Hewlett Packard Enterprise

LOCALITY-BASED OPTIMIZATIONS IN THE CHAPEL COMPILER

Engin Kayraklioglu, Elliot Ronaghan

CHIUW 2021 – June 4, 2021

LOCALITY-BASED OPTIMIZATIONS IN THE CHAPEL COMPILER

- Chapel's high-level features enable the compiler to argue about locality
- Versions 1.23 and 1.24 added two new optimizations

Added in 1.23: Automatic Local Access

- Faster distributed array access in forall bodies

Added in 1.24: Automatic Copy Aggregation

- (Some) Remote array accesses in forall bodies use aggregated communication

"Perilous to us all are the devices of an art deeper than we possess ourselves."

Gandalf*

*The Lord of the Rings: The Two Towers, J.R.R. Tolkien

AUTOMATIC LOCAL ACCESS

AUTOMATIC LOCAL ACCESS

Introduction

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

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

distributed array access overheads

With optimization

Access to A with index i is strength-reduced

- **Before:** if A[i] is a local is checked during execution
- **After:** that check is avoided

Potential workarounds without the optimization

```
forall (a, i) in zip(A, A.domain) do
  a = calculate(i);
```

'zip' may not be intuitive

```
forall i in A.domain do
  A.localAccess(i) = calculate(i);
```

clunky way to access an array

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

- 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);    // 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
```

- The compiler will clone loops if there are one or more dynamic candidates
  - This might increase compilation time
  - We have not observed noticeable compilation slowdowns in real use cases

AUTOMATIC LOCAL ACCESS

Pitfalls

There's an intervening `on` statement

```
forall i in A.domain do
  on Locales[X] do
    A[i] = calculate(i);
```

Array index is not the loop index

```
forall i in A.domain {
  const k = i;
  A[k] = calculate(i);
}
```

Zippered forall that uses packed index

```
forall idx in zip(D, E) { }
forall (x, y) in zip(D, E) { }
```

Array is a forall intent

```
forall i in D with (ref A) do
  A[i] = calculate(i);
```

AUTOMATIC LOCAL ACCESS
Performance Impact

- Global STREAM with array indexing:

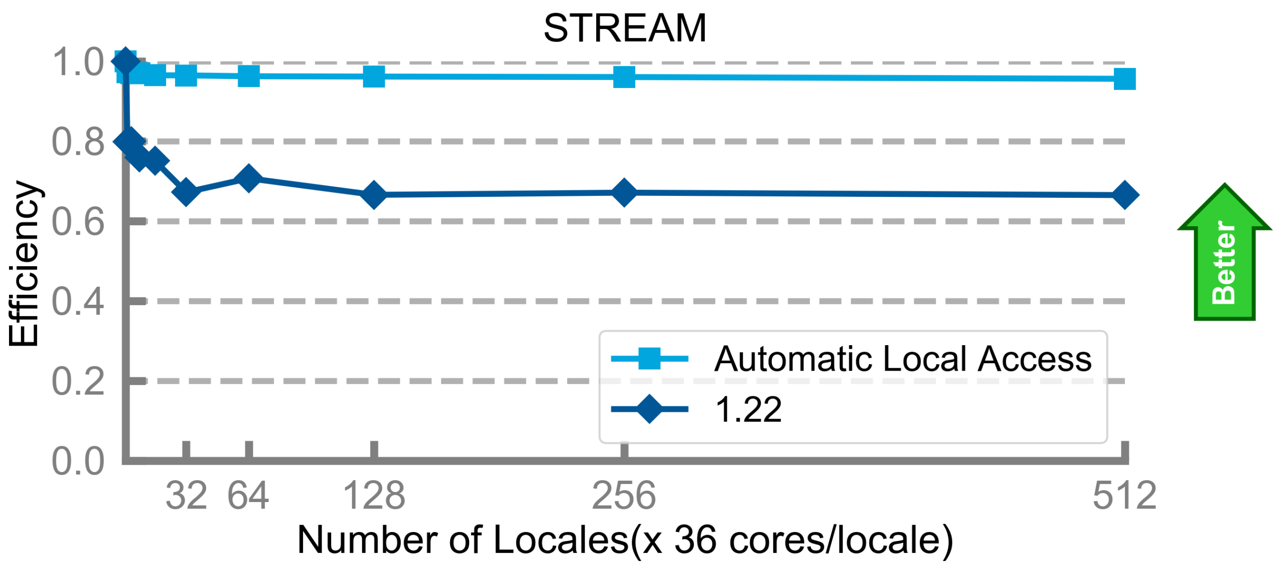
```
forall i in ProblemSpace do
  A[i] = B[i]+ alpha * C[i];
```

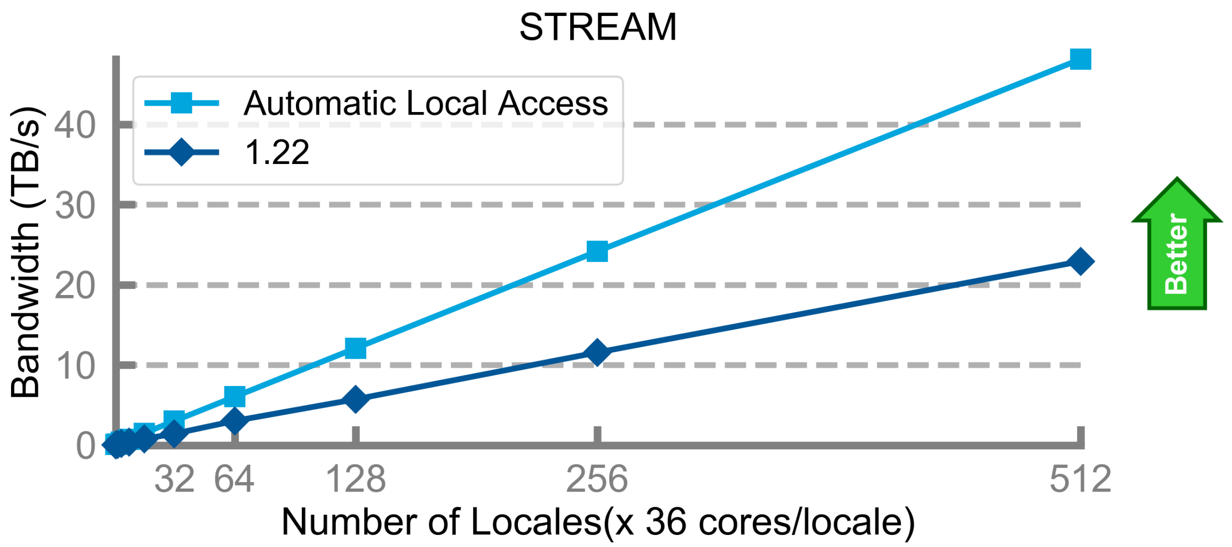
now essentially performs like other idioms:

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

or:

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





AUTOMATIC COPY AGGREGATION

AUTOMATIC COPY AGGREGATION
Introduction

```
var D = newBlockDom({1..N});
var A: [D] int;
var B: [D] int;
forall i in D do
  A[i] = B[calculate(i)];
```

fine-grained remote access causes overheads

With optimization

Irregular access to B is copy-aggregated

- **Before:** each remote access is a message
- **After:** data is buffered locally, moved in bulk

Potential application-specific solution

```
forall i in D with (var agg = newSrcAggregator(A.eltType){
  agg.copy(A[i], B[computeIndex(i)]);
}
```

but how do you write an efficient “aggregator”?

less expressive code

AUTOMATIC COPY 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 COPY AGGREGATION

Pitfalls

Arbitrary operations are not aggregated

```
forall i in A.domain do
  A[i] = B[calculate(i)] + 3;
```

Only the last statement in the body is analyzed

```
forall i in A.domain {
  A[i] = B[calculate(i)];   ✗
  C[i] = D[calculate(i)];   ✓
}
```

Fully-local aggregation can hurt performance

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

either don't use --auto-aggregation,

or "trick" the locality analysis:

```
forall i in A.domain {
  const k = i;
  A[k] = B[getALocalIndex(i)];
}
```

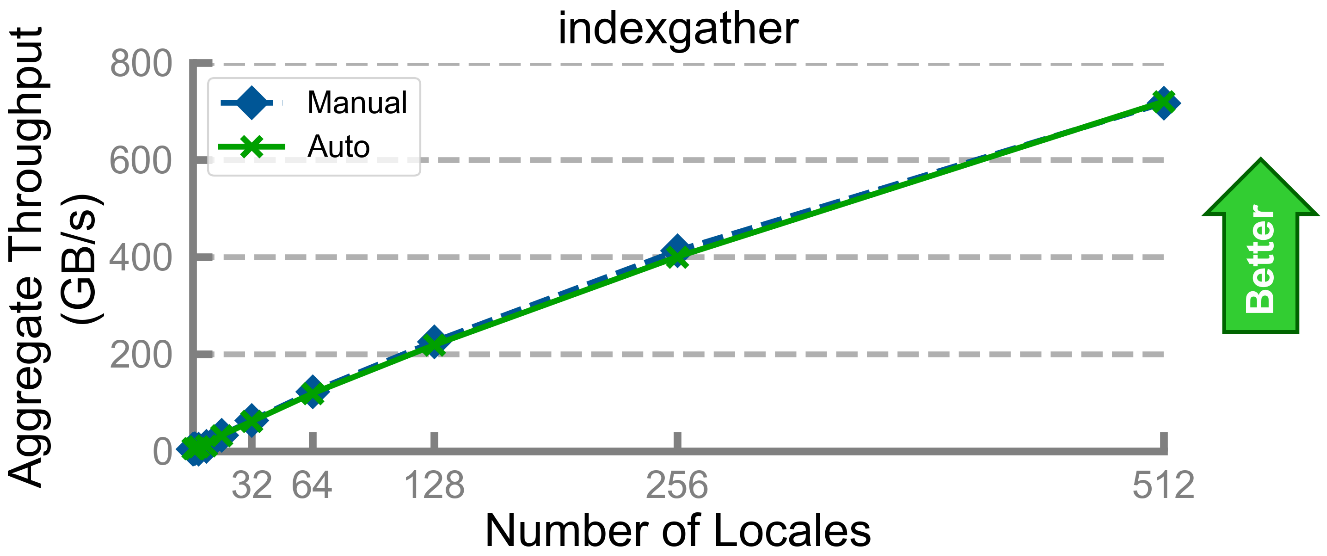
AUTOMATIC COPY AGGREGATION

Impact

- Bale indexgather benefits greatly from aggregation
- '--auto-aggregation' reaches the same performance as the manual version
  - No user effort is needed

– Benchmark kernel:

```
forall i in D2 do
  tmp[i] = A[rindex[i]];
```



– Benchmark kernel with manual aggregation:

```
forall i in D with (var agg = new SrcAggregator(int)) do
  agg.copy(tmp[i], A[rindex[i]]);
```

SUMMARY

Automatic Local Access

- Improves indexed access to distributed arrays in forall bodies
- On-by-default
  - Some control flags: --no-auto-local-access, --report-auto-local-access, --no-dynamic-auto-local-access

Automatic Copy Aggregation

- Aggregates fine-grained copies in the last statements of forall bodies
- Off-by-default
  - Some control flags: --auto-aggregation, --report-auto-aggregation

Future work

- Automatic atomic operation aggregation
- Improve worst-case performance of automatic copy aggregation
- Investigate extending support to arbitrary statements in forall bodies

THANK YOU

engin@hpe.com