Chapel: Task Parallelism

Brad Chamberlain

PRACE Winter School
12 February 2009

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

```
begin-stmt:
  begin stmt
```

- Semantics
  - Create 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
```

```
good bye
hello world
```


Synchronization: `sync`-types

- Syntax

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

- Semantics
  - Default read blocks until written (until "full")
  - Default write blocks until read (until "empty")
- Example: A critical section

```
var lock$: sync bool;   // lock$ is uninitialized so empty

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

Synchronization: `single`-types

- Syntax

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

- Semantics
  - Default read blocks until written (until "full")
  - Can only be written once
- Examples

```
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: `cobegin`

- Syntax

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

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

```
cobegin {
  consumer(1);
  consumer(2);
  producer();
}
```


`cobegin` is Unnecessary

Any cobegin-statement

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

can be rewritten in terms of begin-statements

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

Concurrent loops: `coforall`

- Syntax

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

- Semantics
  - Loops over *iterator-expr* creating a concurrent tasks 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 `sync`

While the cobegin statement is static,

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

the sync statement is dynamic.

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

```
def call1 {
  begin backgroundTask1();
  writeln("in call1");
}

def call2 {
  begin backgroundTask2();
  writeln("in call2");
}
```

Program termination is defined with an implicit sync-statement.

```
sync main();
```

Programs can be terminated early by calling **`exit`** or **`halt`**.

Limiting Concurrency: `serial`

- Syntax

```
serial-stmt:
  serial expr stmt
```

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

```
def search(i: int) {
  // search node i
  serial i > 8 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
  - 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[i] += 1;
}
```

```
atomic {
  newnode.next = insertpt;
  newnode.prev = insertpt.prev;
  insertpt.prev.next = newnode;
  insertpt.prev = newnode;
}
```

Races and Memory Consistency

- Example

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

```
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

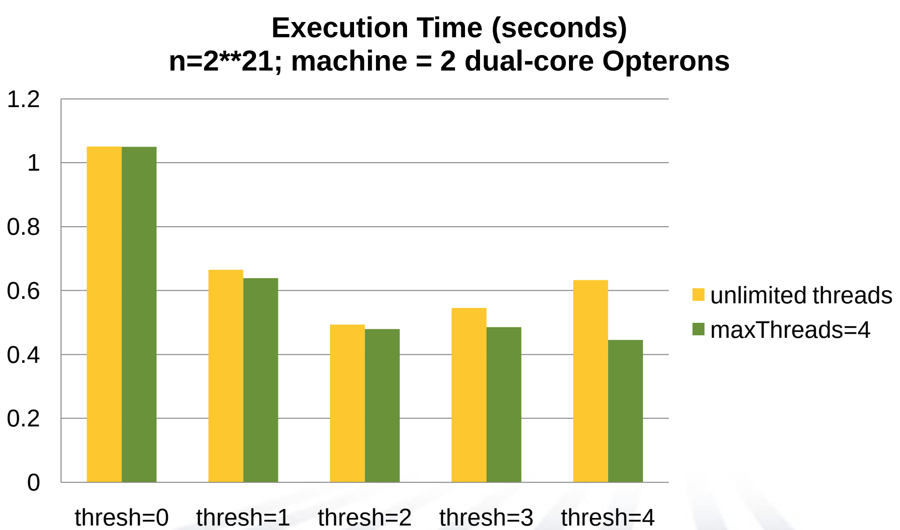
Using the current implementation

- **`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

```
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);
    }
  }
}
```

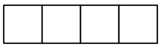

Effect of Threads/Tasks on Performance



Producer-Consumer Example

s: size of the buffer

n: number of exchanges

buff$ 

```
var buff$: [0..#s] sync int;
cobegin {
  producer();
  consumer();
}
def producer() {
  [i in 0..#n] buff$(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?