value to v
- writeFF(v: t) wait until full, leave full, sets value to v
- writeXF(v: t) non-blocking, leave full, sets value to v
- reset() non-blocking, leave empty, resets value
- isFull: bool non-blocking, returns true iff full

- Default read: readFF
- Default 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: \texttt{cobegin}

- **Syntax**
  
  \begin{verbatim}
  cobegin-stmt: 
  cobegin \{ stmt-list \}
  \end{verbatim}

- **Semantics**
  
  - Invokes a concurrent task for each statement in \textit{stmt-list}
  - Waits for all tasks to complete before continuing (implicit join)

- **Example**
  
  \begin{verbatim}
  cobegin { 
  consumer(1);
  consumer(2);
  producer();
  }
  \end{verbatim}

\textbf{cobegin is Unnecessary}

Any cobegin-statement

\begin{verbatim}
 cobegin { 
 stmt1();
 stmt2();
 stmt3();
 }
\end{verbatim}

can be rewritten in terms of begin-statements

\begin{verbatim}
 var s1$, s2$, s3$: sync bool;
 begin { stmt1(); s1$ = true; }
 begin { stmt2(); s2$ = true; }
 begin { stmt3(); s3$ = true; }
 s1$; s2$; s3$;
\end{verbatim}
Concurrent loops: `coforall`

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

- **Semantics**
  - Loops over `iterator-expr` creating a concurrent task for each iteration
  - Waits for all tasks to complete before continuing (implicit join)

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

- **Note:** `coforall` is also unnecessary

Synchronizing Sub-Tasks: `sync`-statements

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

- **Semantics**
  - Executes `stmt`
  - Waits on all `dynamically-encountered` `begin`-statements

- **Example**
  ```
  sync {
    for i in 1..numConsumers {
      begin consumer(i);
    }
    producer();
  }
  ```
Program Termination and \texttt{sync}

While the \texttt{cobegin} statement is static,
\begin{verbatim}
cobegin 
    call1();
    call2();

def call1 
    begin 
        backgroundTask1();
        writeln(“in call1”);
    end

def call2 
    begin 
        backgroundTask2();
        writeln(“in call2”);
    end
\end{verbatim}

the \texttt{sync} statement is dynamic.
\begin{verbatim}
sync 
    begin 
        call1();
    end

    begin 
        call2();
    end

sync main();
\end{verbatim}

Program termination is defined with an implicit \texttt{sync}-statement.

Programs can be terminated early by calling \texttt{exit} or \texttt{halt}.

Limiting Concurrency: \texttt{serial}

- **Syntax**

  \texttt{serial-stmt:}
  \begin{verbatim}
  serial expr stmt
  \end{verbatim}

- **Semantics**
  - Evaluates \texttt{expr} and then executes \texttt{stmt}
  - If the expression is true, enters serial mode
  - When in serial mode, all concurrency will be executed sequentially

- **Example**

  \begin{verbatim}
  def search(i: int) {
    // search node i
    serial i > 8 cobegin {
      search(i*2);
      search(i*2+1);
    }
  }
  \end{verbatim}
Outline

- Primitive Task Parallel Constructs
- Structured Task Parallel Constructs
- Atomic Transactions and Memory Consistency
  - The `atomic` statement
  - Race conditions and memory consistency
- Implementation Notes and Examples

Atomic Transactions

- Syntax
  ```
  atomic-stmt:
  atomic stmt
  ```

- Semantics
  - Executes `stmt` so that it appears to run instantaneously
  - No other task sees a partial result of this statement

- Example
  ```
  atomic {
      A[1] += 1;
  }
  ```
  ```
  atomic {
      newnode.next = insertpt;
      newnode.prev = insertpt.prev;
      insertpt.prev.next = newnode;
      insertpt.prev = newnode;
  }
  ```
Races and Memory Consistency

- Example

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

- Expected Outputs
  - 00
  - 01
  - 11

- What about?
  - 10

Data-Race-Free Programs

- A program without data races is sequentially consistent.

A multi-processing system has sequential consistency if “the results of any executions 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 data races is undefined.

- Synchronization is achieved in two ways:
  - By reading or writing variables of sync or single types
  - 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
  - Produce-Consumer Buffer Example

Using the current implementation

- **CHPL_THREADS**: Environment variable for threading
  - Default for most platforms is `pthreads`
  - Current alternatives include `none` and `mta`

- `maxThreads`: Configuration variable for limiting concurrency
  - Use `--maxThreads=#` to specify a limit on the number of threads
  - Default for `maxThreads` is system-dependent (0 for unlimited)

- Current task scheduling policy
  - Once a task is assigned to a thread it runs to completion
  - If all threads are running and all tasks are blocked, the program will deadlock
    - In the future, blocked threads will be used to run other tasks…
Quick Sort in Chapel

```chapel
def quickSort(arr: [],
               depth: int, // depth at which to serialize
               low: int = arr.domain.low,
               high: int = arr.domain.high,
               thresh: int = lg2(numCores())) {
  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, low, pivotLoc-1);
      quickSort(arr, depth+1, pivotLoc+1, high);
    }
  }
}
```

Effect of Threads/Tasks on Performance

![Graph showing the effect of threads/tasks on performance.](image)

**Execution Time (seconds)**

$n=2^{21}$; machine = 2 dual-core Opterons

- Unlimited threads
- maxThreads=4
Producer-Consumer Example

s: size of the buffer
n: number of exchanges

```chapel
def producer() {
    [i in 0..#n] buff$(i%s) = i;
}
def consumer() {
    var i = 0;
    do {
        var value = buff$(i);
        process(value);
        i = (i+1)%s;
    } while value != n-1;
}
```

Task Parallelism Status

- **Stable features:**
  - `begin`, `cobegin`, `coforall` statements
  - `sync`, `single` types
  - `sync`, `serial` statements

- **Incomplete features:**
  - performance of task parallelism is reasonable, but could be improved

- **Unimplemented features:**
  - `atomic` statements (collaborating with U/Notre Dame, ORNL; UIUC)
  - the memory consistency model is not currently enforced

- **Future directions:**
  - differentiate between “may” and “must” `cobegins` and `begins`
  - ability to use a `serial` statement to turn parallelism back on
  - ability for threads to set aside blocked tasks
  - implement threading interface on user-level threads package(s)
  - runtime task throttling, load balancing, work stealing
Questions?