Array Improvements

Chapel version 1.20

September 19, 2019

chapel_info@cray.com

chapel-lang.org

@ChapelLanguage

Outline

- Array Slice Improvements
- Bulk-Transfer Improvements
- Scan Improvements
- Sparse Domain Improvements

Array Slice Improvements

Array Slices: Background

- Chapel supports array slices as a means of referring to a subset of an array:

```
… A[lo..hi] …
… A[myDomain] …
```

- However, slices have traditionally been expensive, esp. for distributed arrays…
  - communication to create a distributed domain representing the slicing indices
  - communication to create a distributed view representing the array slice
  - communication to move array elements around
    - e.g., in assignment contexts:

```
A[i..#sliceSize] = B[j..#sliceSize];
```


Array Slices: This Effort

- Reduced the overheads caused by slicing arrays
  - Changed slices to be governed by their slicing domain
  - Implemented a technique that reduces other slice-related overheads
  - Optimized data transfers between array slices using bulk-transfer
    - (see next section)
- Improved the expressiveness of array slicing:
  - Enabled sparse slicing of dense arrays
  - Reduced barriers to supporting slices of associative arrays

Governing Slices by their Domains

Slice Governance: Background

- Historically, Chapel has created a snapshot of the domain slicing an array:
  - a slice expression like this:

    … `A[Dom]` …
  - essentially becomes:

```
const tmpSliceDom = A.domain[Dom];    // intersect A's domain with Dom
… A[tmpSliceDom] …
```

- This could be very expensive:
  - if 'A' is distributed, a new distributed domain was created for 'tmpSliceDom'
  - if 'Dom' requires O(*n*) storage, a full copy of that storage was created
    - e.g., sparse or associative domains

Slice Governance: This Effort

Concept: rather than creating new domains, have slices refer to the original:

- a slice expression like this:

```
… A[Dom] …
```

- now essentially becomes:

```
ref tmp = newSliceView(A, Dom);   // represent A being sliced by Dom
… tmp …
```

Implications:

- Overheads associated with creating new domains are eliminated
- Changes slice behavior in some cases

Slice Governance: Semantic Impact

Consider the following example:

```
var D = {1..10};
ref Aslice = A[D]; // capture a reference to a slice
D = {1..20};       // change the slicing domain
writeln(Aslice);   // what should happen here?
```

Previously: since the slice used a copy of ‘D’, 10 elements were printed

Now: since it’s governed by ‘D’ itself, 20 elements are

Slice Governance: Semantic Impact

Consider the following example:

```
const DLoc = {lo..hi};
const DDist = newBlockDom({lo..hi});
var B = A[DLoc];
var C = A[DDist];
```

Previously:

- ‘B’ and ‘C’ would have been distributed arrays, each with its own domain

Now:

- ‘B’ is a new local array whose domain is ‘DLoc’
- ‘C’ is a new distributed array whose domain is ‘DDist’

Slice Governance: Semantic Impact

- The preceding examples represent changes to the language, yet powerful ones
  - Can create views of data that respond to dynamic changes:

```
ref SeattleIDs = ID[EmployeesInSeattle];
// …can modify `EmployeesInSeattle' as the program runs…
… SeattleIDs …   // this will always refer to the IDs of those employees
```

  - Can now distinguish between slice copies that should be localized vs. not:

```
const smallChunk = {lo..#4, lo..#4};
const bigChunk = newBlockDom({2..n-1, 2..n-1});
var Ablock = A[smallChunk];   // I just want a small, local 4x4 array
var AInner = A[bigChunk];     // I want to keep this array distributed
```

  - They are also more flexible and consistent with existing Chapel semantics

Slice Governance: Performance Impact

Creating a slice like this…

```
ref mySlice = myDistArray[myDistDom];
```

…results in the following communications (for 16 locales):

version	role	on	nonblocking on	fast on	put	get
1.19	originating locale	4	numLocales-1	0	0	0
	typical locale	0-4	0	1	1	128

Slice Governance: Performance Impact

Creating a slice like this…

```
ref mySlice = myDistArray[myDistDom];
```

…results in the following communications (for 16 locales):

version	role	on	nonblocking on	fast on	put	get
1.19	originating locale	4	numLocales-1	0	0	0
	typical locale	0-4	0	1	1	128
with slice governance	originating locale	**2**	**0**	0	0	0
	typical locale	**0-2**	0	**0**	**0**	**0**

- Reduced communication is due to no longer creating a copy of ‘myDistDom’
- Cost of operating on the slice expression is unchanged

Slice Governance: Sparse Slicing

- This change also trivially enabled sparse slicing of dense arrays, which we've…
  - …intended to support since day one
  - …advertised in talks for years
  - …never actually supported until now

```
const D = {1..10, 1..10};
const Diag: sparse subdomain(D)
          = [i in 1..10] (i,i);
var A: [D] int;

A[Diag] = 1;
writeln(A);
```

```
1 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 1
```

Lazy Slicing

Reducing slice overheads

Lazy Slicing: Background

Creating a slice like this…

```
ref mySlice = myDistArray[myDistDom];
```

…still results in the following communications, involving all locales:

version	role	on	nonblocking on	fast on	put	get
with slice governance	originating locale	2	0	0	0	0
	typical locale	0-2	0	0	0	0

These remaining communications are due to “proactive slicing”:

- When slicing a distributed array, we tell every locale about the slice
- But what if most of them don’t care…?
  - e.g., the slice doesn’t even involve their sub-arrays

Lazy Slicing: This Effort

Concept: only tell locales about slices on a need-to-know basis

- As an example:

```
ref mySlice = myDistArray[myDistDom];
// only the current locale needs to know about mySlice here…
forall a in mySlice do // other locales do here, but only if they own a piece
  a += 1.0;
```

Approach:

- when creating slices, only represent them locally
- forward / serialize them across on-clauses
  - note that this is cheap for distributed domain/array slices (send IDs only)

Lazy Slicing: Impact (creating slices)

Creating a slice like this…

```
ref mySlice = myDistArray[myDistDom];
```

…no longer requires any communication under lazy slicing:

version	role	on	nonblocking on	fast on	put	get
1.19	originating locale	4	numLocales-1	0	0	0
	typical locale	0-4	0	1	1	128
with slice governance	originating locale	2	0	0	0	0
	typical locale	0-2	0	0	0	0
with lazy slicing as well	originating locale	**0**	0	0	0	0
	typical locale	**0**	0	0	0	0

Lazy Slicing: Impact (using slices)

Moreover, using the slice…

```
forall a in mySlice do
  a += 1.0;
```

…does not change communications relative to standard practice:

version	role	on	nonblocking on	fast on	put	get
1.19	originating locale	0	15	0	0	0
	typical locale	0	0	1	0	0
with lazy slicing	originating locale	0	15	0	0	0
	typical locale	0	0	1	0	0

The payloads of the “nonblocking ons” do change modestly to represent the slice

Lazy Slicing: Status, Next Steps

Status: Lazy slicing is not enabled by default on master today

- While performance improves in many cases, others exchange on's for get's
  - e.g., zippered iteration involving slices
- Users can opt-in by compiling with `-schpl_serializeSlices=true`

Next Steps:

- Enable distributed arrays and domains to generally be forwarded
- Permit tuples to be forwarded / serialized if their elements can be
  - (these are used for zippered iteration)

Array Slices: Status, Next Steps

Status: Array slices have improved in Chapel 1.20

- in terms of semantics and performance
- particularly for slices that use domains (rather than ranges)

Next Steps:

- Turn on lazy slicing by default
- Reduce overheads for slicing using ranges
  - “easier” because there’s no pre-existing domain that could change
  - challenging because it requires lazily creating a new distributed domain

Bulk-Transfer Improvements

Bulk-Transfer Improvements: Background

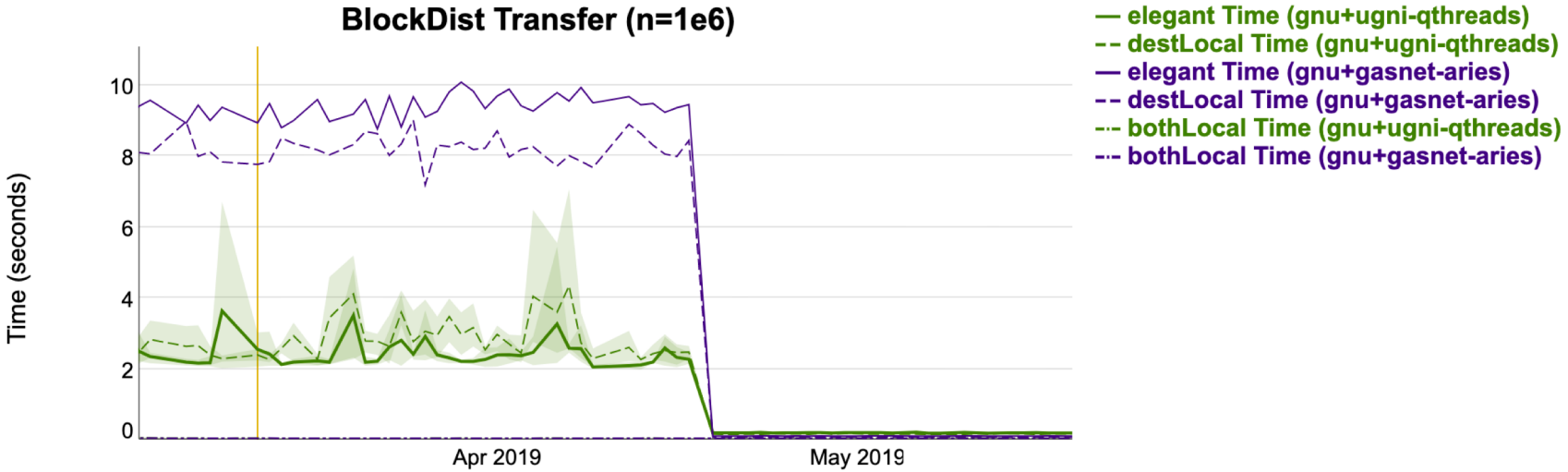
- Support for bulk-transfer of block-distributed arrays has existed since 2013
  - Contributed by external developers
- Not enabled by default due to lack of testing
- Could be enabled with config param
  - "-suseBulkTransferDist"

Bulk-Transfer Improvements: This Effort, Impact

This Effort: Enable bulk-transfer by default for BlockDist

- Simplified implementation and added tests to improve confidence
- Added config param 'disableBlockDistBulkTransfer' to disable optimization

Impact: Improved performance without requiring knowledge of special flags



Bulk-Transfer Improvements: Next Steps

- Find ways to reduce metadata communication in bulk-transfer implementation
  - e.g., caching remote array metadata
  - Optimize placement of tasks used to initiate bulk-transfers
  - on-statements and PUTs vs. GETs

- Improve performance of related array assignment features
  - e.g., reduce cost of creating an array slice

Scan Improvements

Scans: Background, This Effort

Background:

- Chapel has supported a scan (parallel prefix) operator since the outset
- Yet, historically, it has not received much attention
- 1.19 added an opt-in parallel implementation for block and local 1D arrays

This Effort:

- enabled the parallel implementation by default
- optimized the implementation for single-task runs (when the system is loaded)
- fixed a bug affecting scans of array slices (e.g., `+ scan A[lo..hi]`)
- fixed a bug affecting scan operators other than '+'

Scans: Impact, Status, Next Steps

Impact:

- The scan operator is now scalable for local and Block 1D arrays

Next Steps:

- Design and support other flavors of scan:
  - exclusive scans
  - segmented scans
  - multidimensional scans (via partial scans or wraparound)
- Parallelize scans for other expression types:
  - other array layouts and distributions
  - other shape-ful expressions (e.g., `+ scan (A:int)')

Sparse Domain Improvements

Sparse Domains: Background, This Effort

Background: Index addition to sparse domains is an expensive operation

- There is support for adding indices in bulk:

```
spsDom += arrayOfIndices;
```

- However this has some limitations:
  - Users have to manually create an array to store indices before adding
  - Distributed index addition is bottlenecked by local sort

This Effort: Added two new ways of adding indices

- Buffered index addition
- Local index addition (for distributed arrays)

Sparse Domains: Status, Next Steps

Status:

- Buffered index addition:

```
var idxBuf = spsDom.makeIndexBuffer(size=N);   // create buffer
for idx in someIndexGenerator() do
  idxBuf.add(idx); // buffer will be flushed as it gets full
idxBuf.commit();    // commit the remaining indices in the buffer
```

- Local index addition:

```
coforall l in Locales do on l do
  distSpsDom.bulkAdd(getLocalIndices(), addOn=here);
```

Next Steps:

- Support unbounded index buffers

FORWARD LOOKING STATEMENTS

This presentation may contain forward-looking statements that involve risks, uncertainties and assumptions. If the risks or uncertainties ever materialize or the assumptions prove incorrect, the results of Hewlett Packard Enterprise Company and its consolidated subsidiaries ("Hewlett Packard Enterprise") may differ materially from those expressed or implied by such forward-looking statements and assumptions. All statements other than statements of historical fact are statements that could be deemed forward-looking statements, including but not limited to any statements regarding the expected benefits and costs of the transaction contemplated by this presentation; the expected timing of the completion of the transaction; the ability of HPE, its subsidiaries and Cray to complete the transaction considering the various conditions to the transaction, some of which are outside the parties' control, including those conditions related to regulatory approvals; projections of revenue, margins, expenses, net earnings, net earnings per share, cash flows, or other financial items; any statements concerning the expected development, performance, market share or competitive performance relating to products or services; any statements regarding current or future macroeconomic trends or events and the impact of those trends and events on Hewlett Packard Enterprise and its financial performance; any statements of expectation or belief; and any statements of assumptions underlying any of the foregoing. Risks, uncertainties and assumptions include the possibility that expected benefits of the transaction described in this presentation may not materialize as expected; that the transaction may not be timely completed, if at all; that, prior to the completion of the transaction, Cray's business may not perform as expected due to transaction-related uncertainty or other factors; that the parties are unable to successfully implement integration strategies; the need to address the many challenges facing Hewlett Packard Enterprise's businesses; the competitive pressures faced by Hewlett Packard Enterprise's businesses; risks associated with executing Hewlett Packard Enterprise's strategy; the impact of macroeconomic and geopolitical trends and events; the development and transition of new products and services and the enhancement of existing products and services to meet customer needs and respond to emerging technological trends; and other risks that are described in our Fiscal Year 2018 Annual Report on Form 10-K, and that are otherwise described or updated from time to time in Hewlett Packard Enterprise's other filings with the Securities and Exchange Commission, including but not limited to our subsequent Quarterly Reports on Form 10-Q. Hewlett Packard Enterprise assumes no obligation and does not intend to update these forward-looking statements.

THANK YOU

QUESTIONS?


chapel_info@cray.com


@ChapelLanguage


chapel-lang.org

cray.com


@cray_inc


linkedin.com/company/cray-inc-/