Language and Compiler Improvements

Chapel Team, Cray Inc.
Chapel version 1.11
April 2, 2015
This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.
Outline

- Task Intent Improvements
- Improved Const Checking
- Standalone Parallel Iterators
- Compilation Time Improvements
- Other Major Language Compiler Changes
- Language/Compiler Priorities and Next Steps
Task Intent Improvements
Task Intents: Background

- **task intents** (since v1.8) control how outer variables are passed into task functions

- **default task intent** prevents certain common data races
  - e.g. snapshot $i$ when `begin` task is created, rather than reading later
    
    ```
    var i = 0;
    while i < 10 {
        begin f(i); // guaranteed to see each of f(0), f(1), ..., f(9)
        i += 1;
    }
    ```

- **ref intent** allows sharing variables between tasks
  - e.g. producer-consumer
    
    ```
    cobegin with (ref data) {
        while ... { lock$=1; produce(data); lock$; }
        while ... { lock$=1; consume(data); lock$; }
    }
    ```
Task Intents: Summary of This Effort

- additional task intents are now available
- task intents are now supported on forall loops
  - providing the same protection from data races
- reduce intents are introduced for forall loops
Task Intents: Additional Intents

- these intents are now available as task intents:
  - `in`, `const`, `const in`, `const ref`, `ref`

- e.g. `in` intent introduces a task-private variable

```plaintext
proc divide_and_conquer(low, high, ...) {
  cobegin with (in low, in high) {
    // task 1
    high = (high + low) / 2;
    divide_and_conquer(low, high, ...);
  }
  // task 2
  low = (high + low) / 2 + 1;
  divide_and_conquer(low, high, ...);
}

low and high: start with incoming values, task can adjust as needed
```
Task Intents: Support for `forall` Loops

- task intents are now supported on `forall` loops
  - applied to task constructs in the underlying parallel iterator

```latex
\begin{verbatim}
var A: [1..10] real;
var p, q: real;
forall a in A with (ref q) {
  p += a;
  ...
  q += 1;
  ...
}
\end{verbatim}
```

parallel iteration creates tasks

unintentional race on `p` is a compile-time error

allowed, as requested by user via `ref q` intent
Task Intents: Introducing reduce Intents

- **variables marked with reduce intent:**
  - inside loop: task-private "shadow" variable, initialized to identity value
  - at task completion: accumulated into "outer" variable

```plaintext
var total = 0;
forall a in myIter() with (+ reduce total) {
    total += a;
}
writeln("total = ", total);
```

- **prototype implementation is completed**
  - supported reduction ops: + * && || & ^

  *task-private "shadow" variable*
  sequential access from iterations of given task
  no need for sync or atomic guards

  *"outer" variable*
  accumulates per-task values atomically
Task/Reduce Intents Discussion: Keep Initial Value?

- currently: outer variable’s initial value is discarded
- need to keep it to support nested parallelism naturally
  - example: nested loops

```plaintext
var total: real;
forall x1 in iter_dim1() with (+ reduce total) {
  forall x2 in iter_dim2() with (+ reduce total) {
    total += x1
  }
}
writeln("total = ", total);
```

when starting inner loop, need to retain values accumulated into total so far
Task/Reduce Intents Discussion: Types

● what if: type of reduction result differs from type of individual values?
  ● example: min-k reduction compute k smallest values
    ● values: real
    ● result: k*real

  var result: ??;
  forall a in myIter() with (min10 reduce result) {
    result ?? a;
  }
  writeln("result = ", result);

● even more interesting when input, state, and output types all differ
  ● e.g., given a list of coordinates, produce ID of most populated octant

what is the type of result in these three places?
proposal: 10*real
Task/Reduce Intents Discussion: Shadow Variable

- **shadow variable – task-private or iteration-private?**
  - example: min-k reduction
    compute k smallest values seen

  - if task-private:
    - `result` accumulates partial results for each *task*
    - pro: consistent with other task intents
    - cons: within the loop user has to accumulate explicitly

  - if iteration-private:
    - `result` contains the value to accumulate for each *iteration*
    - pro: accumulation is taken care of by reduction author
    - cons: different behavior than other task intents

```c
forall a in myIter() with (min10 reduce result) {
    min10_accumulate(result, a);
    result = a;
}
```

- the choice is not expected to affect performance
Task/Reduce Intents Current Limitations

Initial implementation of reduce intents in 1.11

- only the standard operator reductions are supported
  - supported: + * && || & | ^
  - need to support: user-defined reductions; min, max, minloc, maxloc

- only forall loops are supported
  - need to support: reduce intents with begin, cobegin, coforall

- iterators that yield outside of task-parallel constructs are not yet supported
  - this includes iterators that invoke other iterators via for or forall
    - which in turn includes our standard domain, array iterators

- reduction result and individual values must be of same type
  - need to relax this restriction
Task Intents: Status and Next Steps

Status:
● task intents are implemented
● … and supported in forall loops
● … with initial implementation of some reduce intents

Next Steps:
● finalize semantics of reduce intents
● syntax for reduce intents with user-defined reductions
● finalize implementation of reduce intents
● implement standard reductions using reduce intents and forall
● optimize performance of reductions
● design language support for partial reductions
Improved Const checking
**Improved Const Checking**

**Background:** `const` variables and formals prevent unintentional modification, enable optimizations

**This Effort:** Add compile-time errors for these cases:
- a field that is a tuple or an array
  
  ```
  record R { var tupleField: 3*int;
      const arrayField: [1..n] real; }
  var varR: R; varR.arrayField[5] = 3.14;
  const constR: R; constR.tupleField = (1,2,3);
  ```

- a `var` alias of a `const` array or domain
  
  ```
  const A: [1..n] real; var Aalias => A;
  ```

**Impact:** Alert users to new task- and forall intents

```
var r: R;
forall i in 1..n do
  r.tupleField += (1,1,1);
```
Standalone Parallel Iterators
Standalone Par Iters: Background

- **Zippered forall loops use leader-follower iterators**
  - leader iterators: create parallelism, assign iterations to tasks
  - follower iterators: serially execute work generated by leader

- **Given…**
  
  ```
  forall (a,b,c) in zip(A,B,C) do
    a = b + alpha * c;
  
  ... A is defined to be the leader
  ... A, B, and C are all defined to be followers
  ```

- **Leaders/followers need to be general for zippered iteration**
  - Leader normalizes the index space via 0-shifting and densifying
  - Yields a tuple of ranges, even in 1D case
  - Followers translate normalized indices back to their index sets
Standalone Par Iters: Background

Historically, all forall loops implemented as leader-follower:

**Leader iterator**

```cpp
iter myiter(param tag: iterKind)
    where tag == iterKind.leader {
        coforall i in 0..#ntasks {
            yield densify(zeroShift(getBlock(i)));
        }
    }
```

**Follower iterator**

```cpp
iter myiter(followThis, param tag: iterKind)
    where tag == iterKind.follower {
        for i in unZeroShift(undensify(followThis)) {
            yield i;
        }
    }
```

**User code**

```cpp
forall i in myiter() {
    body(i);
}
```

**Compiler rewrites as:**

```cpp
for block in myiter(iterKind.leader) {
    for i in myiter(block, iterKind.follower) {
        body(i);
    }
}
Standalone Par Iters: Background

Historically, all forall loops implemented as leader-follower:

**Leader iterator**

```plaintext
iter myiter(param tag: iterKind)
   where tag == iterKind.leader {
      coforall i in 0..#ntasks {
         yield densify(zeroShift(getBlock(i)));
      }
   }
```

**Follower iterator**

```plaintext
iter myiter(followThis, param tag: iterKind)
   where tag == iterKind.follower {
      for i in unZeroShift(undensify(followThis)) {
         yield i;
      }
   }
```

**User code**

```plaintext
forall i in myiter() {
   body(i);
}
```

**After iterator inlining:**

```plaintext
coforall i in 0..#ntasks {
   const followThis = densify(zeroShift(getBlock(i)));
   const myBlock = unZeroShift(undensify(followThis));
   for i in myBlock {
      body(i);
   }
}
Standalone Par Iters: Background

- **For non-zippered loops would like to...**
  - Simplify iterator implementation
  - Avoid overheads due to normalizing in leader and follower
    - it’s pointless to pay these costs when you are your only follower

- **A standalone parallel iterator should:**
  - Create the appropriate amount of parallelism
  - Walk indices serially within each parallel task
  - Yield each index individually
Standalone Par Iters: This Effort

- **Modify forall loop implementation**
  - If appropriate standalone iterator is defined, call it in a single loop
  - If not, fall back to the traditional leader/follower idiom

- **Define standalone parallel iterators for built-in types**
  - Ranges
  - Rectangular Domains/Arrays
  - Associative Domains/Arrays
  - Sparse Domains/Arrays
Standalone Par Iters: This effort

Standalone iterator

\[
\begin{align*}
\text{iter } \text{myiter}(\text{param tag: iterKind}) \\
\text{where } \text{tag} == \text{iterKind.standalone} \{ \\
\text{coforall i in 0..#ntasks } \{ \text{// create parallelism} \\
\text{for j in getBlock(i) } \{ \text{// walk indices for this task} \\
\text{yield j; } \text{// yield individual indices} \\
\}\} \\
\}\}
\end{align*}
\]

User code

```
forall i in myiter() { 
    body(i);
}
```

Compiler rewrites as:

```
for i in myiter(iterKind.standalone) { 
    body(i);
}
```
Standalone Par Iters: This effort

**Standalone iterator**

```plaintext
iter myiter(param tag: iterKind)
    where tag == iterKind.standalone {
        coforall i in 0..#ntasks { // create parallelism
            for j in getBlock(i) { // walk indices for this task
                yield j; // yield individual indices
            }
        }
    }
```

**User code**

```plaintext
forall i in myiter() {
    body(i);
}
```

**After iterator inlining:**

```plaintext
coforall i in 0..#ntasks {
    const myBlock = getBlock(i);
    for j in myBlock do
        body(j);
}
```
Standalone Par Iters: Impact

forall loop generated code size reduced

forall i in 1..10 do writeln(i);
Standalone Par Iters: Status

Status:

- Standalone parallel iterators are supported by the compiler
  - yet, not currently used when loops access outer vars with non-ref intents
    - compiler limitation to be addressed as future work
- Several built-in types now support standalone parallel iterators
  - Associative domains/arrays
  - Ranges
  - Rectangular domains/arrays
  - Sparse domains/arrays
- Some iterator functions now support standalone variants:
  - e.g., glob() iterator
- Also used in other cases where zippering is inappropriate/expensive
  - e.g., walkdirs(), and findfiles() iterators
  - e.g., diamond-tiling iterators used in Colorado State’s ICS 2015 paper*

Standalone Par Iters: Next Steps

Next Steps:

- Implement standalone parallel iterators for more cases
  - AdvancedIters module
  - RandomStream class
  - Distributed domains and arrays
- Ensure standalone iterators are always utilized when available
- Look for optimization opportunities in standalone iterators
- Tackle other aspects of Leader-Follower 2.0 design
- Tighten up “try token” capability which is a big hammer
  - either using constrained generics
  - or more precise “can resolve call” queries
Compilation Time Improvements
Compilation Time Improvements

**Background:** Chapel compile-times are slower than we’d like

*#1 issue:* generated code size, which is overly verbose due to…
  …overly normalized IR resulting in too many unnecessary temps
  …heavy use of generics for core features and lack of related optimizations
  ● e.g., if a generic class’s method does not use generic aspects, don’t specialize it
  …“on by default” features that may not be used in the typical case

**This Effort:** Remove some low-hanging fruit

● By default, Chapel compilations use a runtime “task table”
  ● Used to track tasks for deadlock detection, Ctrl-C task reporting
  ● Implemented using a Chapel associative array
  ● Not a frequently used feature

● So, why not turn off by default and require a compiler flag to enable?
  ● Tradeoff: Speeds most compiles, but requires recompile when desired
  ● --[no-]task-tracking flag controls behavior
Compilation Time Improvements: Impact

**Impact:** Compilation time and code size improved

- Task table eliminated by default on this day
Compilation Time Improvements: Next Steps

Next Steps:

● Look for other similarly low-hanging cases
● Eliminate unnecessary temporaries
  ● Move away from current normalization strategy?
  ● Collapse unnecessary temporaries prior to codegen?
● Optimize methods that are unnecessarily generic
Other Major Language/Compiler Changes
Other Language/Compiler Changes

- **Deprecated placeholder ‘refvar’ syntax**
  - use ‘ref’ instead, e.g.:
    ```
    var a: int;
    ref b = a;
    ```

- **Deprecated use of ‘var’ return intent**
  - use ‘ref’ instead, e.g.:
    ```
    proc foo() ref { ... } // was: proc foo() var { ... }
    ```

- **Deprecated ‘type select’ statement**
  - use ‘select x.type’ instead, e.g.:
    ```
    select x.type { ... } // was: type select x { ... }
    ```

- **read(sync-variable) now generates an error**
  - symmetric with `write(sync-variable)` as of 1.10
Language/Compiler Priorities and Next Steps
Language/Compiler Priorities and Next Steps

● **Improve Reductions:**
  ● complete support for reduce intents
  ● re-implement global-view reductions using reduce intents
  ● optimize performance
  ● support partial reductions

● **Improve Parallel Iterators:**
  ● Use standalone iterators in all applicable cases
  ● Continue adoption of standalone iterators in standard modules
  ● Design Leader-follower 2.0 and a “try token” replacement

● **Address remaining cases of missing const checking**

● **Create story for type selecting unions/dynamic types**

● **Look for additional compile-time improvements**
  ● while continuing to focus primarily on performance of executables
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, URIKA, and YARCDATA. 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.

Copyright 2014 Cray Inc.