Runtime Optimizations for Irregular Applications in Chapel

Thomas B. Rolinger (UMD/LPS), Christopher D. Krieger (LPS), Alan Sussman (UMD)

Contact: tbrolin@cs.umd.edu

CHIUW 2021

COMPUTER SCIENCE
UNIVERSITY OF MARYLAND

LPS
The Laboratory for Physical Sciences

Outline

- Motivation and background
  - Irregular applications
  - Inspector-executor technique
- High-level design of inspector-executor
- Performance evaluation
  - NAS-CG, moldyn, PageRank
- Future work

Outline

- Motivation and background
  - Irregular applications
  - Inspector-executor technique
- High-level design of inspector-executor
- Performance evaluation
  - NAS-CG, moldyn, PageRank
- Future work

1.) Motivation and Background

- **Memory wall:** processor speeds outpace rate at which data can be fetched from memory
  - leads to data starvation of compute resources
- Even worse for **irregular applications**
  - sparse, unstructured memory access patterns found in graph analytics
  - lack of spatial/temporal locality leads to fine-grained, remote communication
  - memory access patterns **not known at compile time**
    - requires **runtime-based** optimizations

1.) Motivation and Background (cont.)

- Inspector-executor technique
  - **inspector** → analyze a kernel of interest (memory access pattern, loop iteration dependencies, etc.)
  - **executor** → generate an optimized version of the kernel that utilizes the inspector’s analysis (loop reordering, data reordering, etc.)

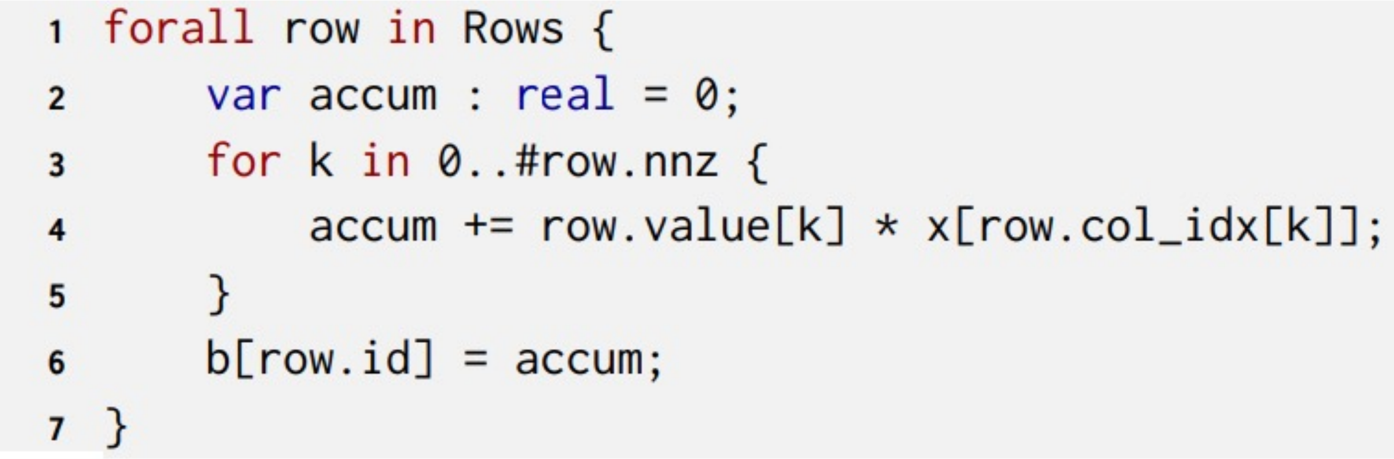
1.) Motivation and Background (cont.)

- Inspector-executor technique
  - **inspector** → analyze a kernel of interest (memory access pattern, loop iteration dependencies, etc.)
  - **executor** → generate an optimized version of the kernel that utilizes the inspector's analysis (loop reordering, data reordering, etc.)
- To achieve performance gains, the overhead of the inspector needs to be **amortized over multiple executions** of the kernel
  - kernel does not change between iterations
  - examples: conjugate gradient, molecular dynamics simulations, PageRank
- The inspector and executor can be **generated by the compiler**
  - in this preliminary work, we **hand-code** the inspector and executor to demonstrate the potential of the optimization

Outline

- Motivation and background
  - Irregular applications
  - Inspector-executor technique
- High-level design of inspector-executor
- Performance evaluation
  - NAS-CG, moldyn, PageRank
- Future work

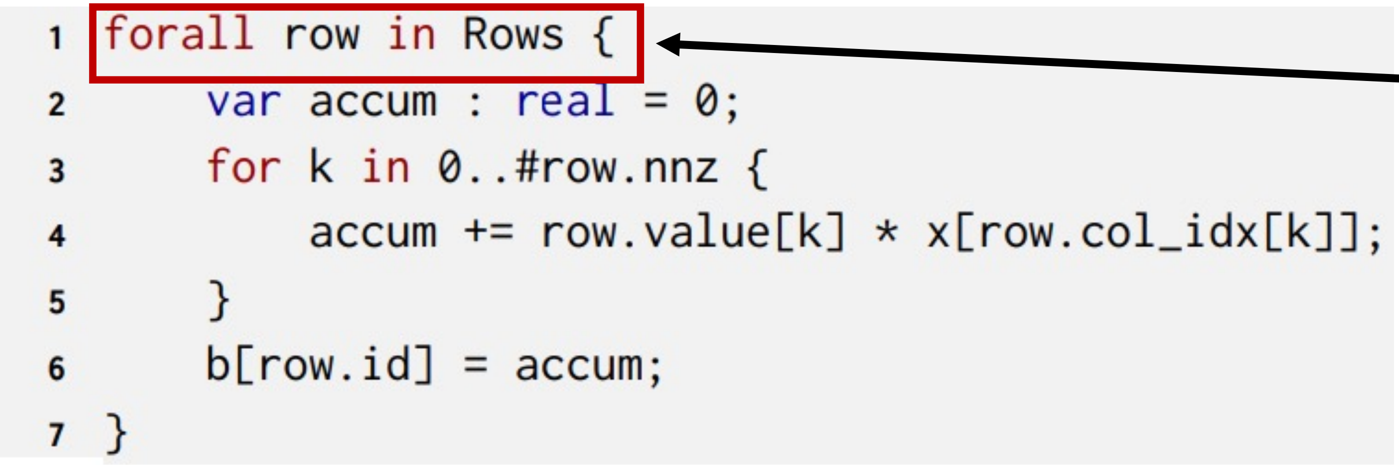
2.) High-level Design of Inspector-executor



```
forall row in Rows {
    var accum : real = 0;
    for k in 0..#row.nnz {
        accum += row.value[k] * x[row.col_idx[k]];
    }
    b[row.id] = accum;
}
```

Sparse Matrix-Vector Multiply (SpMV) kernel

2.) High-level Design of Inspector-executor



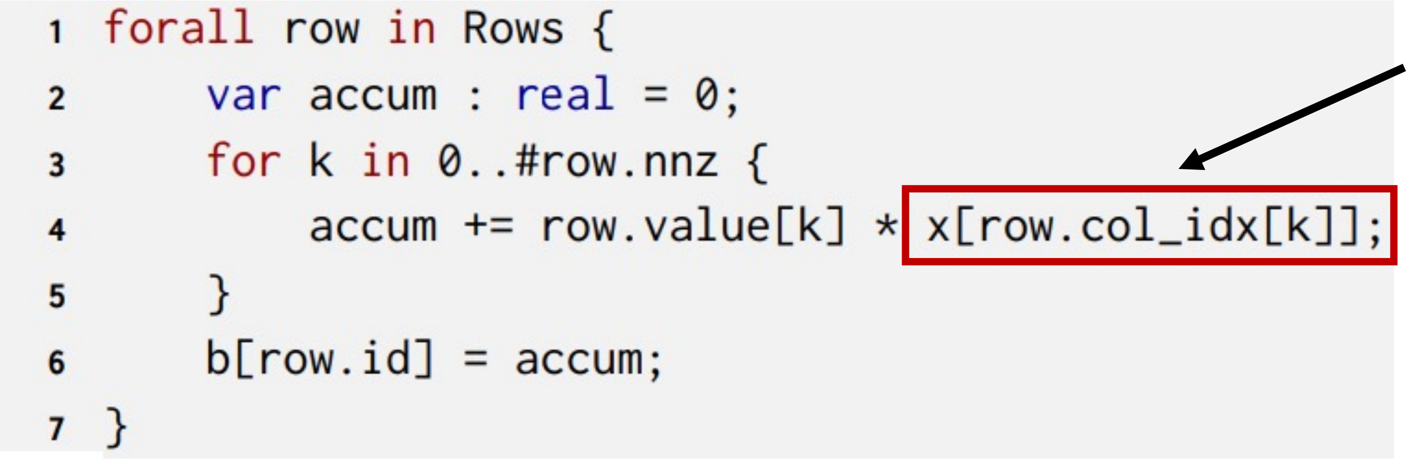
```
forall row in Rows {
    var accum : real = 0;
    for k in 0..#row.nnz {
        accum += row.value[k] * x[row.col_idx[k]];
    }
    b[row.id] = accum;
}
```

Sparse Matrix-Vector Multiply (SpMV) kernel

Rows is a block distributed array

A given **row** is operated on the locale where it is stored

2.) High-level Design of Inspector-executor



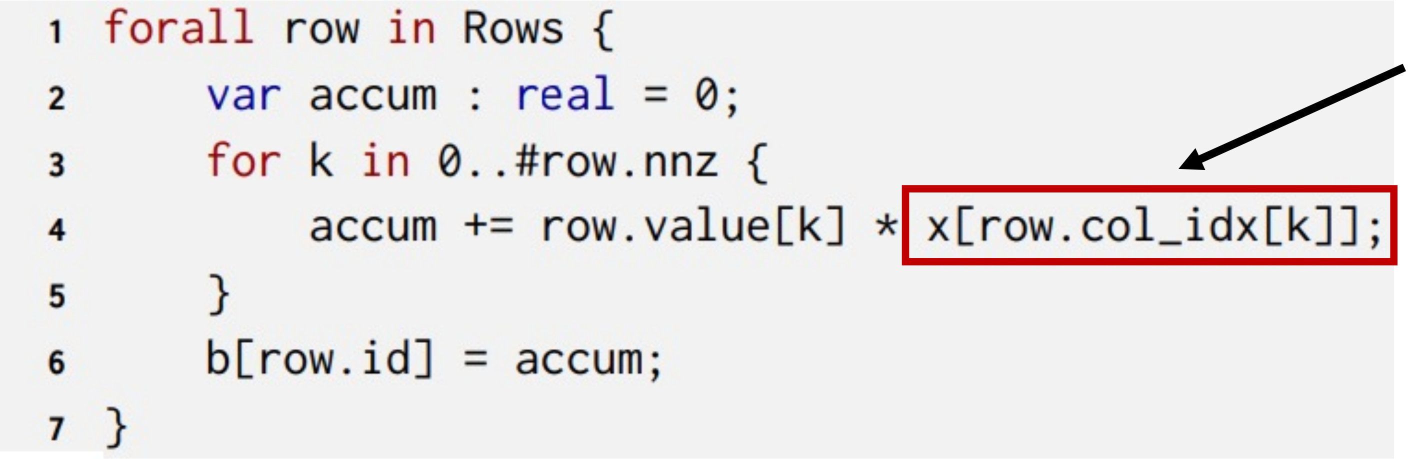
```
forall row in Rows {
    var accum : real = 0;
    for k in 0..#row.nnz {
        accum += row.value[k] * x[row.col_idx[k]];
    }
    b[row.id] = accum;
}
```

Sparse Matrix-Vector Multiply (SpMV) kernel

x is a block distributed array
→ fine-grained remote accesses

indirect access pattern **not known** at compile time

2.) High-level Design of Inspector-executor



```
forall row in Rows {
    var accum : real = 0;
    for k in 0..#row.nnz {
        accum += row.value[k] * x[row.col_idx[k]];
    }
    b[row.id] = accum;
}
```

Sparse Matrix-Vector Multiply (SpMV) kernel

x is a block distributed array
→ fine-grained remote accesses

indirect access pattern **not known** at compile time

GOAL: Eliminate all remote accesses to **x** during the kernel

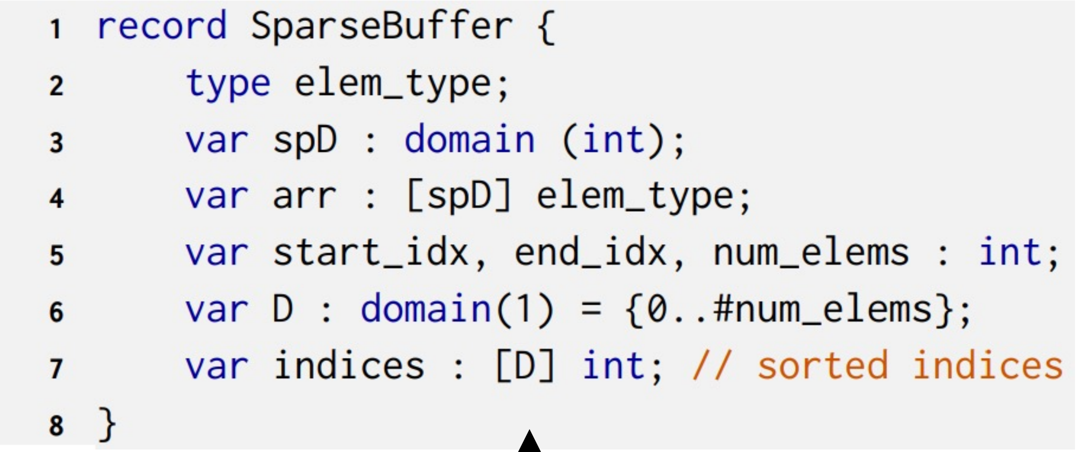
APPROACH:

- **inspect** which **col_idx[k]** result in remote accesses to **x** for a given locale
- **replicate** the remote elements on that locale and access those copies instead

→ Construct a mapping from **col_idx[k]** to **x[col_idx[k]]** for remote accesses

2.) High-level Design of Inspector-executor (cont.)

- Replicating remote elements: **associative arrays**
    - **Keys:** col_idx[k] values (i.e., indices)
    - **Values:** x[col_idx[k]] elements (i.e., remote values)



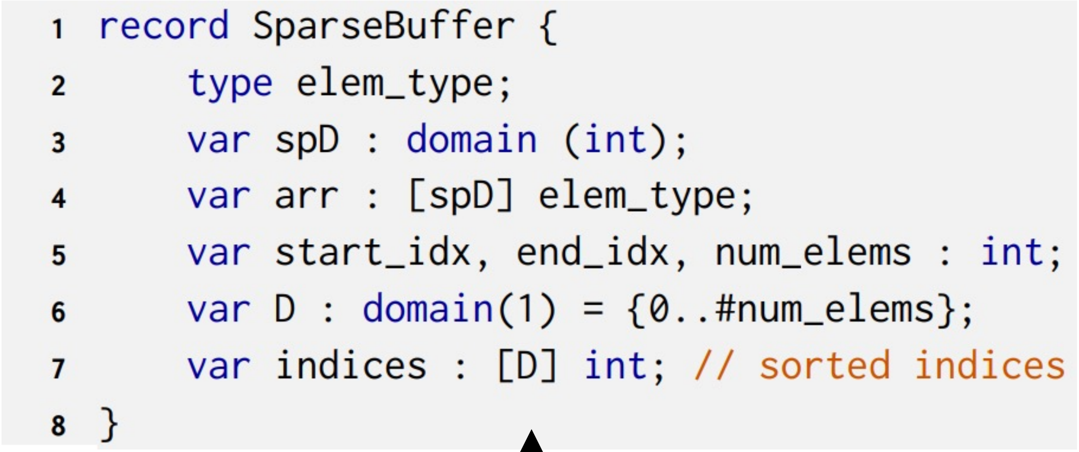
```
record SparseBuffer {
    type elem_type;
    var spD : domain (int);
    var arr : [spD] elem_type;
    var start_idx, end_idx, num_elems : int;
    var D : domain(1) = {0..#num_elems};
    var indices : [D] int; // sorted indices
}
```

Each locale stores a **SparseBuffer** record to keep track of the remote elements it will need

spD is the associative domain, **arr** is the array declared over the associative domain

2.) High-level Design of Inspector-executor (cont.)

- Replicating remote elements: **associative arrays**
  - **Keys:** col_idx[k] values (i.e., indices)
  - **Values:** x[col_idx[k]] elements (i.e., remote values)
- **Pros**
  - clean way to store **sparse** indices
  - faster than Chapel’s sparse domains/arrays
  - automatically **ignores duplicates**
  - can directly use the original **col_idx[k]** indices as look-ups



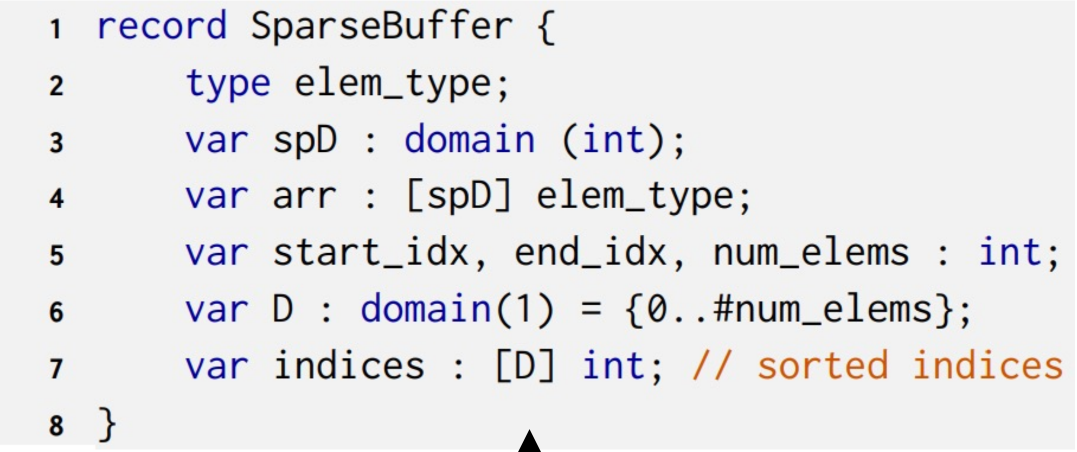
```
record SparseBuffer {
     type elem_type;
     var spD : domain (int);
     var arr : [spD] elem_type;
     var start_idx, end_idx, num_elems : int;
     var D : domain(1) = {0..#num_elems};
     var indices : [D] int; // sorted indices
}
```

Each locale stores a **SparseBuffer** record to keep track of the remote elements it will need

spD is the associative domain, **arr** is the array declared over the associative domain

2.) High-level Design of Inspector-executor (cont.)

- Replicating remote elements: **associative arrays**
  - **Keys:** col_idx[k] values (i.e., indices)
  - **Values:** x[col_idx[k]] elements (i.e., remote values)
- **Pros**
  - clean way to store **sparse** indices
  - faster than Chapel's sparse domains/arrays
  - automatically **ignores duplicates**
  - can directly use the original **col_idx[k]** indices as look-ups
- **Cons**
  - **slower** access time vs. default arrays (~2-3x)
  - **more** memory usage vs. default arrays (~ 10%)

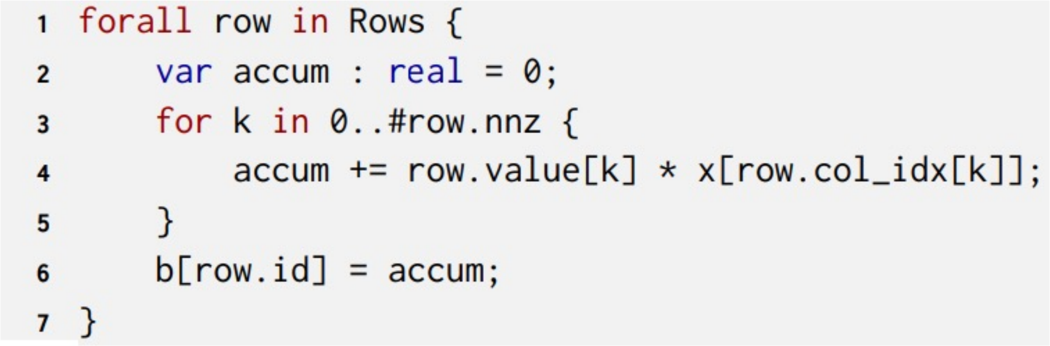


```
record SparseBuffer {
    type elem_type;
    var spD : domain (int);
    var arr : [spD] elem_type;
    var start_idx, end_idx, num_elems : int;
    var D : domain(1) = {0..#num_elems};
    var indices : [D] int; // sorted indices
}
```

Each locale stores a **SparseBuffer** record to keep track of the remote elements it will need

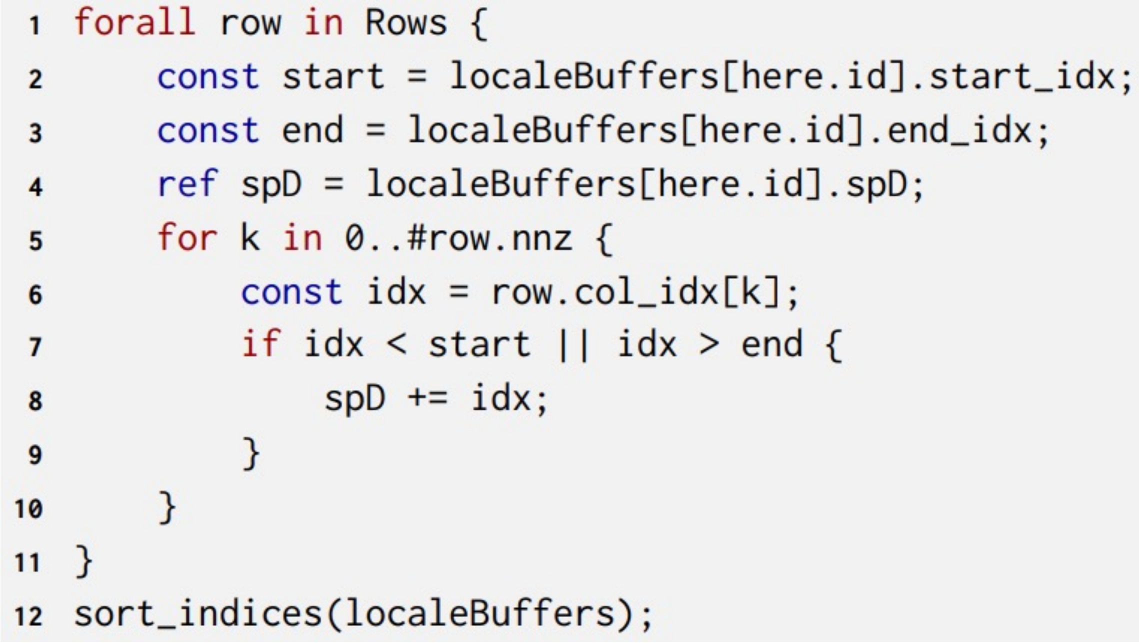
spD is the associative domain, **arr** is the array declared over the associative domain

3.) High-level Design of Inspector-executor (cont.)



```
forall row in Rows {
    var accum : real = 0;
    for k in 0..#row.nnz {
        accum += row.value[k] * x[row.col_idx[k]];
    }
    b[row.id] = accum;
}
```

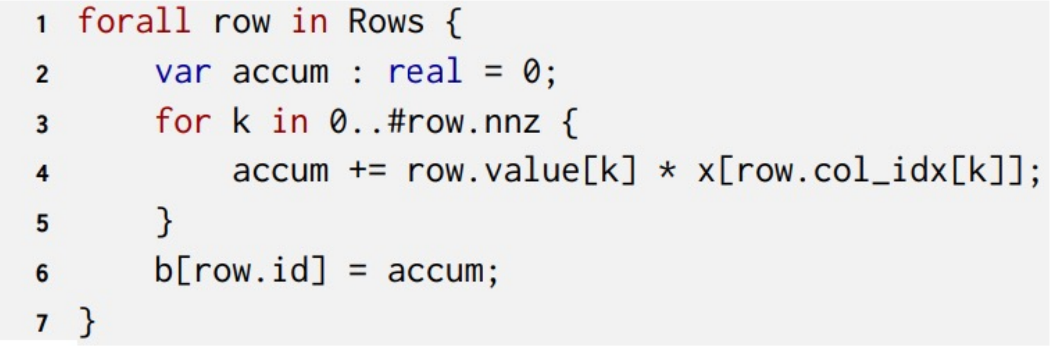
original kernel



```
forall row in Rows {
    const start = localeBuffers[here.id].start_idx;
    const end = localeBuffers[here.id].end_idx;
    ref spD = localeBuffers[here.id].spD;
    for k in 0..#row.nnz {
        const idx = row.col_idx[k];
        if idx < start || idx > end {
            spD += idx;
        }
    }
}
sort_indices(localeBuffers);
```

inspector

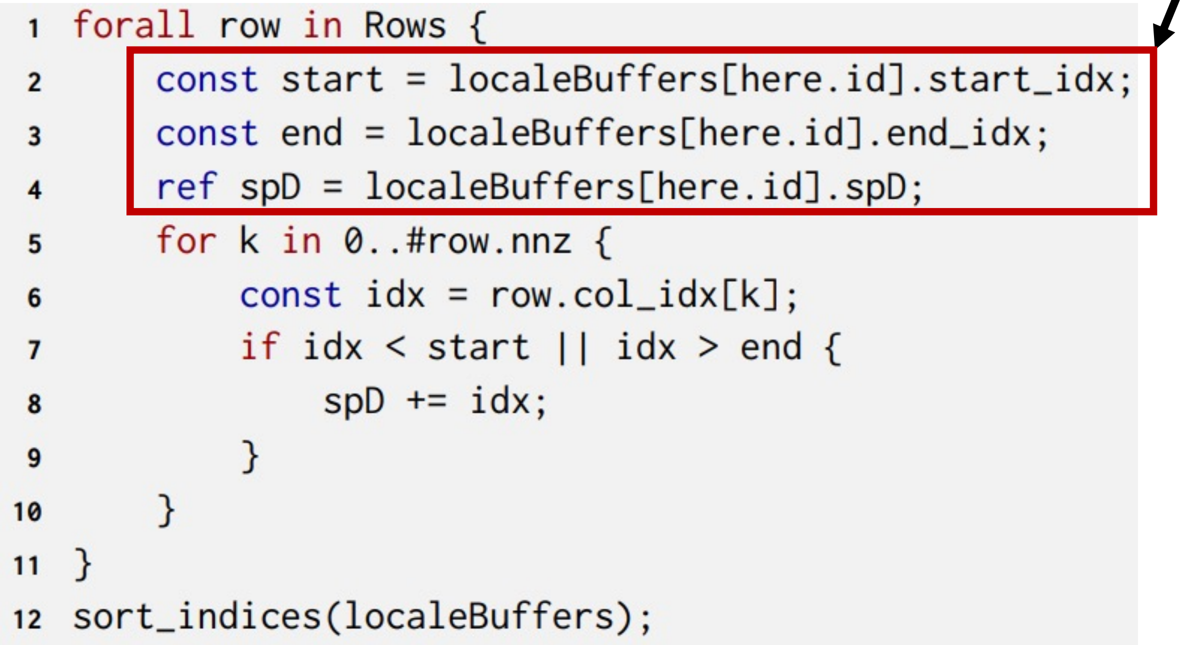
3.) High-level Design of Inspector-executor (cont.)



```
forall row in Rows {
    var accum : real = 0;
    for k in 0..#row.nnz {
        accum += row.value[k] * x[row.col_idx[k]];
    }
    b[row.id] = accum;
}
```

original kernel

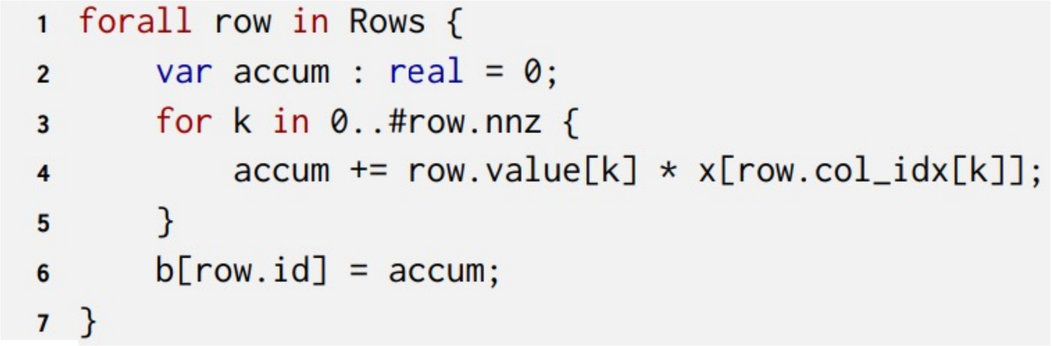
- **localeBuffers:** stores each locale’s **SparseBuffer**
- **start/end**: bounds on the locale’s local partition of **x**
- **spD**: a locale’s associative domain



```
forall row in Rows {
    const start = localeBuffers[here.id].start_idx;
    const end = localeBuffers[here.id].end_idx;
    ref spD = localeBuffers[here.id].spD;
    for k in 0..#row.nnz {
        const idx = row.col_idx[k];
        if idx < start || idx > end {
            spD += idx;
        }
    }
}
sort_indices(localeBuffers);
```

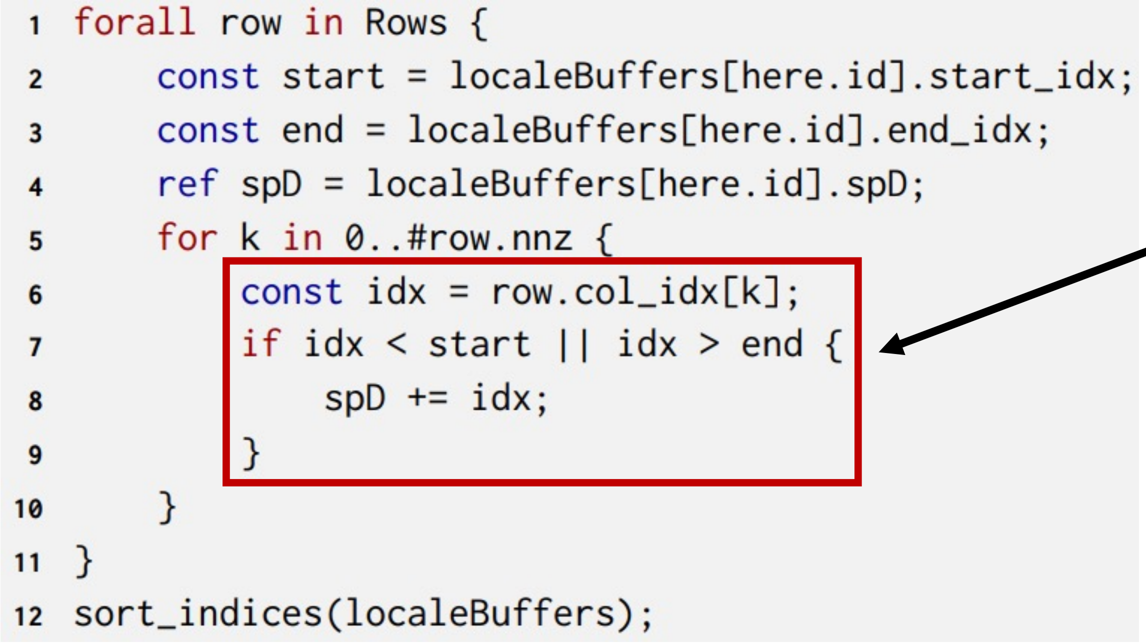
inspector

3.) High-level Design of Inspector-executor (cont.)



```
forall row in Rows {
    var accum : real = 0;
    for k in 0..#row.nnz {
        accum += row.value[k] * x[row.col_idx[k]];
    }
    b[row.id] = accum;
}
```

original kernel



```
forall row in Rows {
    const start = localeBuffers[here.id].start_idx;
    const end = localeBuffers[here.id].end_idx;
    ref spD = localeBuffers[here.id].spD;
    for k in 0..#row.nnz {
        const idx = row.col_idx[k];
        if idx < start || idx > end {
            spD += idx;
        }
    }
}
sort_indices(localeBuffers);
```

inspector

Bounds check for remote accesses

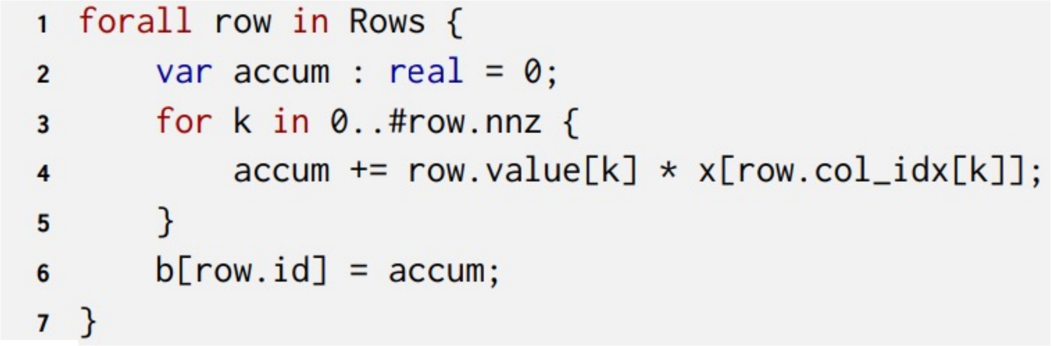
- assumes **block distribution**
- could use **.contains()** on the local subdomain but we observed significant **performance loss**
- **future work:** more general, but efficient, approach?

Does not perform actual remote communication

spD is modified by multiple tasks **concurrently**

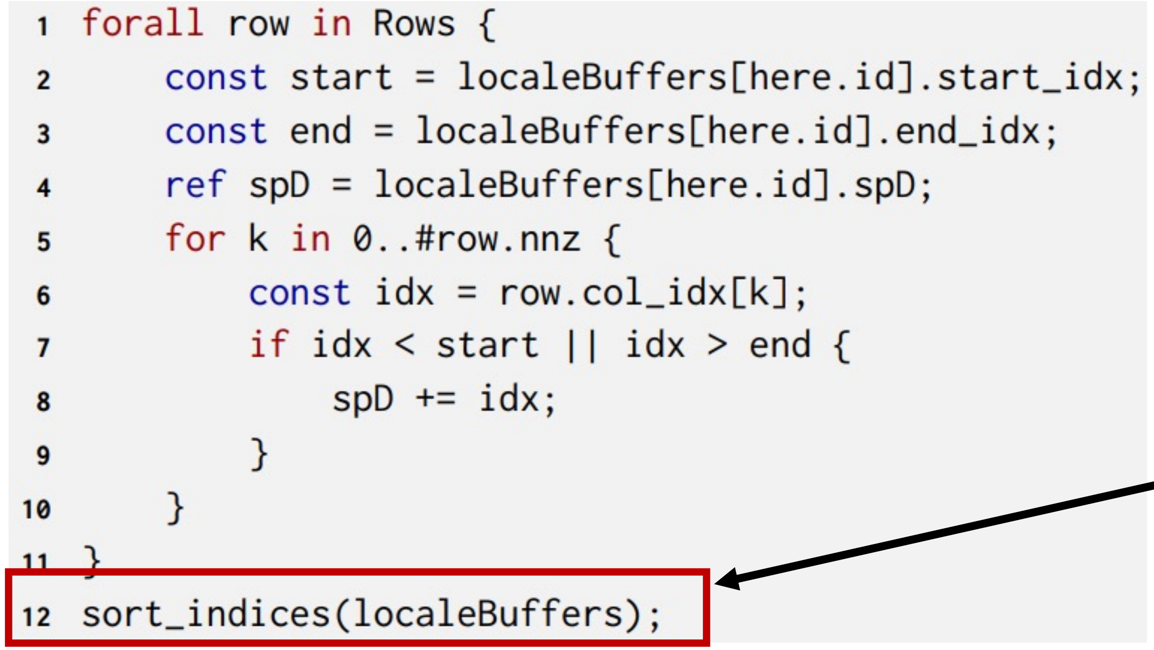
- **forall** loop performs both shared- and distributed-memory parallelism → multiple tasks spawned on each locale
- by default, associative domains provide **parallel safety** for this operation

3.) High-level Design of Inspector-executor (cont.)



```
forall row in Rows {
    var accum : real = 0;
    for k in 0..#row.nnz {
        accum += row.value[k] * x[row.col_idx[k]];
    }
    b[row.id] = accum;
}
```

original kernel



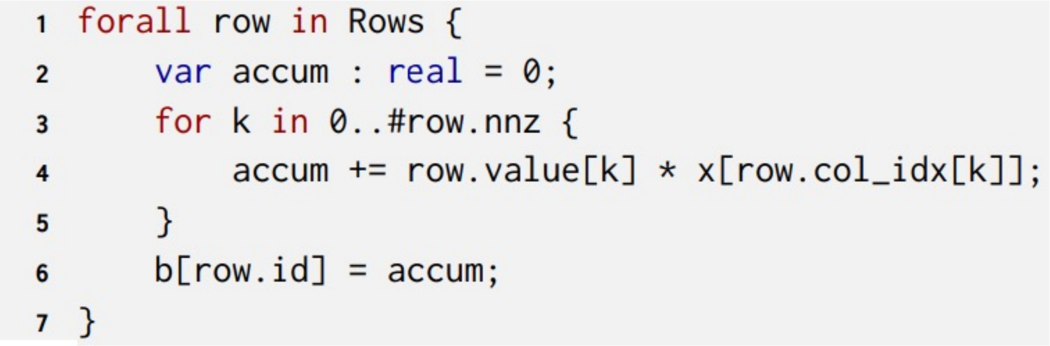
```
forall row in Rows {
    const start = localeBuffers[here.id].start_idx;
    const end = localeBuffers[here.id].end_idx;
    ref spD = localeBuffers[here.id].spD;
    for k in 0..#row.nnz {
        const idx = row.col_idx[k];
        if idx < start || idx > end {
            spD += idx;
        }
    }
}
sort_indices(localeBuffers);
```

inspector

Optimization: create **sorted** array of each locale's associative domain (i.e., their indices)

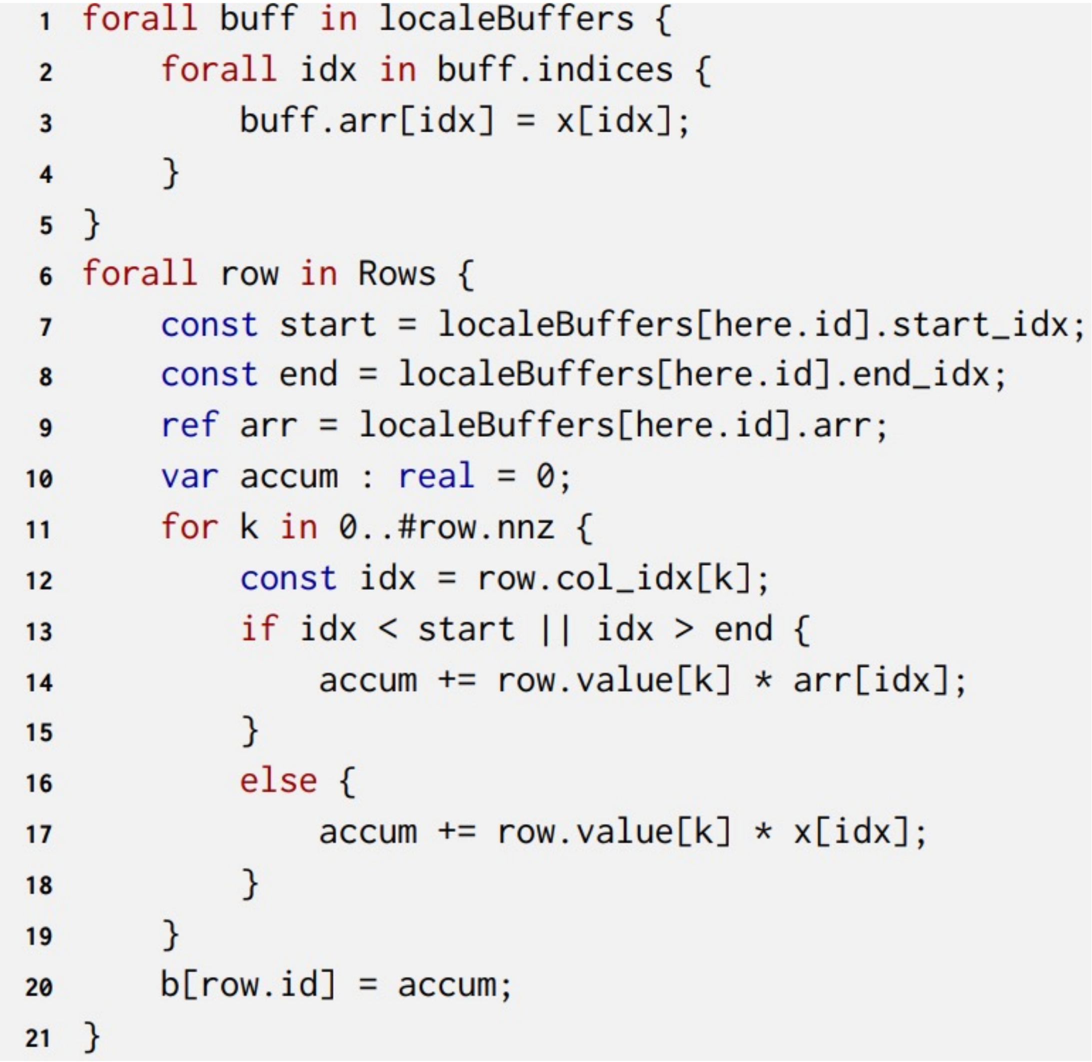
- see next slide for why this is important

3.) High-level Design of Inspector-executor (cont.)



```
forall row in Rows {
    var accum : real = 0;
    for k in 0..#row.nnz {
        accum += row.value[k] * x[row.col_idx[k]];
    }
    b[row.id] = accum;
}
```

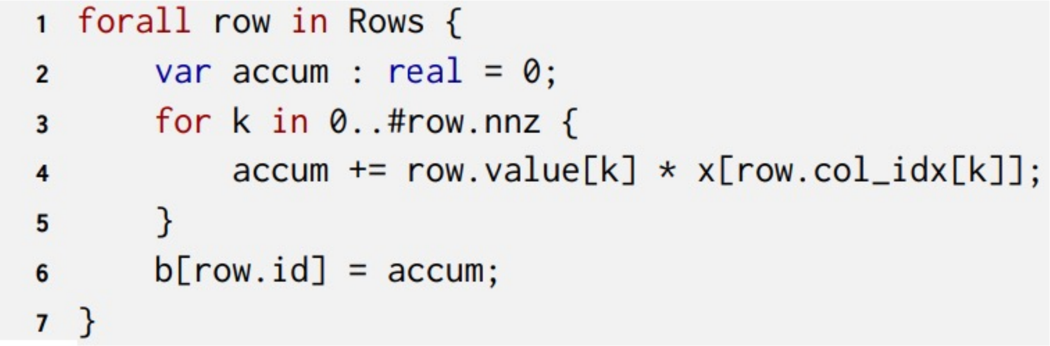
original kernel



```
forall buff in localeBuffers {
    forall idx in buff.indices {
        buff.arr[idx] = x[idx];
    }
}
forall row in Rows {
    const start = localeBuffers[here.id].start_idx;
    const end = localeBuffers[here.id].end_idx;
    ref arr = localeBuffers[here.id].arr;
    var accum : real = 0;
    for k in 0..#row.nnz {
        const idx = row.col_idx[k];
        if idx < start || idx > end {
            accum += row.value[k] * arr[idx];
        }
        else {
            accum += row.value[k] * x[idx];
        }
    }
    b[row.id] = accum;
}
```

executor

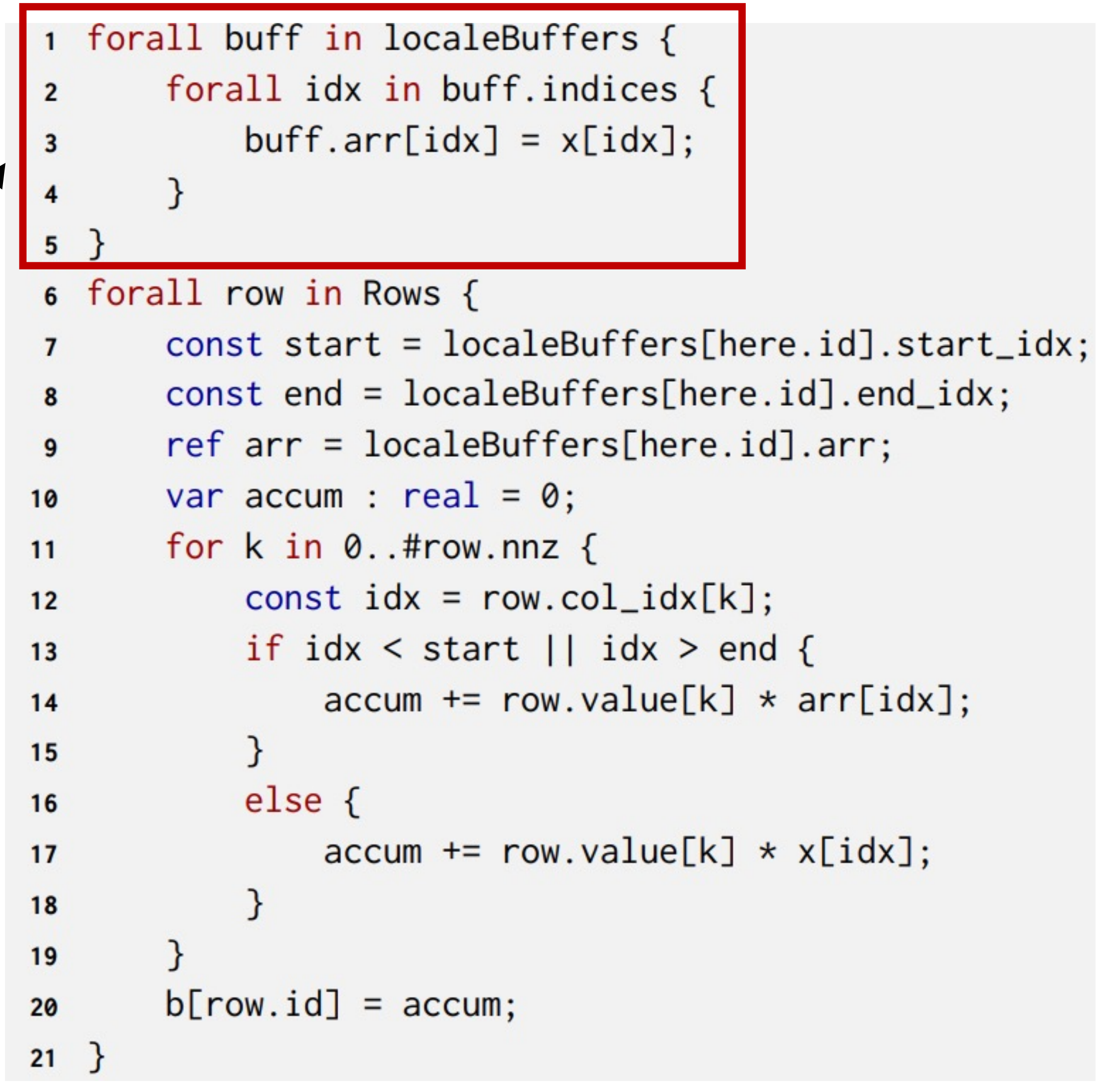
3.) High-level Design of Inspector-executor (cont.)



```
forall row in Rows {
    var accum : real = 0;
    for k in 0..#row.nnz {
        accum += row.value[k] * x[row.col_idx[k]];
    }
    b[row.id] = accum;
}
```

original kernel

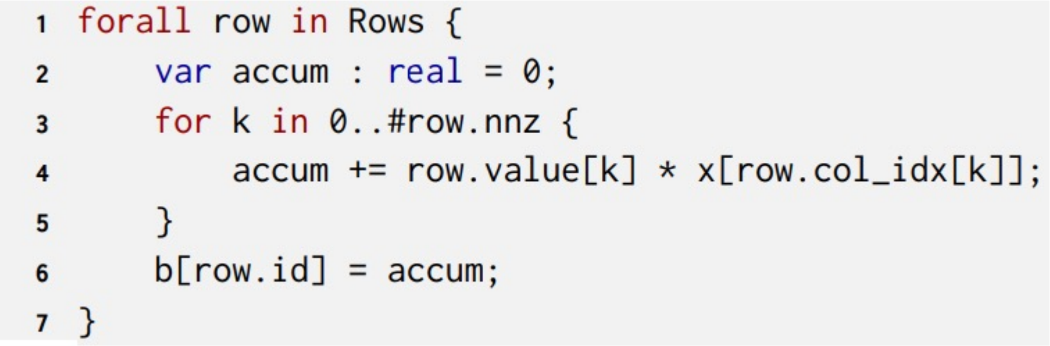
Update/gather the original values from **x** to each locale's replicated copy
→ values most likely changed outside of the kernel



```
forall buff in localeBuffers {
    forall idx in buff.indices {
        buff.arr[idx] = x[idx];
    }
}
forall row in Rows {
    const start = localeBuffers[here.id].start_idx;
    const end = localeBuffers[here.id].end_idx;
    ref arr = localeBuffers[here.id].arr;
    var accum : real = 0;
    for k in 0..#row.nnz {
        const idx = row.col_idx[k];
        if idx < start || idx > end {
            accum += row.value[k] * arr[idx];
        }
        else {
            accum += row.value[k] * x[idx];
        }
    }
    b[row.id] = accum;
}
```

executor

3.) High-level Design of Inspector-executor (cont.)



```
forall row in Rows {
    var accum : real = 0;
    for k in 0..#row.nnz {
        accum += row.value[k] * x[row.col_idx[k]];
    }
    b[row.id] = accum;
}
```

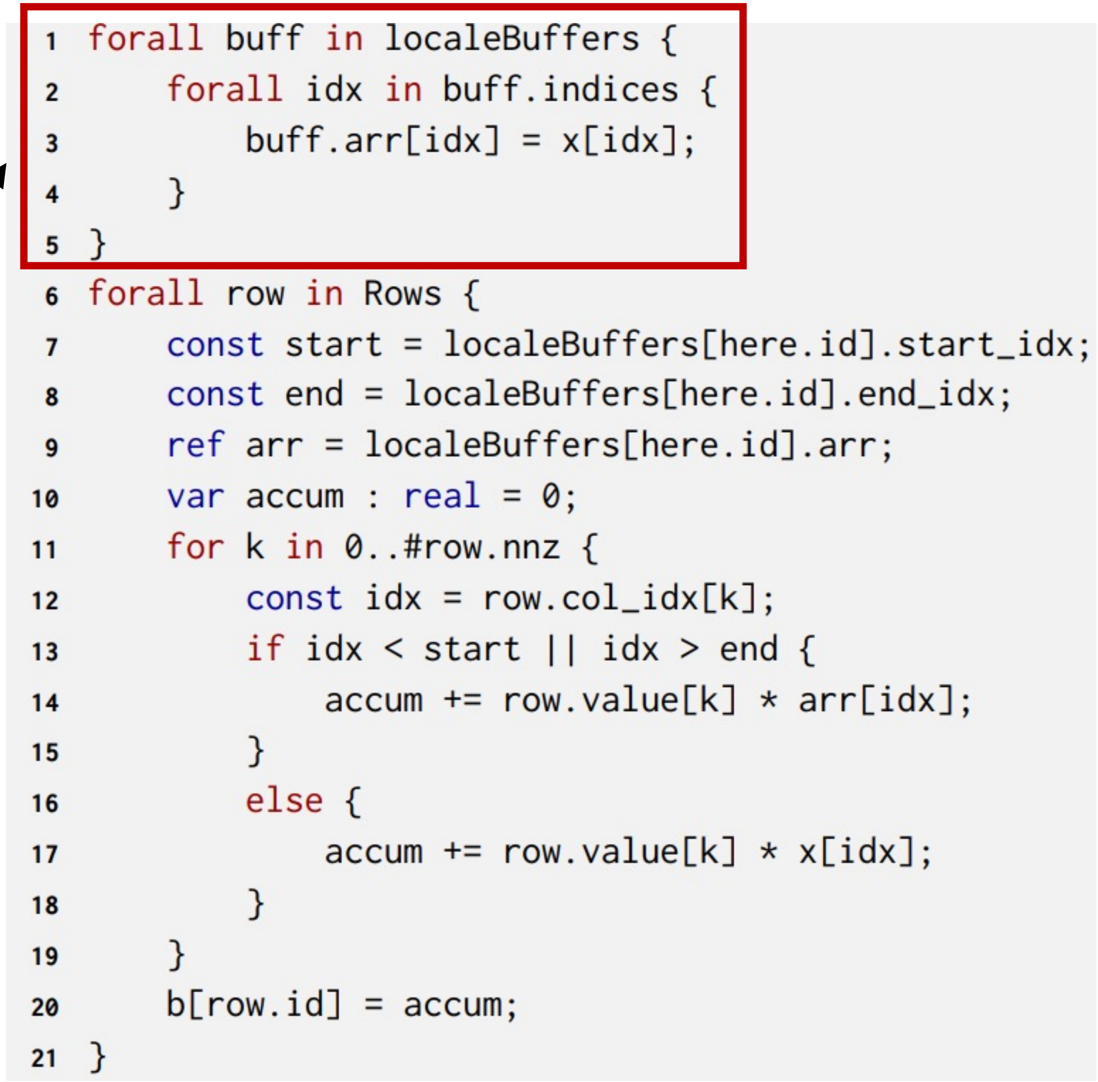
original kernel

Update/gather the original values from **x** to each locale's replicated copy

→ values most likely changed outside of the kernel

All updates are **remote reads**. But since each remote element is **stored only once**, we do a single remote read and get "unlimited" local accesses during the kernel

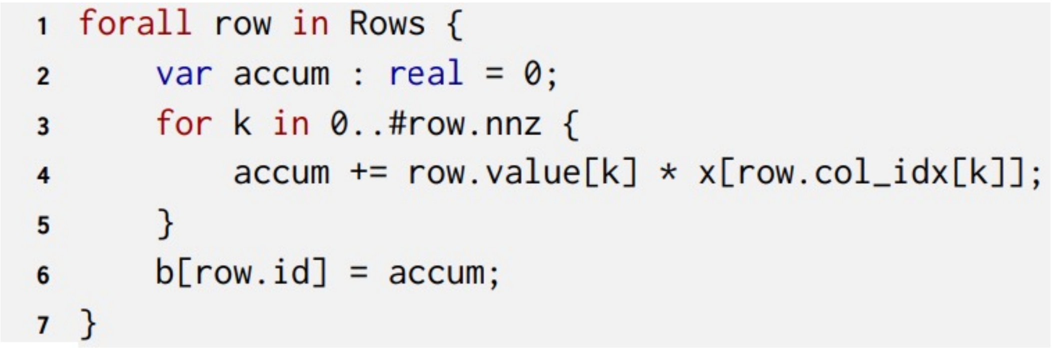
→ **this is the key to our approach achieving performance gains**



```
forall buff in localeBuffers {
    forall idx in buff.indices {
        buff.arr[idx] = x[idx];
    }
}
forall row in Rows {
    const start = localeBuffers[here.id].start_idx;
    const end = localeBuffers[here.id].end_idx;
    ref arr = localeBuffers[here.id].arr;
    var accum : real = 0;
    for k in 0..#row.nnz {
        const idx = row.col_idx[k];
        if idx < start || idx > end {
            accum += row.value[k] * arr[idx];
        }
        else {
            accum += row.value[k] * x[idx];
        }
    }
    b[row.id] = accum;
}
```

executor

3.) High-level Design of Inspector-executor (cont.)



```
forall row in Rows {
    var accum : real = 0;
    for k in 0..#row.nnz {
        accum += row.value[k] * x[row.col_idx[k]];
    }
    b[row.id] = accum;
}
```

original kernel

Update/gather the original values from **x** to each locale’s replicated copy

→ values most likely changed outside of the kernel

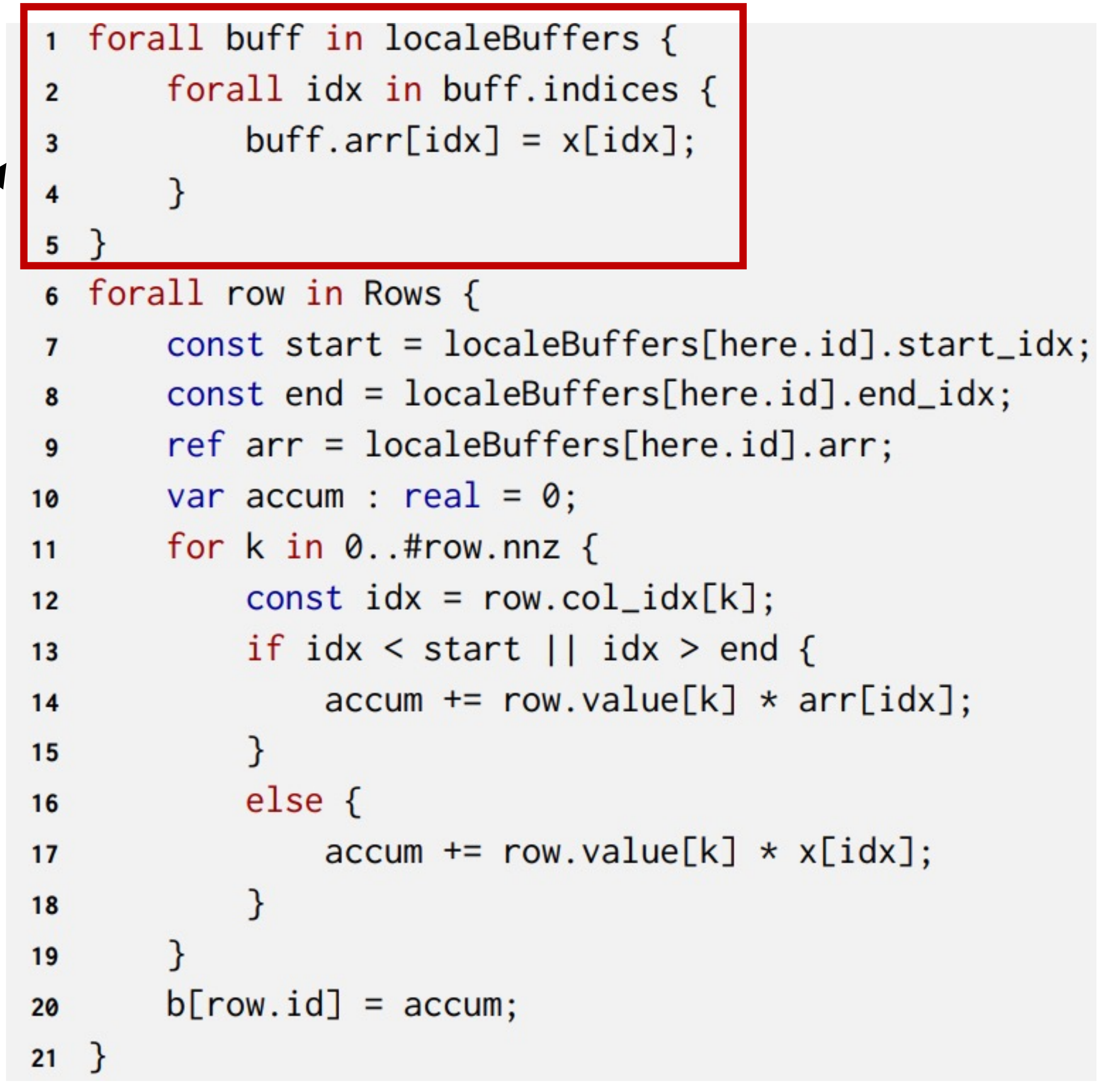
All updates are **remote reads**. But since each remote element is **stored only once**, we do a single remote read and get “unlimited” local accesses during the kernel

→ **this is the key to our approach achieving performance gains**

.indices is a sorted array of the associative array’s keys

- associative array indices are **unsorted**, so directly iterating over them leads to **poor locality for Chapel’s remote cache**

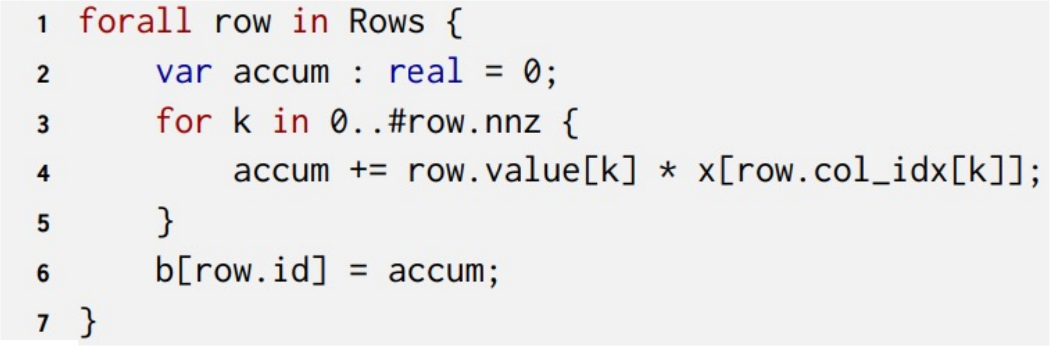
→ observed as much as a **22x speed-up vs. not sorting**



```
forall buff in localeBuffers {
    forall idx in buff.indices {
        buff.arr[idx] = x[idx];
    }
}
forall row in Rows {
    const start = localeBuffers[here.id].start_idx;
    const end = localeBuffers[here.id].end_idx;
    ref arr = localeBuffers[here.id].arr;
    var accum : real = 0;
    for k in 0..#row.nnz {
        const idx = row.col_idx[k];
        if idx < start || idx > end {
            accum += row.value[k] * arr[idx];
        }
        else {
            accum += row.value[k] * x[idx];
        }
    }
    b[row.id] = accum;
}
```

executor

3.) High-level Design of Inspector-executor (cont.)

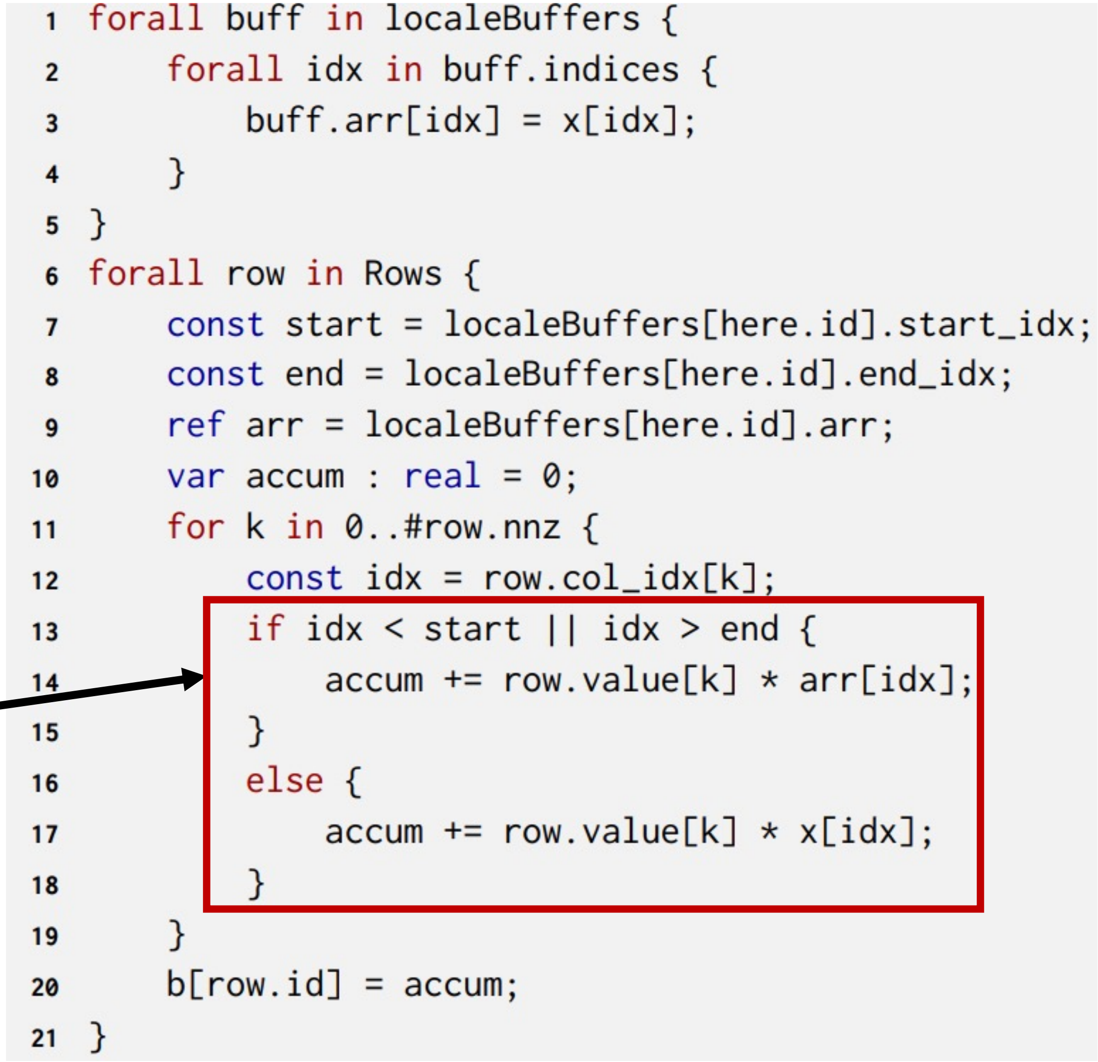


```
forall row in Rows {
    var accum : real = 0;
    for k in 0..#row.nnz {
        accum += row.value[k] * x[row.col_idx[k]];
    }
    b[row.id] = accum;
}
```

original kernel

Same **bounds check** as inspector

- if the access will be **remote**, then access the associative array (**arr**)



```
forall buff in localeBuffers {
    forall idx in buff.indices {
        buff.arr[idx] = x[idx];
    }
}
forall row in Rows {
    const start = localeBuffers[here.id].start_idx;
    const end = localeBuffers[here.id].end_idx;
    ref arr = localeBuffers[here.id].arr;
    var accum : real = 0;
    for k in 0..#row.nnz {
        const idx = row.col_idx[k];
        if idx < start || idx > end {
            accum += row.value[k] * arr[idx];
        }
        else {
            accum += row.value[k] * x[idx];
        }
    }
    b[row.id] = accum;
}
```

executor

Outline

- Motivation and background
  - Irregular applications
  - Inspector-executor technique
- High-level design of inspector-executor
- Performance evaluation
  - NAS-CG, moldyn, **PageRank** → see our paper for moldyn and NAS-CG results
- Future work

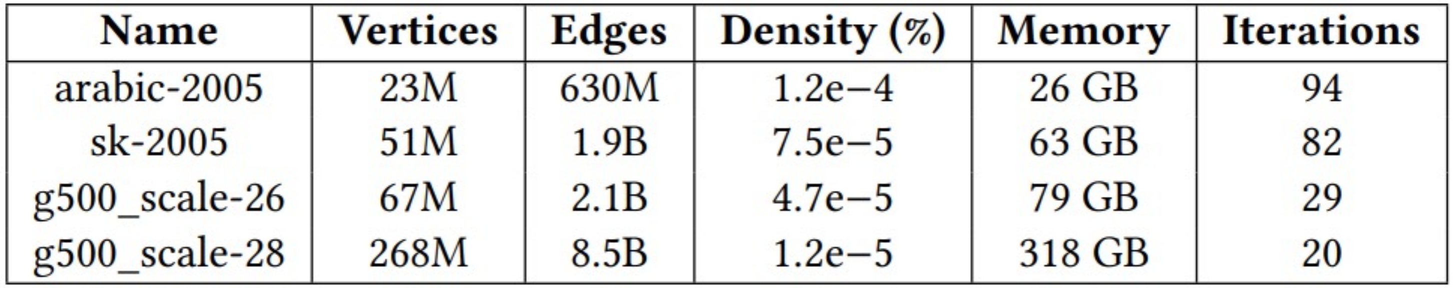
3.) Performance Evaluation: Setup

- System:
  - 16 node FDR Infiniband Cluster
  - Each node →512GB DDR4, 2x Intel Xeon E5-2650v3 (20 cores total)
  - Hyperthreading enabled
- Chapel:
  - 1.24.1, LLVM 11.0.1
  - --fast and --cache-remote
  - GASNet over Infiniband
- Results:
  - **average** over multiple trials (coefficient of variation does not exceed 0.07)
- Comparisons:
  - **Baseline** → no inspector-executor optimization
  - **Replicate-all** → no inspector performed, just give each locale a full copy of the array
- Will refer to inspector-executor as **I/E**

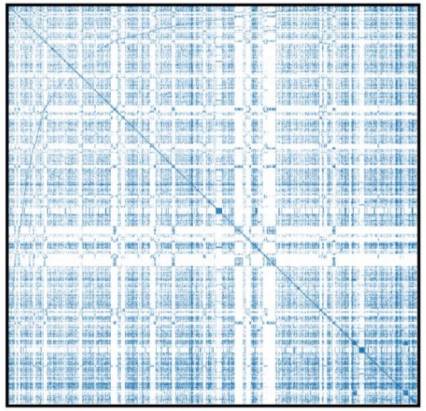
3.) Performance Evaluation: PageRank

- Evaluate two real web-graphs and two Graph500 graphs (https://graph500.org/)
- Execute **until convergence**: tolerance of **1e-10**, damping factor of **0.85**
- Baseline only runs 1 iteration of Graph500 graphs
    - for 2 locales, estimated to require 20 days for all iterations on g500_scale-28
    - baseline results are extrapolated from single iteration runtimes

Table 3: Data Sets for PageRank



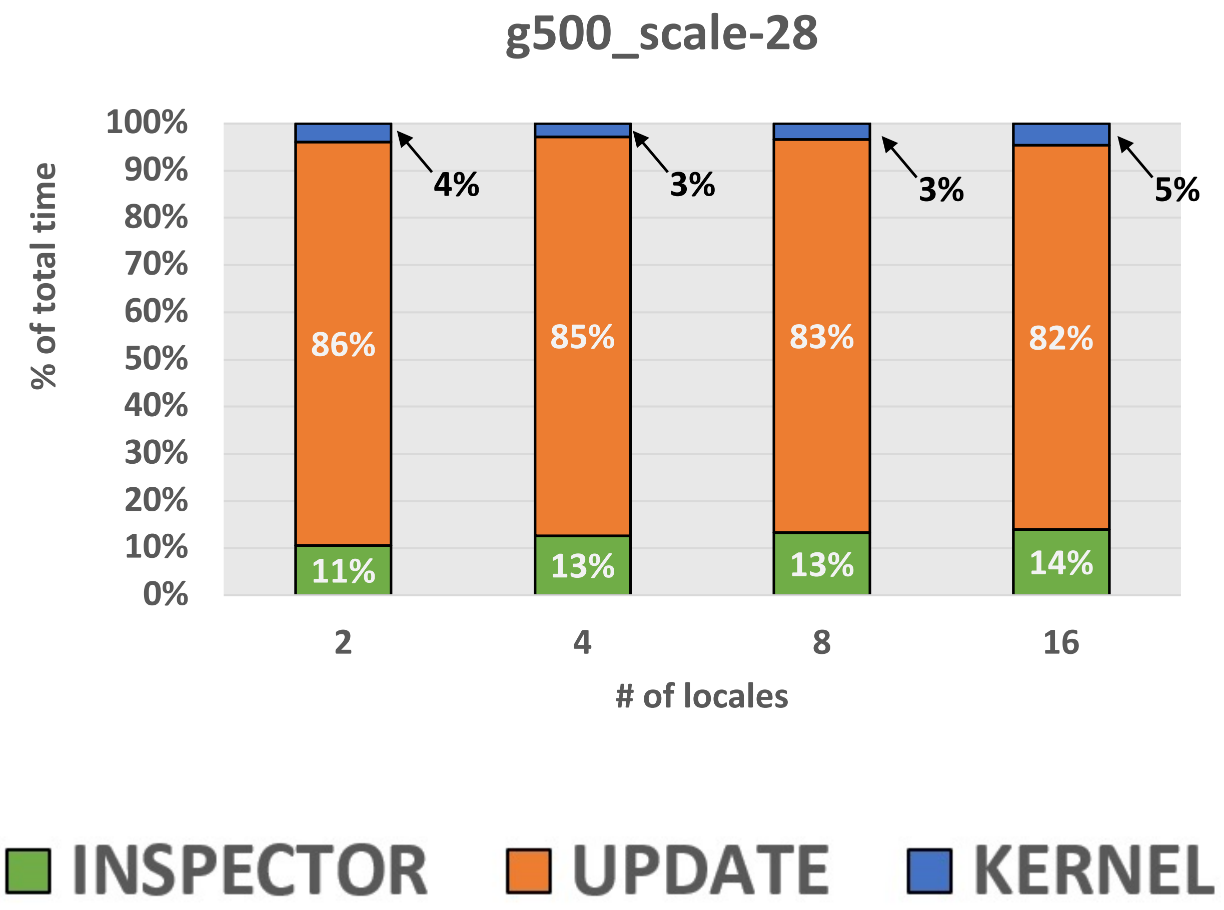
Name	Vertices	Edges	Density (%)	Memory	Iterations
arabic-2005	23M	630M	1.2e−4	26 GB	94
sk-2005	51M	1.9B	7.5e−5	63 GB	82
g500_scale-26	67M	2.1B	4.7e−5	79 GB	29
g500_scale-28	268M	8.5B	1.2e−5	318 GB	20



PageRank: arabic-2005

3.) Performance Evaluation: PageRank (cont.)

- Inspector runtime overhead:
    - **geomean** overhead of 5% relative to the total execution time
- Low overhead due to many iterations, allowing for overhead to be amortized



3.) Performance Evaluation: PageRank (cont.)

- I/E memory usage:
  - **geomean increase** in memory over the baseline of **80%**
  - **high memory** usage is due to the **large Graph500 graphs**
    - memory usage increase for real-world graphs is **42%**

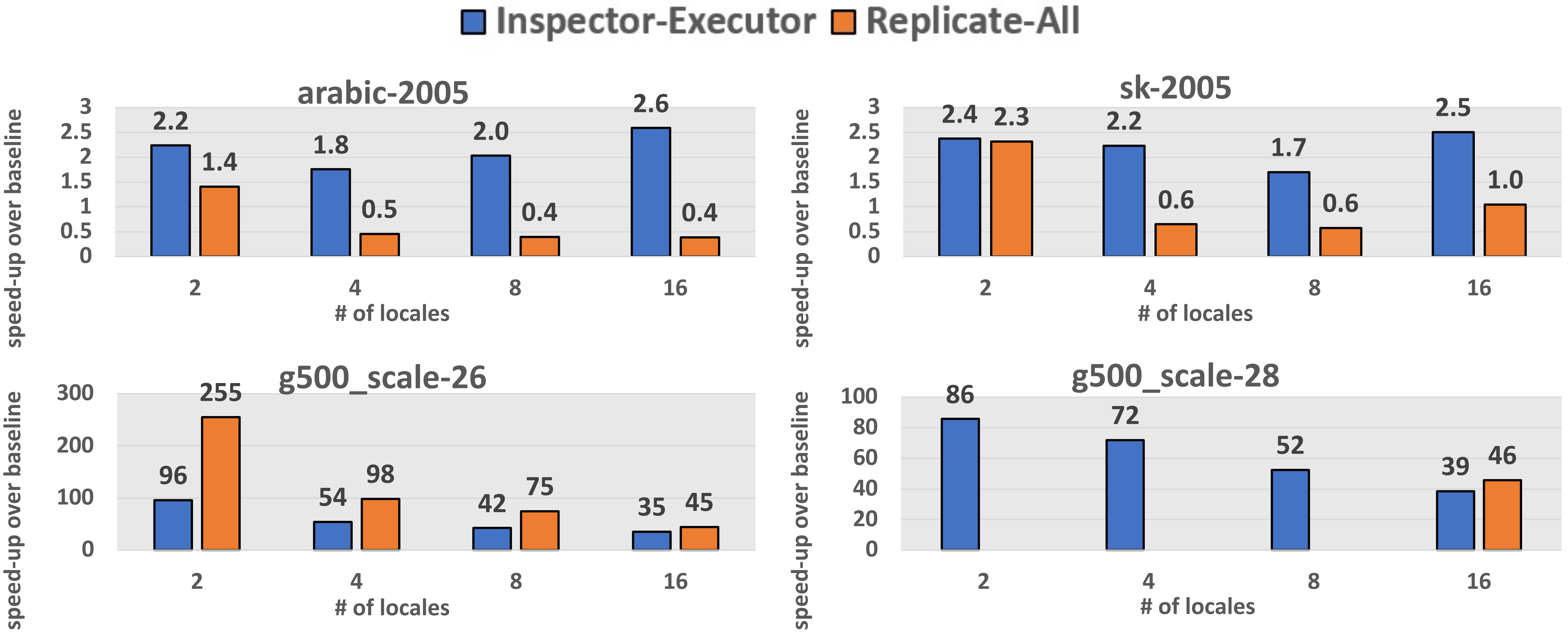
3.) Performance Evaluation: PageRank (cont.)

- I/E memory usage:
  - **geomean increase** in memory over the baseline of **80%**
  - **high memory** usage is due to the **large Graph500 graphs**
    - memory usage increase for real-world graphs is **42%**
- Replicate-all memory usage:
  - **geomean increase** in memory over baseline of **606%**
  - **cannot run** g500_scale-28 on 2, 4, or 8 locales →**out of memory**
  - real-world graph memory usage increase is **565%**

Key Point: I/E replicates **less data** than replicate-all

- I/E only replicates what will be accessed remotely
- replicate-all replicates **EVERYTHING**

PageRank Runtime Speed-ups



- I/E: geomean speed-up of **11x**
- Replicate-all: geomean speed-up of **5x**

PageRank Runtime Speed-ups

I/E exploits data reuse

- single remote get per remote element gives us "unlimited" local accesses

I/E replicates less data

- spends less time in the gather/update phase than replicate-all

I/E slower on Graph500 graphs vs replicate-all

- I/E needs to replicate **virtually all the elements**
- Performance now bounded by access costs to **associative arrays vs. default arrays**

3.) Performance Evaluation: PageRank (cont.)

- Noteworthy comparisons
    - For two locales:
        - baseline estimated to require **20 days** to run all iterations on g500_scale-28
        - **I/E does it in 6 hours**
    - For 16 locales:
        - baseline estimated to require **41 hours**
        - **I/E does it in 1 hour**

3.) Performance Summary

- Note far right column
  - relatively few iterations required until I/E is on par, or faster, than baseline

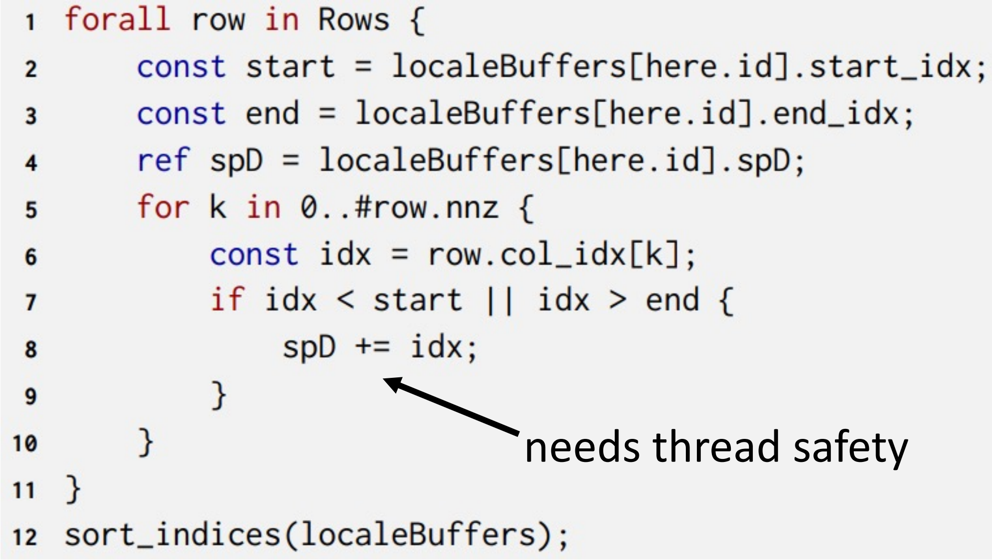
Application	Average Memory Overhead	Average Inspector Overhead	Average Runtime Speed-up	Max # of Iterations to Break Even with Baseline
NAS-CG	6%	4%	27x	**2**
moldyn	4%	24%	8x	**1**
PageRank	80%	5%	11x	**4**

Outline

- Motivation and background
  - Irregular applications
  - Inspector-executor technique
- High-level design of inspector-executor
- Applying the inspector-executor
- Performance evaluation
  - NAS-CG, moldyn, PageRank
- Future work

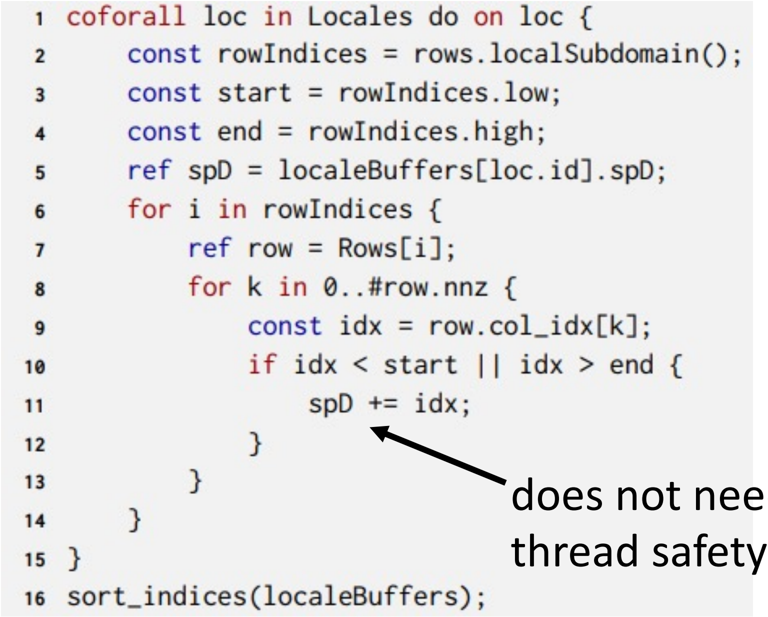
4.) Future Work

- Optimizing the optimization:
    - transform **forall to coforall** for inspector to speed-up associative domain operation
        - forall loop over distributed array will spawn multiple tasks per locale
        - need **parallel-safety** for associative domain (parSafe=true)
        - Use a **coforall** instead, allowing us to set parSafe=false
        - Reduces parallelism but still gives us net performance gains (as much as **6x faster**)
        - Generally, this transformation can be done, but not always true



```
forall row in Rows {
    const start = localeBuffers[here.id].start_idx;
    const end = localeBuffers[here.id].end_idx;
    ref spD = localeBuffers[here.id].spD;
    for k in 0..#row.nnz {
        const idx = row.col_idx[k];
        if idx < start || idx > end {
            spD += idx;
        }
    }
}
sort_indices(localeBuffers);
```

needs thread safety



```
coforall loc in Locales do on loc {
    const rowIndices = rows.localSubdomain();
    const start = rowIndices.low;
    const end = rowIndices.high;
    ref spD = localeBuffers[loc.id].spD;
    for i in rowIndices {
        ref row = Rows[i];
        for k in 0..#row.nnz {
            const idx = row.col_idx[k];
            if idx < start || idx > end {
                spD += idx;
            }
        }
    }
}
sort_indices(localeBuffers);
```

does not need thread safety

4.) Future Work (cont.)

- Optimizing the optimization:
  - use **aggregation** for the update/gathers before the kernel
  - use **default arrays** instead of associative arrays
    - more efficient memory accesses
    - requires building a new index mapping from **indirection array** to indices in the default array
    - gets much uglier than the associative array approach, so there’s a tradeoff between performance and what the compiler could automatically generate

4.) Future Work (cont.)

- Optimizing the optimization:
  - use **aggregation** for the update/gathers before the kernel
  - use **default arrays** instead of associative arrays
    - more efficient memory accesses
    - requires building a new index mapping from **indirection array** to indices in the default array
    - gets much uglier than the associative array approach, so there's a tradeoff between performance and what the compiler could automatically generate
- Compiler automation:
  - user driven (pragmas) or have the compiler try to find suitable kernels?
- More applications please!
  - not ideal for the optimization developer to write the test cases
  - if you have irregular applications that could benefit from runtime optimizations (not just inspector-executor), **contact us! tbrolin@cs.umd.edu**

Conclusions

- Inspector-executor shows promise for irregular applications in Chapel
- Speed-ups as high as **224x**
- Take application runtimes from **days to hours**
- Does not rely on low-level details to be exposed in the source code
  - our goal with the baseline implementations was to write them in the **most natural way**, sticking to the “on-paper” description of the algorithms

Runtime Optimizations for Irregular Applications in Chapel

Thomas B. Rolinger (UMD/LPS), Christopher D. Krieger (LPS), Alan Sussman (UMD)

Contact: tbrolin@cs.umd.edu

CHIUW 2021

COMPUTER SCIENCE
UNIVERSITY OF MARYLAND

The Laboratory for Physical Sciences