LOCALITY-BASED OPTIMIZATIONS IN THE CHAPEL COMPILER

Engin Kayrakliouglu, Elliot Ronaghan, Michael P. Ferguson, Bradford L. Chamberlain
Hewlett Packard Enterprise
engin@hpe.com

The 34th International Workshop on Languages and Compilers for Parallel Computing
October 13, 2021

WHAT IS CHAPEL?

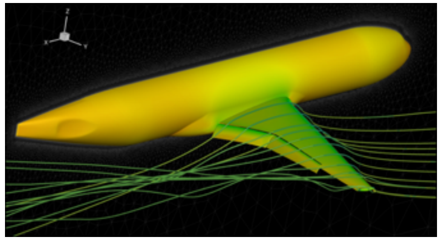
Chapel: A modern parallel programming language

- portable & scalable
- open-source & collaborative

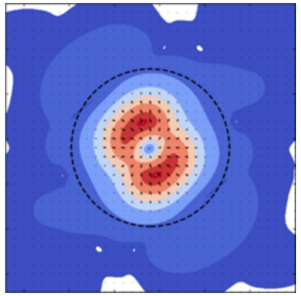
Goals:

- Support general parallel programming
- Make parallel programming at scale far more productive

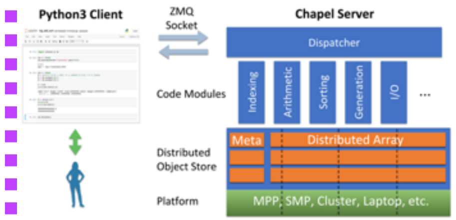
NOTABLE CURRENT APPLICATIONS OF CHAPEL



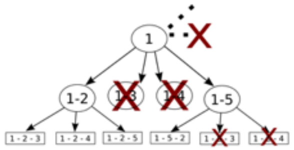
CHAMPS: 3D Unstructured CFD
Éric Laurendeau, Simon Bourgault-Côté, Matthieu Parenteau, et al.
École Polytechnique Montréal
~48k lines of Chapel



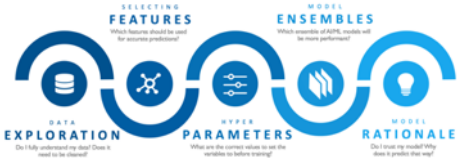
ChplUltra: Simulating Ultralight Dark Matter
Nikhil Padmanabhan, J. Luna Zagorac, Richard Easther, *et al.*
Yale University / University of Auckland



Arkouda: NumPy at Massive Scale
Mike Merrill, Bill Reus, et al.
US DOD
~16k lines of Chapel



ChOp: Chapel-based Optimization
Tiago Carneiro, Nouredine Melab, *et al.*
INRIA Lille, France



CrayAI: Distributed Machine Learning
Hewlett Packard Enterprise

?

Your Project Here?

CHAPEL

Distributed Memory Programming in Chapel

A “domain” is an index set in Chapel
This is from 1 to n, inclusive

Chapel arrays are declared over domains
This is an integer array over ‘myDomain’

```
var myDomain = {1..n};
var myArray: [myDomain] int;
```

A ‘for’ loop executes sequentially on the locale that started it
This iterates over array elements

```
for element in myArray do
  element = 1;
```

Distributed Memory Programming in Chapel

```
use BlockDist;

var myDomain = {1..n} dmapped Block({1..n});
var myArray: [myDomain] int;


for element in myArray do
  element = 1;
```

The 'dmapped' clause can be used to make a domain distributed
Now, 'myDomain' is block-distributed

No change is necessary for array declaration
Now, 'myArray' is block-distributed

'for' loops are always sequential and executes on the initiating locale
This loop behaves the same; accesses to 'element' can be remote

CHAPEL

Distributed Memory Programming in Chapel

```
use BlockDist;

var myDomain = {1..n} dmapped Block({1..n});
var myArray: [myDomain] int;
```

A ‘forall’ loop can be distributed and parallel depending on the iterator it executes over

Now, this loop is parallel and distributed identically to ‘myArray’

```
forall element in myArray do
  element = 1;
```

This implies that accesses to ‘element’ are always local

CHAPEL
For More Information

- **Michelle Strout's invited talk:** "Separating Parallel Performance Concerns Using Chapel"
  - 10:40 EDT, today

- **Web page:** chapel-lang.org
- **Development** on GitHub: github.com/chapel-lang/chapel
- **Mailing list** on Discourse: chapel.discourse.group
- **Public chat** on Gitter: gitter.im/chapel-lang/chapel
- **Questions** on StackOverflow: stackoverflow.com/questions/tagged/chapel
- **Talks** on YouTube: youtube.com/c/ChapelParallelProgrammingLanguage
- **News** on Twitter: @ChapelLanguage

AUTOMATIC LOCAL ACCESS

AUTOMATIC LOCAL ACCESS

Before This Optimization

- Three common idioms for implementing STREAM Triad in Chapel

```
var D = newBlockDom(1..n);
var A, B, C: [D] int;
```

Idiom 1

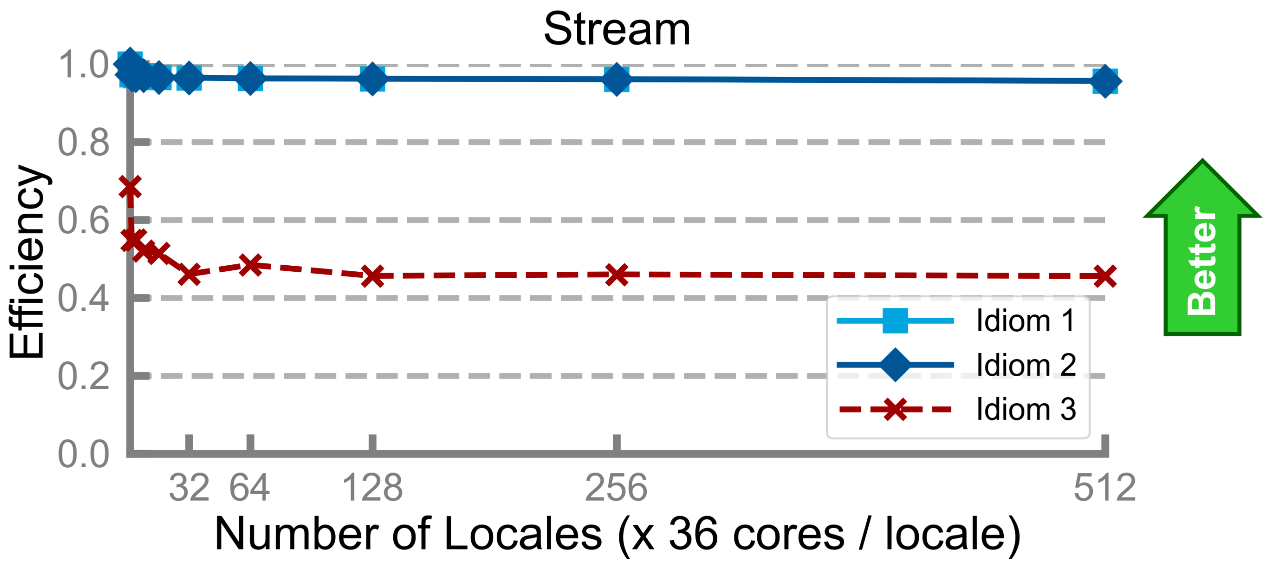
```
A = B + alpha * C;
```

Idiom 2

```
forall (a,b,c) in zip(A,B,C) do
  a = b + alpha * c;
```

Idiom 3

```
forall i in D do
  A[i] = B[i] + alpha * C[i];
```



Numbers are collected from a Cray XC with Aries interconnect and ugni communication layer

AUTOMATIC LOCAL ACCESS

Locality Check Overhead

```
var D = newBlockDom(1..n);
var A, B, C: [D] int;
```

Idiom 3

```
forall i in D do
  A[i] = B[i] + alpha * C[i];
```

```
proc array.access(idx: int) {
  if isLocalIndex(idx) then
    localAccess(idx);
  else
    nonLocalAccess(idx);
}
```

A per-access check is the source of overhead

- This check can be avoided for all 3 accesses
- Because;
  - The ‘forall’ distribution is aligned with A’s
    … because the loop is over A.domain
  - The loop index is the same as the access index
  - B and C’s distribution is aligned with A’s
    … because they all share the same domain

AUTOMATIC LOCAL ACCESS

Examples

```
var D = newBlockDom({1..N});
var A: [D] int, B: [D] int;
```

```
forall i in D do
  A[i] = i;
```

optimized!

the array is indexed using the loop index

the array has the same domain as the loop

```
forall i in D do
  A[i] = B[i];
```

optimized!

the arrays are indexed using the loop index

the arrays have the same domain as the loop

```
forall i in A.domain do
  A[i] = B[i];
```

optimized!

the arrays are indexed using the loop index

loop is run over a domain query

the arrays have the same domain as the loop

AUTOMATIC LOCAL ACCESS
Dynamic Checks and Loop Versioning

- If the compiler cannot determine the domain of an array:
  - Equality of domains will be checked at execution time
  - Depending on that, an optimized or unoptimized version of the loop will be run

```
var A = newBlockArr({1..N}, int);
var B = newBlockArr({1..N}, int);    // currently we can't infer 'B' has the same domain as 'A'
forall i in A.domain do
  A[i] = calculate(B[i]);    // B[i] is local if A.domain == B.domain
                             // that can only be confirmed at execution time
```

- Terminology
  - ‘A[i]’ is a static candidate
  - ‘B[i]’ is a dynamic candidate

AUTOMATIC LOCAL ACCESS

Dynamic Checks and Loop Versioning

```
var A = newBlockArr({1..N}, int);
var B = newBlockArr({1..N}, int);
forall i in A.domain do
  A[i] = calculate(B[i]);
```

Static checks are created for both arrays

Dynamic check is created only for B

```
var A = newBlockArr({1..N}, int);
var B = newBlockArr({1..N}, int);
param staticCheckA = canUseLocalAccess(A, A.domain);
param staticCheckB = canUseLocalAccess(B, A.domain);
if staticCheckA || staticCheckB {
  const dynamicCheckB = canUseLocalAccessDyn(B, A.domain);
  if dynamicCheckB then
    forall i in A.domain do
      A.localAccess[i] = calculate(B.localAccess[i]);
  else
    forall i in A.domain do
      A.localAccess[i] = calculate(B[i]);
} else {
  forall i in A.domain do
    A[i] = calculate(B[i]);
}
```

AUTOMATIC LOCAL ACCESS

Dynamic Checks and Loop Versioning

```
var A = newBlockArr({1..N}, int);
var B = newBlockArr({1..N}, int);
forall i in A.domain do
  A[i] = calculate(B[i]);
```

Will be executed if

- A passes static checks
- B passes static and dynamic checks

Will be executed if

- A passes static checks
- B fails static or dynamic checks

Will be executed if

- Neither array passes static checks

```
var A = newBlockArr({1..N}, int);
var B = newBlockArr({1..N}, int);
param staticCheckA = canUseLocalAccess(A, A.domain);
param staticCheckB = canUseLocalAccess(B, A.domain);
if staticCheckA || staticCheckB {
  const dynamicCheckB = canUseLocalAccessDyn(B, A.domain);
  if dynamicCheckB then
    forall i in A.domain do
      A.localAccess[i] = calculate(B.localAccess[i]);
  else
    forall i in A.domain do
      A.localAccess[i] = calculate(B[i]);
} else {
  forall i in A.domain do
    A[i] = calculate(B[i]);
}
```

AUTOMATIC LOCAL ACCESS

After This Optimization: STREAM Triad

Reaches 96% efficiency at-scale

All idioms perform similarly

```
var D = newBlockDom(1..n);
var A, B, C: [D] int;
```

Idiom 1

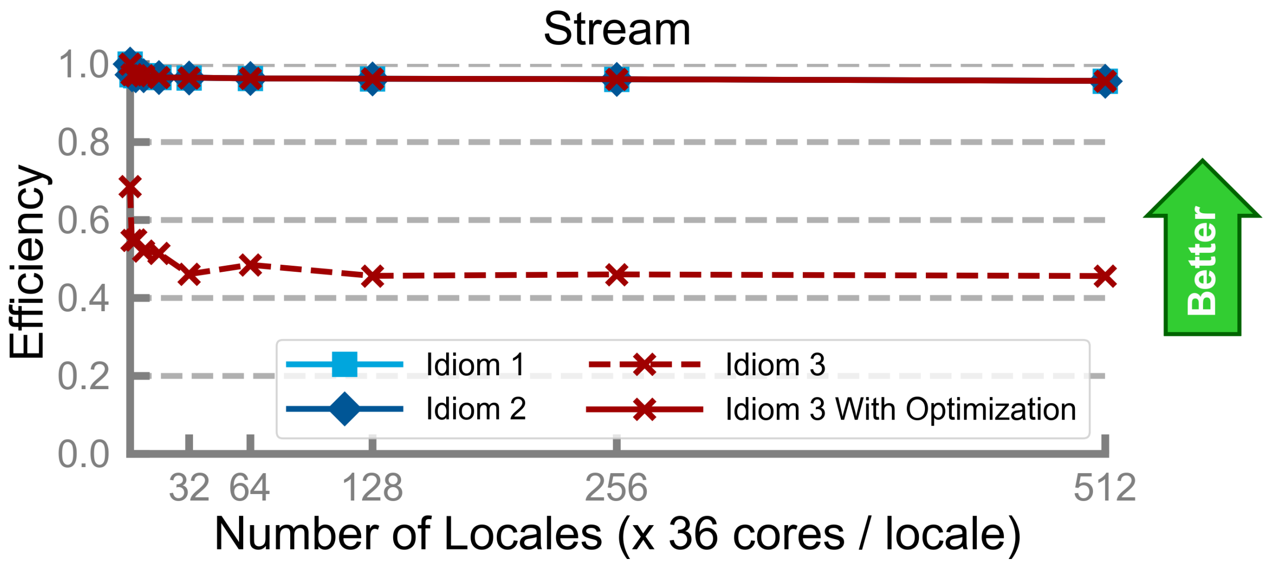
```
A = B + alpha * C;
```

Idiom 2

```
forall (a,b,c) in zip(A,B,C) do
  a = b + alpha * c;
```

Idiom 3

```
forall i in A.domain do
  A[i] = B[i] + alpha * C[i];
```



Numbers are collected from a Cray XC with Aries interconnect and ugni communication layer

AUTOMATIC LOCAL ACCESS

After This Optimization: NAS Parallel Benchmarks - FT

- Explicit 'localAccess' calls are no longer needed in NPB-FT

Kernel with 'localAccess' calls

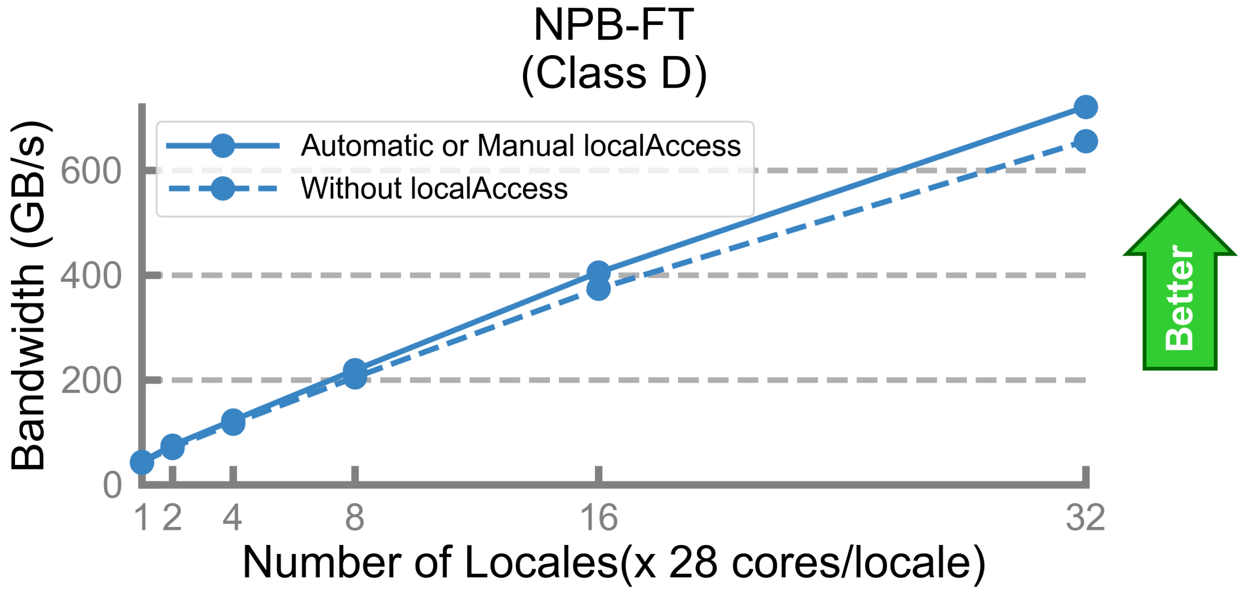
```
forall ijk in DomT {
  const elt = V.localAccess[ijk] *
              T.localAccess[ijk];

  V.localAccess[ijk] = elt;
  Wt.localAccess[ijk] = elt;
}
```

Kernel without 'localAccess' calls

```
forall ijk in DomT {
  const elt = V[ijk] *
              T[ijk];

  V[ijk] = elt;
  Wt[ijk] = elt;
}
```

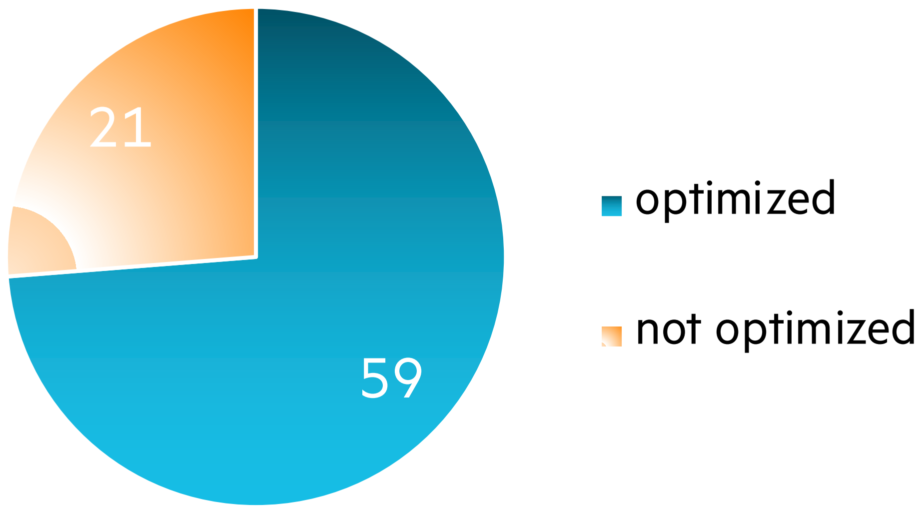


Numbers are collected from a Cray XC with Aries interconnect and ugni communication layer

AUTOMATIC LOCAL ACCESS

After This Optimization: chplUltra

- chplUltra[1] is an Ultralight Dark Matter simulator written in Chapel
- We removed all explicit calls to *localAccess*
  - 80 places in total
    - 59 are optimized automatically
    - 21 were not optimized
  - The patterns where the optimization does not fire
    - 10 locality hard to detect due to complex alignments
    - 7 array access indices are not loop indices
    - 4 is not inside forall loops



[1] Nikhil Padmanabhan et al. "Simulating Ultralight Dark Matter in Chapel". In: 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). May 2020,

AUTOMATIC AGGREGATION

AUTOMATIC AGGREGATION
Before This Optimization

- The *indexgather* benchmark from the bale[2] study tests random access performance

```
var D = newBlockDom(1..n);
var Src, Dst, Inds: [D] int;
```

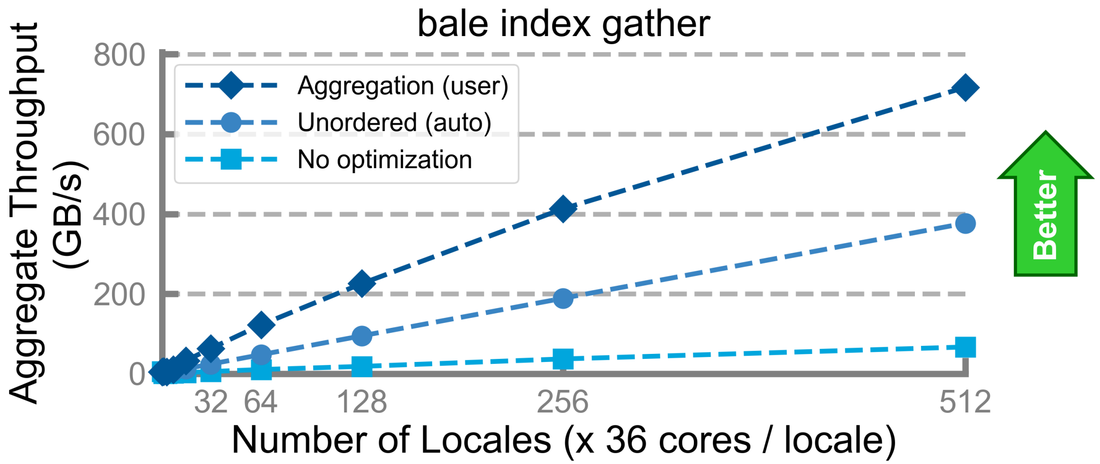
Straightforward *indexgather*

```
forall (d, i) in zip(Dst, Inds) do
  d = Src[i];
```

Communication can be done in any order

The Chapel compiler already had "unordered forall" optimization

Numbers are collected from a Cray XC with Aries interconnect and ugni communication layer



A Manual Approach for Data Aggregation

```
forall (d, i) in zip(Dst, Inds) with (var agg = new SrcAggregator(int)) do
  agg.copy(d, Src[i]);
```

[2] https://github.com/jdevinney/bale

AUTOMATIC AGGREGATION

Connecting the Dots for Automatic Aggregation

Straightforward *indexgather*

```
forall (d, i) in zip(Dst, Inds) do
  d = Src[i];
```

What does it take to make this transition automatically?

A Manual Approach for Data Aggregation

```
forall (d, i) in zip(Dst, Inds) with (var agg = new SrcAggregator(int)) do
  agg.copy(d, Src[i]);
```

Q: Is it safe to complete communication in any order?

A: unordered forall optimization does that analysis

Q: Is exactly one side of the operation local?

A: automatic local access optimization does that analysis

Q: Do you have means to aggregate the data?

A: aggregator objects can do that

AUTOMATIC AGGREGATION

Examples

```
var D = newBlockDom({1..N});
var A: [D] int, B: [D] int;
```

```
forall i in D do
  A[i] = B[computeIndex(i)];
```

aggregated!

destination of copy is local

source of copy is likely not local

```
forall (a, i) in zip(A, 0..) do
  B[computeIndex(i)] = a;
```

source is yielded by the first iterand, must be local

aggregated!

destination is likely not local

```
forall (i,a) in zip(A.domain, A) do
  A[computeIndex(i)] = a;
```

source is yielded by the second iterand

but it is aligned with the first one

aggregated!

destination is likely not local

AUTOMATIC AGGREGATION

After This Optimization: *indexgather*

- The *indexgather* benchmark from the bale study tests random access performance

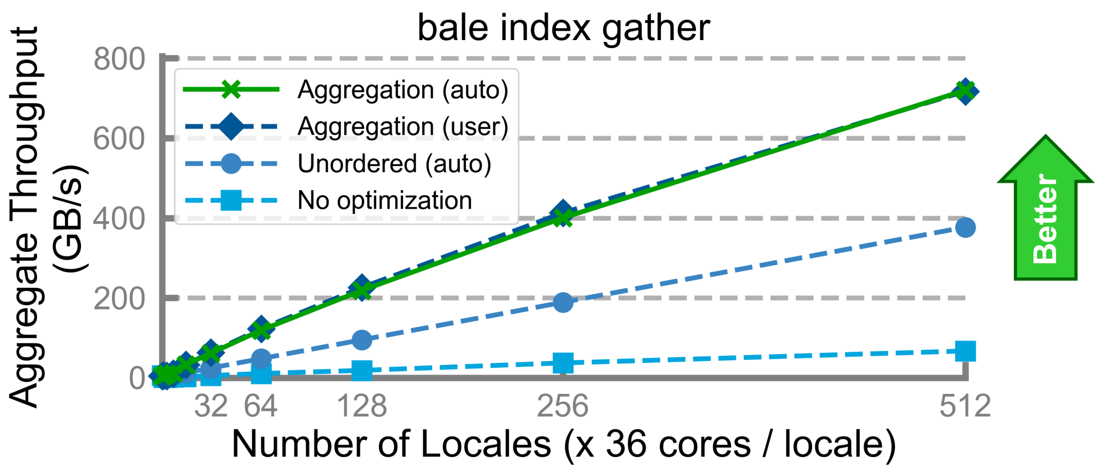
```
var D = newBlockDom(1..n);
var Src, Dst, Inds: [D] int;
```

Straightforward *indexgather*

```
forall (d, i) in zip(Dst, Inds) do
  d = Src[i];
```

Straightforward version performs identical to the manually-aggregated version

Numbers are collected from a Cray XC with Aries interconnect and ugni communication layer



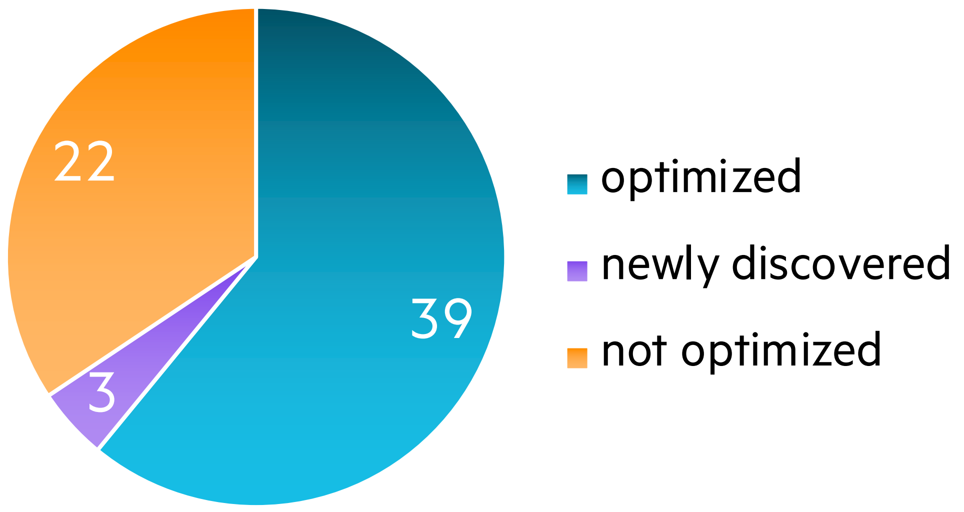
A Manual Approach for Data Aggregation

```
forall (d, i) in zip(Dst, Inds) with (var agg = new SrcAggregator(int)) do
  agg.copy(d, Src[i]);
```

AUTOMATIC AGGREGATION

After This Optimization: Arkouda

- Arkouda[3] is a data analytics tool that has a Python client and a server implemented in Chapel
- We removed all the manual aggregation from the source
  - 61 places in total
    - 39 are optimized automatically
    - 22 are not optimized
  - 3 cases that were not using aggregators are now optimized
  - The patterns where the aggregation does not fire:
    - 9: aggregation is not based on ‘forall’ loops
    - 6: compiler cannot prove that unordered operation is safe
    - 3: locality is hard to detect
    - 2: aggregated copy is not in the last statement of the body
    - 1: one side of the assignment is defined within the loop body
    - 1: needs further investigation



[3] https://github.com/Bears-R-Us/arkouda

LIMITATIONS & NEXT STEPS

Automatic Local Access

- Can we do the same optimizations when the index is a complex expression?
  - Today: Access must be at same index as the loop index

Automatic Aggregation

- Support arbitrary operations
  - Today: Limited to copy operations (i.e., ‘=’ operator)
- Improve worst-case performance
  - Today: If everything is local, aggregation adds overhead and can reduce performance by half
- Investigate multi-hop aggregation
  - Today: Per-locale buffers can have a large memory footprint significantly at-scale
- Expose aggregation as a user-facing language feature
  - Today: The aggregator objects are not in the documented part of the standard library

SUMMARY

- We have discussed two locality-based optimizations in the Chapel compiler

Automatic Local Access

- Avoids locality checks while accessing distributed arrays using indices
- Added to Chapel in version 1.23 (1.25 was released few weeks ago)
- On-by-default

Automatic Aggregation

- Aggregates some remote copy operations
- Added to Chapel in version 1.24 (1.25 was released few weeks ago)
- Off-by-default, enable with --auto-aggregation

THANK YOU!

engin@hpe.com