Compiler Optimization for Irregular Memory Access Patterns in PGAS Programs

Thomas B. Rolinger (UMD/LPS)

Christopher D. Krieger (LPS)

Alan Sussman (UMD)

Contact: tbrolin@cs.umd.edu

LCPC 2022

COMPUTER SCIENCE
UNIVERSITY OF MARYLAND

Motivation

- Why target irregular memory accesses?
- Why target Partitioned Global Address Space (PGAS) programs?
- Why are compiler optimizations needed?

Motivation: Irregular memory accesses

Irregular memory access patterns are prevalent in:

- graph analytics
- sparse linear algebra
- PDE solvers
- some machine learning applications

In these applications, performance is often **memory-bound**

- irregular memory accesses are **performance bottlenecks**

Motivation: Irregular memory accesses

Irregular memory access patterns are prevalent in:

- graph analytics
- sparse linear algebra
- PDE solvers
- some machine learning applications

data is large, sparse and unstructured

In these applications, performance is often **memory-bound**

- irregular memory accesses are **performance bottlenecks**

Motivation: Irregular memory accesses

Irregular memory access patterns are prevalent in:

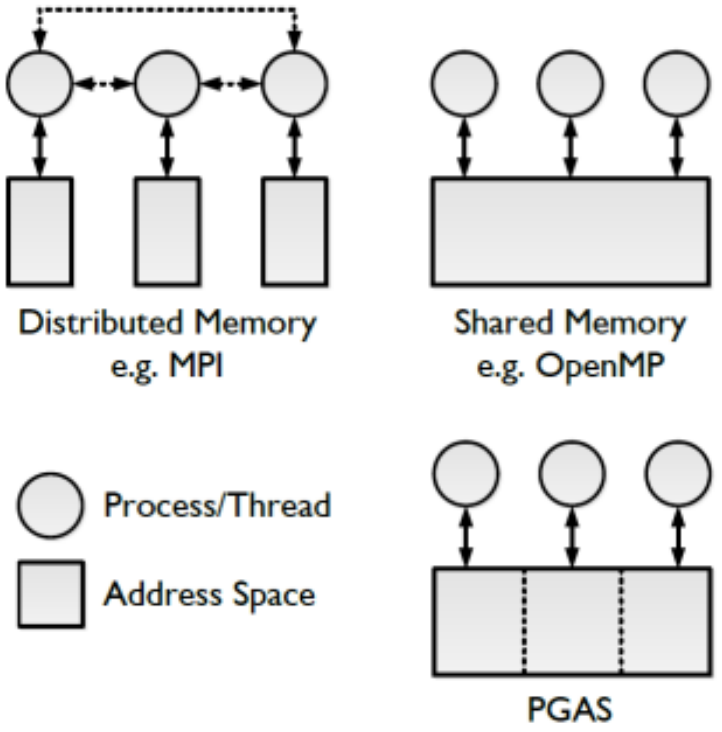
- graph analytics
- sparse linear algebra
- PDE solvers
- some machine learning applications

In these applications, performance is often **memory-bound**

- irregular memory accesses are **performance bottlenecks**

Partitioned Global Address Space (PGAS)

- Presents a view of a distributed memory system that resembles a **single shared address space**
- Each node "owns" a partition of the address space, so there is still the concept of **local vs remote memory**
- Provides **one-sided communication** (puts/gets), allowing for languages/libraries that implement the PGAS model to hide data distribution/communication details

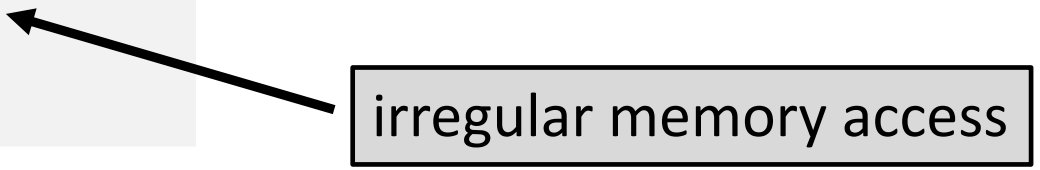


Sparse Matrix Vector Multiply (SpMV)
OpenMP shared-memory

```
1 #pragma omp parallel for
2 for(int i = 0; i < numrows; i++) {
3   double accum = 0.0;
4   for(int k = rowPtr[i]; k < rowPtr[i+1]; k++) {
5     accum += values[k] * x[col_idx[k]];
6   }
7   b[i] = accum;
8 }
```

Sparse Matrix Vector Multiply (SpMV)
OpenMP shared-memory

```
1 #pragma omp parallel for
2 for(int i = 0; i < numrows; i++) {
3   double accum = 0.0;
4   for(int k = rowPtr[i]; k < rowPtr[i+1]; k++) {
5     accum += values[k] * x[col_idx[k]];
6   }
7   b[i] = accum;
8 }
```



Sparse Matrix Vector Multiply (SpMV) OpenMP shared-memory

```
#pragma omp parallel for
for(int i = 0; i < numrows; i++) {
  double accum = 0.0;
  for(int k = rowPtr[i]; k < rowPtr[i+1]; k++) {
    accum += values[k] * x[col_idx[k]];
  }
  b[i] = accum;
}
```

Sparse Matrix Vector Multiply (SpMV) MPI+OpenMP distributed- and shared-memory

```
int *recvCounts = (int*)malloc(sizeof(int)*numRanks);
int *displs = (int*)malloc(sizeof(int)*numRanks);
MPI_Allgather(&localNumRows, 1, MPI_INT, recvCounts, 1, MPI_INT, MPI_COMM_WORLD);
recv_counts[rank] = localNumRows;
int currOffset = 0;
for (int i = 0; i < numRanks; i++) {
  displs[i] = currOffset;
  currOffset += recvCounts[i];
}

MPI_Allgatherv(locX, localNumRows, MPI_DOUBLE, x, recvCounts, displs, MPI_DOUBLE, MPI_COMM_WORLD);
#pragma omp parallel for
for (int i = 0; i < localNumRows; i++) {
  double accum = 0.0;
  for (int k = rowPtr[i]; k < rowPtr[i+1]; k++) {
    accum += values[k] * x[col_idx[k]];
  }
  locB[i] = accum;
}
```

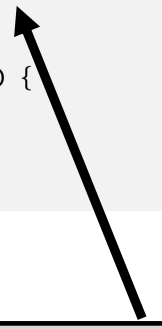
Sparse Matrix Vector Multiply (SpMV) OpenMP shared-memory

```
#pragma omp parallel for
for(int i = 0; i < numrows; i++) {
  double accum = 0.0;
  for(int k = rowPtr[i]; k < rowPtr[i+1]; k++) {
    accum += values[k] * x[col_idx[k]];
  }
  b[i] = accum;
}
```

Sparse Matrix Vector Multiply (SpMV) MPI+OpenMP distributed- and shared-memory

```
int *recvCounts = (int*)malloc(sizeof(int)*numRanks);
int *displs = (int*)malloc(sizeof(int)*numRanks);
MPI_Allgather(&localNumRows, 1, MPI_INT, recvCounts, 1, MPI_INT, MPI_COMM_WORLD);
recv_counts[rank] = localNumRows;
int currOffset = 0;
for (int i = 0; i < numRanks; i++) {
  displs[i] = currOffset;
  currOffset += recvCounts[i];
}

MPI_Allgatherv(locX, localNumRows, MPI_DOUBLE, x, recvCounts, displs, MPI_DOUBLE, MPI_COMM_WORLD);
#pragma omp parallel for
for (int i = 0; i < localNumRows; i++) {
  double accum = 0.0;
  for (int k = rowPtr[i]; k < rowPtr[i+1]; k++) {
    accum += values[k] * x[col_idx[k]];
  }
  locB[i] = accum;
}
```



MPI_Allgatherv → full replication of **x** on every process, since we don't know which remote values we need

Sparse Matrix Vector Multiply (SpMV)
OpenMP shared-memory

```
#pragma omp parallel for
for(int i = 0; i < numrows; i++) {
  double accum = 0.0;
  for(int k = rowPtr[i]; k < rowPtr[i+1]; k++) {
    accum += values[k] * x[col_idx[k]];
  }
  b[i] = accum;
}
```

Sparse Matrix Vector Multiply (SpMV)
MPI+OpenMP distributed- and shared-memory

```
int *recvCounts = (int*)malloc(sizeof(int)*numRanks);
int *displs = (int*)malloc(sizeof(int)*numRanks);
MPI_Allgather(&localNumRows, 1, MPI_INT, recvCounts, 1, MPI_INT, MPI_COMM_WORLD);
recv_counts[rank] = localNumRows;
int currOffset = 0;
for (int i = 0; i < numRanks; i++) {
  displs[i] = currOffset;
  currOffset += recvCounts[i];
}

MPI_Allgatherv(locX, localNumRows, MPI_DOUBLE, x, recvCounts, displs, MPI_DOUBLE, MPI_COMM_WORLD);
#pragma omp parallel for
for (int i = 0; i < localNumRows; i++) {
  double accum = 0.0;
  for (int k = rowPtr[i]; k < rowPtr[i+1]; k++) {
    accum += values[k] * x[col_idx[k]];
  }
  locB[i] = accum;
}
```

Sparse Matrix Vector Multiply (SpMV)
Chapel + Partitioned Global Address Space (PGAS)

```
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;
}
```

Sparse Matrix Vector Multiply (SpMV)
OpenMP shared-memory

```
#pragma omp parallel for
for(int i = 0; i < numrows; i++) {
  double accum = 0.0;
  for(int k = rowPtr[i]; k < rowPtr[i+1]; k++) {
    accum += values[k] * x[col_idx[k]];
  }
  b[i] = accum;
}
```

Sparse Matrix Vector Multiply (SpMV)
MPI+OpenMP distributed- and shared-memory

```
int *recvCounts = (int*)malloc(sizeof(int)*numRanks);
int *displs = (int*)malloc(sizeof(int)*numRanks);
MPI_Allgather(&localNumRows, 1, MPI_INT, recvCounts, 1, MPI_INT, MPI_COMM_WORLD);
recv_counts[rank] = localNumRows;
int currOffset = 0;
for (int i = 0; i < numRanks; i++) {
  displs[i] = currOffset;
  currOffset += recvCounts[i];
}

MPI_Allgatherv(locX, localNumRows, MPI_DOUBLE, x, recvCounts, displs, MPI_DOUBLE, MPI_COMM_WORLD);
#pragma omp parallel for
for (int i = 0; i < localNumRows; i++) {
  double accum = 0.0;
  for (int k = rowPtr[i]; k < rowPtr[i+1]; k++) {
    accum += values[k] * x[col_idx[k]];
  }
  locB[i] = accum;
}
```

looks like shared-memory code

Sparse Matrix Vector Multiply (SpMV)
Chapel + Partitioned Global Address Space (PGAS)

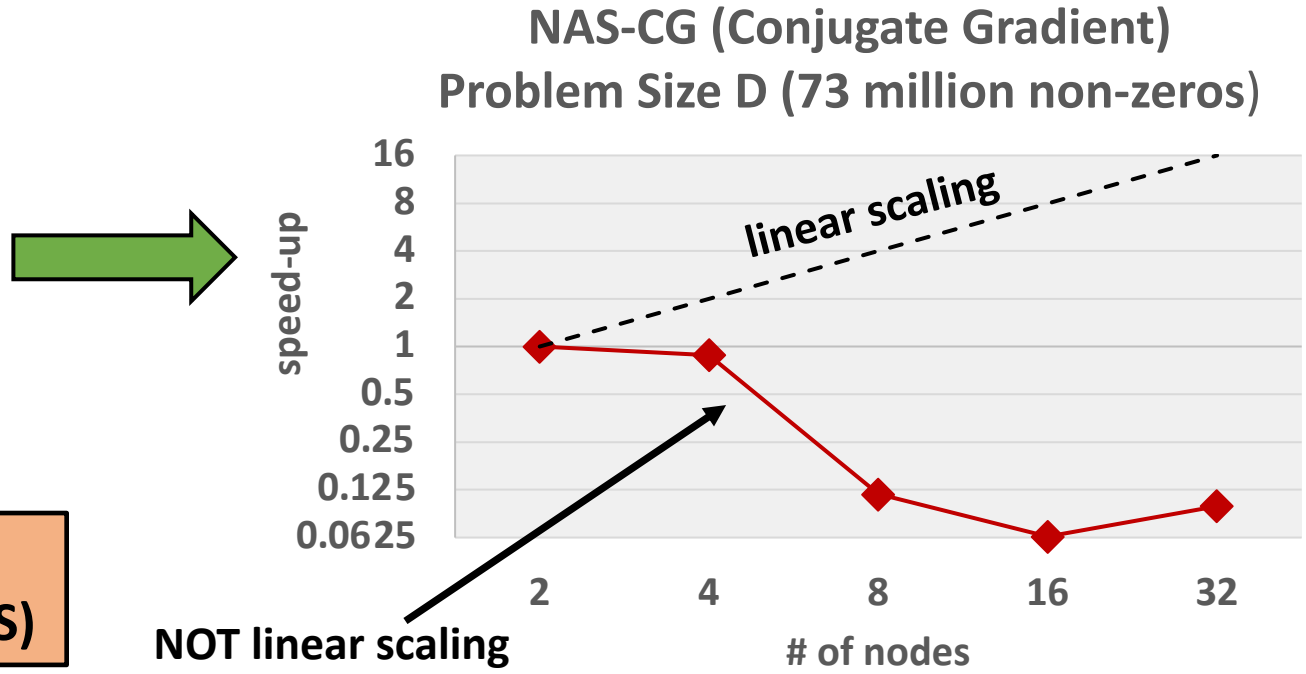
```
forall row in Rows {
  const id = row.id;
  var accum : real = 0;
  for k in row.columnOffset {
    accum += values[k] * x[col_idx[k]];
  }
  b[id] = accum;
}
```

executes across multiple nodes but no replication of x

Motivation: Irregular memory accesses in PGAS

```
1 forall row in Rows {
2   const id = row.id;
3   var accum : real = 0;
4   for k in row.columnOffset {
5     accum += values[k] * x[col_idx[k]];
6   }
7   b[id] = accum;
8 }
```

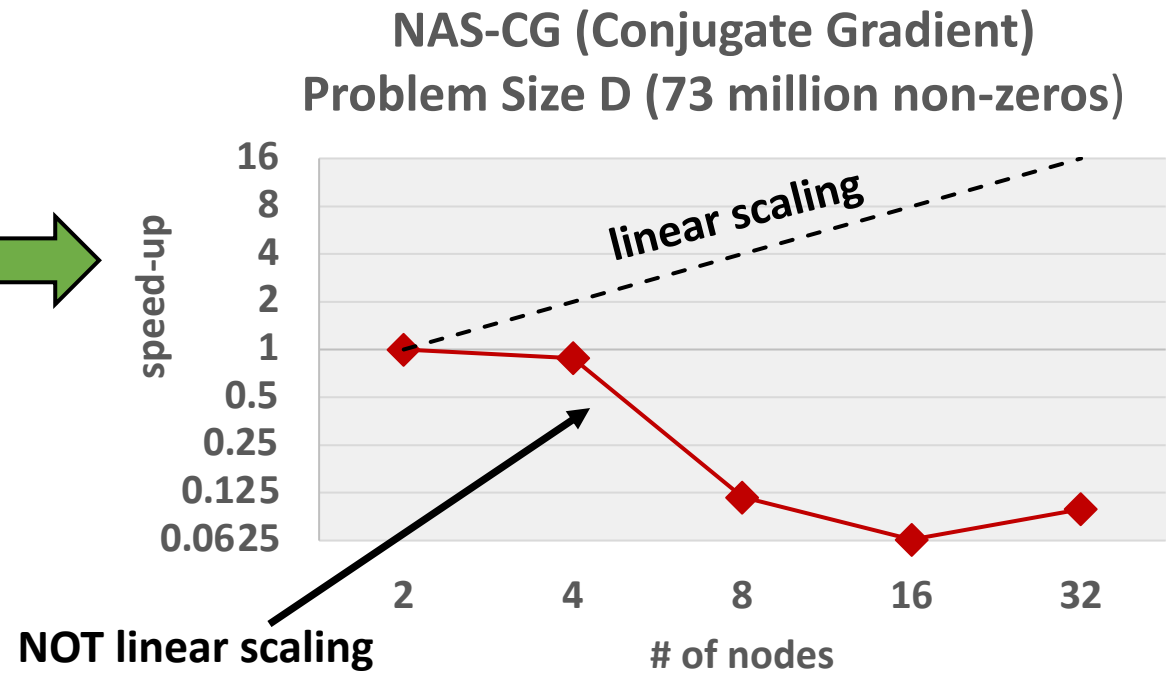
Sparse Matrix Vector Multiply (SpMV)
Chapel + Partitioned Global Address Space (PGAS)



Motivation: Irregular memory accesses in PGAS

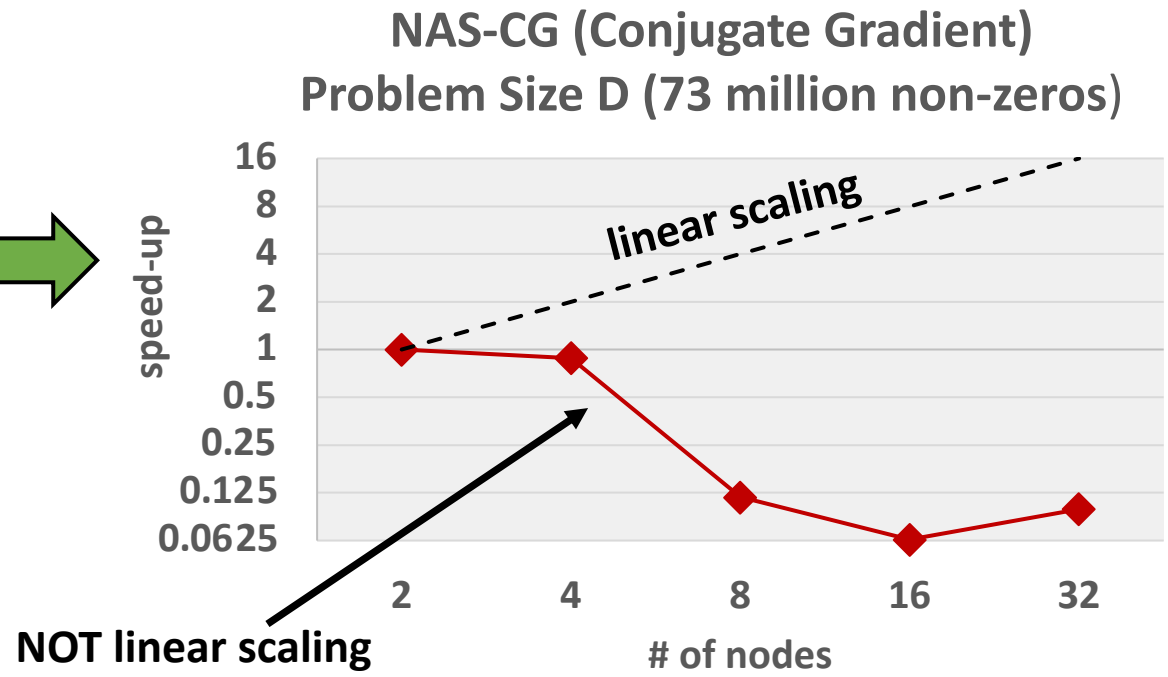
```
1 forall row in Rows {
2   const id = row.id;
3   var accum : real = 0;
4   for k in row.columnOffset {
5     accum += values[k] * x[col_idx[k]];
6   }
7   b[id] = accum;
8 }
```

(implicit) fine grain remote communication



Motivation: Irregular memory accesses in PGAS

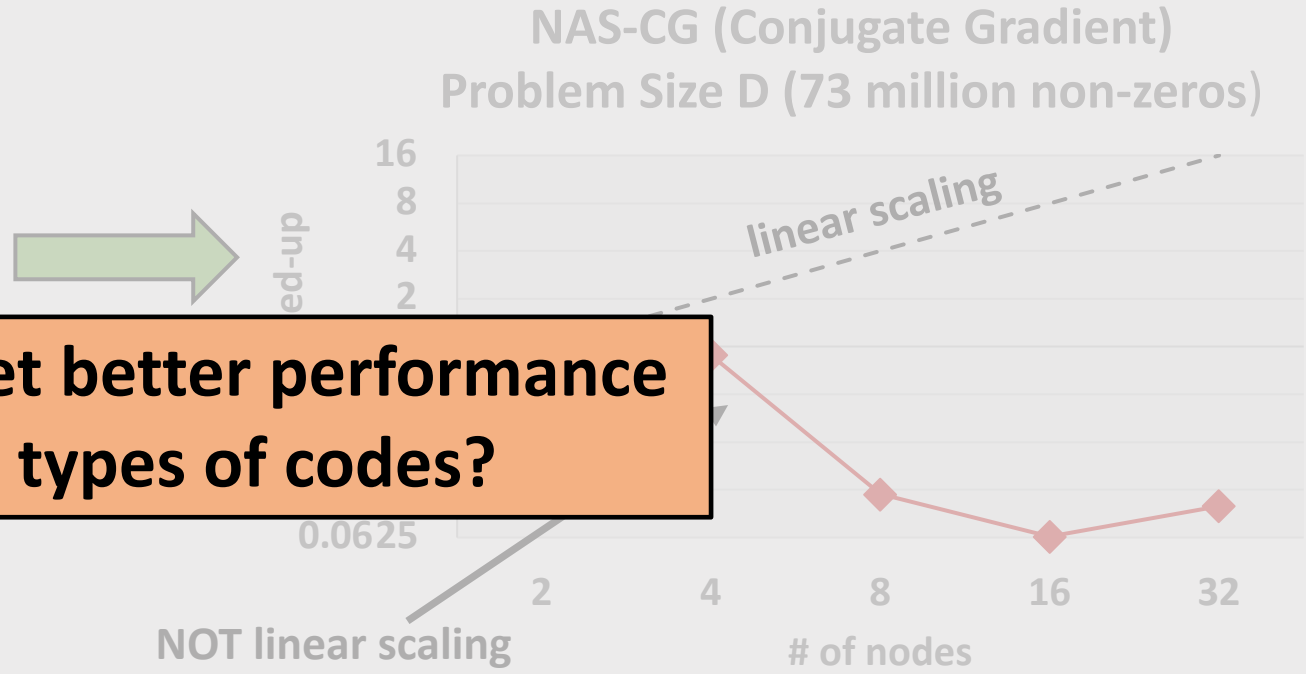
```
1 forall row in Rows {
2   const id = row.id;
3   var accum : real = 0;
4   for k in row.columnOffset {
5     accum += values[k] * x[col_idx[k]];
6   }
7   b[id] = accum;
8 }
```



High productivity does not always lead to high performance

Motivation: Irregular memory accesses in PGAS

```
1 forall row in Rows {
2   const id = row.id;
3   var accum : real = 0;
4   for k in row.columnOffset {
5     accum += values[k] * x[col_idx[k]];
6   }
7   b[id] = accum;
8 }
```



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

High productivity does not always lead to high performance

Motivation: Irregular memory accesses in PGAS

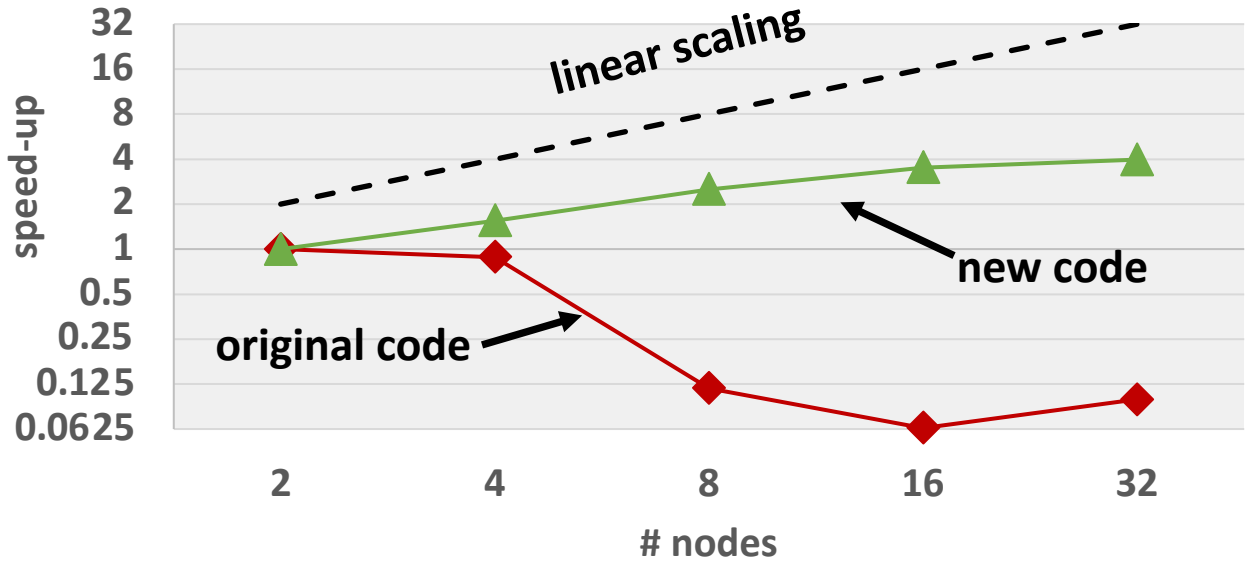
```
forall row in Rows {
  const id = row.id;
  var accum : real = 0;
  for k in row.columnOffset {
    accum += values[k] * x[col_idx[k]];
  }
  b[id] = accum;
}
```

Just a simple transformation…

```
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;
  }
}
```

Motivation: Irregular memory accesses in PGAS

```
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;
  }
}
```



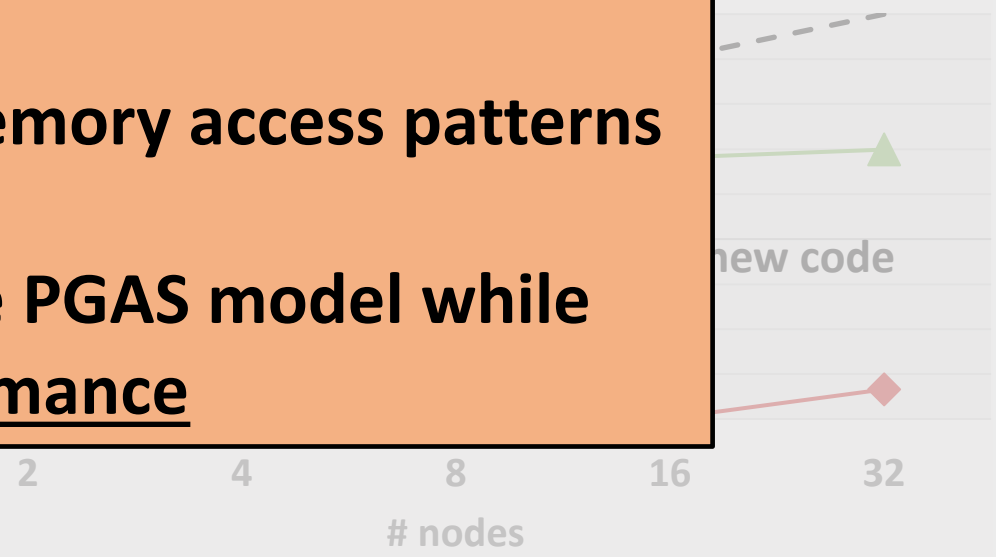
Manual optimizations can drastically improve performance

Motivation: Irregular memory accesses in PGAS

```
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
        const id
        var accu
        for k in
          inspec
        }
      }
    }
    sortReplIndi
    inspectorOf
    inspectorOf
  }
  executorPreamb
  forall row in

    const id = r
    var accum :
    for k in row
      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:
Automatic optimization of irregular memory access patterns

Preserve the high productivity of the PGAS model while achieving better performance

Manual optimizations can drastically improve performance

Outline

- Brief introduction Chapel
- Optimization: selective data replication
- Implementation within compiler:
  - Code transformations
  - Static analysis
- Performance evaluation:
  - NAS-CG
  - PageRank
- Future work and conclusions

Chapel

- Chapel Parallel Programming Language
  - high-level language that implements the PGAS model
  - **designed for productive parallel computing at scale**
- Terminology
  - **task:** set of computations that can be executed in parallel
  - **locale**: machine resources on which tasks execute (i.e., a node in a cluster)

Chapel

- Chapel Parallel Programming Language
  - high-level language that implements the PGAS model
  - **designed for productive parallel computing at scale**
- Terminology
  - **task:** set of computations that can be executed in parallel
  - **locale**: machine resources on which tasks execute (i.e., a node in a cluster)

```
1 var D : domain(1) = {0..5};
2 var data : [D] int;
3 data[0] = 1;
```

  - domain **D** with indices 0 through 5
  - array **data** defined over **D**

Chapel

- Chapel Parallel Programming Language
  - high-level language that implements the PGAS model
  - **designed for productive parallel computing at scale**
- Terminology
  - **task:** set of computations that can be executed in parallel
  - **locale**: machine resources on which tasks execute (i.e., a node in a cluster)

```
1 var D = newBlockDom({0..#16});
2 var A : [D] int;
3 // or
4 var A = newBlockArr({0..#16}, int);
```

  - **block distributed** domain **D**
  - access to **remote** array elements looks **no different** than accessing **local** elements

Chapel

- Chapel Parallel Programming Language
  - high-level language that implements the PGAS model
  - **designed for productive parallel computing at scale**
- Terminology
  - **task:** set of computations that can be executed in parallel
  - **locale**: machine resources on which tasks execute (i.e., a node in a cluster)

```
1 forall row in Rows {
2   const id = row.id;
3   var accum : real = 0;
4   for k in row.columnOffset {
5     accum += values[k] * x[col_idx[k]];
6   }
7   b[id] = accum;
8 }
```

- **forall** loop provides data parallelism
- **Rows** is a block distributed array
- task assigned the iteration for a given **row** will execute **on the locale** where **row** is
  - ***locale affinity*** defined implicitly by the loop iterand

Optimization: Selective Data Replication

- We focus on accesses of the form **A[B[i]]** in **forall** loops
  - **A** is a distributed array and the values in **B** are not known until runtime
  - more complex access patterns are supported (**see paper for details**)

Optimization: Selective Data Replication

- We focus on accesses of the form **A[B[i]]** in **forall** loops
  - **A** is a distributed array and the values in **B** are not known until runtime
  - more complex access patterns are supported (**see paper for details**)
- **Goal**: replicate remotely accessed elements of **A** so they can be accessed 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

Optimization: Selective Data Replication

- We focus on accesses of the form **A[B[i]]** in **forall** loops
  - **A** is a distributed array and the values in **B** are not known until runtime
  - more complex access patterns are supported (**see paper for details**)
- **Goal**: replicate remotely accessed elements of **A** so they can be accessed 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:
  - access-of-interest is **read-only** in the **forall** loop

Optimization: Selective Data Replication

- W
- G
  a
- Requirements:
  - access-of-interest is **read-only** in the **forall** loop

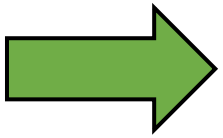
The **inspector-executor** technique has been around for a **long time**

We **leverage** the prior work on inspector-executors and **adapt** them for **Chapel+PGAS**

- we work within **Chapel's compiler**, before it lowers the code to LLVM
- requires different approaches to statically reason about **domains/arrays**, **forall loops** and **implicit communication**

Compiler: Code Transformations

```
1 forall i in B.domain {
2   C[i] += A[B[i]];
3 }
```



```
1  if doOptimization {
2    if doInspector(A, B) {
3      inspectorPreamble(A);
4      forall i in inspectorIter(B.domain) {
5        inspectAccess(A, B[i]);
6      }
7      inspectorOff(A, B);
8    }
9    executorPreamble(A);
10   forall i in B.domain {
11     C[i] += executeAccess(A, B[i]);
12   }
13 }
14 else {
15   forall i in B.domain {
16     C[i] += A[B[i]];
17   }
18 }
```

Compiler: Code Transformations (cont.)

Inspector: performs memory access analysis

- 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**

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

- **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

```
1  if doOptimization {
2    if doInspector(A, B) {
3      inspectorPreamble(A);
4      forall i in inspectorIter(B.domain) {
5        inspectAccess(A, B[i]);
6      }
7      inspectorOff(A, B);
8    }
9    executorPreamble(A);
10   forall i in B.domain {
11     C[i] += executeAccess(A, B[i]);
12   }
13 }
14 else {
15   forall i in B.domain {
16     C[i] += A[B[i]];
17   }
18 }
```

Compiler: Static Analysis

- Everything just discussed for code transformations only holds if the optimization **CAN** and **SHOULD** be applied:
  - optimization needs to **maintain correct program results**
  - optimization should **improve program performance**
- The goal of our static analysis is to achieve the above, all without requiring user intervention
  - no code annotations, hints, pragmas, etc.
  - just a **compiler flag** that is turned on when compiling the program

Compiler: Static Analysis (cont.)

- When could code transformations produce **incorrect behavior**?
  - if the **inspector is not executed** when needed, the replicated copies will be out of date
  - → need to be able to statically determine when the inspector should execute

Assume candidate access A[B[i]]

1. **forall** must iterate over a distributed array or domain
2. **i** from **A[B[i]]** must be *yielded* by the loop that contains **A[B[i]]** and that loop must iterate over a domain/array
3. **A**, **B,** and their **domains** cannot be modified in the **forall**

Compiler: Static Analysis (cont.)

- **(1)** ensures the optimization can reason about **which locales** the task execute on

```
1 forall i in B.domain {
2   C[i] += A[B[i]];
3 }
```

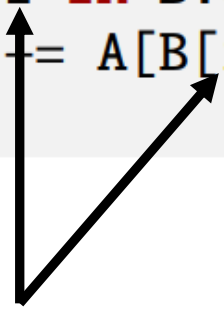

Controls locale affinity of the loop.
If B.domain changes, we know that the locale affinity could change.

Assume candidate access A[B[i]]

1. **forall** must iterate over a distributed array or domain
2. **i** from **A[B[i]]** must be *yielded* by the loop that contains **A[B[i]]** and that loop must iterate over a domain/array
3. **A**, **B,** and their **domains** cannot be modified in the **forall**

Compiler: Static Analysis (cont.)

```
1 forall i in B.domain {
2   C[i] += A[B[i]];
3 }
```



- **(3)** ensures the optimization can reason about the values of **i**, specifically **when they would change** (i.e., when the array/domain changes)

Assume candidate access A[B[i]]

1. **forall** must iterate over a distributed array or domain
2. **i** from **A[B[i]]** must be *yielded* by the loop that contains **A[B[i]]** and that loop must iterate over a domain/array
3. **A**, **B,** and their **domains** cannot be modified in the **forall**

Compiler: Static Analysis (cont.)

```
1 forall i in B.domain {
2   C[i] += A[B[i]];
3   B[i] += 1;
4 }
```

Assume candidate access A[B[i]]

1. **forall** must iterate over a distributed array or domain
2. **i** from **A[B[i]]** must be *yielded* by the loop that contains **A[B[i]]** and that loop must iterate over a domain/array
3. **A**, **B,** and their **domains** cannot be modified in the **forall**

- **(4)** ensures that the values in **B** analyzed by the inspector **will be the same** as those used in the executor and original loop

Compiler: Static Analysis (cont.)

Assume candidate access A[B[i]]

ributed array or

d by the loop that

o must iterate over a

not be modified in

```
1 forall
2   C[i]
3   B[i]
4 }
```

The optimization **statically identifies modifications** to the arrays/domains and inserts code to **turn on flags at runtime** to indicate that the inspector needs to be executed

- **(4)** ensures that the values in **B** analyzed by the inspector **will be the same** as those used in the executor and original loop

Compiler: Static Analysis (cont.)

- When could code transformations produce **poor performance**?
  - if the **inspector** is executed each time the **forall** is executed

Assume candidate access A[B[i]]

I. **forall** must be nested in an outer serial loop
II. **B** nor its **domain** can be modified within the outer loop that the **forall** is nested
III. **A's domain** cannot be modified within the outer loop that the **forall** is nested in

Compiler: Static Analysis (cont.)

- When could code transformations produce **poor performance**?
  - if the **inspector** is executed each time the **forall** is executed

Assume candidate access A[B[i]]

I. **forall** must be nested in an outer serial loop
II. **B** nor its **domain** can be modified within the outer loop that the **forall** is nested
III. **A's domain** cannot be modified within the outer loop that the **forall** is nested in

- **(I)** ensures that the **forall** is likely to be executed multiple times

```
1 for step in 0..#numSteps {
2   forall i in B.domain {
3     C[i] += A[B[i]];
4   }
5 }
```

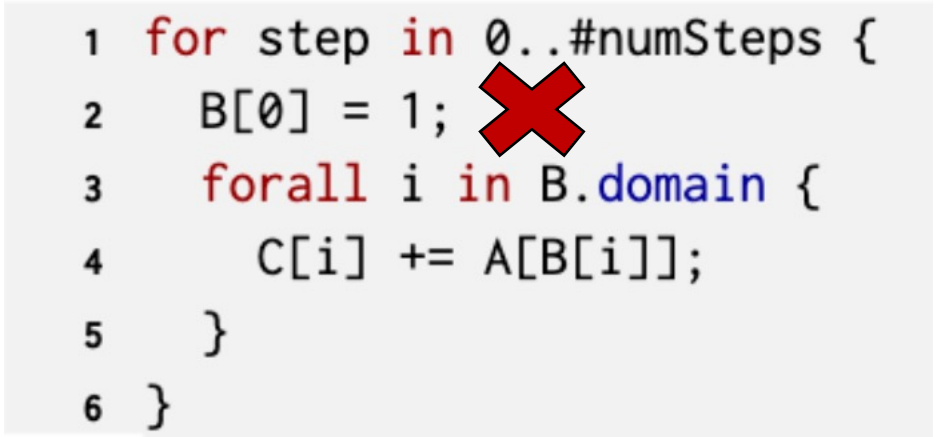
Compiler: Static Analysis (cont.)

- When could code transformations produce **poor performance**?
  - if the **inspector** is executed each time the **forall** is executed

Assume candidate access A[B[i]]

I. **forall** must be nested in an outer serial loop
II. **B** nor its **domain** can be modified within the outer loop that the **forall** is nested
III. **A's domain** cannot be modified within the outer loop that the **forall** is nested in

- **(II)** ensures the inspector will not execute each time the **forall** is performed (if **B** or its **domain** is modified, the **inspector must be rerun**)



Compiler: Static Analysis (cont.)

- When could code transformations produce **poor performance**?
  - if the **inspector** is executed each time the **forall** is executed

Assume candidate access A[B[i]]

I. **forall** must be nested in an outer serial loop
II. **B** nor its **domain** can be modified within the outer loop that the **forall** is nested
III. **A's domain** cannot be modified within the outer loop that the **forall** is nested in

- **(III)** is the same as (**II**) but for **A's domain**; changes to **A**'s values do not change the access pattern **A[B[i]]** but changes to the domain can change where elements of **A** are located

Compiler: Static Analysis (cont.)

- Additional analyses (see paper for details):
  - **Non-affine expression analysis** to identify candidate accesses beyond **A[B[i]]**
  - **Interprocedural analysis** to track modifications to arrays/domains across function calls
  - **Alias analysis** to track modifications to arrays/domains across to any aliases created
  - **Call path analysis** to identify invalid call paths to the function containing the **forall** and “turning off” the optimization at runtime for the invalid paths

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 Chapel 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

- **Each outer iteration performs 26 SpMVs**
- **Problem size C performs 75 outer iterations**
- **Other sizes perform 100 outer iterations**
- **No changes to A[B[i]] access pattern during CG**

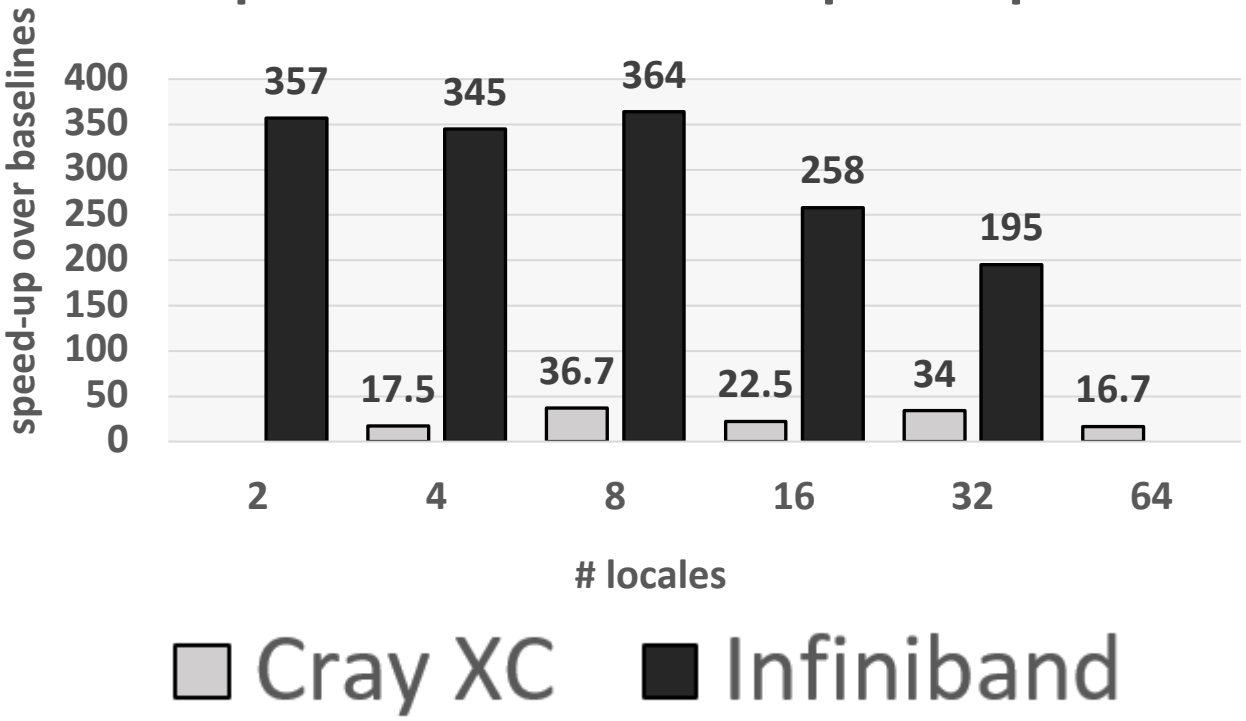
NAS-CG Data sets

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

Locales	Cray XC				Infiniband			
	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

NAS-CG Problem Size E Optimization Runtime Speed-ups



Take-aways:

- "—" means not enough memory, "NA" means not enough nodes
- **high degree 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

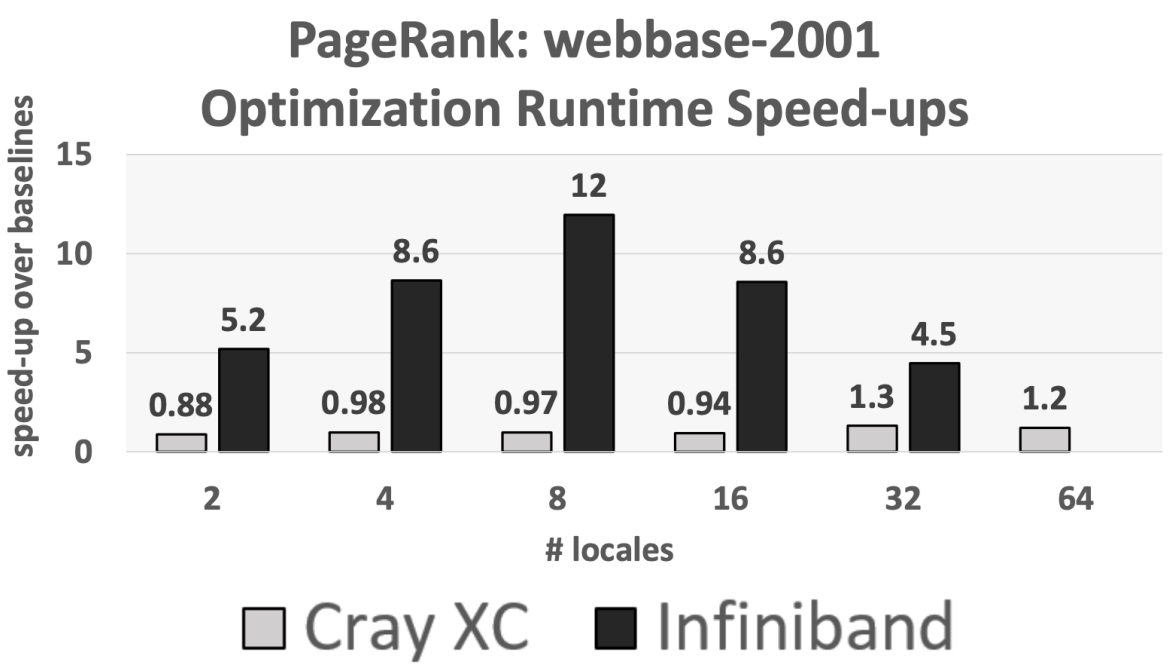
- **PageRank runs until a convergence threshold is met**
- **# of iterations depends on graph structure**

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		Infiniband	
	webbase-2001	sk-2005	webbase-2001	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



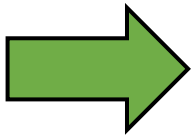
Take-aways:

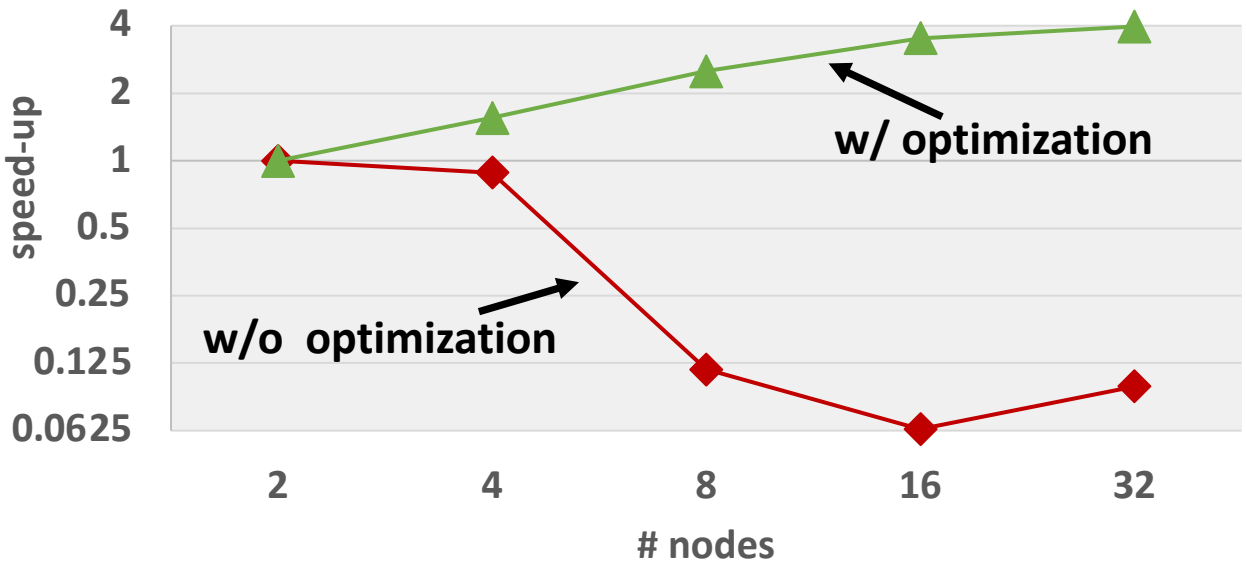
- **smaller speed-ups** overall due to **fewer iterations** than NAS-CG and **less data reuse**
  - because of both the algorithm and the graphs
- speed-ups on the Cray can be negative when the data reuse is low (webbase-2001)
- nevertheless, still significant speed-ups overall

Future Work

- **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
- Future work: **additional optimizations**
  - **adaptive prefetching** for Chapel's remote cache
  - **remote data aggregation** for remote writes
  - end goal is **a single framework** that can apply all these optimizations **automatically**, deciding which one to apply considering the specific scenario

```
1 forall row in Rows {
2   const id = row.id;
3   var accum : real = 0;
4   for k in row.columnOffset {
5     accum += values[k] * x[col_idx[k]];
6   }
7   b[id] = accum;
8 }
```





Optimization **preserves** original **high productivity** of code but achieves significantly **better performance**:

- runtimes reduced from **hours/days to minutes**

Acknowledgements: Chapel team for providing access to the Cray system and compiler support