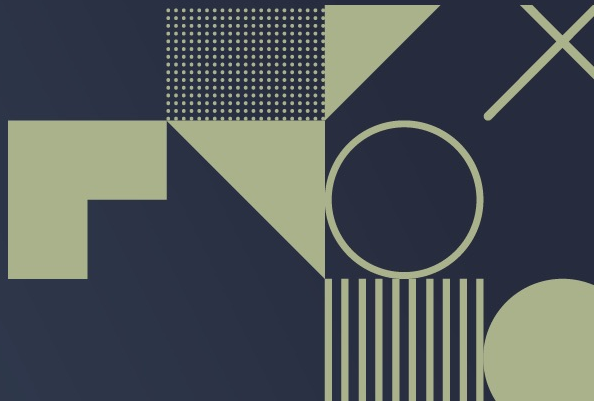
Towards High Productivity and Performance for Irregular Applications in Chapel

Thomas B. Rolinger *University of Maryland*
Joseph Craft *Laboratory for Physical Sciences*
Christopher D. Krieger *Laboratory for Physical Sciences*
Alan Sussman *University of Maryland*

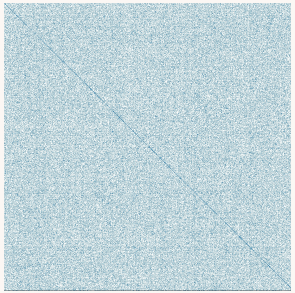


Agenda

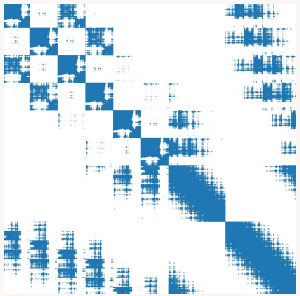
- Background
  - Irregular applications
  - Chapel programming language
  - Productivity-performance tradeoffs
- Application Studies
  - Breadth First Search (BFS) – message aggregation
  - PageRank – selective data replication
- Conclusions and Future Work

Background: Irregular Applications

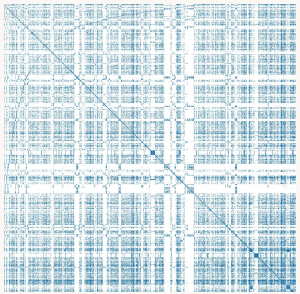
- Defining feature:
  - fine-grained, sparse memory access patterns (i.e., A[B[i]]) **not known until runtime**
- On distributed-memory systems, irregular accesses to distributed data leads to **fine-grained remote communication**
- Example applications:



Conjugate Gradient



Molecular Dynamics Simulation



Graph Analytics

Runtime Optimizations for Irregular Applications in Chapel, CHIUW 2021

Background: Chapel Programming Language

- Implements a Partitioned Global Address Space (**PGAS**)
- **High-level** constructs for both shared- and distributed-memory programming
- Executes tasks and distributes data across **locales**
  - Can think of a locale as a compute node in a cluster
- **Implicit** remote communication
  - accesses to remote elements of a distributed array look no different from local accesses

Background: Chapel Programming Language

- Implements a Partitioned Global Address Space (**PGAS**)
- **High-level** constructs for both shared- and distributed-memory programming
- Executes tasks and distributes data across **locales**
  - Can think of a locale as a compute node in a cluster
- **Implicit** remote communication
  - accesses to remote elements of a distributed array look no different from local accesses

```
forall vertex in Graph {
    for i in 0..#vertex.degree {
        Graph[vertex.neighbors[i]] = ...
    }
}
```

- **Graph** is a block distributed array of vertices
- **forall** loop iterates over vertices in **parallel**, creating tasks **on the locale** where the vertex is located
- **3 lines of code** that performs shared- and distributed-memory parallel graph kernel – **good productivity**
- but has **fine-grained remote communication** – **bad performance**

Background: Productivity-Performance Tradeoffs

- Productivity: ✓
  - writing distributed irregular applications is “easy” due to not having to use **explicit communication** for access patterns that are **not known until runtime**
- Performance: ✗
  - distributed irregular applications will naturally contain fine-grained remote communication
  - today’s HPC systems are **not optimized** for this

Our paper: how to **bridge the gap** between **productivity and performance** for irregular applications in Chapel?

Agenda

- Background
  - Irregular applications
  - Chapel programming language
  - Productivity-performance tradeoffs
- **Application Studies**
  - **Breadth First Search (BFS) – message aggregation**
  - **PageRank – selective data replication**
- Conclusions and Future Work

Application Study: Breadth First Search (BFS)

- A fundamental algorithm in graph analytics
- **Algorithm**: starting from a **root** vertex, visit each neighbor of the root and then visit each of their neighbors, etc. → proceeds level-by-level
- **Difficult to parallelize**
  - relies on a **queue** data structure to store vertices for processing at each iteration

```
var currQs = newBlockArr(LocaleSpace, domain(int));
var nextQs = newBlockArr(LocaleSpace, domain(int));
currQs[G[root].locale.id] += root;
var curr_lvl = 1;

while true {
  var work : bool;
  coforall cq in currQs with (|| reduce work) do on cq {
    forall u in cq with (|| reduce work) {
      if G[u].dist == -1 {
        G[u].dist = curr_lvl;
        work = true;
        for v in G[u].neighbors {
          nextQs[G[v].locale.id] += v;
        }
      }
    }
  }
  coforall (cq, nq) in zip(currQs, nextQs) do on cq {
    cq <=> nq; // <=> is the swap operator
    nq.clear();
  }
  if !work then break;
  curr_lvl += 1;
}
```

High-productivity BFS code in Chapel

Application Study: Breadth First Search (BFS)

- A fundamental algorithm in graph analytics
- **Algorithm**: starting from a **root** vertex, visit each neighbor of the root and then visit each of their neighbors, etc. → proceeds level-by-level
- **Difficult to parallelize**
  - relies on a **queue** data structure to store vertices for processing at each iteration

```
var currQs = newBlockArr(LocaleSpace, domain(int));
var nextQs = newBlockArr(LocaleSpace, domain(int));
currQs[G[root].locale.id] += root;
var curr_lvl = 1;

while true {
  var work : bool;
  coforall cq in currQs with (|| reduce work) do on cq {
    forall u in cq with (|| reduce work) {
      if G[u].dist == -1 {
        G[u].dist = curr_lvl;
        work = true;
        for v in G[u].neighbors {
          nextQs[G[v].locale.id] += v;
        }
      }
    }
  }
  coforall (cq, nq) in zip(currQs, nextQs) do on cq {
    cq <=> nq; // <=> is the swap operator
    nq.clear();
  }
  if !work then break;
  curr_lvl += 1;
}
```

High-productivity BFS code in Chapel

Each locale stores two queues:
currQ holds vertices to visit at current level.
nextQ holds vertices to visit at next level.

A queue is represented as an **associative domain** (i.e., dictionary) → allows for safe concurrent insertions

Application Study: Breadth First Search (BFS)

- A fundamental algorithm in graph analytics
- **Algorithm**: starting from a **root** vertex, visit each neighbor of the root and then visit each of their neighbors, etc. → proceeds level-by-level
- **Difficult to parallelize**
  - relies on a **queue** data structure to store vertices for processing at each iteration

```
var currQs = newBlockArr(LocaleSpace, domain(int));
var nextQs = newBlockArr(LocaleSpace, domain(int));
currQs[G[root].locale.id] += root;
var curr_lvl = 1;

while true {
  var work : bool;
  coforall cq in currQs with (|| reduce work) do on cq {
    forall u in cq with (|| reduce work) {
      if G[u].dist == -1 {
        G[u].dist = curr_lvl;
        work = true;
        for v in G[u].neighbors {
          nextQs[G[v].locale.id] += v;
        }
      }
    }
  }
  coforall (cq, nq) in zip(currQs, nextQs) do on cq {
    cq <=> nq; // <=> is the swap operator
    nq.clear();
  }
  if !work then break;
  curr_lvl += 1;
}
```

coforall loop creates a task on each locale and **forall** loop processes all vertices in a given locale's **currQ** in parallel

High-productivity BFS code in Chapel

Application Study: Breadth First Search (BFS)

- A fundamental algorithm in graph analytics
- **Algorithm**: starting from a **root** vertex, visit each neighbor of the root and then visit each of their neighbors, etc. → proceeds level-by-level
- **Difficult to parallelize**
  - relies on a **queue** data structure to store vertices for processing at each iteration

```
var currQs = newBlockArr(LocaleSpace, domain(int));
var nextQs = newBlockArr(LocaleSpace, domain(int));
currQs[G[root].locale.id] += root;
var curr_lvl = 1;

while true {
  var work : bool;
  coforall cq in currQs with (|| reduce work) do on cq {
    forall u in cq with (|| reduce work) {
      if G[u].dist == -1 {
        G[u].dist = curr_lvl;
        work = true;
        for v in G[u].neighbors {
          nextQs[G[v].locale.id] += v;
        }
      }
    }
  }
  coforall (cq, nq) in zip(currQs, nextQs) do on cq {
    cq <=> nq; // <=> is the swap operator
    nq.clear();
  }
  if !work then break;
  curr_lvl += 1;
}
```

High-productivity BFS code in Chapel

for each neighbor **v** of vertex **u**, add it to the **nextQ** on the locale where **v** is located (**G** is a block distributed array of vertices)
→ **fine-grained remote write**

Application Study: Breadth First Search (BFS)

- **Problem**: improve performance without sacrificing productivity
- **Solution**: leverage well-known benefits[1] of message **aggregation**
- "Buffer" remote writes and perform them in bulk for a given destination locale
- Can **hide** all the aggregation details in a Chapel module, leaving the application code more or less unchanged
  - potential to generalize aggregation and apply it **automatically** to programs

```
forall u in cq with (|| reduce work,
                     var agg = new DstAggregator(int))
{
  if G[u].dist == -1 {
    G[u].dist = curr_lvl;
    work = true;
    for v in G[u].neighbors {
      agg.copy(v, G[v].locale.id);
    }
  }
}
```

BFS using built-in aggregation

[1] *Communication Optimizations for Fine-grained UPC Applications, W.-Y. Chen, C. Iancu, and K. Yelick, PACT'05*

Application Study: Breadth First Search (BFS)

- **Problem**: improve performance without sacrificing productivity
- **Solution**: leverage well-known benefits[1] of message **aggregation**
- "Buffer" remote writes and perform them in bulk for a given destination locale
- Can **hide** all the aggregation details in a Chapel module, leaving the application code more or less unchanged
  - potential to generalize aggregation and apply it **automatically** to programs

```
forall u in cq with (|| reduce work,
                     var agg = new DstAggregator(int))
{
  if G[u].dist == -1 {
    G[u].dist = curr_lvl;
    work = true;
    for v in G[u].neighbors {
      agg.copy(v, G[v].locale.id);
    }
  }
}
```

each task created via the **forall** gets its own aggregator instance **agg.**

We **extended** Chapel's built-in **CopyAggregation** module to support this type of aggregation

BFS using built-in aggregation

Application Study: Breadth First Search (BFS)

- **Problem**: improve performance without sacrificing productivity
- **Solution**: leverage well-known benefits[1] of message **aggregation**
- "Buffer" remote writes and perform them in bulk for a given destination locale
- Can **hide** all the aggregation details in a Chapel module, leaving the application code more or less unchanged
  - potential to generalize aggregation and apply it **automatically** to programs

```
forall u in cq with (|| reduce work,
                     var agg = new DstAggregator(int))
{
  if G[u].dist == -1 {
    G[u].dist = curr_lvl;
    work = true;
    for v in G[u].neighbors {
      agg.copy(v, G[v].locale.id);
    }
  }
}
```

BFS using built-in aggregation

buffer the vertex ID we want to write and the locale where we want to write it to.

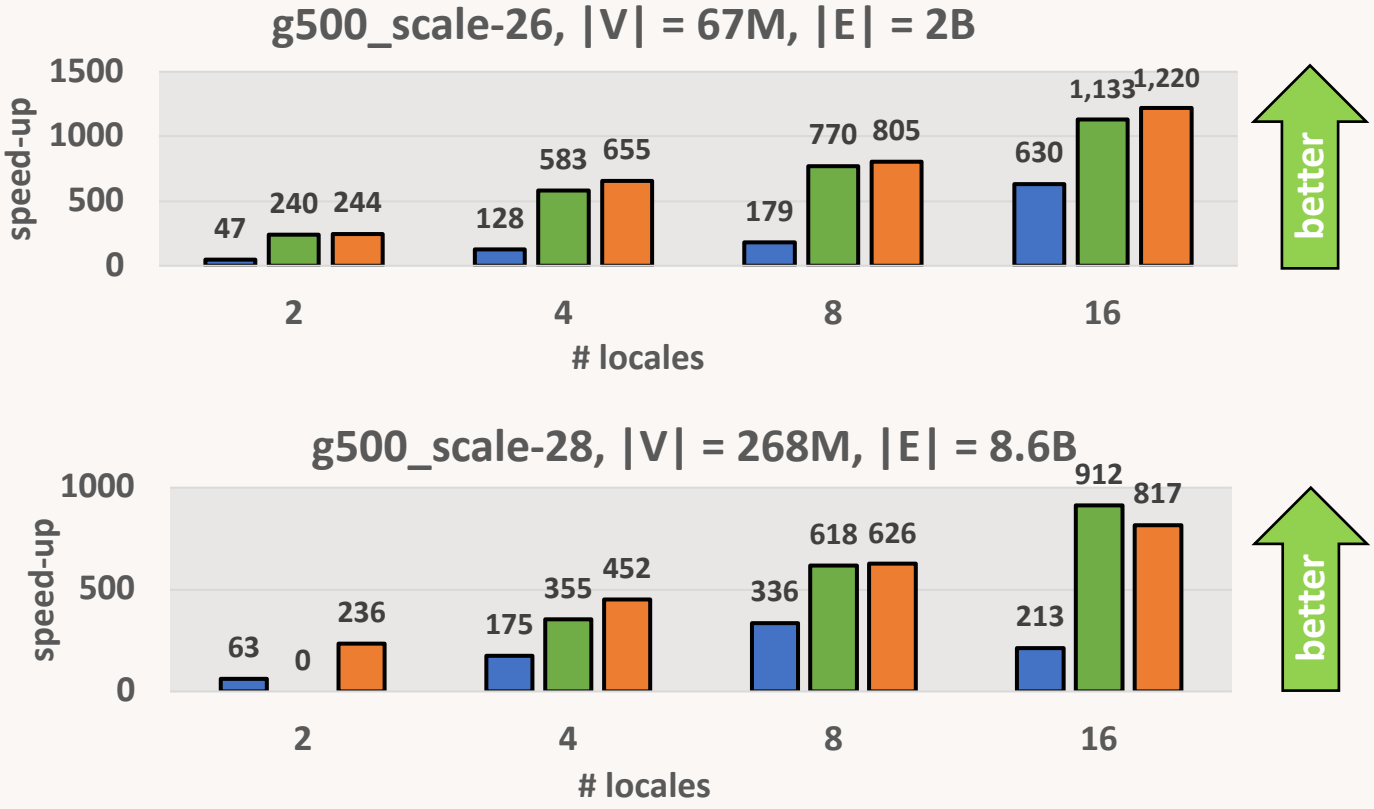
When a given task's aggregation buffer is **full** for some destination locale, it performs a **flush** of all the vertex IDs to that locale's **nextQ**

See paper for more details on how the flush is performed

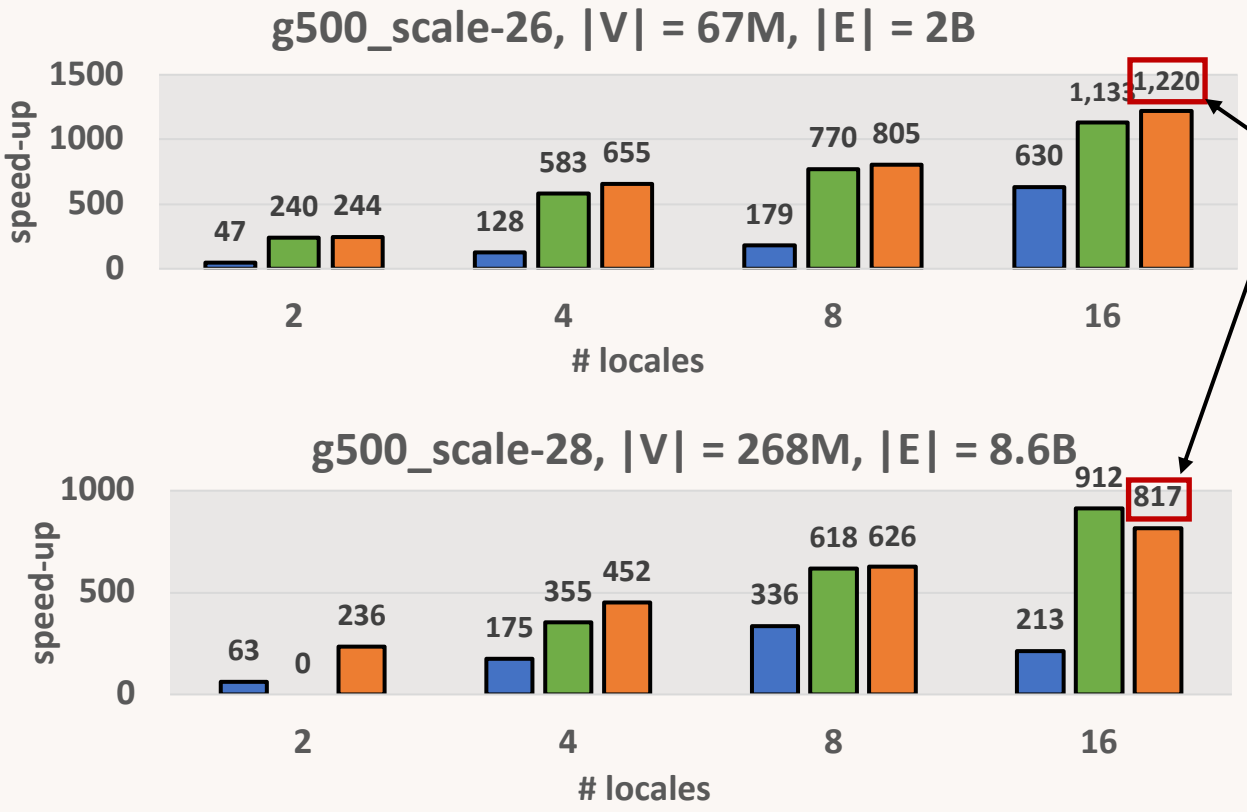
Application Study: Breadth First Search (BFS)

- Performance evaluation:
  - 16 node **FDR-Infiniband** cluster, 20 cores per node, 512 GB of memory per node
  - Chapel v1.24.1, GASnet, ibv comm substrate
- Compare high productivity Chapel code to:
  - built-in aggregation Chapel code
    - **fixed size** aggregation buffers, flushes happen “behind the scenes”
  - manual aggregation using MPI+OpenMP (2-sided communication)
    - aggregation buffers **grow dynamically** to accommodate all remote writes in a given iteration, performs a **single flush**
  - manual aggregation Chapel code
    - same aggregation approach as MPI+OpenMP but implemented in Chapel
    - do we improve performance by **sacrificing productivity** in Chapel?

BFS Runtime Speed-ups over High Productivity Chapel



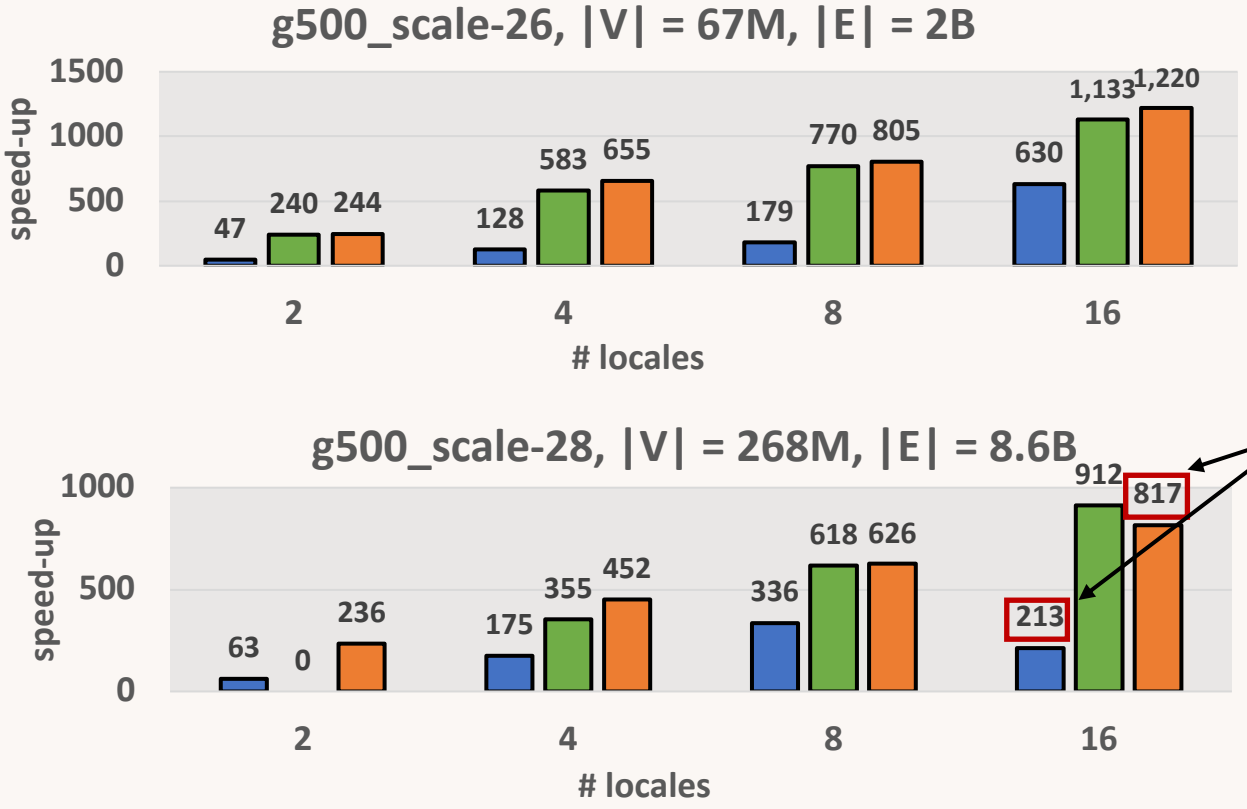
BFS Runtime Speed-ups over High Productivity Chapel



- **Significant speed-ups** over high-productivity Chapel
  - built-in aggregation requires **minimal changes** to user's code

MPI+OpenMP | Chapel Manual Aggr | Chapel Built-in Aggr

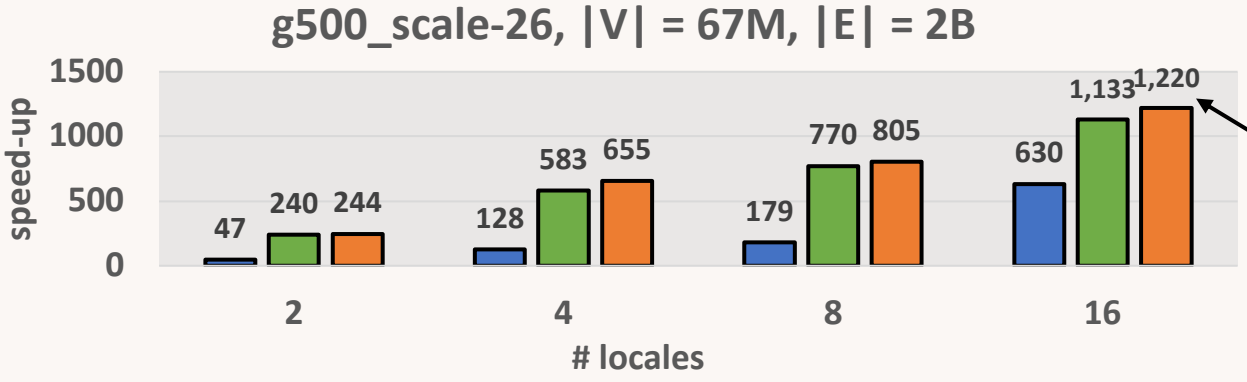
BFS Runtime Speed-ups over High Productivity Chapel

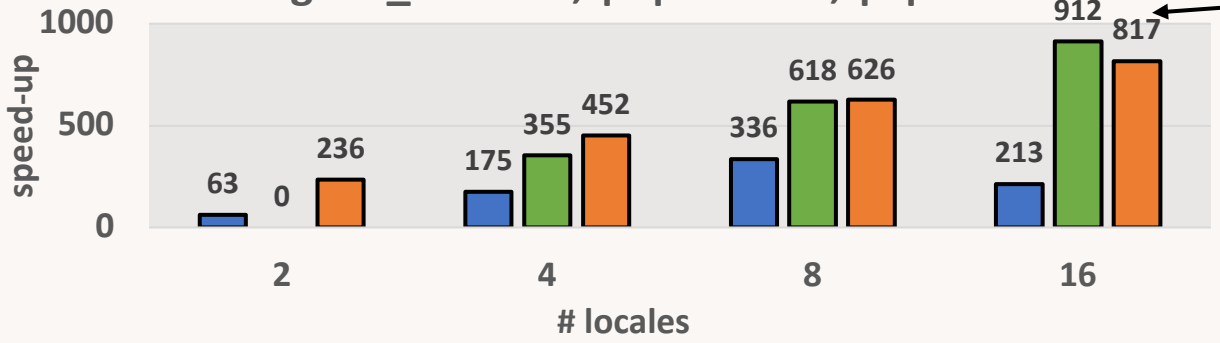


- **Significant speed-ups** over high-productivity Chapel
  - built-in aggregation requires **minimal changes** to user's code
- **MPI+OpenMP slower** than the optimized Chapel codes
  - largely due to differences in **memory allocator** used, which is crucial for buffer resizing in all approaches

MPI+OpenMP Chapel Manual Aggr Chapel Built-in Aggr

BFS Runtime Speed-ups over High Productivity Chapel

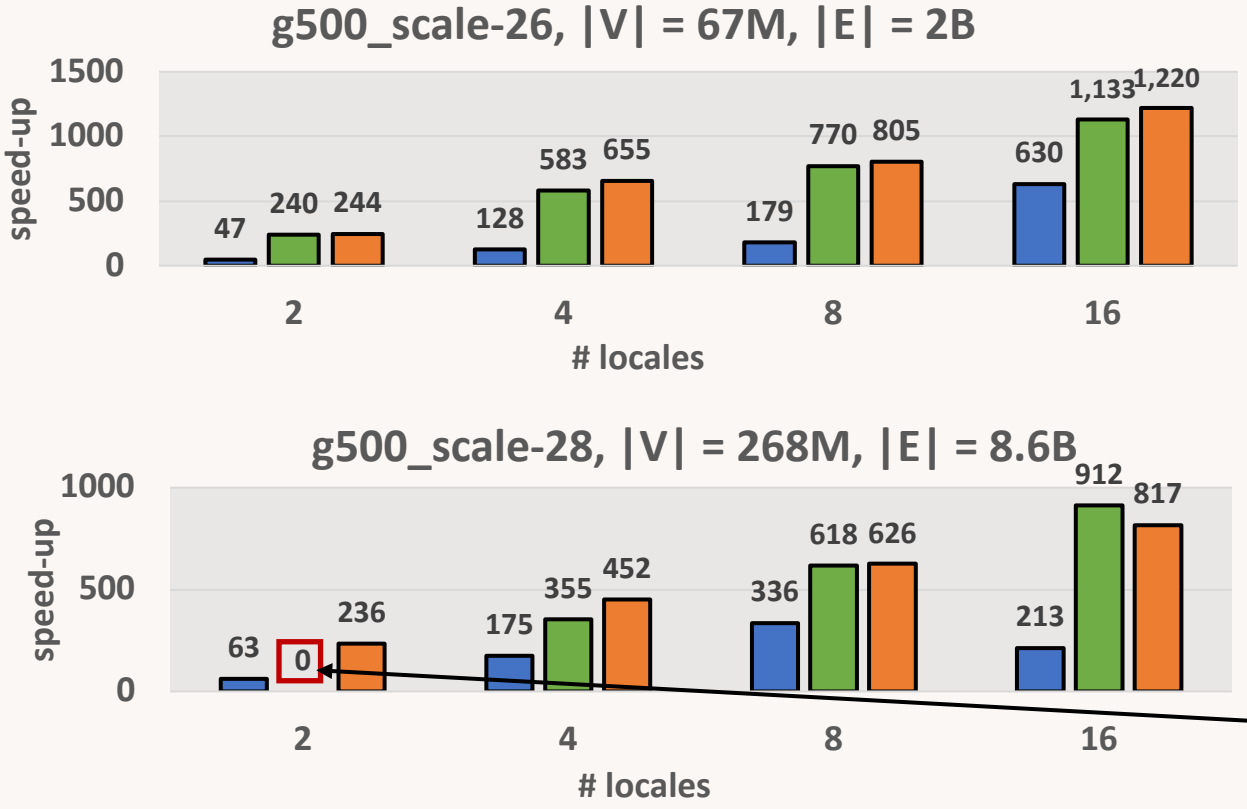




- Built-in aggregation is **generally faster** than manual aggregation
  - built-in aggregation uses **fixed size** buffers and flushes when full
  - manual approach **dynamically grows** buffers to accommodate all remote-writes in an iteration
- **Overhead of flushes** becomes apparent on large data sets and high locale counts

MPI+OpenMP **Chapel Manual Aggr** **Chapel Built-in Aggr**

BFS Runtime Speed-ups over High Productivity Chapel



- Built-in aggregation is **generally faster** than manual aggregation
  - built-in aggregation uses **fixed size** buffers and flushes when full
  - manual approach **dynamically grows** buffers to accommodate all remote-writes in an iteration
  - **Overhead of flushes** becomes apparent on large data sets and high locale counts
- Manual aggregation **runs out of memory** on g500_scale-28 on 2 locales
  - too much data to aggregate all-at-once in an iteration

MPI+OpenMP **Chapel Manual Aggr** **Chapel Built-in Aggr**

Application Study: PageRank

- **Purpose:** compute a ranking/importance factor for each vertex in the graph based on its incoming edges/links
- Originally developed for website ranking in search engines, but often used in more general graph analytics
- **Iterative**
  - performs kernel on same graph until convergence

```
forall v in G {
  var val = 0.0;
  for i in 0..#v.inDegree {
    ref u = G[v.inNeighbors[i]];
    val += u.readPR/u.outDegree;
  }
  v.writePR = (val*d)+((1-d)/numVerts) + sinkVal;
}
```

High-productivity PageRank kernel in Chapel

Application Study: PageRank

- **Purpose:** compute a ranking/importance factor for each vertex in the graph based on its incoming edges/links
- Originally developed for website ranking in search engines, but often used in more general graph analytics
- **Iterative**
  - performs kernel on same graph until convergence

G is a block distributed array of vertex records, where each vertex stores an array of incoming neighbor IDs

```
forall v in G {
  var val = 0.0;
  for i in 0..#v.inDegree {
    ref u = G[v.inNeighbors[i]];
    val += u.readPR/u.outDegree;
  }
  v.writePR = (val*d)+((1-d)/numVerts) + sinkVal;
}
```

High-productivity PageRank kernel in Chapel

Application Study: PageRank

- **Purpose:** compute a ranking/importance factor for each vertex in the graph based on its incoming edges/links
- Originally developed for website ranking in search engines, but often used in more general graph analytics
- **Iterative**
  - performs kernel on same graph until convergence

```
forall v in G {
  var val = 0.0;
  for i in 0..#v.inDegree {
    ref u = G[v.inNeighbors[i]];
    val += u.readPR/u.outDegree;
  }
  v.writePR = (val*d)+((1-d)/numVerts) + sinkVal;
}
```

irregular access to **G** to access each neighbor of vertex **v**

this access pattern **does not change** throughout all executions of this kernel

High-productivity PageRank kernel in Chapel

Application Study: PageRank

- **Problem**: improve the performance without sacrificing productivity
- **Solution**: use **inspector-executor** technique to perform selective data replication

[1] *Runtime Optimizations for Irregular Applications in Chapel,* CHIUW 2021

Application Study: PageRank

- **Problem**: improve the performance without sacrificing productivity
- **Solution**: use **inspector-executor** technique to perform selective data replication
- **Inspector**: routine that performs **one-time** analysis of the memory access pattern in the kernel to determine which data is remotely accessed and create **local copies** of that data
  - in PageRank, the data replicated are the vertex pagerank values
- **Executor**: modified version of the kernel that reads from the local copies rather than performing remote reads
  - before each kernel execution, perform an update to **"refresh"** the local copies
  - **exploits data reuse** → perform one remote read to refresh and reuse that local value many times in the kernel

[1]*Runtime Optimizations for Irregular Applications in Chapel,* CHIUW 2021

Application Study: PageRank

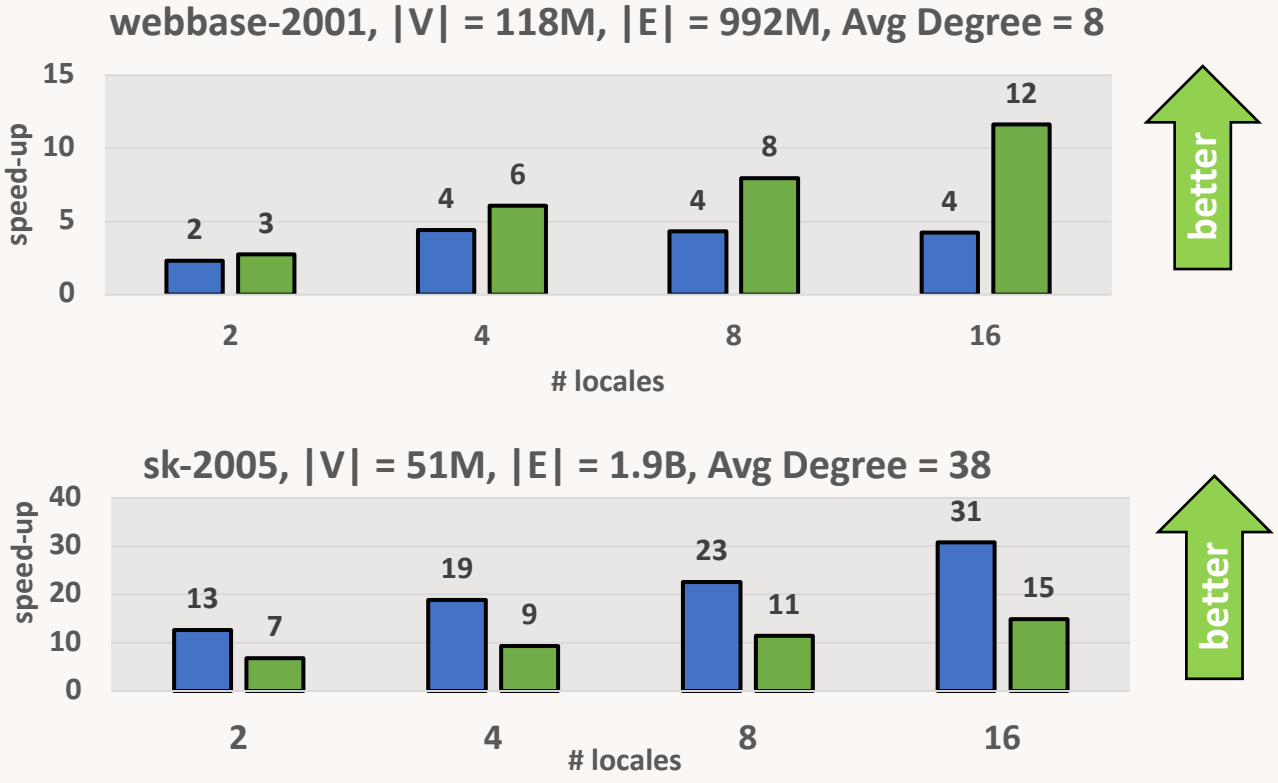
- **Problem**: improve the performance without sacrificing productivity
- **Solution**: use **inspector-executor** technique to perform selective data replication
- **Inspector**: routine that performs **one-time** analysis of the memory access pattern in the kernel to determine which data is remotely accessed and create **local copies** of that data
  - in PageRank, the data replicated are the vertex pagerank values
- **Executor**: modified version of the kernel that reads from the local copies rather than performing remote reads
  - before each kernel execution, perform an update to **"refresh"** the local copies
  - **exploits data reuse** → perform one remote read to refresh and reuse that local value many times in the kernel
- An inspector-executor can be implemented as a **compiler optimization**, which will be the focus of our future work
  - For now, we **hand-generate** the code that the compiler would generate
  - Refer to the paper and our CHIUW 2021 paper[1] for more details

[1] *Runtime Optimizations for Irregular Applications in Chapel,* CHIUW 2021

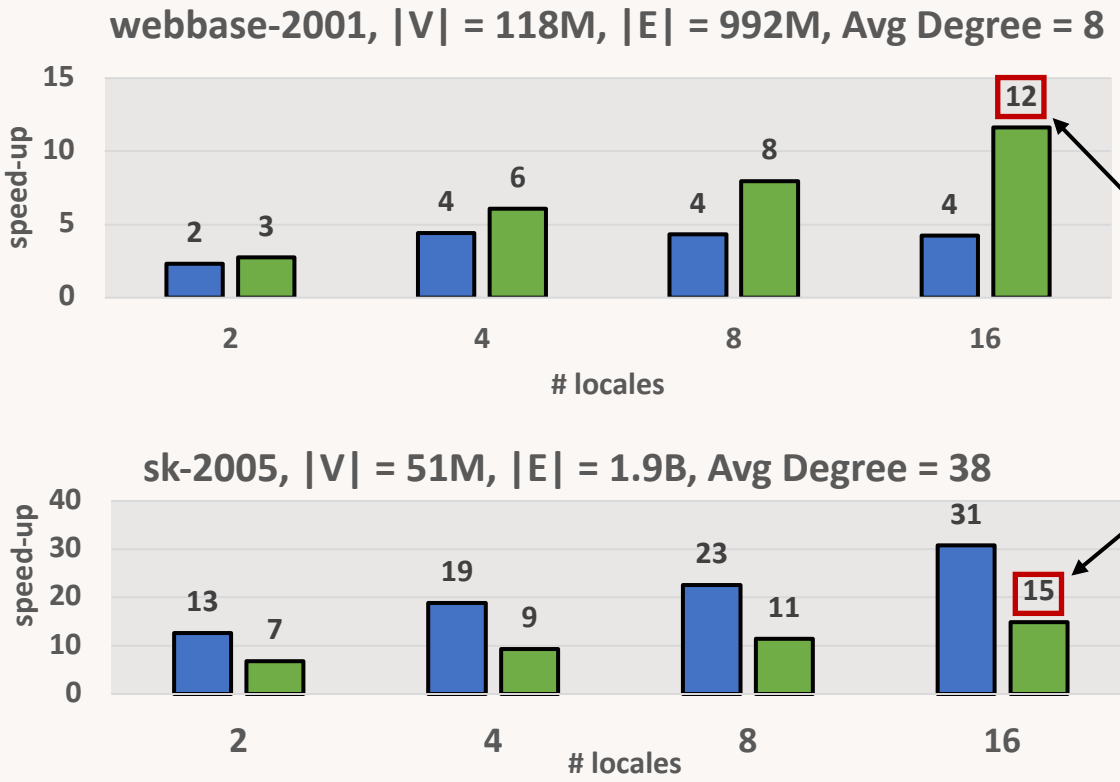
Application Study: PageRank

- Performance evaluation:
  - 16 node **Cray XC** cluster, 44 cores per node, 128 GB of memory per node
  - Chapel v1.24.1, ugni, **aries** comm substrate
- Compare high productivity Chapel code to:
  - inspector-executor (**I/E**) optimized Chapel code
    - only replicates the pagerank values that are accessed remotely
    - **includes cost** of the inspector analysis
  - MPI+OpenMP code with **full-data replication** (see paper for details)
    - each rank replicates **ALL** of the pagerank values, since no runtime analysis is performed

PageRank Runtime Speed-ups over High Productivity Chapel

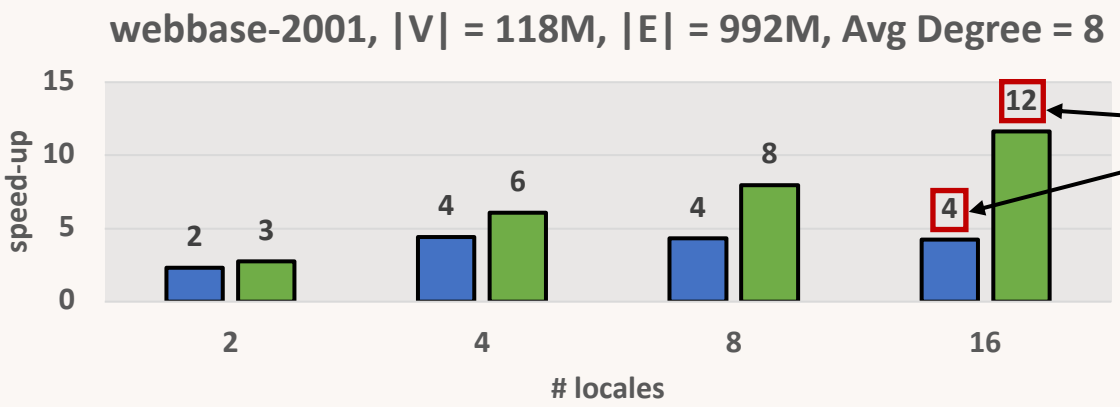


PageRank Runtime Speed-ups over High Productivity Chapel



- **Large speed-ups** over high-productivity Chapel
  - even though the **Aries** interconnect is **optimized for small messages**

PageRank Runtime Speed-ups over High Productivity Chapel

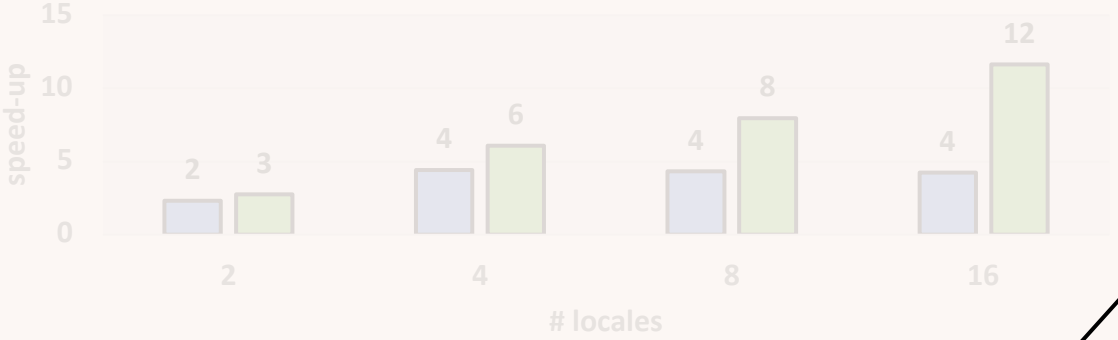


- **MPI+OpenMP slower** than optimized Chapel code **on webbase-2001**
  - this graph is **very sparse** with low average degree per vertex
- As a result, there is **less remote communication** per locale
  - I/E will replicate only what is remotely accessed
  - MPI+OpenMP does **full replication**
- → I/E performs less communication, leading to better performance
  - even considering the **time to execute the inspector**

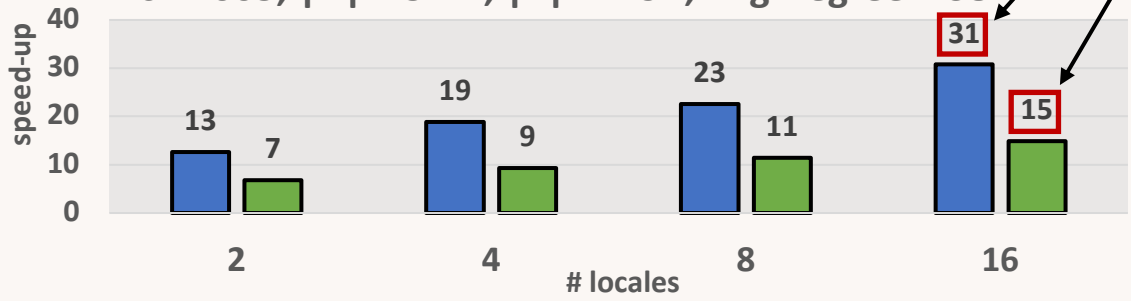
MPI+OpenMP **Chapel I/E**

PageRank Runtime Speed-ups over High Productivity Chapel

webbase-2001, |V| = 118M, |E| = 992M, Avg Degree = 8



sk-2005, |V| = 51M, |E| = 1.9B, Avg Degree = 38



MPI+OpenMP Chapel I/E

- **MPI+OpenMP faster** than optimized Chapel code **on sk-2005**
  - this graph is **not as sparse** with **high average degree** per vertex compared to webbase-2001
- As a result, I/E replicates a **comparable amount** of data as MPI+OpenMP
  - now **overhead** of inspector comes into play
- Other factors contributing to performance loss vs. MPI+OpenMP
  - **wide-pointer references** in Chapel code lead to 1.23x slow-down

Conclusions and Future Work

- High productivity ≠ high performance for irregular applications in Chapel
  - **implicit fine-grained communication**
- We explored techniques that leave productivity intact but **significantly improve performance**
  - message aggregation, selective data replication
  - **1,219x** speed-up for BFS, **22x** speed-up for PageRank
- Future work
  - perform these optimizations **automatically**
  - look at irregular applications with other data access patterns

Towards High Productivity and Performance for Irregular Applications in Chapel

Thomas B. Rolinger *University of Maryland*
Joseph Craft *Laboratory for Physical Sciences*
Christopher D. Krieger *Laboratory for Physical Sciences*
Alan Sussman *University of Maryland*