Language and Compiler Improvements

Chapel Team, Cray Inc.
Chapel version 1.11
April 2, 2015

Safe Harbor Statement

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. shapshot *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

```
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

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

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

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

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

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

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

when starting inner loop, need to retain values accumulated into `total` *so far*

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

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

what is the type of `result` *in these three places?*
proposal: `10*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

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

```
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);
```

modifications are not allowed

- a `var` alias of a `const` array or domain

```
const A: [1..n] real; var Aalias => A;
```

cannot `var`-alias a `const` array

Impact: Alert users to new task- and forall intents

```
var r: R;
forall i in 1..n do
  r.tupleField += (1,1,1);
```

`forall` intents: `r` is passed by default intent
`r`'s fields are `const`, cannot modify them

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

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

Follower iterator

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

User code

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

Compiler rewrites as:

```
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

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

Follower iterator

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

User code

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

After iterator inlining:

```
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

```
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

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

Compiler rewrites as:

```
for i in myiter(iterKind.standalone) {
  body(i);
}
```

Standalone Par Iters: This effort

Standalone iterator

```
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

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

After iterator inlining:

```
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);
```

Leader-follower

```
chpl_build_bounded_range(INT64(1
), INT64(10),
ret_to_arg_ref_tmp__chpl);
  chpl__iterLF = call_tmp_chpl2;
  _ic__F0_this_chpl =
call_tmp_chpl2;
  _autoCopy_tmp__chpl =
chpl_rt_buildLocaleID(chpl_nodeID,
local_c_sublocid_any_chpl);
  call_tmp_chpl3 =
chpl__initCopy_chpl_rt_localeID_t(_
autoCopy_tmp__chpl);
  call_tmp_chpl4 =
chpl__initCopy_chpl_rt_localeID_t(c
all_tmp_chpl3);
  call_tmp_chpl5 =
chpl_localeID_to_locale(&call_tmp_
chpl4);
  _virtual_method_tmp__chpl =
((object)(call_tmp_chpl5))-
>chpl__cid;

((int64_t(*)(locale))chpl_vmtable[((I
NT64(5) *
_virtual_method_tmp__chpl) +
INT64(1))])(call_tmp_chpl5);
  this_chpl = &_ic__F0_this_chpl;
  call_tmp_chpl6 =
length(this_chpl);
  v_chpl = call_tmp_chpl6;
  call_tmp_chpl7 =
chpl_task_getSerial();
  if (call_tmp_chpl7) {
    tmp_chpl = INT64(1);
  } else {
    call_tmp_chpl8 =
_computeNumChunks_chpl2(call_t
mp_chpl6);
    tmp_chpl = call_tmp_chpl8;
  }
  numChunks_chpl = tmp_chpl;
  call_tmp_chpl9 = (tmp_chpl ==
INT64(1));
  if (call_tmp_chpl9) {
    call_tmp_chpl10 =
(call_tmp_chpl6 - INT64(1));
    ret_to_arg_ref_tmp__chpl2 =
&call_tmp_chpl11;

chpl_build_bounded_range(INT64(0
), call_tmp_chpl10,
ret_to_arg_ref_tmp__chpl2);
    _ic__F0_this_chpl2 =
call_tmp_chpl2;
    myFollowThis_chpl =
call_tmp_chpl11;
    ret_chpl =
(&_ic__F0_this_chpl2)->_low;
    ret_chpl2 =
(&_ic__F0_this_chpl2)->_high;
    call_tmp_chpl12 = (ret_chpl >
ret_chpl2);
    if (call_tmp_chpl12) {
      tmp_chpl2 = false;
    } else {
      tmp_chpl2 = true;
    }
    call_tmp_chpl13 = (! tmp_chpl2);
    if (call_tmp_chpl13) {
      ret_chpl3 =
(&_ic__F0_this_chpl2)->_low;
      ret_chpl4 =
(&_ic__F0_this_chpl2)->_high;
      call_tmp_chpl14 = (ret_chpl3 >
ret_chpl4);
      if (call_tmp_chpl14) {
        ret_chpl5 =
(&myFollowThis_chpl)->_low;
        ret_chpl6 =
(&myFollowThis_chpl)->_high;
        call_tmp_chpl15 = (ret_chpl5
> ret_chpl6);
        call_tmp_chpl16 = (!
call_tmp_chpl15);
        if (call_tmp_chpl16) {
          halt("zippered iteration with a
range has non-equal lengths",
INT64(2), "forallRange.chpl");
        }
      } else {
        halt("iteration over a range
with no first index", INT64(2),
"forallRange.chpl");
      }
    }
    ret_chpl7 =
(&myFollowThis_chpl)->_low;
    ret_chpl8 =
(&myFollowThis_chpl)->_high;
    call_tmp_chpl17 = (ret_chpl7 >
ret_chpl8);
    if (call_tmp_chpl17) {
      tmp_chpl3 = false;
    } else {
      tmp_chpl3 = true;
    }
    call_tmp_chpl18 = (! tmp_chpl3);
    if (call_tmp_chpl18) {
      ret_chpl9 =
(&myFollowThis_chpl)->_low;
      ret_chpl10 =
(&myFollowThis_chpl)->_high;
      call_tmp_chpl19 = (ret_chpl9 >
ret_chpl10);
      call_tmp_chpl20 = (!
call_tmp_chpl19);
      if (call_tmp_chpl20) {
        halt("zippered iteration over a
range with no first index", INT64(2),
"forallRange.chpl");
      }
    }
    _ref_tmp__chpl =
&myFollowThis_chpl;
    call_tmp_chpl21 =
length(_ref_tmp__chpl);
    ret_to_arg_ref_tmp__chpl3 =
&r_chpl;

chpl_build_bounded_range2(INT64(
1), INT64(0),
ret_to_arg_ref_tmp__chpl3);
    call_tmp_chpl22 =
(call_tmp_chpl21 != INT64(0));
    if (call_tmp_chpl22) {
      ret_chpl11 =
(&myFollowThis_chpl)->_low;
      this_chpl2 =
&_ic__F0_this_chpl2;
      call_tmp_chpl23 =
orderToIndex(this_chpl2,
ret_chpl11);
      call_tmp_chpl24 =
(call_tmp_chpl21 - INT64(1));
      call_tmp_chpl25 =
(call_tmp_chpl23 +
call_tmp_chpl24);
      ret_chpl12 =
(&myFollowThis_chpl)->_high;
      this_chpl3 =
&_ic__F0_this_chpl2;
      call_tmp_chpl26 =
orderToIndex(this_chpl3,
ret_chpl12);
      call_tmp_chpl27 =
(call_tmp_chpl25 ==
call_tmp_chpl26);
      assert_chpl(call_tmp_chpl27);
      ret_to_arg_ref_tmp__chpl4 =
&call_tmp_chpl28;

chpl_build_bounded_range(call_tm
p_chpl23, call_tmp_chpl25,
ret_to_arg_ref_tmp__chpl4);
      r_chpl = call_tmp_chpl28;
    }
    _ic__F0_this_chpl3 = r_chpl;
    ret_chpl13 =
(&_ic__F0_this_chpl3)->_low;
    ret_chpl14 =
(&_ic__F0_this_chpl3)->_high;
    end_chpl = ret_chpl14;
    CHPL_PRAGMA_IVDEP
    for (i_chpl = ret_chpl13; ((i_chpl
<= end_chpl)); i_chpl += INT64(1))
{
      writeln_chpl4(i_chpl);
    }
  } else {
    call_tmp_chpl29 =
sizeof(chpl_chpl___EndCount_obje
ct);
    cast_tmp_chpl =
chpl_here_alloc(call_tmp_chpl29,
INT16(17));
    this_chpl4 =
((chpl___EndCount)(cast_tmp_chpl
));
    ((object)(this_chpl4))->chpl__cid
= chpl__cid_chpl___EndCount;
    (this_chpl4)->i =
_init_class_tmp__chpl;
    (this_chpl4)->taskCnt =
INT64(0);
    (this_chpl4)->taskList = NULL;
    ret_chpl15 = type_tmp_chpl;
    _ref_tmp__chpl2 = &ret_chpl15;

atomic_init_int_least64_t(_ref_tmp_
_chpl2, INT64(0));
    (&this_chpl5)->_v = ret_chpl15;
    wrap_call_tmp_chpl =
_construct_atomic_int64(ret_chpl15
, &this_chpl5);
    (this_chpl4)->i =
wrap_call_tmp_chpl;
    (this_chpl4)->taskCnt =
INT64(0);
    ret_chpl16 = NULL;
    (this_chpl4)->taskList =
ret_chpl16;
    wrap_call_tmp_chpl2 =
_construct__EndCount(&wrap_call_
tmp_chpl, INT64(0), ret_chpl16,
this_chpl4);
    _coforallCount_chpl =
wrap_call_tmp_chpl2;
    ret_to_arg_ref_tmp__chpl5 =
&call_tmp_chpl30;

chpl_build_partially_bounded_range
(INT64(0),
ret_to_arg_ref_tmp__chpl5);
    ret_to_arg_ref_tmp__chpl6 =
&call_tmp_chpl31;

chpl___POUND_(&call_tmp_chpl30
, tmp_chpl,
ret_to_arg_ref_tmp__chpl6);
    _ic__F0_this_chpl4 =
call_tmp_chpl31;
    ret_chpl17 =
(&_ic__F0_this_chpl4)->_low;
    ret_chpl18 =
(&_ic__F0_this_chpl4)->_high;
    end_chpl2 = ret_chpl18;
    for (i_chpl2 = ret_chpl17;
((i_chpl2 <= end_chpl2)); i_chpl2
+= INT64(1)) {

_upEndCount(_coforallCount_chpl);
      rvfDerefTmp_chpl =
_coforallCount_chpl;
      chpl_here_alloc_size =
sizeof(chpl__class_localscoforall_fn
_chpl_object);
      chpl_here_alloc_tmp =
chpl_here_alloc(chpl_here_alloc_si
ze, INT16(23));
      _args_forcoforall_fn_chpl =
((_class_localscoforall_fn_chpl)(chp
l_here_alloc_tmp));
      (_args_forcoforall_fn_chpl)-
>_0_v_chpl = v_chpl;
      (_args_forcoforall_fn_chpl)-
>_1_numChunks_chpl =
numChunks_chpl;
      (_args_forcoforall_fn_chpl)-
>_2__yieldedIndex_chpl = i_chpl2;
      (_args_forcoforall_fn_chpl)-
>_3_rvfDerefTmp_chpl =
rvfDerefTmp_chpl;
      (_args_forcoforall_fn_chpl)-
>_4_rvfDerefTmp_chpl =
chpl__iterLF;
      /*** wrapcoforall_fn_chpl ***/
chpl_taskListAddCoStmt(INT32(-2),
INT64(1),
((void*)(_args_forcoforall_fn_chpl)),
&(((_args_forcoforall_fn_chpl)-
>_3_rvfDerefTmp_chpl)->taskList),
chpl_nodeID, INT64(2),
"forallRange.chpl");
    }

chpl_taskListProcess((_coforallCou
nt_chpl)->taskList, INT64(2),
"forallRange.chpl");

_waitEndCount(_coforallCount_chpl
);
    delete_tmp_chpl =
_coforallCount_chpl;
    _field_destructor_tmp__chpl =
&((delete_tmp_chpl)->i);
    call_tmp_chpl32 =
&((_field_destructor_tmp__chpl)-
>_v);

atomic_destroy_int_least64_t(call_t
mp_chpl32);
    call_tmp_chpl33 =
((void*)(delete_tmp_chpl));
    chpl_here_free(call_tmp_chpl33);
  }
```

Standalone

```
chpl_build_bounded_range(INT64(1),
INT64(10), ret_to_arg_ref_tmp__chpl);
  _ic__F0_this_chpl = call_tmp_chpl2;
  this_chpl = &_ic__F0_this_chpl;
  call_tmp_chpl3 = length(this_chpl);
  len_chpl = call_tmp_chpl3;
  call_tmp_chpl4 =
chpl_task_getSerial();
  if (call_tmp_chpl4) {
    tmp_chpl = INT64(1);
  } else {
    call_tmp_chpl5 =
_computeNumChunks_chpl2(call_tmp
_chpl3);
    tmp_chpl = call_tmp_chpl5;
  }
  numChunks_chpl = tmp_chpl;
  call_tmp_chpl6 = (tmp_chpl <=
INT64(1));
  if (call_tmp_chpl6) {
    _ic__F0_this_chpl2 =
_ic__F0_this_chpl;
    ret_chpl = (&_ic__F0_this_chpl2)-
>_low;
    ret_chpl2 = (&_ic__F0_this_chpl2)-
>_high;
    end_chpl = ret_chpl2;
    CHPL_PRAGMA_IVDEP
    for (i_chpl = ret_chpl; ((i_chpl <=
end_chpl)); i_chpl += INT64(1)) {
      writeln_chpl4(i_chpl);
    }
  } else {
    call_tmp_chpl7 =
sizeof(chpl_chpl___EndCount_object);
    cast_tmp_chpl =
chpl_here_alloc(call_tmp_chpl7,
INT16(17));
    this_chpl2 =
((chpl___EndCount)(cast_tmp_chpl));
    ((object)(this_chpl2))->chpl__cid =
chpl__cid_chpl___EndCount;
    (this_chpl2)->i =
_init_class_tmp__chpl;
    (this_chpl2)->taskCnt = INT64(0);
    (this_chpl2)->taskList = NULL;
    ret_chpl3 = type_tmp_chpl;
    _ref_tmp__chpl = &ret_chpl3;

atomic_init_int_least64_t(_ref_tmp__c
hpl, INT64(0));
    (&this_chpl3)->_v = ret_chpl3;
    wrap_call_tmp_chpl =
_construct_atomic_int64(ret_chpl3,
&this_chpl3);
    (this_chpl2)->i =
wrap_call_tmp_chpl;
    (this_chpl2)->taskCnt = INT64(0);
    ret_chpl4 = NULL;
    (this_chpl2)->taskList = ret_chpl4;
    wrap_call_tmp_chpl2 =
_construct__EndCount(&wrap_call_tm
p_chpl, INT64(0), ret_chpl4,
this_chpl2);
    _coforallCount_chpl =
wrap_call_tmp_chpl2;
    ret_to_arg_ref_tmp__chpl2 =
&call_tmp_chpl8;

chpl_build_partially_bounded_range(IN
T64(0), ret_to_arg_ref_tmp__chpl2);
    ret_to_arg_ref_tmp__chpl3 =
&call_tmp_chpl9;
    chpl___POUND_(&call_tmp_chpl8,
tmp_chpl, ret_to_arg_ref_tmp__chpl3);
    _ic__F0_this_chpl3 =
call_tmp_chpl9;
    ret_chpl5 = (&_ic__F0_this_chpl3)-
>_low;
    ret_chpl6 = (&_ic__F0_this_chpl3)-
>_high;
    end_chpl2 = ret_chpl6;
    for (i_chpl2 = ret_chpl5; ((i_chpl2 <=
end_chpl2)); i_chpl2 += INT64(1)) {

      _upEndCount(_coforallCount_chpl);
      this_chpl4 = &_ic__F0_this_chpl;
      rvfDerefTmp_chpl =
_coforallCount_chpl;
      chpl_here_alloc_size =
sizeof(chpl__class_localscoforall_fn_c
hpl_object);
      chpl_here_alloc_tmp =
chpl_here_alloc(chpl_here_alloc_size,
INT16(23));
      _args_forcoforall_fn_chpl =
((_class_localscoforall_fn_chpl)(chpl_h
ere_alloc_tmp));
      (_args_forcoforall_fn_chpl)-
>_0_this_chpl = this_chpl4;
      (_args_forcoforall_fn_chpl)-
>_1_rvfDerefTmp_chpl =
rvfDerefTmp_chpl;
      (_args_forcoforall_fn_chpl)-
>_2__yieldedIndex_chpl = i_chpl2;
      (_args_forcoforall_fn_chpl)-
>_3_numChunks_chpl =
numChunks_chpl;
      (_args_forcoforall_fn_chpl)-
>_4_len_chpl = len_chpl;
      /*** wrapcoforall_fn_chpl ***/
chpl_taskListAddCoStmt(INT32(-2),
INT64(1),
((void*)(_args_forcoforall_fn_chpl)),
&(((_args_forcoforall_fn_chpl)-
>_1_rvfDerefTmp_chpl)->taskList),
chpl_nodeID, INT64(2),
"forallRange.chpl");
    }

chpl_taskListProcess((_coforallCount_
chpl)->taskList, INT64(2),
"forallRange.chpl");

    _waitEndCount(_coforallCount_chpl);
    delete_tmp_chpl =
_coforallCount_chpl;
    _field_destructor_tmp__chpl =
&((delete_tmp_chpl)->i);
    call_tmp_chpl10 =
&((_field_destructor_tmp__chpl)->_v);

atomic_destroy_int_least64_t(call_tmp
_chpl10);
    call_tmp_chpl11 =
((void*)(delete_tmp_chpl));
    chpl_here_free(call_tmp_chpl11);
  }
```

About 50% reduction in generated code size

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*

* I. Bertolacci, C. Olschanowsky, B. Harshbarger, D. Wannacott, B. Chamberlain, and M. Strout. *Parameterized Diamond Tiling for Stencil Computations with Chapel Iterators*. To appear in *ACM International Conference on Supercomputing (ICS)*, 2015.

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

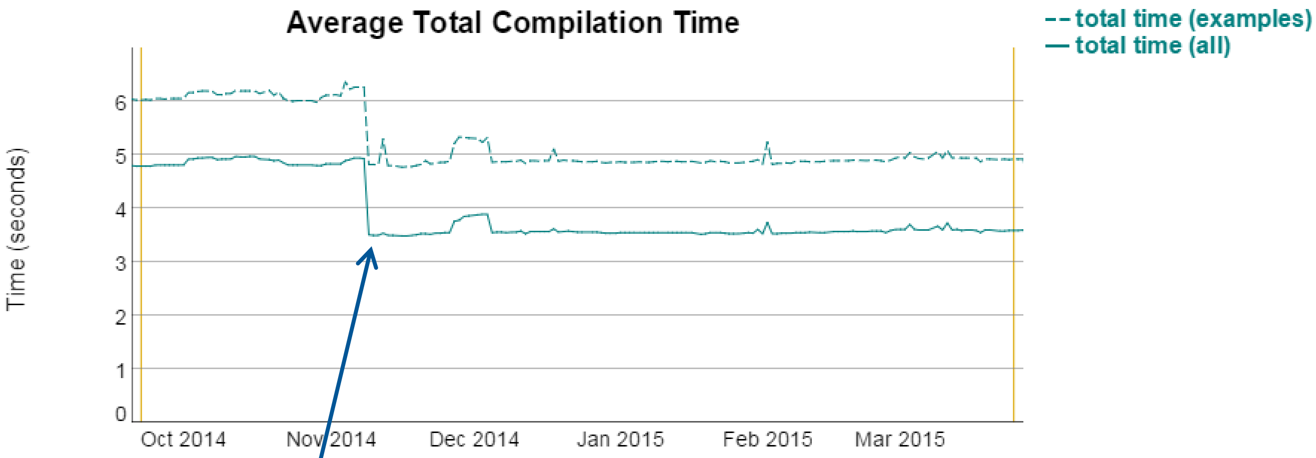
- **#1 issue:** generated code size, which is overly verbose due to…
  - …overly normalized IR resulting in too many unnecessary temps
  - …heavy use of generics for core features and lack of related optimizations
    - e.g., if a generic class's method does not use generic aspects, don't specialize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

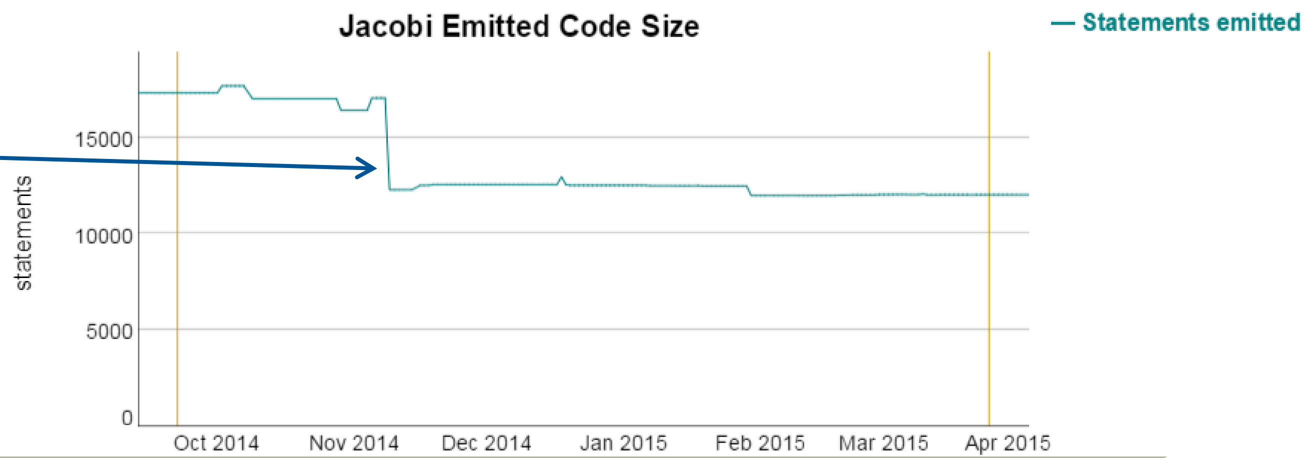
This Effort: Remove some low-hanging fruit

- 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.

http://chapel.cray.com chapel_info@cray.com http://sourceforge.net/projects/chapel/