Data-Centric Locality in Chapel

Ben Harshbarger
Cray Inc.

CHIUW 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

- Background on multi-locale implementation
- 1.10 multi-locale performance issues
- Data-centric approaches to improving performance
- Performance results
- Future work

Background on multi-locale implementation

- 'Wide pointers' represent potentially remote vars

```
typedef struct {
  int localeID: // where this object lives
  myClass addr; // pointer to data
} wide_myClass;
```

- Runtime GETs and PUTs used to read/write data

Compiler-generated C

```
void bar(wide_myClass foo) {
  myClass local_foo;
  local_foo = comm_get(foo.locale, foo.addr);
  // do things with 'local_foo'
}
```

Background on multi-locale implementation

- Runtime will avoid unnecessary communication

```
void* comm_get(int id, void* addr) {
  if (id == here.id) {
    return addr;
  } else {
    // comm layer call
  }
}
```

1.10 multi-locale performance issues

- **Wide pointers are conservatively introduced**
  - Simple implementation
  - Easier to ensure program correctness

- **Local data is often represented with wide pointers**
  - Unnecessary overhead

- **This is particularly bad for arrays**
  - Wide pointer overhead for every array access
  - May prevent back-end C compiler optimizations

1.10 multi-locale performance issues

- **Functions have the most general declaration**

- **If an argument is wide, the function formal will be wide**
  - Insert a temporary if the actual is not wide
  - Particularly bad for member functions

```
void myClass_foo(wide_myClass this) { … }

wide_myClass X;
myClass_foo(X); // 'bar' now has to be wide

myClass Y;
wide_myClass temp;
temp.locale = here.id;
temp.addr = Y;
myClass_foo(temp);
```

1.10 multi-locale performance issues

- **Before 1.11, all fields were conservatively widened**
  - If that field was a class

- **Again, especially bad for arrays**

Chapel code

```
// Simplified internal array representation
class ArrayClass {
  var dom : domain;
  var data : cPtr(int);   // wide pointer
}
```

1.10 multi-locale performance issues

- The ‘local’ block tends to save us in distributed code

```
// Simplified implementation of distributed array access
proc DistArray.access(var idx : int) {
  local {
    if isLocalIdx(idx) then return locData[idx];
  }
  // remote code
}
```

- Assertion that no communication is required
  - Informs the compiler to not insert wide pointers

- Pros: simple implementation, good performance
- Cons: Imprecise, scoping issues

Data-centric improvements

- Problem: too many coarse-grained decisions
- Compiler: “every field must be wide”
- Developer: “everything in this block is local”
- Better: reason about locality on a data-centric basis
- Goal: get rid of the local block

Data-centric improvements – local fields

- **New in 1.11: the "local field" pragma**

- **Allows class designers to assert locality for each field**
  - Only works for class fields within an aggregate type
  - Automatically applied to arrays in an aggregate type

```
// Simplified internal array representation
class ArrayClass {
  var dom : domain;

  pragma "local field"
  var data : cPtr(int);   // compiler can reduce overhead
}
proc ArrayClass.check() {
  return this.locale.id == data.locale.id;
}
```

Data-centric improvements – local fields

- **Applied this pragma to C pointers in DefaultRectangular**
  - DefaultRectangular is…
    …the domain map used to implement local arrays by default
    …also used as the guts of virtually every other domain map (e.g., Block)
  - Its pointers should never point to remote data
  - Represents a significant source of overhead given its widespread use

- **Runtime checks inserted to ensure correctness**
  - Invoked on reads or writes of such fields
  - Generates runtime error if field is assigned remote data
  - Can disable with "--no-local-checks"
    - Or with --no-checks or --fast

Data-centric improvements – functions

- Arrays and domains are implemented as classes

- Compiler tends to widen the "this" argument

```
void myClass_foo(wide_myClass this, …) { … }
```

- Need to insert wide temps if actual isn't wide

Data-centric improvements – functions

- **Possible solution: duplicate member functions**

```
void myClass_foo(wide_myClass this, …) { … }
void myClass_foo(myClass this, …) { … }
```

- **Currently implemented on a dev branch**
  - Complicates implementation
  - Larger generated-C code size
  - Positive performance improvements

Performance results

- **Collected on 64-bit Linux with 2 quad core (8 HT) Intel Xeon processors**
  - 8 cores, 16 threads, 48GB ram
- **Numbers gathered using 1.10 and 1.11 releases**

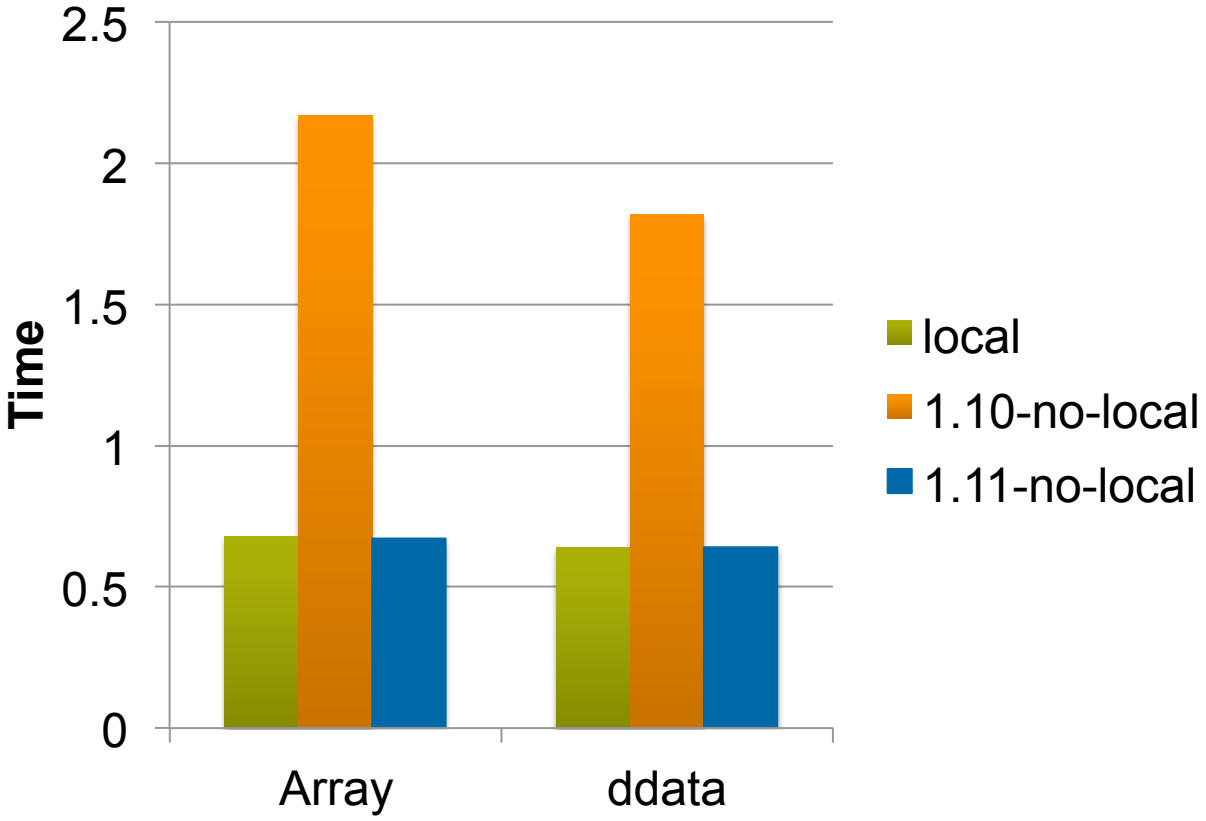
Performance results

- **No multi-locale numbers (yet)**
  - "local" block squashes most distributed overhead

- **These numbers compare local vs no-local**
  - local: compiling without a comm layer, zero wide pointers
  - no-local: inserts wide pointers, even if no comm layer is selected

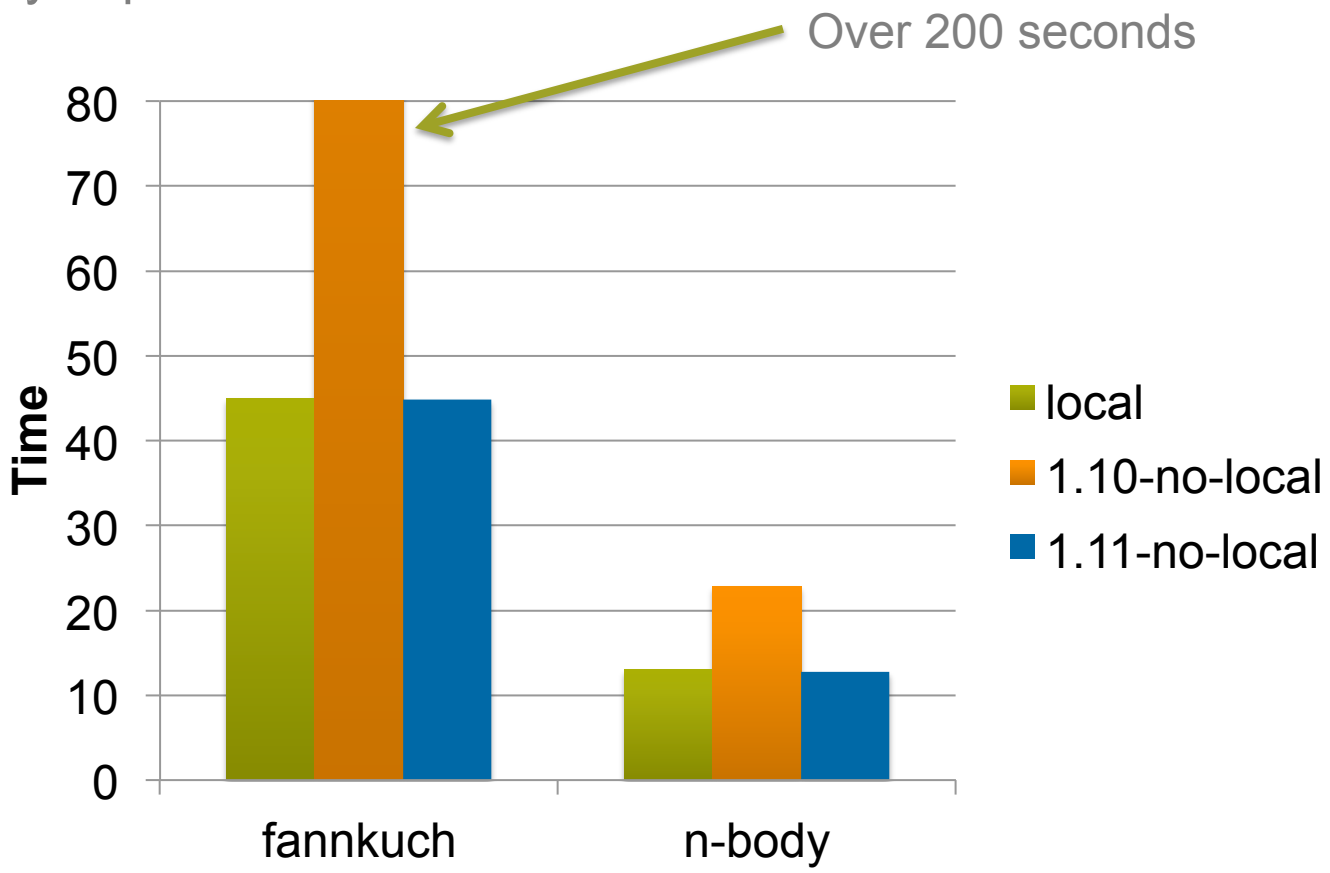
Performance results

- **Serial array iteration**
  - Mostly improved due to local fields



Performance results

- **Computer Language Benchmarks Game**
  - Mostly improved due to local fields



Performance results

- **HPCC STREAM-EP: Background**

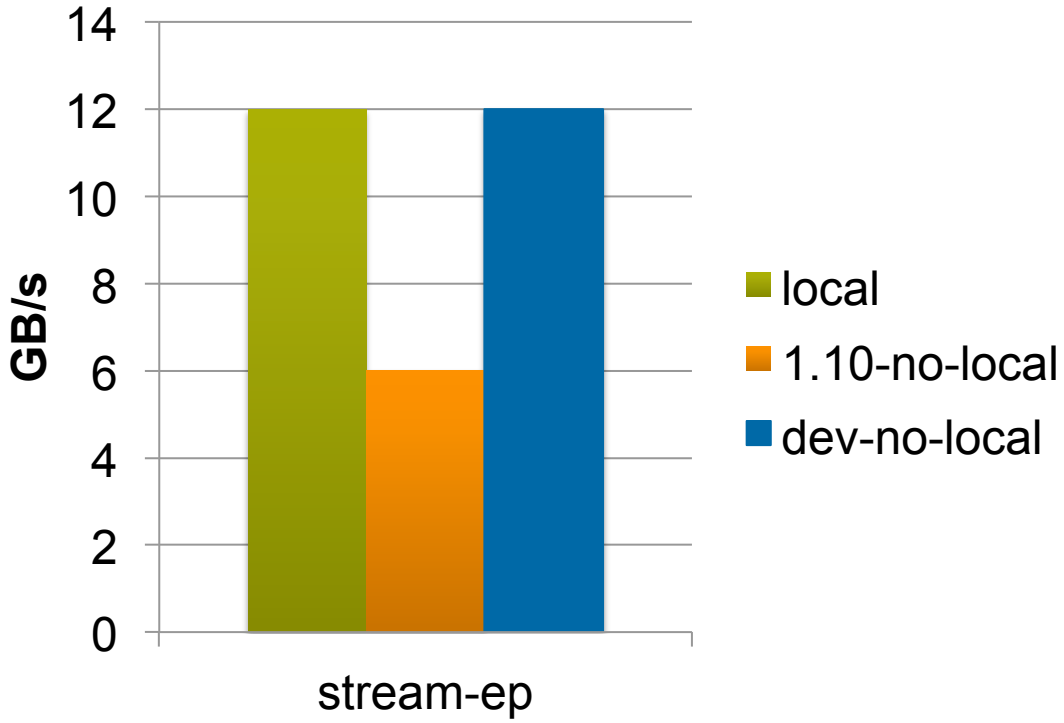
```
coforall loc in Locales do on loc {
  local {  // shouldn't need this, clearly no communication
      var A, B, C : [1..n] real;
      const alpha = 3.0;

      initVectors(B, C);

      for trial in 1..numTrials {
        forall (a, b, c) in zip(A, B, C) do
          a = b + alpha * c;
      }
  }
}
```

Performance results

- **STREAM-EP (without local block)**
  - Bigger is better
  - Mostly improved due to function duplication
  - Gathered with clang 5.1



Future work

- **Use "local field" pragma in more places**

- **Replace pragma with a robust language-level construct**
  - Not just fields
    - Array elements
    - Regularly-scoped variables
  - Still in design phase
    - But here's an idea:

```
var baz : local Foo;

var data : [1..10] local Foo;

// Instead of a pragma…
class Bar {
  var f : local Foo;
}
```

Future work

- **Deprecate the 'local' block**
  - This statement is imprecise
  - Scoping rules limit its applicability
  - We would prefer finer-grained, data-centric locality assertions

- **Support Local Array Views**
  - Often a program wants to only work with local array data
    - typically results in similarly conservative "is this element remote?" checks
  - Doing so today is possible, but a bit clunky
  - Sketch of concept:

```
var myLocArrElts = Arr[local];
…myLocArrElts[i,j]…     // fast local access to A[i,j]; OOB if (i,j) is remote
```

  - Current array-view effort provides a framework for this feature

Future work

- Given "`on foo do` …"

- **Avoid on-statement overhead**
  - If `foo` is local, we can avoid runtime overhead for on-statements
  - Namely, avoid allocating bundled arguments
    - This is important for atomic operations, which have on-statements

- **Optimize `foo` within the on-statement**
  - By definition, the on-statement will execute on `foo`'s locale
  - Thus, we know references to `foo` are local within the on-statement

Summary

- Allowing developers to assert locality is valuable
- The compiler should (and can be) smarter about locality
- These two factors should result in improved performance

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.