Compiler Optimization for Irregular Memory Accesses in Chapel

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

Contact: tbrolin@cs.umd.edu

CHIUW 2022

COMPUTER SCIENCE
UNIVERSITY OF MARYLAND

The Laboratory for Physical Sciences

Motivation: Why target irregular memory accesses?

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

Distributed Sparse Matrix-Vector Multiply (SpMV)

Motivation: Why target irregular memory accesses?

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

Distributed Sparse Matrix-Vector Multiply (SpMV)

Rows is a distributed array

Motivation: Why target irregular memory accesses?

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

Rows is a distributed array

Irregular memory access

x is a distributed array

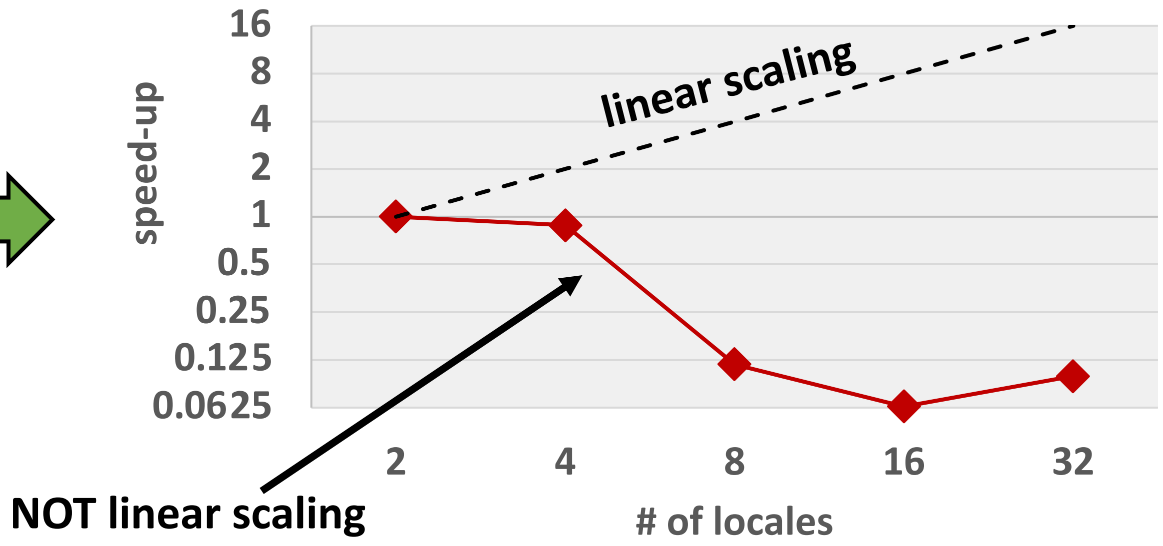
Distributed Sparse Matrix-Vector Multiply (SpMV)

Motivation: Why target irregular memory accesses?

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

Distributed Sparse Matrix-Vector Multiply (SpMV)

NAS-CG (Conjugate Gradient)
Problem Size D (73 million non-zeros)



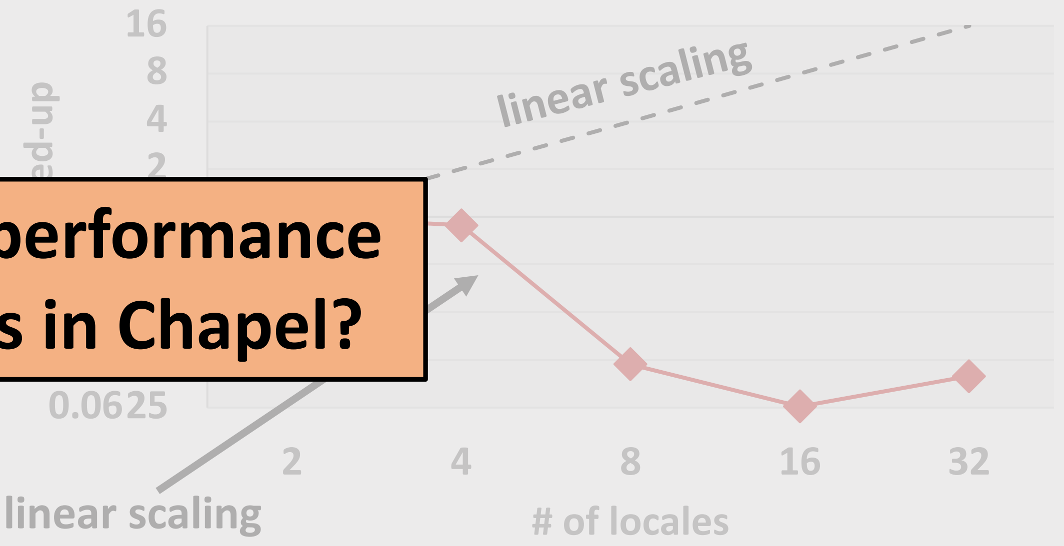
High Productivity does not always lead to High Performance

Motivation: Why target irregular memory accesses?

```
forall row in Rows {
  const id = row.id;
  var accum : real = 0;
  for k in row.columnOffset {
    accum += values[k]
  }
  b[id] = accum;
}
```

Distributed Sparse Matrix-Vector Multiply (SpMV)

NAS-CG (Conjugate Gradient)
Problem Size D (73 million non-zeros)



How can we get better performance for these types of codes in Chapel?

High Productivity does not always lead to High Performance

Motivation: Why target irregular memory accesses?

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

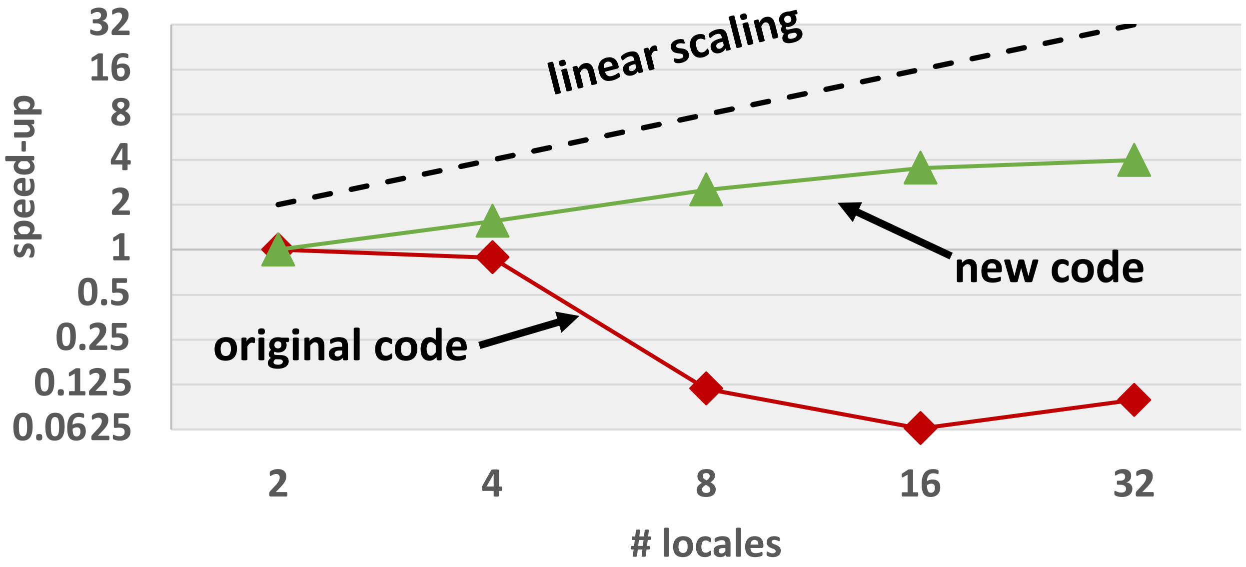
```
if doOptimization {
  if doInspector(x) || doInspector(col_idx) {
    inspectorPreamble(x);
    forall loc in Locales do on loc {
      ref replD = x.replArray.replD;
      ref arrDom = x.replArray.arrDom;
      const ref indices = Rows.localSubdomain();
      for i in indices {
        ref row = Rows[i];
        const id = row.id;
        var accum : real = 0;
        for k in row.columnOffset {
          inspectAccess(replD, arrDom, col_idx[k]);
        }
      }
    }
    sortReplIndices(x);
    inspectorOff(x);
    inspectorOff(col_idx);
  }
  executorPreamble(x);
  forall row in Rows with (ref arrDom = x.replArray.arrDom,
                           ref replArr = x.replArray.replArr) {
    const id = row.id;
    var accum : real = 0;
    for k in row.columnOffset {
      accum += values[k] * executeAccess(arrDom, replArr, x, col_idx[k]);
    }
    b[id] = accum;
  }
}
else {
  forall row in Rows {
    const id = row.id;
    var accum : real = 0;
    for k in row.columnOffset {
      accum += values[k] * x[col_idx[k]];
    }
    b[id] = accum;
  }
}
```

Just a simple transformation...

Motivation: Why target irregular memory accesses?

```
if doOptimization {
  if doInspector(x) || doInspector(col_idx) {
    inspectorPreamble(x);
    forall loc in Locales do on loc {
      ref replD = x.replArray.replD;
      ref arrDom = x.replArray.arrDom;
      const ref indices = Rows.localSubdomain();
      for i in indices {
        ref row = Rows[i];
        const id = row.id;
        var accum : real = 0;
        for k in row.columnOffset {
          inspectAccess(replD, arrDom, col_idx[k]);
        }
      }
    }
    sortReplIndices(x);
    inspectorOff(x);
    inspectorOff(col_idx);
  }
  executorPreamble(x);
  forall row in Rows with (ref arrDom = x.replArray.arrDom,
                           ref replArr = x.replArray.replArr) {
    const id = row.id;
    var accum : real = 0;
    for k in row.columnOffset {
      accum += values[k] * executeAccess(arrDom, replArr, x, col_idx[k]);
    }
    b[id] = accum;
  }
}
else {
  forall row in Rows {
    const id = row.id;
    var accum : real = 0;
    for k in row.columnOffset {
      accum += values[k] * x[col_idx[k]];
    }
    b[id] = accum;
  }
}
```

NAS-CG (Conjugate Gradient)
Problem Size D (73 million non-zeros)

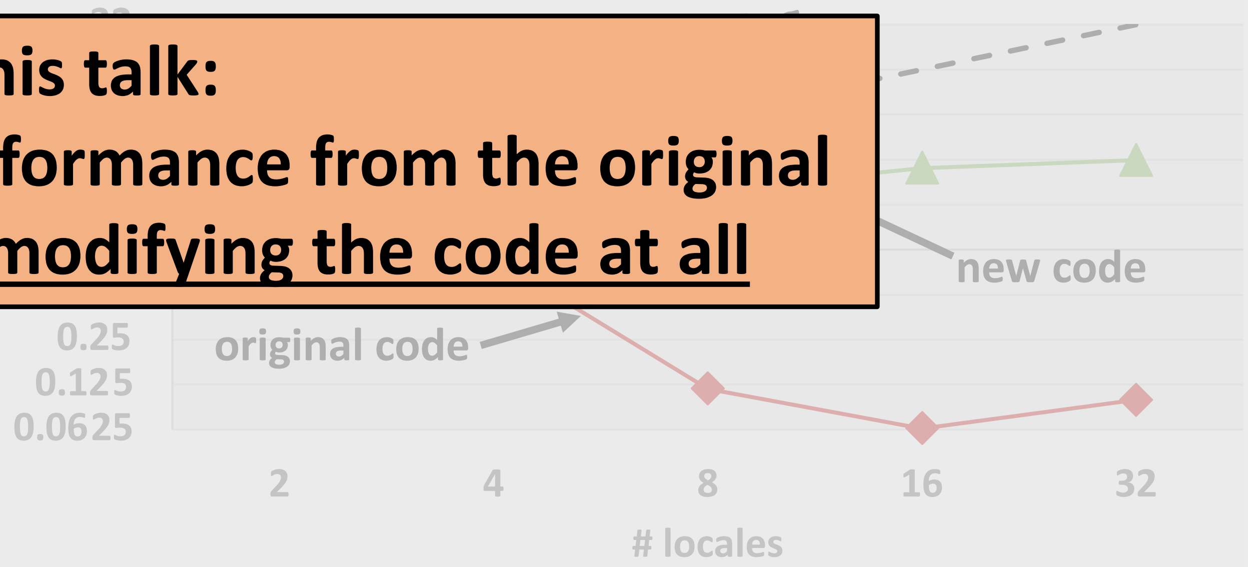


Manual optimizations can drastically improve performance

Motivation: Why target irregular memory accesses?

```
if doOptimization {
  if doInspector(x) || doInspector(col_idx) {
    inspectorPreamble(x);
    forall loc in Locales do on loc {
      ref replD = x.replArray.replD;
      ref arrDom = x.replArray.arrDom;
      const ref indices = Rows.localSubdomain();
      for i in indices {
        ref row = Rows[i];
        const id = row.id;
        var accum : real = 0;
        for k in row.columnOffset
          inspectAccess(replD, arr
        }
      }
    }
    sortReplIndices(x);
    inspectorOff(x);
    inspectorOff(col_idx);
  }
  executorPreamble(x);
  forall row in Rows with (ref arrDom = x.replArray.arrDom,
                           ref replArr = x.replArray.replArr) {
    const id = row.id;
    var accum : real = 0;
    for k in row.columnOffset {
      accum += values[k] * executeAccess(arrDom, replArr, x, col_idx[k]);
    }
    b[id] = accum;
  }
}
else {
  forall row in Rows {
    const id = row.id;
    var accum : real = 0;
    for k in row.columnOffset {
      accum += values[k] * x[col_idx[k]];
    }
    b[id] = accum;
  }
}
```

NAS-CG (Conjugate Gradient)
Problem Size D (73 million non-zeros)



In this talk:
We can achieve this performance from the original "nice" code <u>without modifying the code at all</u>

Manual optimizations can drastically improve performance

Outline

- Optimization: selective data replication
- Implementation within compiler:
  - Code transformations
  - Static analysis
- Performance evaluation:
  - NAS-CG
  - PageRank

Selective Data Replication

- We focus on accesses of the form **A[B[i]]** in **forall** loops
  - **A** is a distributed array
  - the values in **B** are not known until runtime

Selective Data Replication

- We focus on accesses of the form **A[B[i]]** in **forall** loops
  - **A** is a distributed array
  - the values in **B** are not known until runtime
- **Goal**: replicate remotely accessed elements of **A** so they can be used locally in the **forall**
  - **inspector:** runtime analysis that determines remote accesses
  - **executor:** optimized version of the **forall** that redirects remote accesses to the replicated copies
  - both generated by the **compiler** without user intervention

Selective Data Replication

- We focus on accesses of the form **A[B[i]]** in **forall** loops
  - **A** is a distributed array
  - the values in **B** are not known until runtime
- **Goal**: replicate remotely accessed elements of **A** so they can be used locally in the **forall**
  - **inspector:** runtime analysis that determines remote accesses
  - **executor:** optimized version of the **forall** that redirects remote accesses to the replicated copies
  - both generated by the **compiler** without user intervention
- Requirements:
  - the **forall** executes **many times** with the **same access pattern**
  - **A[B[i]]** is on the RHS of an operation (i.e., read-only)

Selective Data Replication

- We focus on accesses of the form **A[B[i]]** in **forall** loops
  - **A** is a distributed array
  - the values in
- **Goal**: replicate … be used locally in the **f**
  - **inspector:** ru
  - **executor:** op … sses to the replicated co
  - both generat
- Requirements
  - the **forall** exe
  - **A[B[i]]** is on the RHS of an operation (i.e., read-only)

We manually implemented this optimization in prior work:

- https://chapel-lang.org/CHIUW/2021/Rolinger.pdf

The optimization is influenced by other prior works:

- "*Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures*", **Das et al.**
- "*Automatic Support for Irregular Computations in a High-Level Language*", **Su and Yelick**
- " *Improving Communication in PGAS Environments: Static and Dynamic Coalescing in UPC*", **Alvanos et al.**

Code Transformations

```
forall i in B.domain {
  C[i] += A[B[i]];
}
```

```
if doOptimization {
  if doInspector(A, B) {
    inspectorPreamble(A);
    forall i in inspectorIter(B.domain) {
      inspectAccess(A, B[i]);
    }
    inspectorOff(A, B);
  }
  executorPreamble(A);
  forall i in B.domain {
    C[i] += executeAccess(A, B[i]);
  }
}
else {
  forall i in B.domain {
    C[i] += A[B[i]];
  }
}
```

Compiler Optimization: Code Transformations

Inspector: performs memory access analysis

Key points:

- inspector should only be performed (1) the **first time** we encounter the loop and (2) anytime the access pattern **A[B[i]]** could have **changed**
- **inspectAccess**() does not issue the remote access to **A**; it just “queries” whether index **B[i]** will be a remote access to **A**

```
if doOptimization {
  if doInspector(A, B) {
    inspectorPreamble(A);
    forall i in inspectorIter(B.domain) {
      inspectAccess(A, B[i]);
    }
    inspectorOff(A, B);
  }
  executorPreamble(A);
  forall i in B.domain {
    C[i] += executeAccess(A, B[i]);
  }
}
else {
  forall i in B.domain {
    C[i] += A[B[i]];
  }
}
```

Compiler Optimization: Code Transformations

Executor: executes the loop but redirects remote accesses to the replicated copies

Key points:

- **executorPreamble()** initializes replicated elements of **A** with values from original array
  - we only replicate an element once, regardless of how many times it is accessed → amortizes the cost of the remote read over multiple local accesses
- **executeAccess()** checks if index **B[i]** will be a remote access to **A**, and if so, returns the local copy.

```
if doOptimization {
  if doInspector(A, B) {
    inspectorPreamble(A);
    forall i in inspectorIter(B.domain) {
      inspectAccess(A, B[i]);
    }
    inspectorOff(A, B);
  }
  executorPreamble(A);
  forall i in B.domain {
    C[i] += executeAccess(A, B[i]);
  }
}
else {
  forall i in B.domain {
    C[i] += A[B[i]];
  }
}
```

Compiler Optimization: Code Transformations

May find later in Chapel's compilation process that the optimization **cannot be applied**

In this case, **doOptimization** is set to false and dead-code elimination will "**undo**" our transformations and fall back to the original **forall** loop.

```
if doOptimization {
  if doInspector(A, B) {
    inspectorPreamble(A);
    forall i in inspectorIter(B.domain) {
      inspectAccess(A, B[i]);
    }
    inspectorOff(A, B);
  }
  executorPreamble(A);
  forall i in B.domain {
    C[i] += executeAccess(A, B[i]);
  }
}
else {
  forall i in B.domain {
    C[i] += A[B[i]];
  }
}
```

Static Analysis

- Everything just discussed for code transformations only holds if the optimization **CAN** and **SHOULD** be applied
- **We need to maintain correct program results**
  - detect when **A[B[i]]** access pattern changes so we can re-run the inspector
- **We should improve program performance**
  - ensure that the **forall** is nested in an outer loop, so it is likely to be executed multiple times → amortizes the cost of the inspector over multiple iterations
  - also need to ensure that the inspector will not have to run EVERY time we execute the **forall**

Static Analysis

- Everything just mentioned for code transformations only holds if the optimization **CAN** and **SHOULD** be applied

Static analysis and code transformations are performed **automatically**

The user **does not** add pragmas, code annotations, hints or anything to their code. They turn on a **compiler flag**

multiple times → amortizes the inspector over multiple iterations

- also need to ensure that the inspector will not have to run EVERY time we execute the **forall**

Performance Evaluation

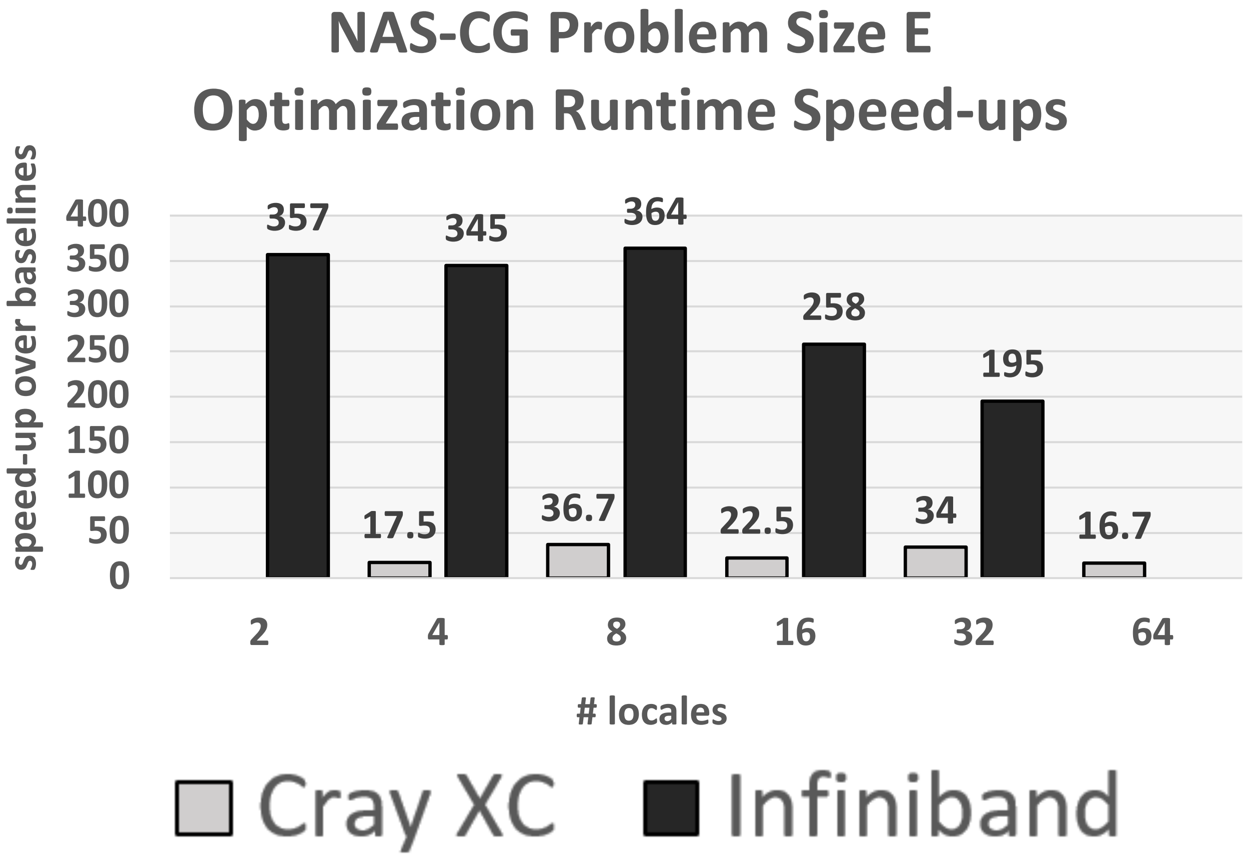
- Applications:
  - **NAS-CG** (conjugate gradient)
  - **PageRank** (iterative SpMV-like operations)
- Systems:
  - **FDR Infiniband**, 20 cores per node, 512 GB of memory per node
  - **Cray XC,** Aries interconnect, 44 cores per node, 128 GB of memory per node
- Experiments:
  - measured **runtime speed-ups** achieved by optimization relative to the original code
  - includes any overhead incurred by the inspector

NAS-CG Data sets

Name	Rows	Non-zeros	Density (%)	# of SpMVs
C	150k	39M	0.17	1950
D	150k	73M	0.32	2600
E	9M	6.6B	0.008	2600
F	54M	55B	0.002	2600

NAS-CG Optimization Speed-ups

	Cray XC				Infiniband			
Locales	C	D	E	F	C	D	E	F
2	3.2	2.8	—	—	8.9	6	357	—
4	3.6	3.4	17.5	—	15.8	10.4	345	—
8	5.7	6.2	36.7	—	115	127	364	—
16	8.6	11	22.5	—	238	330	258	270
32	6.4	8.4	34	52.3	160	240	195	165
64	4.1	4.9	16.7	25.4	NA	NA	NA	NA
geomean	5	5.5	24.1	36.4	57.3	57.5	296	211



Take-aways:

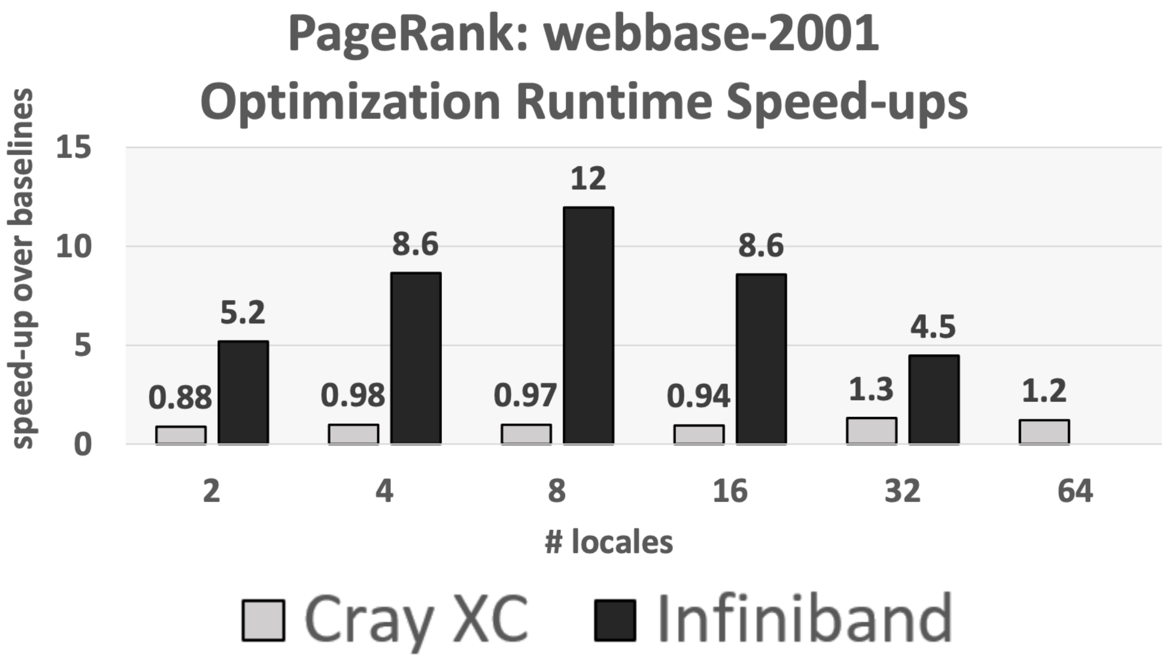
- “—” means not enough memory, “NA” means not enough nodes
- **lots of data reuse** in the kernel, so the optimization performs very well
- **inspector overhead is small** due to many iterations w/o the access pattern changing
- optimization provides **larger gains on Infiniband**
  - higher latency for small messages than Aries

PageRank Data sets

Name	|V|	|E|	Density (%)	Iterations
webbase-2001	118M	992M	7.1e − 6	33
sk-2005	51M	1.9B	7.5e − 5	40

PageRank Optimization Speed-ups

Locales	Cray XC webbase-2001	Cray XC sk-2005	Infiniband webbase-2001	Infiniband sk-2005
2	0.88	1.2	5.2	2
4	0.98	1.6	8.6	7.1
8	0.97	1.3	12	6
16	0.94	1.7	9.6	5.4
32	1.3	1.4	4.5	4.2
64	1.2	2.1	NA	NA
geomean	1.04	1.5	7.3	4.5



Take-aways:

- **smaller speed-ups** overall due to **fewer iterations** than NAS-CG and **less data reuse**
  - because of both the algorithm and the graphs
- speed-ups on the Cray can be negative when the data reuse is low (webbase-2001)
- nevertheless, still significant speed-ups overall

Conclusions

- Optimization is producing promising results: runtimes improved from **hours/days to minutes**
- **Limitations** exists for this type of data replication
  - **forall** must execute multiple times without the memory access pattern changing
  - could use a **lot of memory** for the replication
  - currently limited to **read-only** data
- **Not covered in this talk**:
  - handling **foralls** in procedures with multiple call sites
  - special handling for arrays/domains that are **fields in a record**
  - **inter-procedural analysis** to detect modifications to arrays/domains across calls
  - **alias analysis** to detect modifications to arrays/domains

Future Work

- Use this framework to implement other optimizations
  - **prefetching** for Chapel's remote cache
  - more generalized **aggregation** for remote writes than CopyAggregation
  - end goal is **a single framework** that can apply all these optimizations **automatically**, deciding which one to apply considering the specific scenario
- **Acknowledgements**:
  - Chapel team: extremely helpful and responsive to questions, and facilitated access to the Cray system
  - Specific shout outs to **Engin K., Vass L., Elliot R., Brad C., Michelle S.**

Bonus: BFS Results

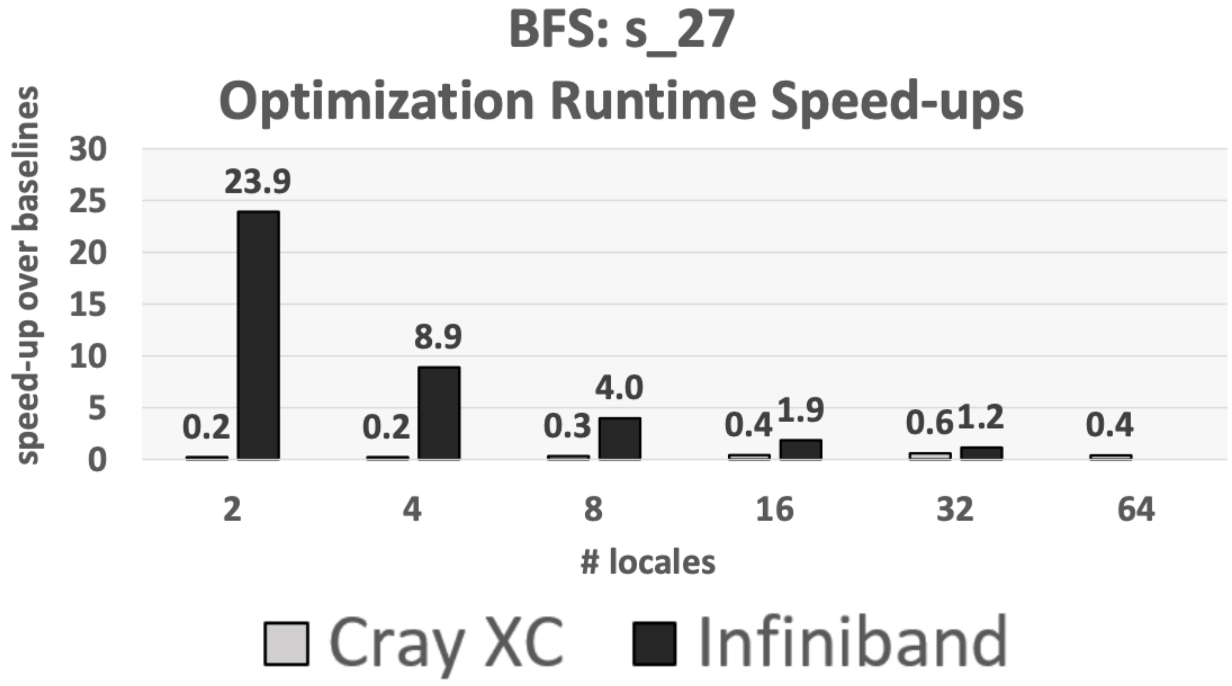
- Implemented Breadth First Search (BFS) as a series of SpMV-like operations
- Not necessarily the best approach for raw performance but provides an interesting experiment
  - relatively few iterations are performed, so the cost of the inspector becomes more prominent

BFS Data sets

Name	|V|	|E|	Density (%)	Iterations
s_25	33M	1B	9.5e − 5	6
s_26	67M	2.1B	4.8e − 5	7
s_27	134M	4.2B	2.3e − 5	7

BFS Optimization Speed-ups

Locales	Cray XC			Infiniband		
	s_25	s_26	s_27	s_25	s_26	s_27
2	0.23	0.26	0.23	30.3	27.6	23.9
4	0.22	0.27	0.24	11.7	13.6	8.9
8	0.3	0.29	0.32	6.5	7.1	4
16	0.28	0.32	0.42	3.9	2.8	1.9
32	0.39	0.42	0.6	2.6	1.2	1.2
64	0.32	0.31	0.4	NA	NA	NA
geomean	0.28	0.31	0.35	7.5	6.2	4.5



Take-aways:

- **poor performance on the Cray**
  - not enough iterations to amortize the cost of the inspector
- **positive gains on the Infiniband** system despite the few number of iterations
  - performance increases dwindle as we increase the number of locales (inspector runtime does not scale as well as the runtime of each iteration)