Array, Domain, & Domain Map Improvements

Chapel Team, Cray Inc.

Chapel version 1.16

October 5, 2017

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

- Block Sparse Locality Improvements
- Replicated Distribution Improvements
- Associative Array Locking Improvements
- Array View Improvements
- Compressed Sparse Row/Column Layouts (CSR/CSC)
- Optimizing Sparse bulkAdd
- Bulk Array Expansion
- Parallel Array Initialization
- Other Array, Domain, Domain Map Improvements

Block-Sparse Locality Improvements

Block-Sparse Locality: Background

Background:

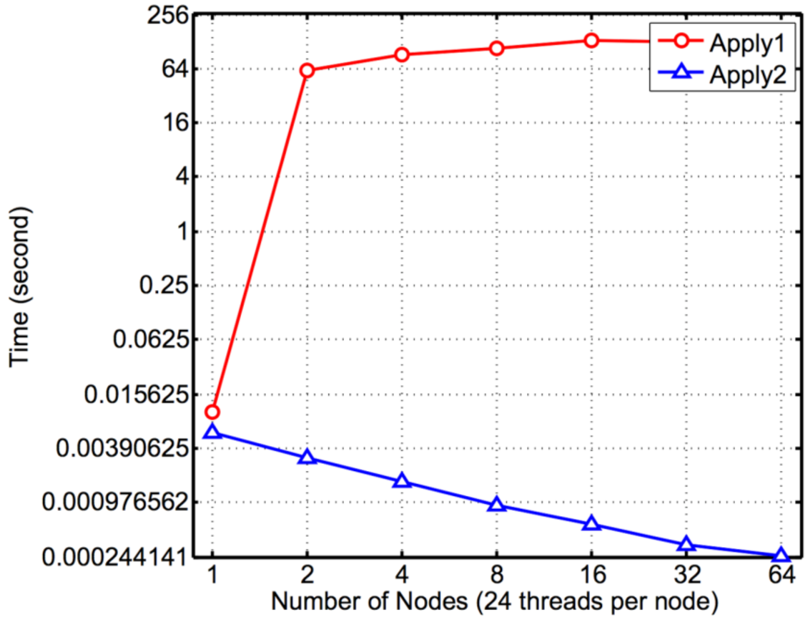
- Block-distributed sparse arrays were introduced in Chapel 1.14
- Performance was far from ideal
  - a known issue / TODO: lack of privatization
  - a significant bug, revealed in Azad and Buluç's CHIUW 2017 paper, *Towards a GraphBLAS Library in Chapel:*

preferred, high-level version:

```
forall a in spArr do
  a = unaryOp(a);
```

lower-level workaround:

```
coforall locArr in locArrs do
  on locArr do
    forall a in locArr.myElems do
      a = unaryOp(a);
```



Block-Sparse Locality: This Effort

This Effort:

- added privatization to Block-Sparse domains/arrays
  - *privatization:* localizing key descriptors to each target locale
  - contributed by Engin Kayraklioglu (GWU)
- fixed the bug revealed by CHIUW paper
  - parallel array iterator had the wrong “on-clause”, ran everything locally

```
-     coforall locA in locArr do on locArr {
+     coforall locA in locArr do on locA {
        // forward to sparse standalone iterator
        for i in locA.myElems._value.these(tag) {
          yield i;
```

  - unfortunately, not reported prior to publication, so not fixed for Chapel 1.15

Block-Sparse Locality: Impact

Impact:

- simple loops on a 1000 x 1000 matrix, 10% sparse, 4 locales

	`forall ij in SD do` `SA[ij] = …;`				`forall a in SA do` `a = …;`			
	ons	gets	puts	time (100 trials)	ons	gets	puts	time (100 trials)
1.15	3 0 0 0	0 4,227,000 3,675,000 3,675,000	0	107.44 sec	0	225,009 0 0 0	75,000 0 0 0	15.37 sec
1.16	3 0 0 0	0	0	0.162 sec	3 0 0 0	0	0	0.0193 sec

Block-Sparse Locality: Status and Next Steps

Status:

- Block-Sparse domains and arrays are now much more scalable

Next Steps:

- Further evaluation and tuning of sparse operations

Replicated Distribution Improvements

Replicated: Background

Background:

- Chapel has long supported a 'Replicated' distribution
  - concept: each target locale stores the entire domain/array (a 'replicand')
  - for example, each locale would store an *n*-ary domain/array given:

```
const D = {1..n} dmapped ReplicatedDist();
var A: [D] real;
```

- Certain aspects of its behavior have been surprising / unusual
  - e.g., the following loop traversed all elements in all replicands

```
forall a in A do …    // does O(n*numLocales) work rather than O(n)
```

  - e.g., '.size()' would return '*n*numIndices*' rather than simply '*n*'

- Other minor pain points:
  - no way to easily access another locale's replicand
  - distribution class was named 'ReplicatedDist', as was module
    - contrast with 'Block' distribution defined in 'BlockDist' module
  - target locale array had an unusual "consistency" requirement

Replicated: Consistency Improvements

This Effort: Improved naming and behavior

- Replaced 'ReplicatedDist' with 'Replicated'
- Make 'Replicated' operations refer to local data only
  - Consider the following declarations, running on two locales:

```
const D = {1..5} dmapped …;
var A: [D] real;
```

code	ReplicatedDist (1.15)	Replicated (1.16)
`for[all] i in D do…` `for[all] a in A do…`	iterated over all 10 indices / elements, generating communication	iterates over 5 indices / local elements
`D.size() / A.size()`	10	5
`writeln(D);`	{1..5} replicated over LOCALE0 LOCALE1	{1..5}
`writeln(A);`	LOCALE0: 0.0 0.0 0.0 0.0 0.0 LOCALE1: 0.0 0.0 0.0 0.0 0.0	0.0 0.0 0.0 0.0 0.0

Replicated: Other improvements

This Effort: Implemented other improvements

- "Added '.replicand()' method to access a remote locale's copy
  - e.g., could simulate old writeln(A) behavior via:

```
for loc in Locales {
  writeln(loc, ":")
  writeln(A.replicand(loc.id));
}
```

- Removed "consistency" requirement on target locale set
- Added a Replicated-specific primer, improved documentation
  - see: http://chapel.cray.com/docs/1.16/primers/replicated.html

Replicated: Status and Next Steps

Status:

- Preserved 'ReplicatedDist' for 1.16
  - maintains backwards compatibility for existing users
  - generates a deprecation warning
  - will be retired in 1.17

- "But what if I liked the old behavior?"
  - can still get it via manual rewrites:

```
coforall loc in Locales do    // implement old forall behavior
  on loc do
    forall i in D do …
```

Next Steps:

- Gain additional experience with 'Replicated' and improve it as needed

Associative Array Locking Improvements

Associative Locking: Background

Background:

- Traditionally, associative array accesses have been guarded by locks

```
proc AssocArray.access(idx) ref {
  lock$ = true;      // take lock
  ref elt = …;       // take reference to element
  lock$;             // release lock
  return elt;        // return reference
}
```

  - presumably to avoid races in the event that the array was being resized

```
cobegin {
  Age["abe"] += 1;        // update an existing person's age
  People += "billy";      // add a new person, growing the array
}
```

- However, such guards don't actually provide safety
  - the array could still be resized between the return of the ref and its use
  - the lock would need to surround the entire assignment to be effective

Associative Locking: This Effort

This Effort:

- remove locking on associative array accesses
- makes them more similar to other array types
  - e.g., rectangular arrays are also subject to such races

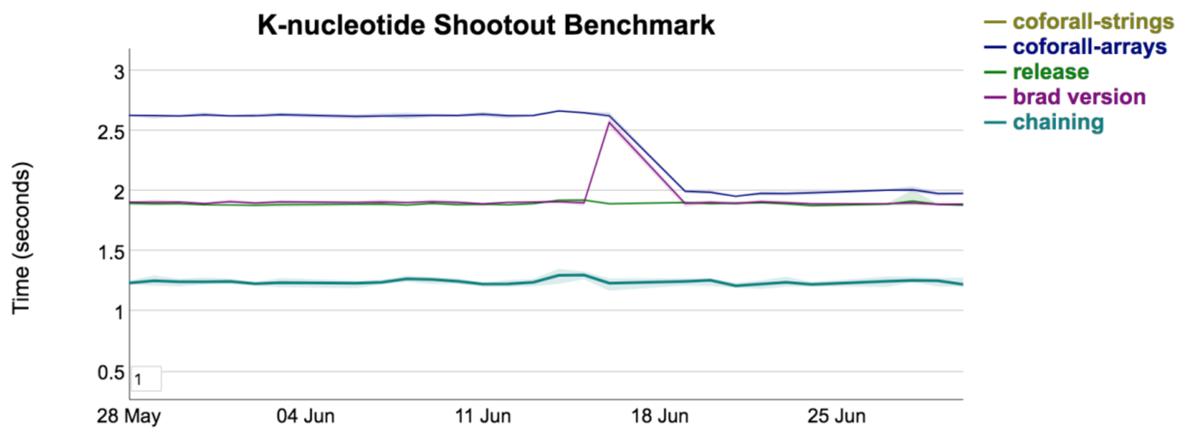
    ```
    cobegin {
      A[i,j] += 1;
      D = {1..2*m, 1..2*n};
    }
    ```

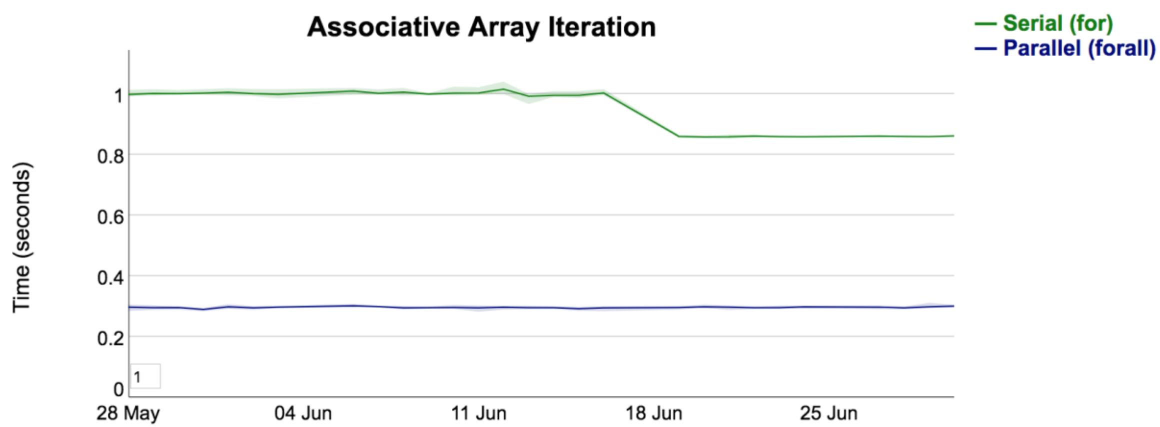
  - such cases have always been considered the user's responsibility

Associative Locking: Impact

Impact:

- improved performance for cases using associative array accesses





Associative Locking: Next Steps

Next Steps:

- Look into techniques to improve safety in such cases
  - compiler analysis?
  - (expensive) opt-in execution-time techniques?

Array View Improvements

Array Views: Background

Background:

- Chapel 1.15 introduced the concept of *array views*
  - implement array slicing, rank-change, and reindexing via indirection
  - made these operations more robust
  - simplified authoring new domain maps
- Some work was left unfinished in 1.15
  - domains/dists of reindexed distributed arrays did not preserve distribution
  - domains/dists of rank-change / reindex arrays were only stored on locale 0

```
var MyDistArr: [{1..m, 1..n} dmapped Block(…)] real;
ref ZeroBasedArr = MyDistArr.reindex({0..#m, 0..#n});
foo(ZeroBasedArr);
proc foo(X: [?D]) {
  var Y: [D] real;       // Y was not distributed as you'd think it would be
  on Locales[2] {
    const s = D.size;  // locale 2 had no local copy of D ⇒ comm. required
  }
}
```

Array Views: This Effort

This Effort:

- Fixed the aforementioned issues
  - domains and distributions of distributed array views…
    - …preserve locality
    - …are privatized

```
var MyDistArr: [{1..m, 1..n} dmapped Block(…)] real;
ref ZeroBasedArr = MyDistArr.reindex({0..#m, 0..#n});
foo(ZeroBasedArr);
proc foo(X: [?D]) {
  var Y: [D] real;      // Y is now distributed like X / MyDistArray
  on Locales[2] {
    const s = D.size; // locale 2 has a local copy of D; can compute locally
  }
}
```

Array Views: Impact and Next Steps

Impact:

- Array views of distributed arrays behave as you'd expect
- Communication requirements have been reduced in such cases

Next Steps (see 1.15 release notes for details):

- Restore ability to pass array views to default initializers
- Fix type query behavior for array views

Compressed Sparse Row/Column Layouts (CSR/CSC)

CSR/CSC

Background: Chapel has supported CSR layouts, but not CSC

- CSR (Compressed Sparse Row) supported by LayoutCSR
- CSC (Compressed Sparse Column) has not been supported
  - useful for interoperability, efficient CSR * CSC matrix multiplication, …

This Effort:

- Generalized LayoutCSR to support CSC as well:

```
// CSC specified through param-argument
var cscD: sparse subdomain(D) dmapped CS(compressRows=false);
// CSR is the default
var csrD: sparse subdomain(D) dmapped CS();
```

Status:

- Replaced LayoutCSR with LayoutCS
- Deprecated LayoutCSR (will be removed in future releases)

Next Steps: improve performance of CSR, CSC layouts

Optimizing Sparse bulkAdd

Sparse bulkAdd

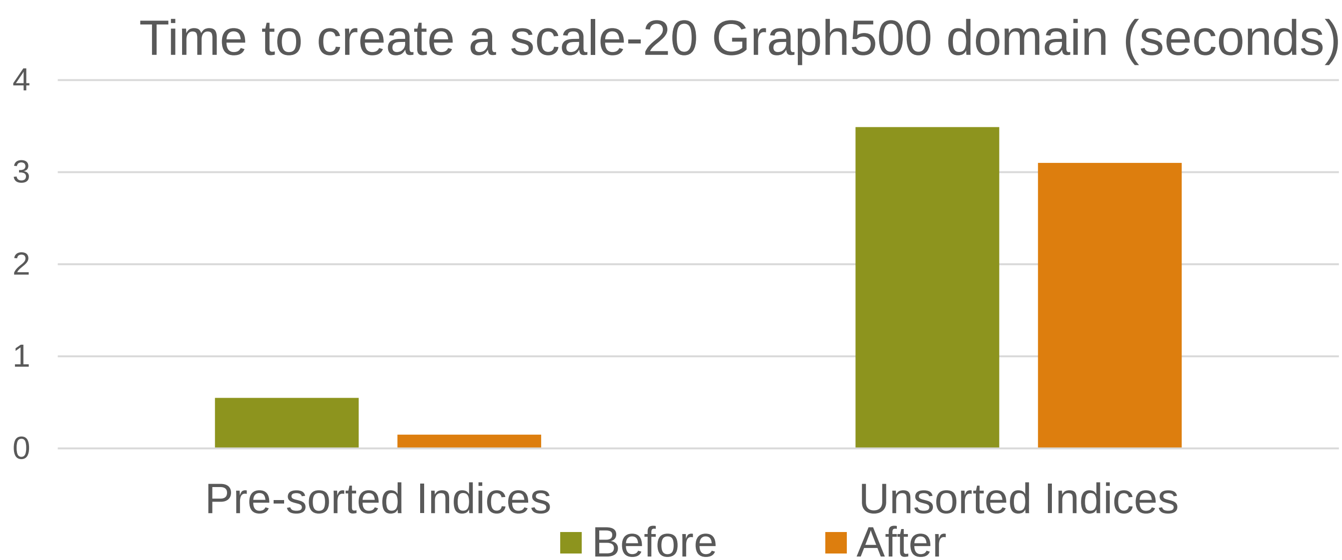
Background: Sparse domains support bulkAdd()

- Adds many new indices at once for efficiency

This Effort: Optimize bulkAdd() when domain is empty

- Contributed by Engin Kayraklioglu

Impact: Improved performance for some use cases:



Bulk Array Expansion

Bulk Array Expansion

Background: Array expansion interfaces had an issue

- Expanding arrays using array arguments resulted in promotion
- The promotion pushed new elements in an undefined order:

```
var A = [1,2,3];
A.push_back([4,5,6]); // Non-deterministic results, e.g. [1, 2, 3, 5, 6, 4]
```

This Effort: Added array overloads for extension methods

- New overloads:

```
array.push_back(arr: [])
array.push_front(arr: [])
array.insert(pos: idxType, arr: [])
```

- Optimized to grow array memory only once

Impact: Arrays can append, prepend, and insert in bulk

- The following code behaves as expected:

```
var A = [1,2,3];
A.push_back([4,5,6]); // A is now: [1,2,3,4,5,6]
```

Parallel Array Initialization

Parallel Array Initialization

Background: We initialize numerical arrays in parallel
- To get correct first-touch on NUMA systems

This Effort: Initialize POD (plain old data) arrays in parallel
- For example: records with numeric fields, tuples, etc.

Impact: No visible improvements in our performance graphs
- Significantly improved performance of a user's nbody simulation

Next steps: Extend parallel initialization to all arrays
- Including arrays-of-arrays
- Also want to permit users to override initialization strategy

Other Array, Domain, Domain Map Improvements

Other Array, Domain, Domain Map Changes

- **Added a version of reindex() that takes range arguments**
  - previously required a domain argument

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, and URIKA. 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.

CRAY CHAPEL

CRAY

THE SUPERCOMPUTER COMPANY