User-Defined Parallel Zippered Iterators in Chapel

Brad Chamberlain, Sung-Eun Choi
Steve Deitz, Angeles Navarro
Cray Inc. / University of Málaga

PGAS 2011: October 17th, 2011

PGAS: Partitioned Global Address Space Languages
(Or perhaps: Partitioned Global *Namespace* Languages)

Concept:

- support a shared namespace
  - "any parallel task can access any lexically visible variable"
- give each variable a well-defined affinity to a system node
  - "local variables are cheaper to access than remote ones"
- founding members: UPC, Co-Array Fortran, Titanium

Strengths:

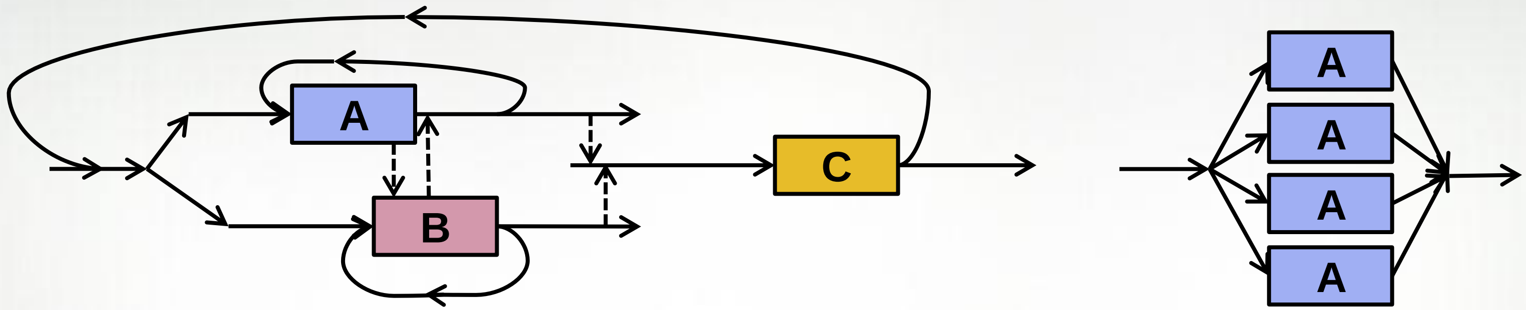
- permits users to specify *what* to transfer rather than *how*
- supports reasoning about locality/affinity to get scalability

Weaknesses (of traditional PGAS languages):

- restricted to SPMD programming and execution models
- limited support for distributed arrays

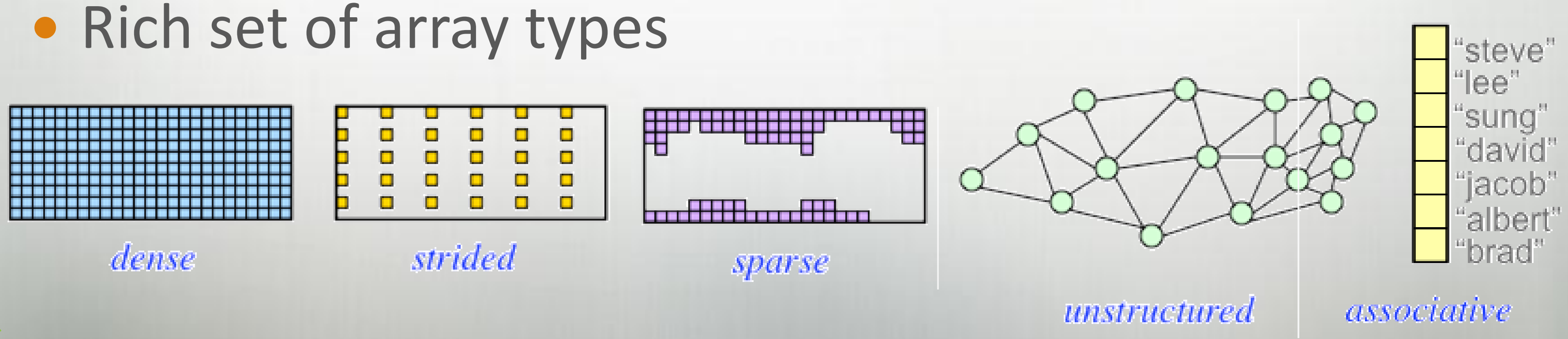
Chapel: A Next-Generation PGAS Language

- General/dynamic/multithreaded parallelism



- Distinct concepts for parallelism vs. locality
  - e.g., *coforall loop* creates tasks, *locale* type represents locality

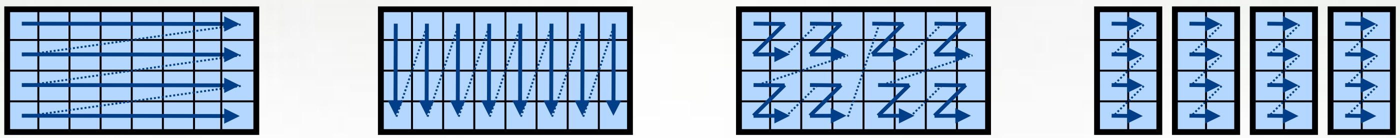
- Rich set of array types



Array Implementation: Questions

Q1: How are arrays laid out in memory?

- Are regular arrays laid out in row- or column-major order? Or…?

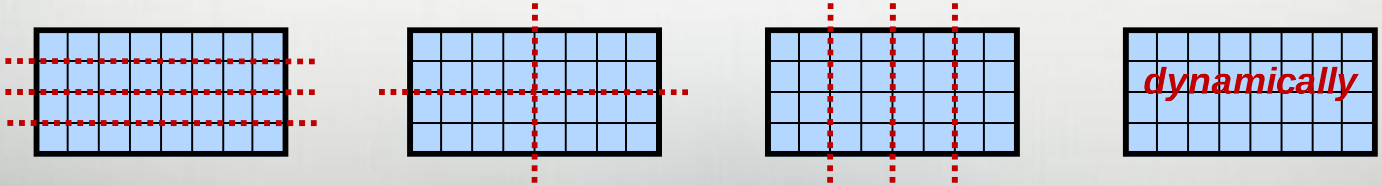


…?

- How are sparse arrays stored? (COO, CSR, CSC, block-structured, …?)
- What memories/memory types are used?

Q2: How are arrays distributed between locales/nodes?

- Completely local to one locale? Or distributed?
- If distributed… In a blocked manner? cyclically? block-cyclically? recursively bisected? dynamically rebalanced? …?

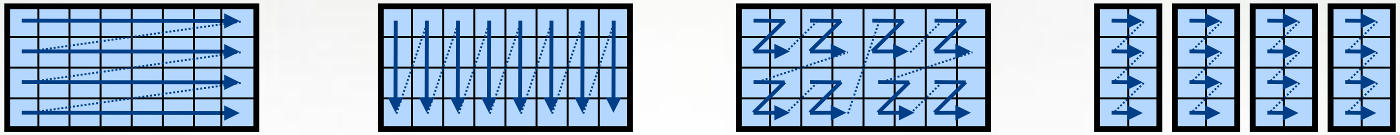


…?

Array Implementation: Questions

Q1: How are arrays laid out in memory?

- Are regular arrays laid out in row- or column-major order? Or…?



…?

- How are sparse arrays stored? (COO, CSR, CSC, block-structured, …?)
- What memories/memory types are used?

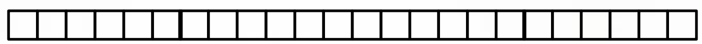
Q2: How are arrays distributed between locales/nodes?

- Completely local to one locale? Or distributed?
- If distributed… In a blocked manner? cyclically? block-cyclically? recursively bisected? dynamically rebalanced? …?

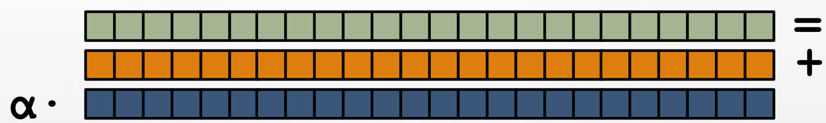
A: Chapel's *domain maps* are designed to give the user full control over such decisions

STREAM Triad in Chapel

```
const ProblemSpace = [1..m];
```



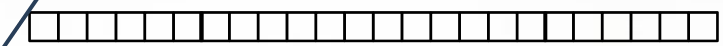
```
var A, B, C: [ProblemSpace] real;
```



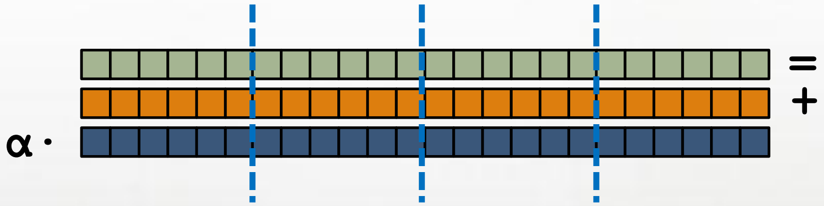

```
A = B + alpha * C;
```

STREAM Triad in Chapel (multicore)

```
const ProblemSpace = [1..m];
```



```
var A, B, C: [ProblemSpace] real;
```

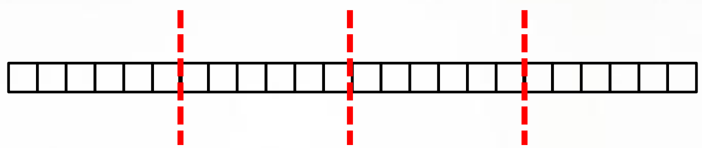



```
A = B + alpha * C;
```

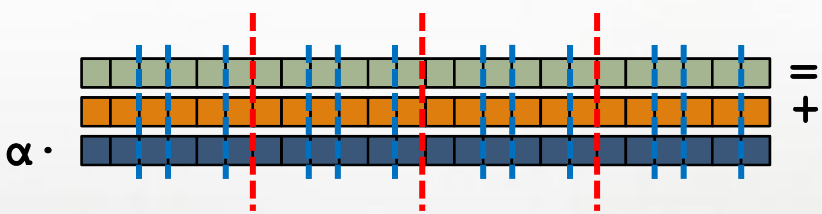
No domain map specified => use default layout
- current locale owns all indices and values
- computation will execute using local processors only

STREAM Triad in Chapel (multinode, blocked)

```
const ProblemSpace = [1..m]
                     dmapped Block(boundingBox=[1..m]);
```



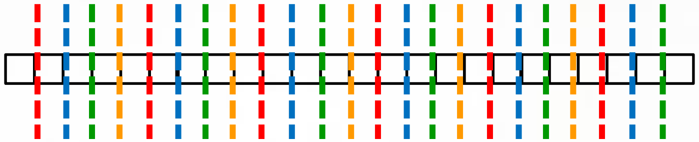
```
var A, B, C: [ProblemSpace] real;
```



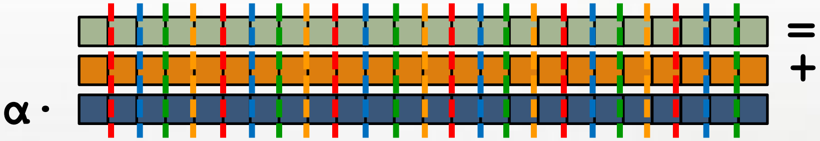

```
A = B + alpha * C;
```

STREAM Triad in Chapel (multinode, cyclic)

```
const ProblemSpace = [1..m]
                     dmapped Cyclic(startIdx=1);
```



```
var A, B, C: [ProblemSpace] real;
```




```
A = B + alpha * C;
```

For More Information on Domain Maps

HotPAR'10: *User-Defined Distributions and Layouts in Chapel*
Chamberlain, Deitz, Iten, Choi; June 2010

CUG 2011: *Authoring User-Defined Domain Maps in Chapel*
Chamberlain, Choi, Deitz, Iten, Litvinov; May 2011

Chapel release:

- Technical notes detailing domain map interface for programmers:
  $CHPL_HOME/doc/technotes/README.dsi
- Current domain maps:
  $CHPL_HOME/modules/dists/*.chpl
  layouts/*.chpl
  internal/Default*.chpl

Motivating Questions for This Paper

Q3: How are data parallel loops implemented?

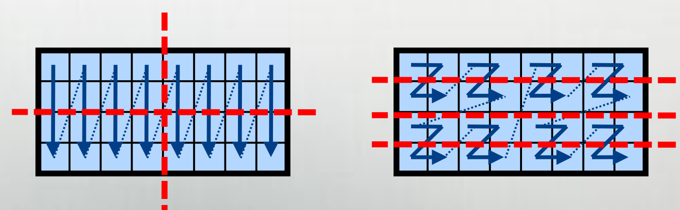
```
forall i in B.domain do B[i] = i/10.0;
forall c in C do c = 3.0;
```

- How many tasks? Where do they execute?
- How is the iteration space divided between the tasks?

Q4: How are parallel zippered loops implemented?

```
forall (a,b,c) in (A,B,C) do
  a = b + alpha * c;
```

- Particularly given that the iterands might have incompatible distributions, memory layouts, and parallelization strategies



A B C

Motivating Questions for This Paper

Q3: How are data parallel loops implemented?

```
forall i in B.domain do B[i] = i/10.0;
forall c in C do c = 3.0;
```

- How many tasks? Where do they execute?
- How is the iteration space divided between the tasks?

Q4: How are parallel zippered loops implemented?

```
forall (a,b,c) in (A,B,C) do
  a = b + alpha * c;
```

- Particularly given that the iterands might have incompatible distributions, memory layouts, and parallelization strategies

A: Chapel's *leader-follower* iterators (the topic of this paper) are designed to give users full control over such decisions

Outline

- ✓ Background and Motivation
- ➢ Quick Introduction to Chapel
- Leader-Follower Iterators
- Results and Summary

What is Chapel?

- An emerging parallel programming language
  - Design and development led by Cray Inc.
  - Started under the DARPA HPCS program
- **Overall goal:** Improve programmer productivity
  - Improve the programmability of parallel computers
  - Match or beat the performance of current programming models
  - Support better portability than current programming models
  - Improve the robustness of parallel codes
- A work-in-progress

Chapel's Implementation

- Being developed as open source at SourceForge
- Licensed as BSD software
- Target Architectures:
  - multicore desktops and laptops
  - commodity clusters
  - Cray architectures
  - systems from other vendors
  - (in-progress: CPU+accelerator hybrids, manycore, …)

A few of Chapel's Motivating Themes

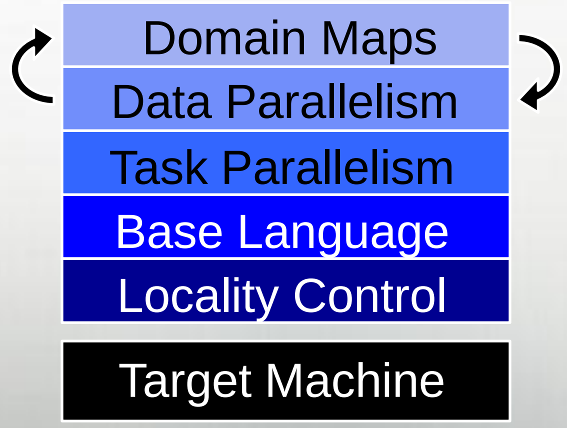
General Parallel Programming

- "any parallel algorithm on any parallel hardware"

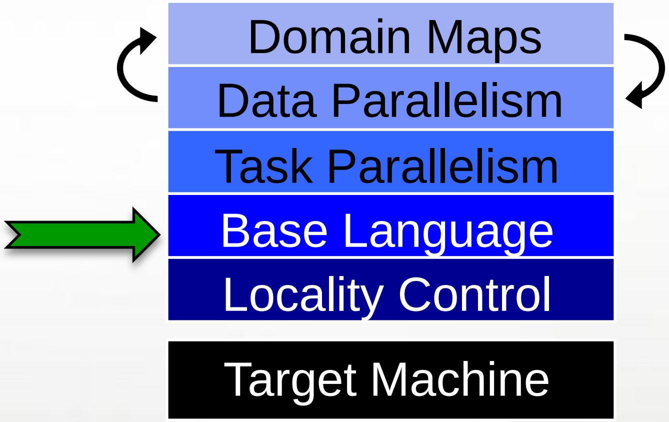
Multiresolution Parallel Programming

- lower levels for control
- higher levels for programmability, productivity

Chapel language concepts



Base Language Features



Iterators

```
iter fibonacci(n) {
  var current = 0,
      next = 1;
  for 1..n {
    yield current;
    current += next;
    current <=> next;
  }
}
```

```
for f in fibonacci(7) do
  writeln(f);
```

```
0
1
1
2
3
5
8
```

```
iter tiledRMO(D, tilesize) {
  const tile = [0..#tilesize,
                0..#tilesize];
  for base in D by tilesize do
    for ij in D[tile + base] do
      yield ij;
}
```

```
const D = [1..n, 1..n];
for ij in tiledRMO(D, 2) do
  write(ij);
```

```
(1,1)(1,2)(2,1)(2,2)
(1,3)(1,4)(2,3)(2,4)
(1,5)(1,6)(2,5)(2,6)
…
(3,1)(3,2)(4,1)(4,2)
```

Zippered Iteration

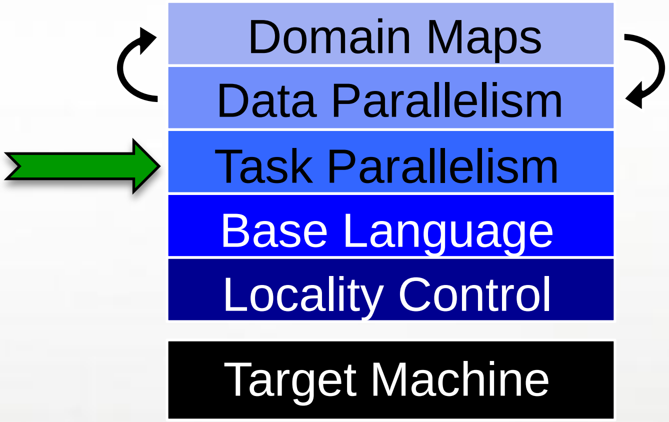
```
var A: [0..9] real;

for (i,j,a) in (1..10, 2..20 by 2, A) do
  a = j + i/10.0;

writeln(A);
```

```
2.1 4.2 6.3 8.4 10.5 12.6 14.7 16.8 18.9 21.0
```

Task Parallel Features



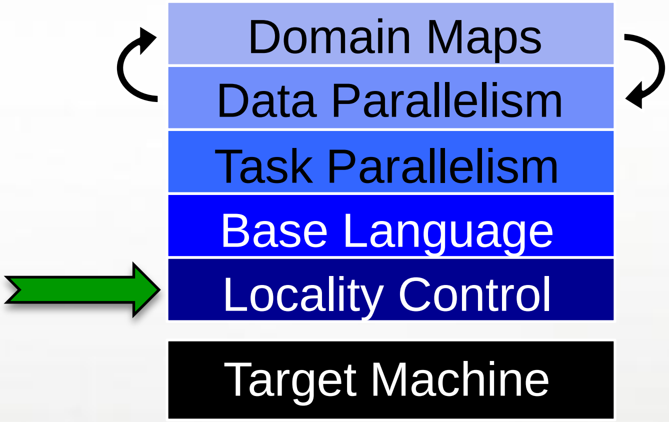
Coforall Loops

```
coforall t in 0..#numTasks do
  writeln("Hello from task ", t, " of ", numTasks);

writeln("All tasks done");
```

```
Hello from task 2 of 4
Hello from task 0 of 4
Hello from task 3 of 4
Hello from task 1 of 4
All tasks done
```

Locality Features



The Locale Type

Definition:

- Abstract unit of target architecture
- Supports reasoning about locality
- Capable of running tasks and storing variables
  - i.e., has processors and memory

Typically: A multi-core processor or SMP node

Coding with Locales

- Specify # of locales when running Chapel programs

```
% a.out --numLocales=8
```

```
% a.out –nl 8
```

- Chapel provides built-in variables representing locales

```
config const numLocales: int = …;
const LocaleSpace = [0..#numLocales];
const Locales: [LocaleSpace] locale;
```

Locales

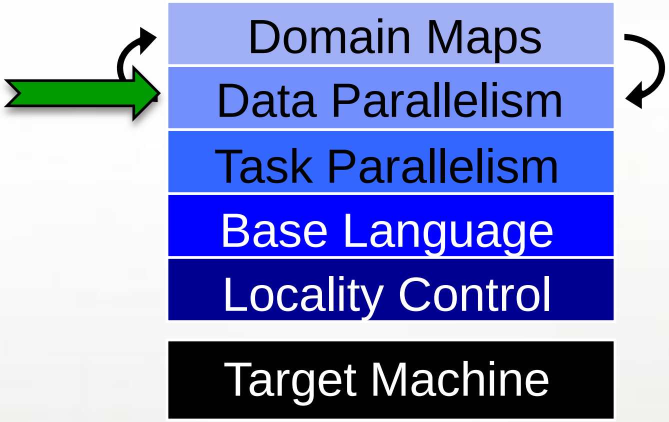
- *On-clauses* support placement of computations:

```
writeln("on locale 0");
on Locales[1] do
  writeln("now on locale 1");
writeln("on locale 0 again");
```

```
on A[i,j] do
  bigComputation(A);

on node.left do
  search(node.left);
```

Data Parallel Features



Forall Loops

```
forall a in A do
  writeln("Here is an element of A: ", a);
```

How many tasks?

- (That's what we're here to figure out!)
- In practice, typically 1 ≤ #Tasks << #Iterations)

```
forall (a, i) in (A, 1..n) do
  a = i/10.0;
```

Forall-loops may be zippered, like for-loops

- Corresponding iterations must match up
- (But how?!)

Previous Work

Other languages have supported zippered iteration…

- …but have either been serial
  - **(e.g., Python, Ruby, …)**
- …or parallel, yet only supporting a small number of built-in zipperable types/parallelization strategies
  - **(e.g., NESL, HPF, ZPL, …)**

Outline

- ✓ Background and Motivation
- ✓ Quick Introduction to Chapel
- ➢ Leader-Follower Iterators
- Results and Summary

Leader-Follower Iterators: Definition

- Chapel defines all zippered forall loops in terms of leader-follower iterators:
  - *leader iterators:* create parallelism, assign iterations to tasks
  - *follower iterators:* serially execute work generated by leader

- Given…

```
forall (a,b,c) in (A,B,C) do
  a = b + alpha * c;
```

…A is defined to be the *leader*

…*A, B,* and *C* are all defined to be *followers*

Leader-Follower Iterators: Rewriting

- *Conceptually*, the Chapel compiler translates:

```
forall (a,b,c) in (A,B,C) do
  a = b + alpha * c;
```

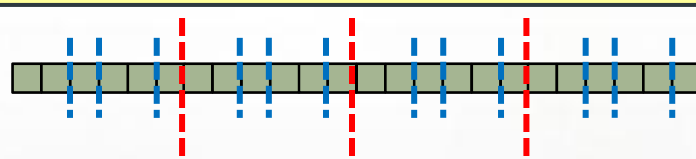
  into:

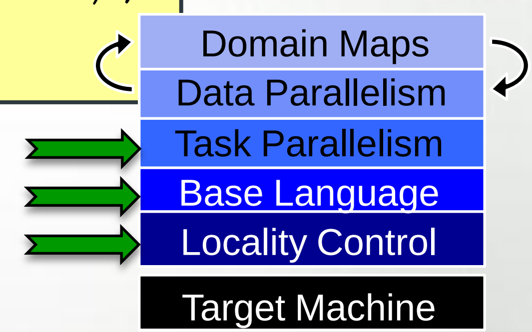
```
inlined A.lead() iterator, which yields work...
  for (a,b,c) in (A.follow(work),
                  B.follow(work)
                  C.follow(work)) do
    a = b + alpha * c;
```

Writing Leaders and Followers

Leader iterators are defined using task/locality features:

```
iter BlockArr.lead() {
  coforall loc in Locales do
    on loc do
      coforall tid in here.numCores do
        yield computeMyChunk(loc.id, tid);
}
```





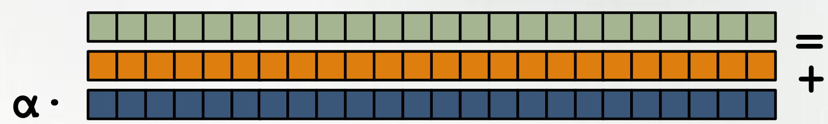
Follower iterators simply use serial features:

```
iter BlockArr.follow(work) {
  for i in work do
    yield accessElement(i);
}
```

Leader-Follower Iterators: Rewriting

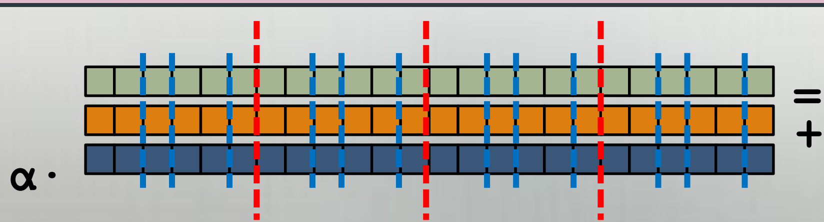
- Given the previous leader iterators…

```
forall (a,b,c) in (A,B,C) do
  a = b + alpha * c;
```



…would get rewritten by the Chapel compiler as:

```
coforall loc in Locales do
  on loc do
    coforall tid in here.numCores {
      const work = computeMyChunk(loc.id, tid);
      for (a,b,c) in (A.follow(work),
                      B.follow(work)
                      C.follow(work)) do
        a = b + alpha * c;        }
```



Leader-Follower Iterators…

…permit the user to write high-level parallel loops…

- …without tripping over all of the low-level details
- while still able to reason about them semantically

…provide clear answers to our motivating questions:

- Chapel semantics define a leader for each data parallel loop
- Leader iterators decide…
  - how many tasks to use
  - where the tasks execute
  - what work each task owns
- Followers are responsible for yielding corresponding iterations – even if they aren't local
  - gives them control over communication granularity/approach

Controlling Data Parallelism

Q: *“What if I don’t like the approach implemented by an array’s leader iterator?”*

A: Several possibilities…

Controlling Data Parallelism

```
forall (b,a,c) in (B,A,C) do
  a = b + alpha * c;
```

Make something else the leader.

Controlling Data Parallelism

```
const ProblemSize = [1..n] dmapped BlockCyclic(start=1,
                                                blocksize=64);
var A, B, C: [ProblemSize] real;

forall (a,b,c) in (A,B,C) do
  a = b + alpha * C;
```

Change the array's default leader by changing its domain map (perhaps to one that you wrote yourself).

Controlling Data Parallelism

```
forall (a,b,c) in (dynamic(A, chunk=64), B, C) do
  a = b + alpha * c;
```

Invoke some other leader iterator explicitly (perhaps one that you wrote yourself).

Example Leader-Follower Iterators in the Paper

- Statically-blocked leaders and followers
  - local and distributed (single- and multi-locale)
- OpenMP-style dynamic leader iterators
  - dynamic (deal out fixed chunk size)
  - guided (deal out varying chunk sizes)
- Adaptive work-stealing leader
- Pseudo-random number stream follower

(The paper also covers coding conventions and implementation details in more detail than the talk)

Outline

- ✓ Background and Motivation
- ✓ Quick Introduction to Chapel
- ✓ Leader-Follower Iterators
- ➢ Results and Summary

Experimental Results

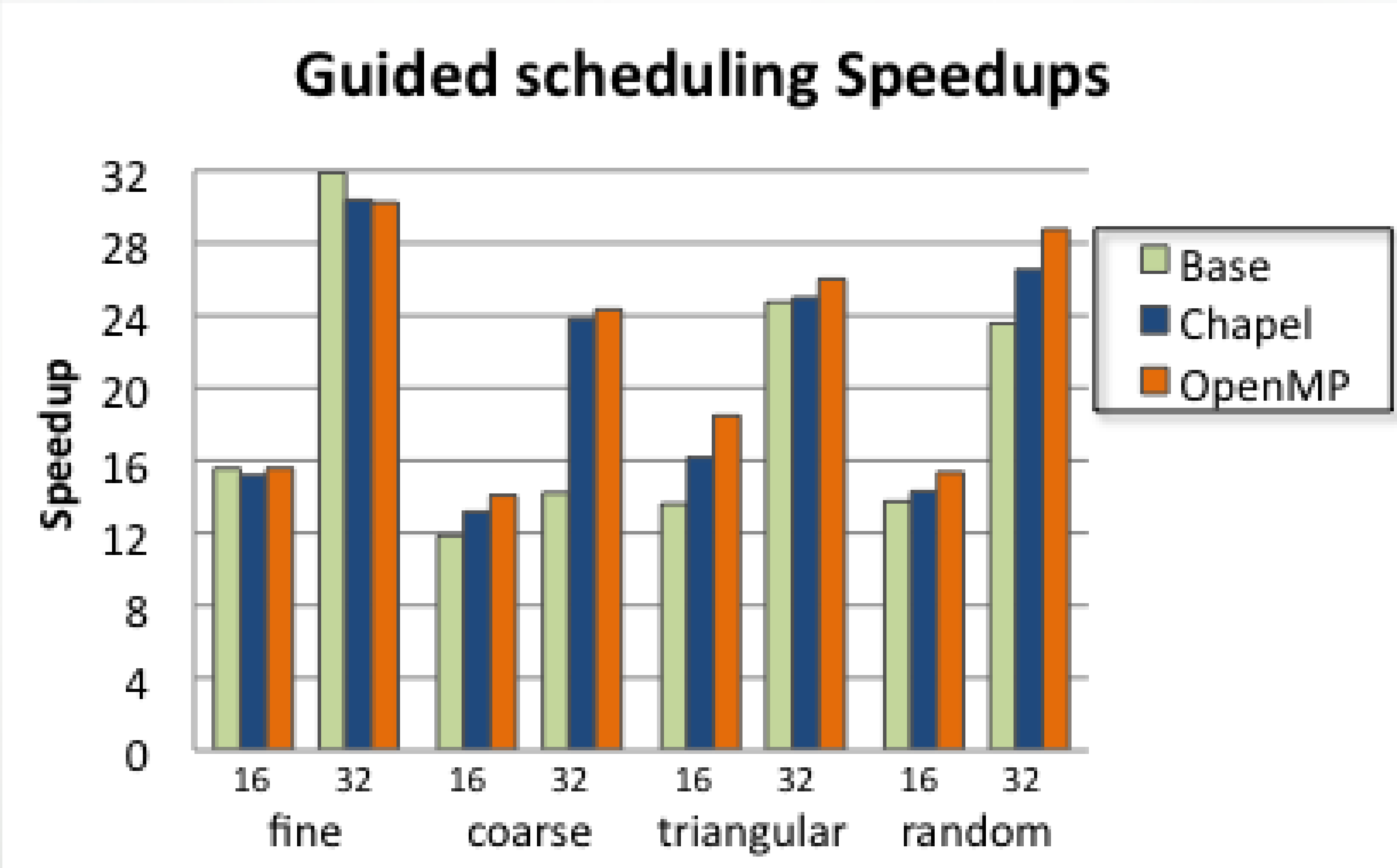
Shared Memory: Chapel vs. OpenMP

- Chapel dynamic vs. OpenMP dynamic
- Chapel guided vs. OpenMP guided
- Chapel adaptive vs. OpenMP guided

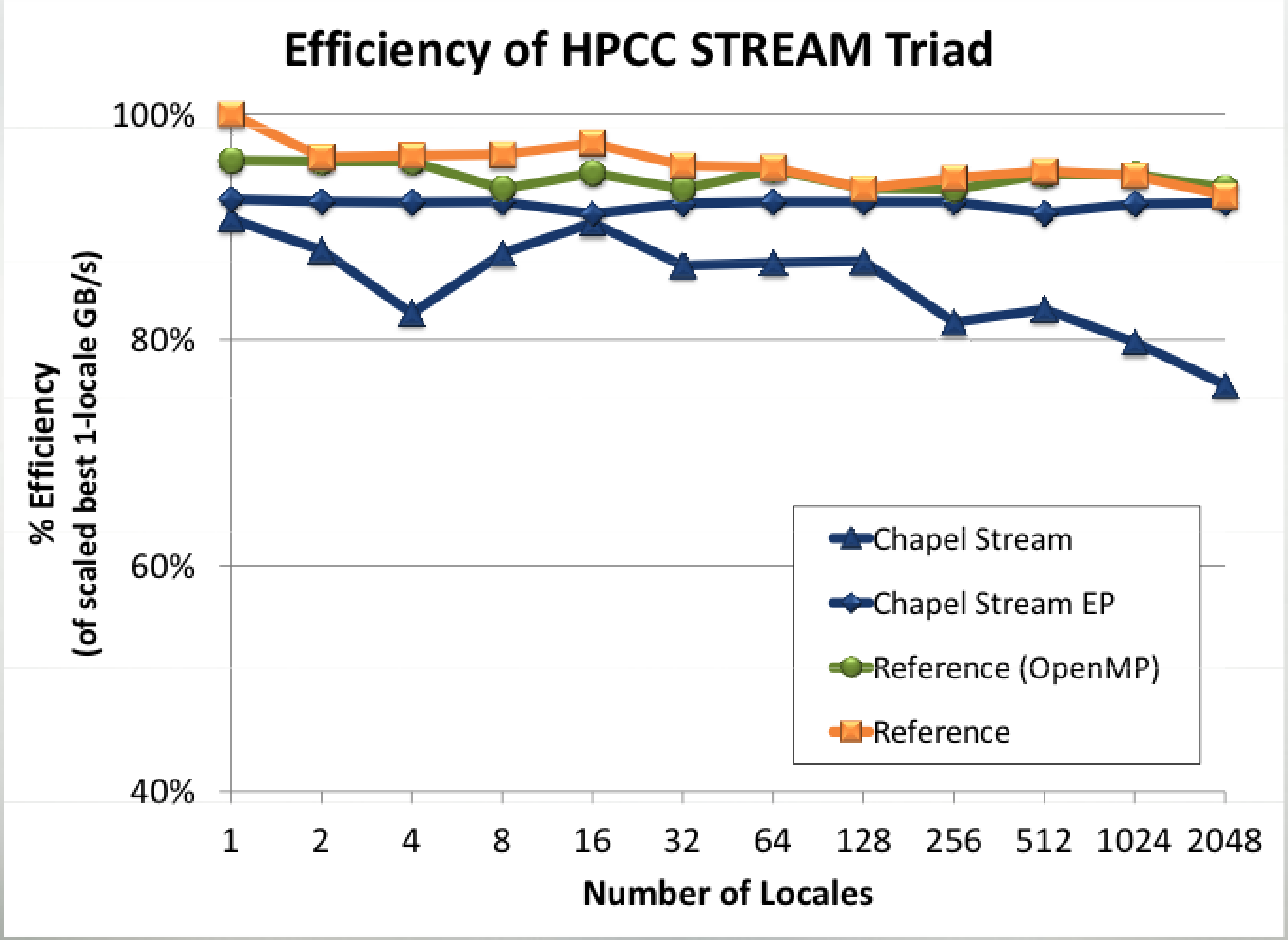
Distributed Memory: HPCC Benchmarks

- STREAM: multi-locale static block leader & followers
- RA: multi-locale static block leader + random follower

Chapel vs. OpenMP Guided



STREAM Triad



Summary

- Leader-follower iterators permit users to write their own recipes for parallel iteration in Chapel
  - Control over granularity, locality, work mapping
  - Shared- or distributed-memory execution
  - Without need to modify compiler or runtime
- Initial performance results support the approach
  - Shared-memory comparable to OpenMP
  - Distributed-memory scales, albeit with loop startup overhead when written in global-view style

Next Steps

- Break leader into two steps to permit amortization of overheads
  - creation of parallelism vs. assignment of work
- Improve support for multidimensional iteration
  - works today, but produces suboptimal loop nests
- Support option to write standalone forall iterators
  - today, they use leader-follower interface which is overkill
- And several other things…

Our Team

- Cray:


Brad Chamberlain

Sung-Eun Choi

Greg Titus

Vass Litvinov

Tom Hildebrandt

- External Collaborators:


Albert Sidelnik

Jonathan Turner

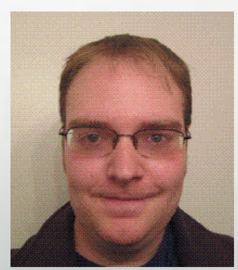
Angeles Navarro

●●●

You? Your Friend/Student/ Colleague?

- Interns:


Jonathan Claridge

Hannah Hemmaplardh

Andy Stone

Jim Dinan

Rob Bocchino

Mack Joyner

●●●

For More Information on Chapel

- **Chapel Home Page** (papers, presentations, tutorials):
  http://chapel.cray.com
- **Chapel Project Page** (releases, mailing lists, code):
  http://sourceforge.net/projects/chapel/
- **General Questions/Info:**
  chapel_info@cray.com (or SourceForge chapel-users list)
- **Upcoming Events:**
  SC11 (November, Seattle WA):
  - **Monday, Nov 14th:** full-day comprehensive Chapel tutorial
  - **Wednesday, Nov 16th:** BoF: Chapel Lightning Talks
  - **Friday, Nov 18th:** half-day outreach Chapel tutorial
  - **throughout:** PGAS booth

http://chapel.cray.com   chapel-info@cray.com   http://sourceforge.net/projects/chapel/