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
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.)

• 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
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

Sparse Matrix-Vector Multiply (SpMV) kernel

```plaintext
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;
}
```

**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

Sparse Matrix-Vector Multiply (SpMV) kernel

```
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;
}
```

- `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 $\text{col}_\text{idx}[k]$ result in remote accesses to $x$ for a given locale
- replicate the remote elements on that locale and access those copies instead

$\Rightarrow$ Construct a mapping from $\text{col}_\text{idx}[k]$ to $x[\text{col}_\text{idx}[k]]$ 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)

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 lookups

Cons
- slower access time vs. default arrays (~2-3x)
- more memory usage vs. default arrays (~10%)

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

```plaintext
1 record SparseBuffer {
2     type elem_type;
3     var spD : domain (int);
4     var arr : [spD] elem_type;
5     var start_idx, end_idx, num elems : int;
6     var D : domain(1) = {0..#num elems};
7     var indices : [D] int; // sorted indices
8 }
```

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

```plaintext
1 record SparseBuffer {
  2     type elem_type;
  3     var spD : domain (int);
  4     var arr : [spD] elem_type;
  5     var start_idx, end_idx, num_elems : int;
  6     var D : domain(1) = {0..#num_elems};
  7     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%)

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

```csharp
1 for all row in Rows {
2   var accum : real = 0;
3   for k in 0..#row.nnz {
4       accum += row.value[k] * x[row.col_idx[k]];
5   }
6   b[row.id] = accum;
7 }

original kernel

```csharp
1 for all row in Rows {
2   const start = localeBuffers[here.id].start_idx;
3   const end = localeBuffers[here.id].end_idx;
4   ref spD = localeBuffers[here.id].spD;
5   for k in 0..#row.nnz {
6       const idx = row.col_idx[k];
7       if idx < start || idx > end {
8           spD += idx;
9       }
10   }
11 }
12 sort_indices(localeBuffers);

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

original kernel

```java
1 for all row in Rows {
2 var accum : real = 0;
3 for k in 0..#row.nnz {
4     accum += row.value[k] * x[row.col_idx[k]];
5 }
6 b[row.id] = accum;
7 }
```

inspector

```java
1 for all row in Rows {
2 const start = localeBuffers[here.id].start_idx;
3 const end = localeBuffers[here.id].end_idx;
4 ref spD = localeBuffers[here.id].spD;
5 for k in 0..#row.nnz {
6     const idx = row.col_idx[k];
7     if idx < start || idx > end {
8         spD += idx;
9     }
10 }
11 }
12 sort_indices(localeBuffers);
```

- **localeBuffers**: stores each locale’s `SparseBuffer`
- **start/end**: bounds on the locale’s local partition of `x`
- **spD**: a locale’s associative domain
3.) High-level Design of Inspector-executor (cont.)

```
1 forall row in Rows {
2     var accum : real = 0;
3     for k in 0..#row.nnz {
4         accum += row.value[k] * x[row.col_idx[k]];
5     }
6     b[row.id] = accum;
7 }
```

**original kernel**

```
1 forall row in Rows {
2     const start = localeBuffers[here.id].start_idx;
3     const end = localeBuffers[here.id].end_idx;
4     ref spD = localeBuffers[here.id].spD;
5     for k in 0..#row.nnz {
6         const idx = row.col_idx[k];
7         if idx < start || idx > end {
8             spD += idx;
9         }
10     }
11     sort_indices(localeBuffers);
12 }
```

**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.)

```plaintext
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**

```plaintext
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);
}
```

**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.)

```c
1 forall row in Rows {
2   var acum : real = 0;
3   for k in 0..#row.nnz {
4       acum += row.value[k] * x[row.col_idx[k]];
5   }
6   b[row.id] = acum;
7 }
```

**original kernel**

```c
1 forall buf in localeBuffers {
2   forall idx in buf.indices {
3       buff.arr[idx] = x[idx];
4   }
5 }
6 forall row in Rows {
7   const start = localeBuffers[here.id].start_idx;
8   const end = localeBuffers[here.id].end_idx;
9   ref arr = localeBuffers[here.id].arr;
10  var acum : real = 0;
11  for k in 0..#row.nnz {
12     const idx = row.col_idx[k];
13     if idx < start || idx > end {
14       acum += row.value[k] * arr[idx];
15     }
16     else {
17       acum += row.value[k] * x[idx];
18     }
19   } b[row.id] = acum;
20 }
```

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

```java
1 for all row in Rows {
2    var accum : real = 0;
3    for k in 0..#row.nnz {
4        accum += row.value[k] * x[row.col_idx[k]];
5    }
6    b[row.id] = accum;
7 }
```

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

**original kernel**

```java
1 for all buff in localeBuffers {
2    for all idx in buff.indices {
3        buff.arr[idx] = x[idx];
4    }
5 }
```

```java
1 for all row in Rows {
2    const start = localeBuffers[here.id].start_idx;
3    const end = localeBuffers[here.id].end_idx;
4    ref arr = localeBuffers[here.id].arr;
5    var accum : real = 0;
6    for k in 0..#row.nnz {
7        const idx = row.col_idx[k];
8        if idx < start || idx > end {
9            accum += row.value[k] * arr[idx];
10        }
11        else {
12            accum += row.value[k] * x[idx];
13        }
14    }
15    b[row.id] = accum;
16 }
```
3.) High-level Design of Inspector-executor (cont.)

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

Original Kernel:

```
for all 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;
}
```

**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.

Executor:

```
for all buff in localeBuffers {
    for all idx in buff.indices {
        buff.arr[idx] = x[idx];
    }
}
for all 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;
}
```

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

```c
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)

```c
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 $1 \times 10^{-10}$, 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>
<thead>
<tr>
<th>Name</th>
<th>Vertices</th>
<th>Edges</th>
<th>Density (%)</th>
<th>Memory</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>arabic-2005</td>
<td>23M</td>
<td>630M</td>
<td>1.2e-4</td>
<td>26 GB</td>
<td>94</td>
</tr>
<tr>
<td>sk-2005</td>
<td>51M</td>
<td>1.9B</td>
<td>7.5e-5</td>
<td>63 GB</td>
<td>82</td>
</tr>
<tr>
<td>g500_scale-26</td>
<td>67M</td>
<td>2.1B</td>
<td>4.7e-5</td>
<td>79 GB</td>
<td>29</td>
</tr>
<tr>
<td>g500_scale-28</td>
<td>268M</td>
<td>8.5B</td>
<td>1.2e-5</td>
<td>318 GB</td>
<td>20</td>
</tr>
</tbody>
</table>
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

![Diagram showing % of total time across different scales](image)
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%

**Key Point**: I/E replicates less data than replicate-all
- I/E only replicates what will be accessed remotely
- replicate-all replicates EVERYTHING
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

**Inspector-Executor** vs **Replicate-All**

- **arabic-2005**
  - # of locales: 2, 4, 8, 16
  - Speed-up over baseline: 2.2, 1.4, 1.8, 0.5

- **sk-2005**
  - # of locales: 2, 4, 8, 16
  - Speed-up over baseline: 2.4, 2.3, 2.2, 0.6

- **g500_scale-26**
  - # of locales: 2, 4, 8, 16
  - Speed-up over baseline: 96, 54, 98, 42

- **g500_scale-28**
  - # of locales: 2, 4, 8, 16
  - Speed-up over baseline: 86, 72, 52, 39

- **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

<table>
<thead>
<tr>
<th>Application</th>
<th>Average Memory Overhead</th>
<th>Average Inspector Overhead</th>
<th>Average Runtime Speed-up</th>
<th>Max # of Iterations to Break Even with Baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAS-CG</td>
<td>6%</td>
<td>4%</td>
<td>27x</td>
<td>2</td>
</tr>
<tr>
<td>moldyn</td>
<td>4%</td>
<td>24%</td>
<td>8x</td>
<td>1</td>
</tr>
<tr>
<td>PageRank</td>
<td>80%</td>
<td>5%</td>
<td>11x</td>
<td>4</td>
</tr>
</tbody>
</table>
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

```c++
for all 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);
}
```

```c++
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);
}
```
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