Hewlett Packard
Enterprise

COMPILING CHAPEL: KEYS TO MAKING PARALLEL PROGRAMMING PRODUCTIVE AT SCALE

Brad Chamberlain

PACT 2020 keynote
October 7, 2020

WHAT IS CHAPEL?

Chapel: A modern parallel programming language

- portable & scalable
- open-source & collaborative

Goals:

- Support general parallel programming
- Make parallel programming at scale far more productive

WHAT DOES "PRODUCTIVITY" MEAN TO YOU?

Recent Graduates:

"Something similar to what I used in school: Python, Matlab, Java, …"

Seasoned HPC Programmers:

"That sugary stuff which I can't use because I need full control to ensure good performance"

Computational Scientists:

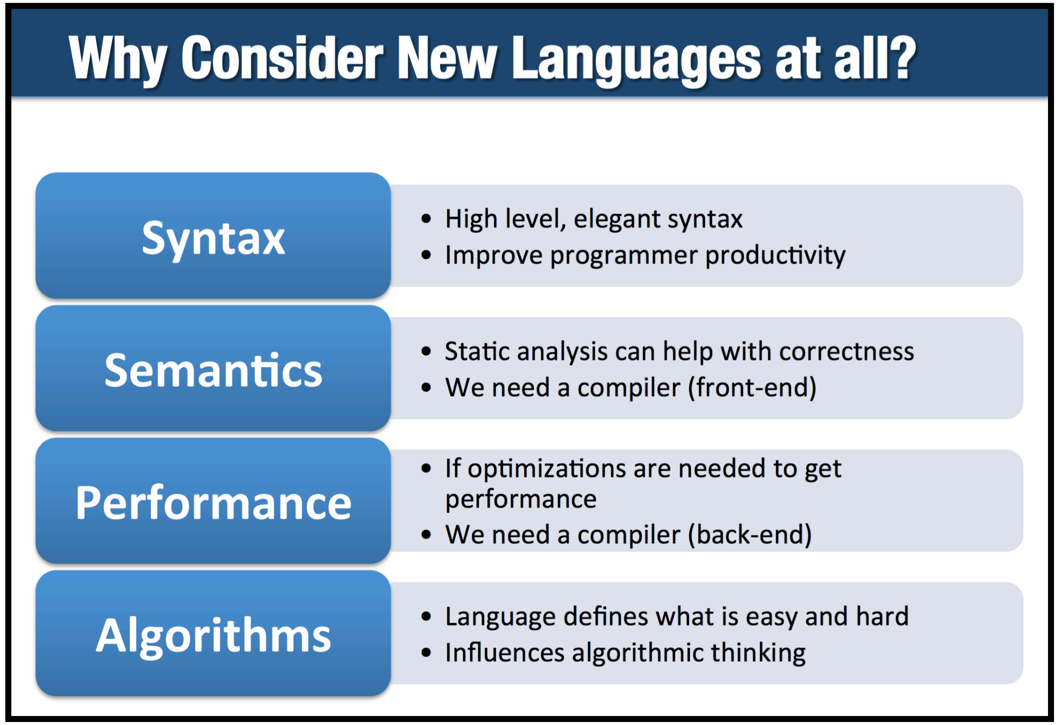
"Something that lets me focus on my science without having to wrestle with architecture-specific details"

Chapel Team:

"Something that lets computational scientists express what they want, without taking away the control that HPC programmers need, implemented in a language that's attractive to recent graduates."

WHY CREATE A NEW LANGUAGE?

- **Because parallel programmers deserve better**
  - the state of the art for HPC is a mish-mash of libraries, pragmas, and extensions
  - parallelism and locality are concerns that deserve first-class language features



[Image Source:
Kathy Yelick's (UC Berkeley, LBNL)
CHIUW 2018 keynote:
Why Languages Matter More Than Ever,
used with permission]

- **And because existing languages don't fit our desires…**

CHAPEL GOALS, RELATIVE TO OTHER LANGUAGES

Chapel aims to be as...

...**programmable** as Python

...**fast** as Fortran

...**scalable** as MPI, SHMEM, or UPC

...**portable** as C

...**flexible** as C++

...**fun** as [your favorite programming language]

THIS TALK VS. TYPICAL CHAPEL TALKS

My typical Chapel talks:

- summarize Chapel's features and accomplishments, with the goal of growing the community

This talk:

- will do a bit of the above…
- but also give you more insights into Chapel's compilation and optimization

Thesis: well-designed languages can improve user productivity <u>and</u> help with compilation

OUTLINE

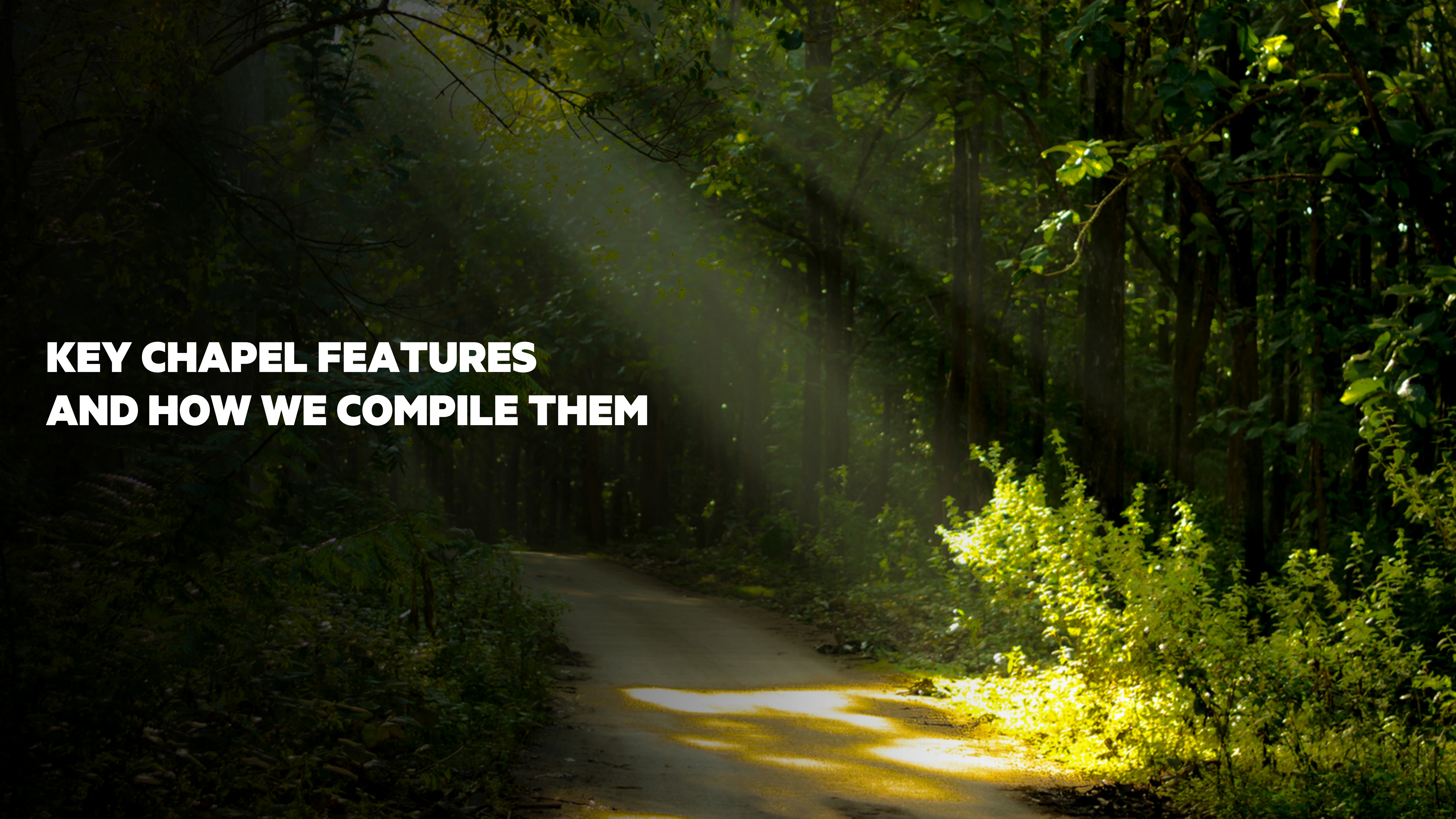
I. Chapel Context & Motivation

II. Compiling Chapel Features
(“nuts and bolts”)

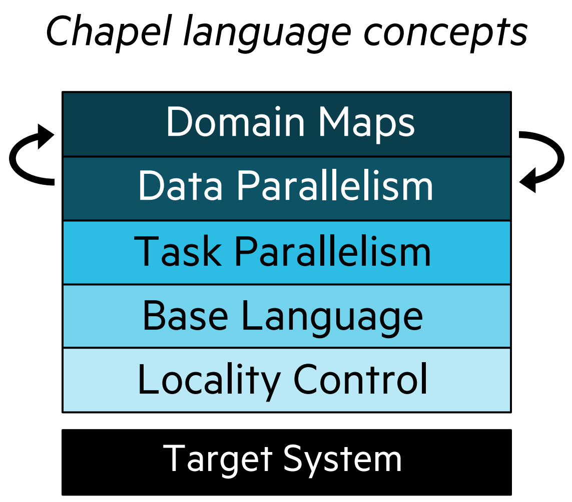
III. Some Chapel Optimizations
(“bells and whistles”)

IV. Summary and Resources

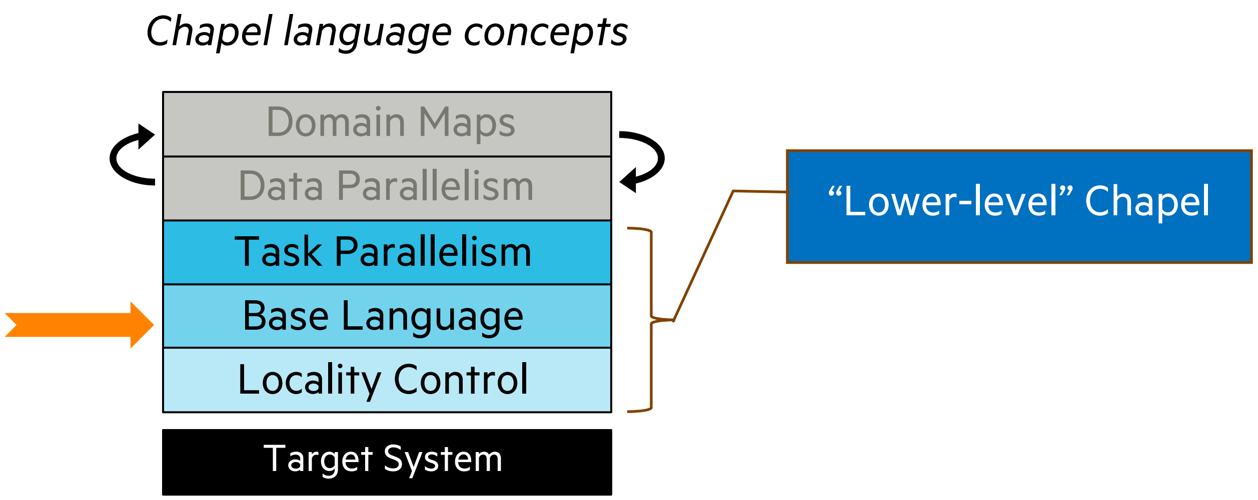
KEY CHAPEL FEATURES AND HOW WE COMPILE THEM



CHAPEL FEATURE AREAS



BASE LANGUAGE



FIBONACCI ITERATION

fib.chpl

```
config const n = 10;

for f in fib(n) do
  writeln(f);

iter fib(x) {
  var current = 0,
      next = 1;

  for i in 1..x {
    yield current;
    current += next;
    current <=> next;
  }
}
```

FIBONACCI ITERATION

fib.chpl

```
config const n = 10;

for f in fib(n) do
  writeln(f);

iter fib(x) {
  var current = 0,
      next = 1;

  for i in 1..x {
    yield current;
    current += next;
    current <=> next;
  }
}
```

Drive this loop
by invoking fib(n)

‘yield’ binds values back
to the loop’s index variable

FIBONACCI ITERATION

fib.chpl

```
config const n = 10;

for f in fib(n) do
  writeln(f);

iter fib(x) {
  var current = 0,
      next = 1;

  for i in 1..x {
    yield current;
    current += next;
    current <=> next;
  }
}
```

And executes the loop's body for that value

'yield' binds values back to the loop's index variable

FIBONACCI ITERATION

fib.chpl

```
config const n = 10;

for f in fib(n) do
  writeln(f);

iter fib(x) {
  var current = 0,
      next = 1;

  for i in 1..x {
    yield current;
    current += next;
    current <=> next;
  }
}
```

And executes the loop's body for that value

Then the iterator continues from where it left off

Until we fall out of it (or return)

```
prompt> chpl fib.chpl
prompt> ./fib --n=20
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
…
```

FIBONACCI ITERATION

fib.chpl

```
config const n = 10;

for f in fib(n) do
  writeln(f);

iter fib(x) {
  var current = 0,
      next = 1;

  for i in 1..x {
    yield current;
    current += next;
    current <=> next;
  }
}
```

```
prompt> chpl fib.chpl
prompt> ./fib --n=20
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
…
```

IMPLEMENTING SERIAL ITERATOR LOOPS

fib.chpl

```
config const n = 10;

for f in fib(n) do
  writeln(f);

iter fib(x) {
  var current = 0,
      next = 1;

  for i in 1..x {
    yield current;
    current += next;
    current <=> next;
  }
}
```

Rewrite the loop
by inlining the iterator

```
{
  const x = n;
  var current = 0,
      next = 1;

  for i in 1..x {
    yield current;
    current += next;
    current <=> next;
  }
}
```

Then rewrite the 'yield'
by inlining the loop's body

```
{
  const x = n;
  var current = 0,
      next = 1;

  for i in 1..x {
    {
      const f = current;
      writeln(f);
    }
    current += next;
    current <=> next;
  }
}
```

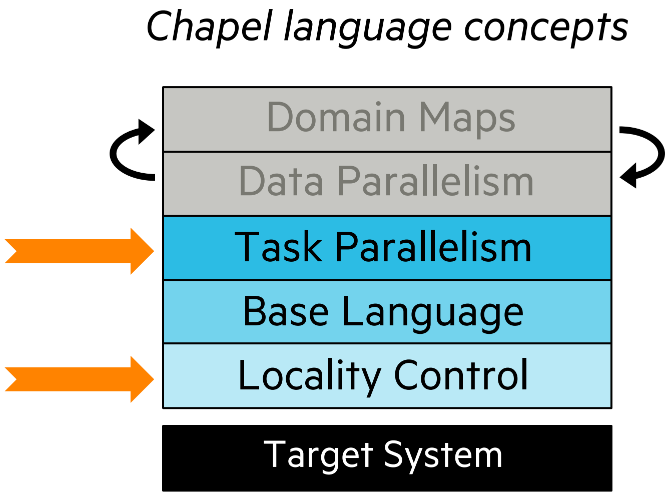
More Advanced Cases
(not covered today)

- zippered iteration
- recursive iterators

OTHER BASE LANGUAGE FEATURES

- **Object-oriented programming** (value- and reference-based)
  - Nilable vs. non-nilable class variables
  - Memory-managed objects
  - Lifetime checking
- **Generic programming / polymorphism**
- **Error-handling**
- **Compile-time meta-programming**
- **Modules** (supporting namespaces)
- **Procedure overloading / filtering**
- **Arguments:** default values, intents, name-based matching, type queries
- and more…

TASK PARALLELISM AND LOCALITY CONTROL



CHAPEL TERMINOLOGY: LOCALES

- Locales can run tasks and store variables
  - Think “compute node” on a parallel system
  - User specifies number of locales on executable’s command-line

```
prompt> ./myChapelProgram --numLocales=4    # or `-nl 4`
```

`Locales` array:

TASK-PARALLEL "HELLO WORLD"

helloTaskPar.chpl

```
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
  writef("Hello from task %n of %n on %s\n",
         tid, numTasks, here.name);
```

TASK-PARALLEL “HELLO WORLD”

helloTaskPar.chpl

```
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
  writef("Hello from task %n of %n on %s\n",
         tid, numTasks, here.name);
```

‘here’ refers to the locale on which we’re currently running

how many processing units (think “cores”) does my locale have?

what’s my locale’s name?

TASK-PARALLEL "HELLO WORLD"

helloTaskPar.chpl

```
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
  writef("Hello from task %n of %n on %s\n",
         tid, numTasks, here.name);
```

a 'coforall' loop executes each iteration as an independent task

```
prompt> chpl helloTaskPar.chpl
prompt> ./helloTaskPar
Hello from task 1 of 4 on n1032
Hello from task 4 of 4 on n1032
Hello from task 3 of 4 on n1032
Hello from task 2 of 4 on n1032
```

TASK-PARALLEL "HELLO WORLD"

helloTaskPar.chpl

```
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
  writef("Hello from task %n of %n on %s\n",
         tid, numTasks, here.name);
```

```
prompt> chpl helloTaskPar.chpl
prompt> ./helloTaskPar
Hello from task 1 of 4 on n1032
Hello from task 4 of 4 on n1032
Hello from task 3 of 4 on n1032
Hello from task 2 of 4 on n1032
```

So far, this is a shared-memory program

Nothing refers to remote locales, explicitly or implicitly

TASK-PARALLEL "HELLO WORLD"

helloTaskPar.chpl

```
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
  writef("Hello from task %n of %n on %s\n",
         tid, numTasks, here.name);
```

TASK-PARALLEL "HELLO WORLD" (DISTRIBUTED VERSION)

helloTaskPar.chpl

```
coforall loc in Locales {
  on loc {
    const numTasks = here.numPUs();
    coforall tid in 1..numTasks do
      writef("Hello from task %n of %n on %s\n",
             tid, numTasks, here.name);
  }
}
```

TASK-PARALLEL "HELLO WORLD" (DISTRIBUTED VERSION)

helloTaskPar.chpl

```
coforall loc in Locales {
  on loc {
    const numTasks = here.numPUs();
    coforall tid in 1..numTasks do
      writef("Hello from task %n of %n on %s\n",
             tid, numTasks, here.name);
  }
}
```

create a task per locale on which the program is running

have each task run 'on' its locale

then print a message per core, as before

```
prompt> chpl helloTaskPar.chpl
prompt> ./helloTaskPar -numLocales=4
Hello from task 1 of 4 on n1032
Hello from task 4 of 4 on n1032
Hello from task 1 of 4 on n1034
Hello from task 2 of 4 on n1032
Hello from task 1 of 4 on n1033
Hello from task 3 of 4 on n1034
Hello from task 1 of 4 on n1035
…
```

IMPLEMENTING COFORALL LOOPS

helloTaskPar.chpl

```
coforall loc in Locales {
  on loc {
    const numTasks = here.numPUs();
    coforall tid in 1..numTasks do
      writef("Hello from task %n of %n on %s\n",
                    tid, numTasks, here.name);
  }
}
```

Rewrite the loop as a serial loop that captures outer-scope variables and creates a task in each iteration

```
for tid in 1..numTasks {
  const argBundle = captureVars(…);
  create_task(code=coforallTask,
                    args=argBundle);
}
```

Outline the loop body

```
proc coforallTask(argBundle) {
  writef("Hello from task %n of %n on %s\n",
              argBundle.tid, argBundle.numTasks, here.name);
}
```

IMPLEMENTING COFORALL LOOPS

helloTaskPar.chpl

```
coforall loc in Locales {
  on loc {
    const numTasks = here.numPUs();
    coforall tid in 1..numTasks do
      writef("Hello from task %n of %n on %s\n",
                tid, numTasks, here.name);
  }
}
```

IMPLEMENTING ON-CLAUSES

helloTaskPar.chpl

```
coforall loc in Locales {
  on loc {
    const numTasks = here.numPUs();
    coforall tid in 1..numTasks do
      writef("Hello from task %n of %n on %s\n",
                tid, numTasks, here.name);
  }
}
```

Outline the body of the on-clause

Rewrite the on-clause as an active message

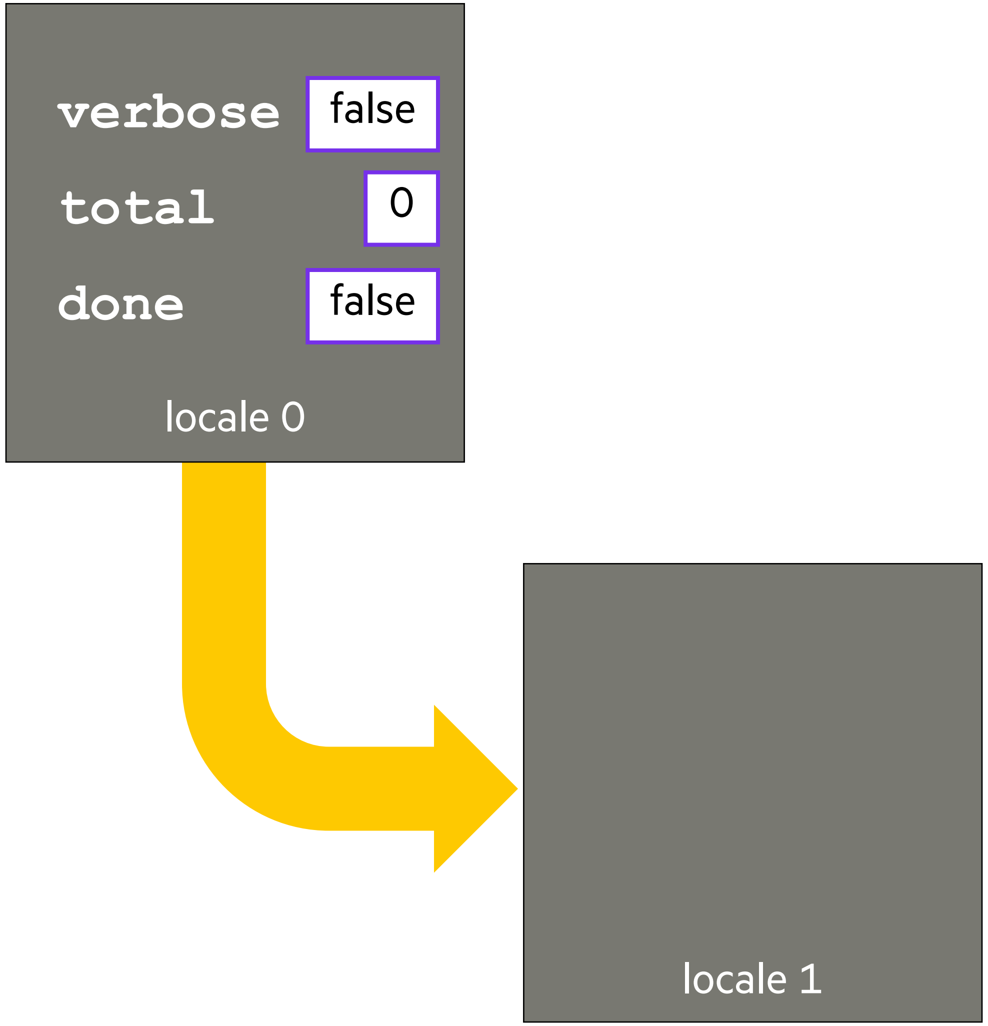
```
const argBundle = captureVars(…);
create_active_msg(locale=loc,
                  code=onTask,
                  args=argBundle);
```

```
proc onTask(argBundle) {
  const numTasks = here.numPUs();
  coforall tid in 1..numTasks do
    writef("Hello from task %n of %n on %s\n",
              tid, numTasks, here.name);
  }
}
```

CAPTURING VARIABLES FOR ON-CLAUSES

onClause.chpl

```
config const verbose = false;
var total = 0,
    done = false;

…

on Locales[1] {
  if !done {
    if verbose then
      writef("Adding locale 1's contribution");
    total += computeMyContribution();
  }
}
```

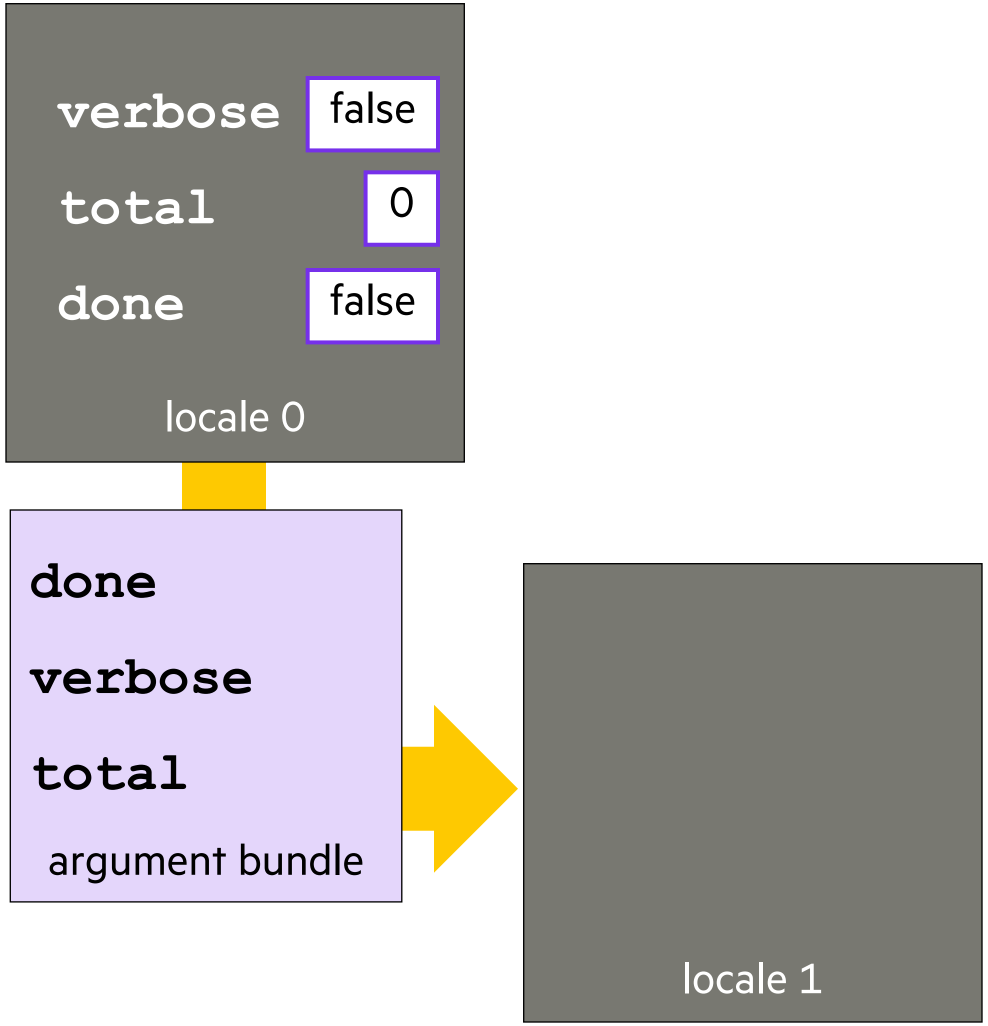


CAPTURING VARIABLES FOR ON-CLAUSES

onClause.chpl

```
config const verbose = false;
var total = 0,
    done = false;

…

on Locales[1] {
  if !done {
    if verbose then
      writef("Adding locale 1's contribution");
    total += computeMyContribution();
  }
}
```

Which outer-scope variables are referenced?



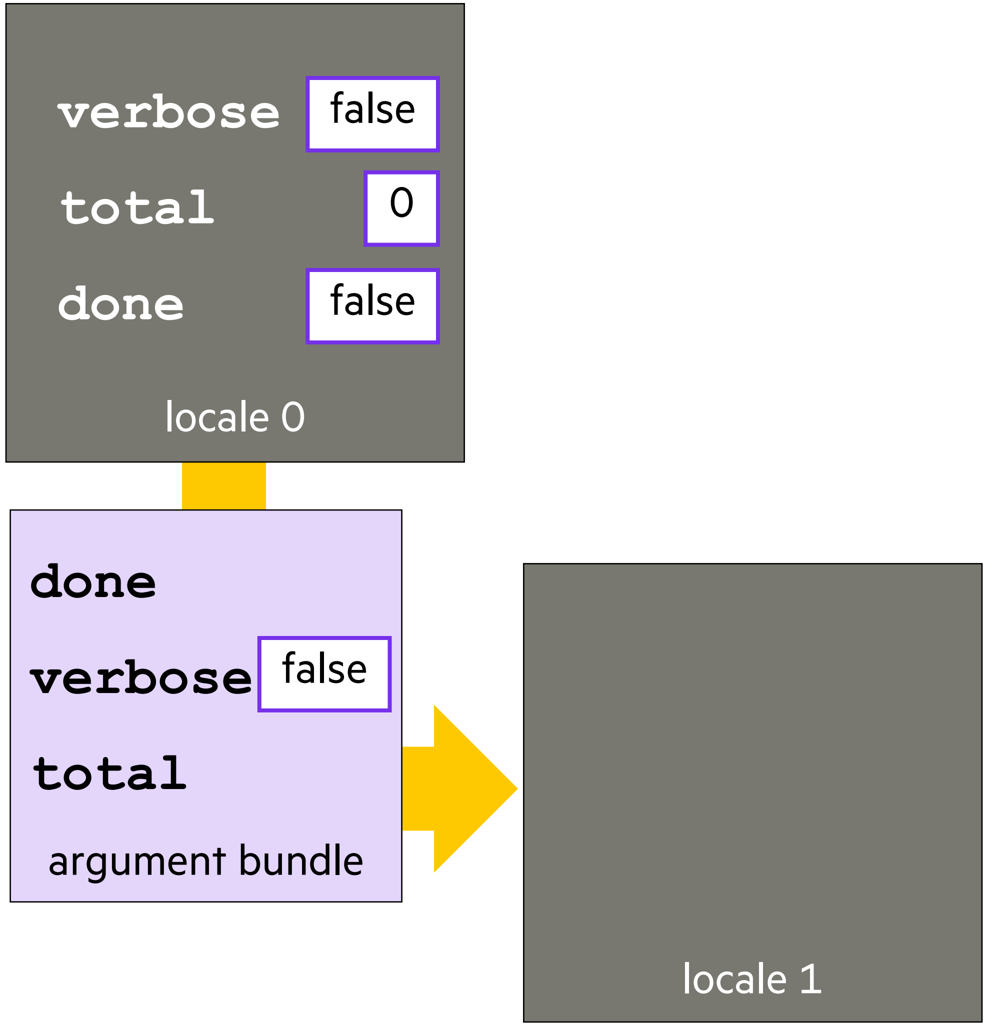
CAPTURING VARIABLES FOR ON-CLAUSES

onClause.chpl

```
config const verbose = false;
var total = 0,
    done = false;

…

on Locales[1] {
  if !done {
    if verbose then
      writef("Adding locale 1's contribution");
    total += computeMyContribution();
  }
}
```

Which outer-scope variables are referenced?

Which are constant? Can capture these by value.



CAPTURING VARIABLES FOR ON-CLAUSES

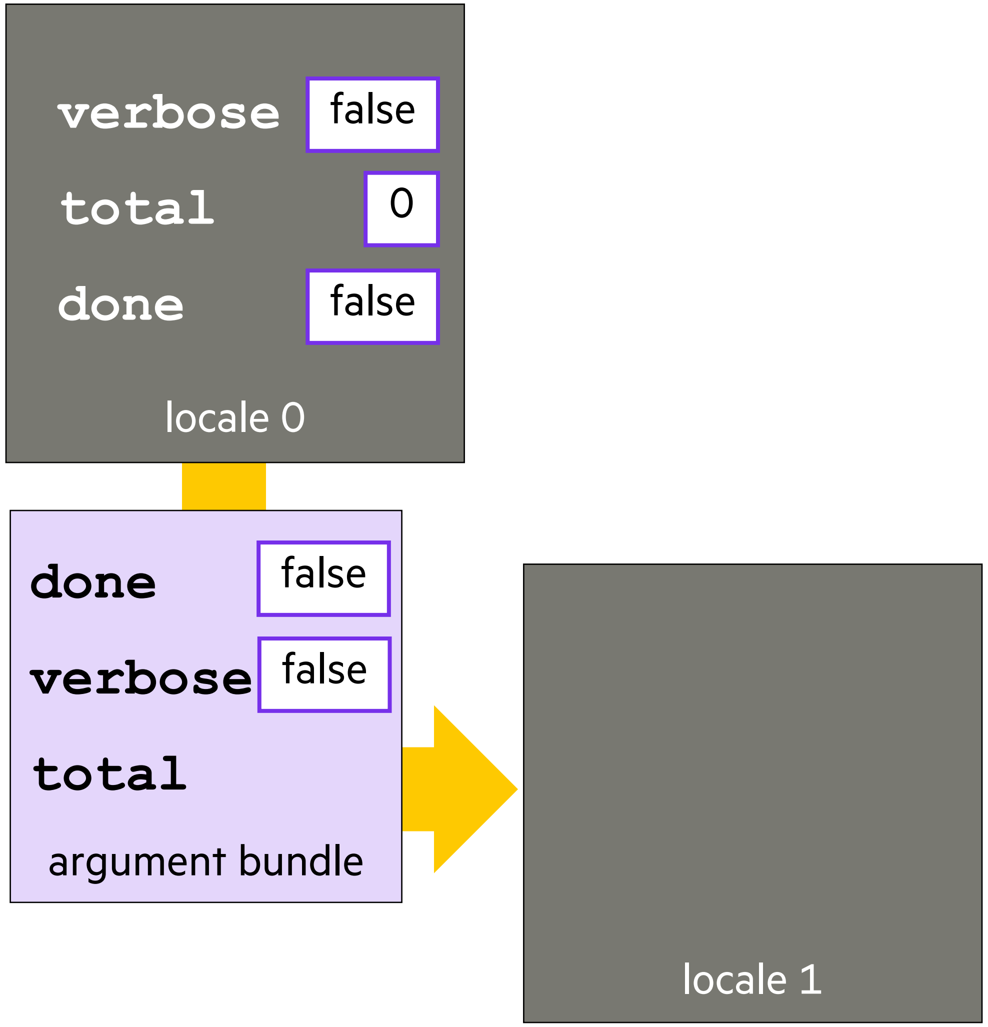
onClause.chpl

```
config const verbose = false;
var total = 0,
    done = false;

…

on Locales[1] {
  if !done {
    if verbose then
      writef("Adding locale 1's contribution");
    total += computeMyContribution();
  }
}
```

Which outer-scope variables are referenced?

Which are constant? Can capture these by value.

Which are MCM-safe to cache? Can capture these by value.



MCM = Memory Consistency Model

CAPTURING VARIABLES FOR ON-CLAUSES

onClause.chpl

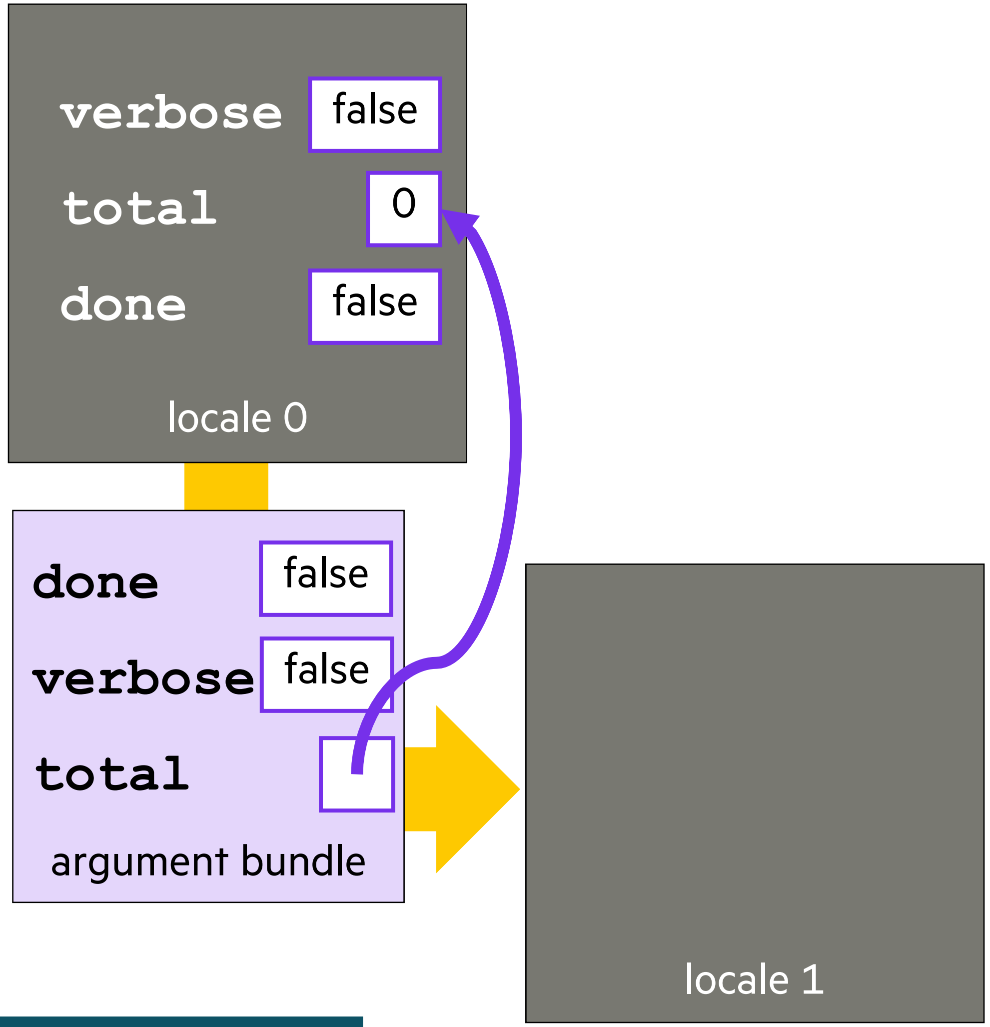
```
config const verbose = false;
var total = 0,
    done = false;

…

on Locales[1] {
  if !done {
    if verbose then
      writef("Adding locale 1's contribution");
    total += computeMyContribution();
  }
}
```

Which outer-scope variables are referenced?

Which are constant? Can capture these by value.

Which are MCM-safe to cache? Can capture these by value.

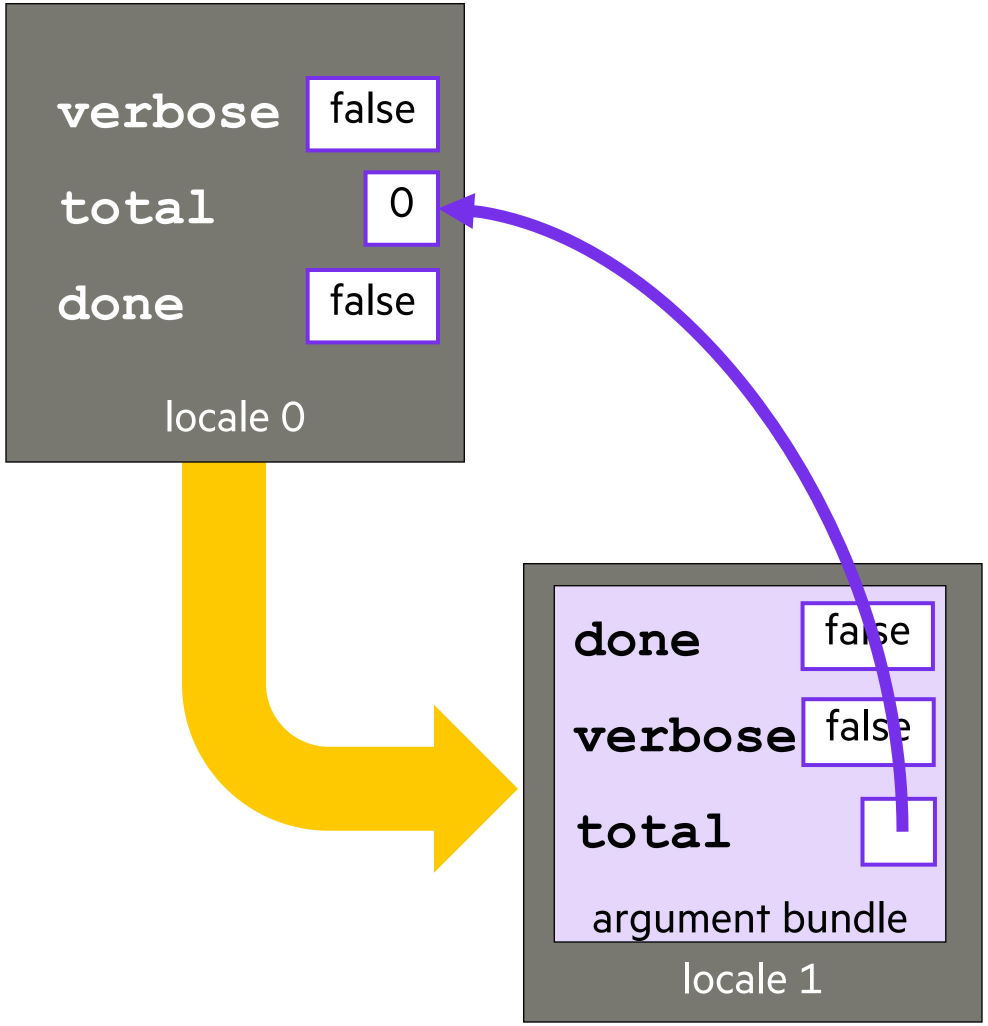
What remains? Have these refer to the original, requiring R[D]MA to access.



CAPTURING VARIABLES FOR ON-CLAUSES

onClause.chpl

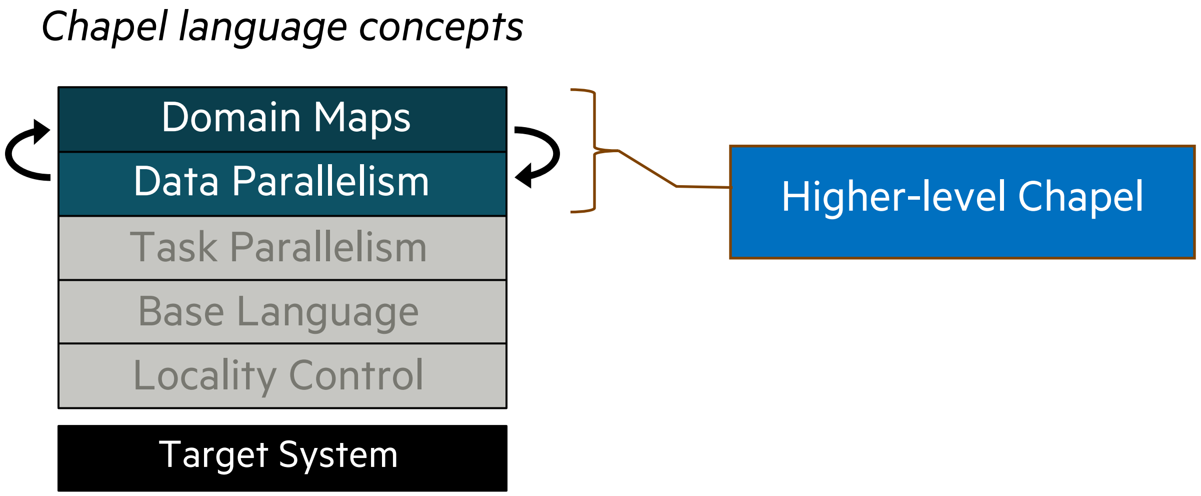
```
config const verbose = false;
var total = 0,
    done = false;

…

on Locales[1] {
  if !done {
    if verbose then
      writef("Adding locale 1’s contribution");
    total += computeMyContribution();
  }
}
```



OTHER TASK PARALLEL FEATURES

- **begin / cobegin statements:** other ways of creating tasks
- **atomic / synchronized variables:** for sharing data & coordinating between tasks
- **task intents / task-private variables:** ways of controlling how variables relate to tasks

DATA PARALLELISM AND DOMAIN MAPS



DATA-PARALLEL ARRAY FILL

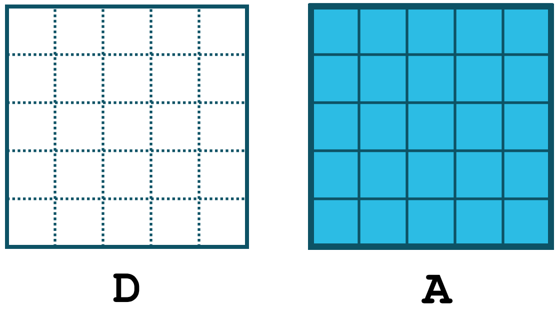
fillArray.chpl

```
config const n = 1000;

var D = {1..n, 1..n};

var A: [D] real;

forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;

writeln(A);
```

DATA-PARALLEL ARRAY FILL

fillArray.chpl

```
config const n = 1000;

var D = {1..n, 1..n};

var A: [D] real;

forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;

writeln(A);
```



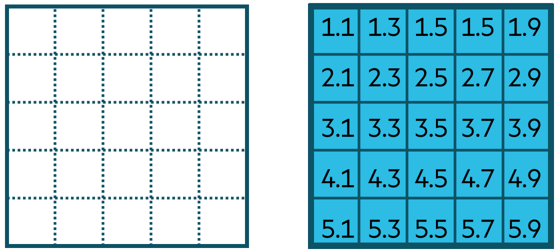
declare a domain, a first-class index set

declare an array over that domain

DATA-PARALLEL ARRAY FILL

fillArray.chpl

```
config const n = 1000;

var D = {1..n, 1..n};

var A: [D] real;

forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;

writeln(A);
```



D A

declare a domain, a first-class index set

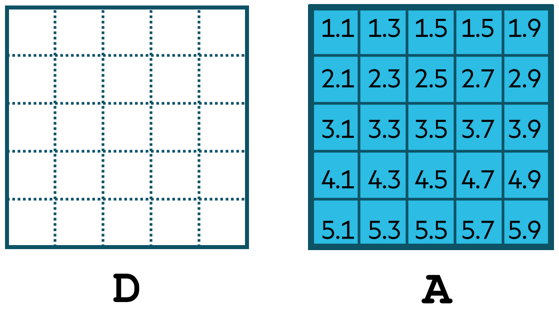
declare an array over that domain

iterate over the domain's indices in parallel, assigning to the corresponding array elements

DATA-PARALLEL ARRAY FILL

fillArray.chpl

```
config const n = 1000;

var D = {1..n, 1..n};

var A: [D] real;

forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;

writeln(A);
```




```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

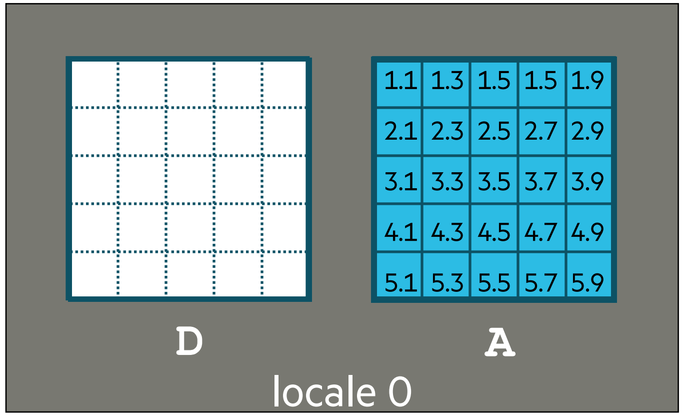
So far, this is a shared-memory program

Nothing refers to remote locales, explicitly or implicitly

DATA-PARALLEL ARRAY FILL

fillArray.chpl

```
config const n = 1000;

var D = {1..n, 1..n};

var A: [D] real;

forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;

writeln(A);
```




```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

So far, this is a shared-memory program

Nothing refers to remote locales, explicitly or implicitly

DATA-PARALLEL ARRAY FILL

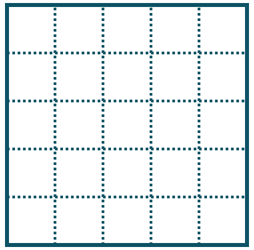
fillArray.chpl

```
config const n = 1000;

var D = {1..n, 1..n};

var A: [D] real;

forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;

writeln(A);
```

DATA-PARALLEL ARRAY FILL (DISTRIBUTED VERSION)

fillArray.chpl

```
use CyclicDist;

config const n = 1000;

var D = {1..n, 1..n}
          dmapped Cyclic(startIdx = (1,1));
var A: [D] real;

forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;

writeln(A);
```



D

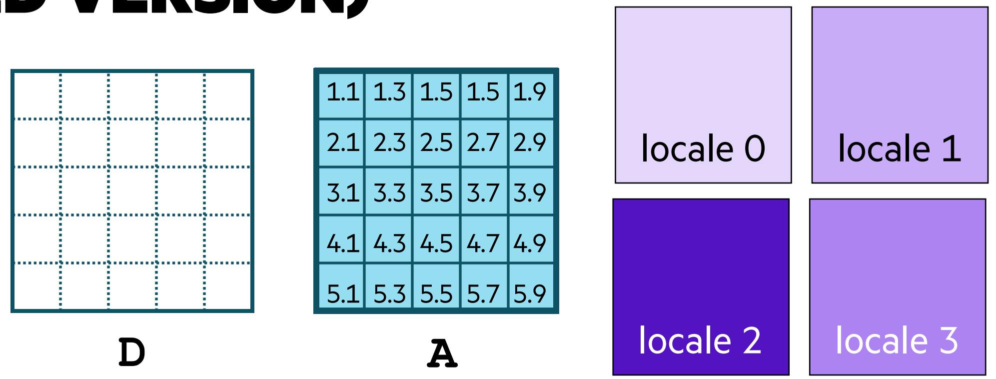
1.1	1.3	1.5	1.5	1.9
2.1	2.3	2.5	2.7	2.9
3.1	3.3	3.5	3.7	3.9
4.1	4.3	4.5	4.7	4.9
5.1	5.3	5.5	5.7	5.9

A

DATA-PARALLEL ARRAY FILL (DISTRIBUTED VERSION)

fillArray.chpl

```
use CyclicDist;

config const n = 1000;

var D = {1..n, 1..n}
        dmapped Cyclic(startIdx = (1,1));
var A: [D] real;

forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;

writeln(A);
```



apply a domain map, specifying how to implement…

…the domain's indices,

…the array's elements,

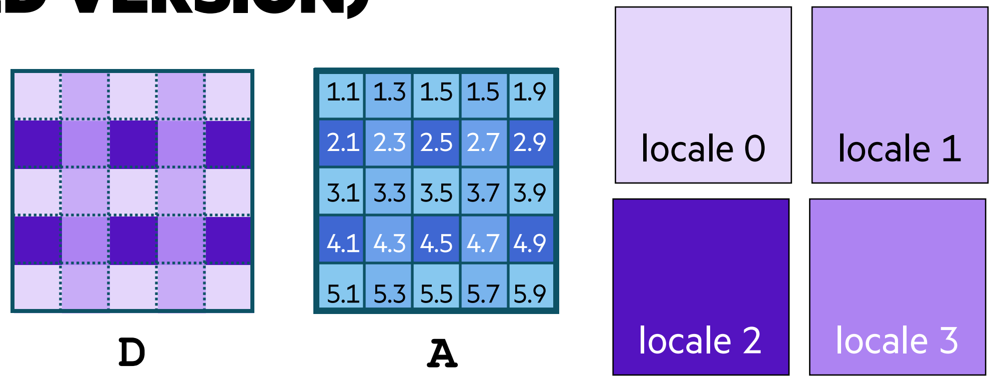
…the loop's iterations,

…on the program's locales

DATA-PARALLEL ARRAY FILL (DISTRIBUTED VERSION)

fillArray.chpl

```
use CyclicDist;

config const n = 1000;

var D = {1..n, 1..n}
          dmapped Cyclic(startIdx = (1,1));
var A: [D] real;

forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;

writeln(A);
```



apply a domain map, specifying how to implement…

…the domain's indices,

…the array's elements,

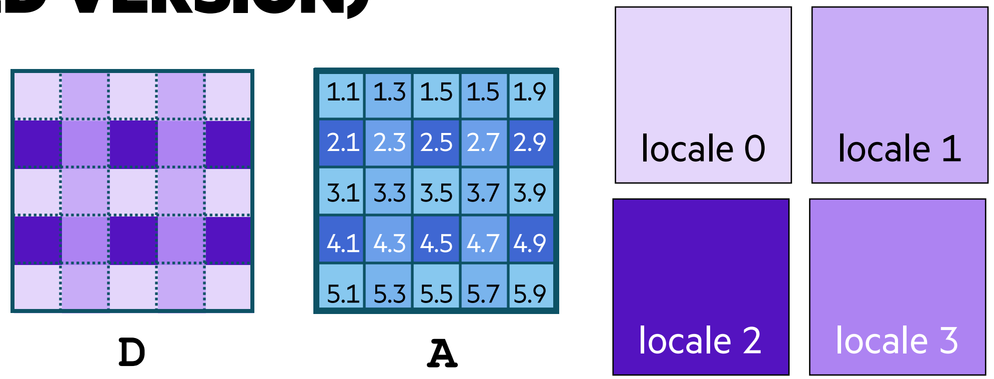
…the loop's iterations,

…on the program's locales

DATA-PARALLEL ARRAY FILL (DISTRIBUTED VERSION)

fillArray.chpl

```
use CyclicDist;

config const n = 1000;

var D = {1..n, 1..n}
          dmapped Cyclic(startIdx = (1,1));
var A: [D] real;

forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;

writeln(A);
```

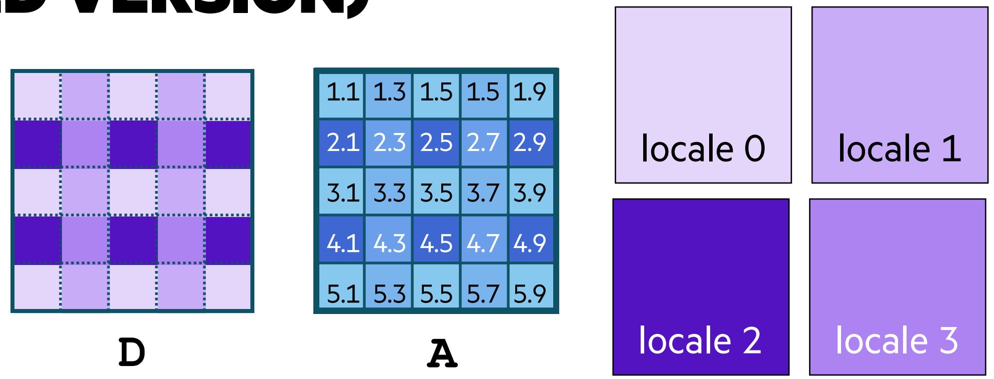



```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

DATA-PARALLEL ARRAY FILL (DISTRIBUTED VERSION)

fillArray.chpl

```
use CyclicDist;

config const n = 1000;

var D = {1..n, 1..n}
          dmapped Cyclic(startIdx = (1,1));
var A: [D] real;

forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;

writeln(A);
```




```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5 --numLocales=4
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

DATA-PARALLEL ARRAY FILL (DISTRIBUTED VERSION)

fillArray.chpl

```
use CyclicDist;

config const n = 1000;

var D = {1..n, 1..n}
        dmapped Cyclic(startIdx = (1,1));
var A: [D] real;

forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;

writeln(A);
```

IMPLEMENTING DATA PARALLELISM IN CHAPEL

fillArray.chpl

```
…
var D = {1..n, 1..n}
          dmapped Cyclic(…=(1,1));
var A: [D] real;

forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
…
```

IMPLEMENTING DATA PARALLELISM IN CHAPEL

fillArray.chpl

```
…
var D = {1..n, 1..n}
        dmapped Cyclic(…=(1,1));
var A: [D] real;

forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
…
```

Data-parallel features are lowered to method calls on domain map objects

```
const Dmap = new Cyclic(startIdx=(1,1));
var D = Dmap.newDomain(idxs={1..n, 1..n});
var A = D.newArray(eltType=real);

for (i,j) in D.defaultParIterator() do
  A[i,j] = i + (j - 0.5)/n;
```

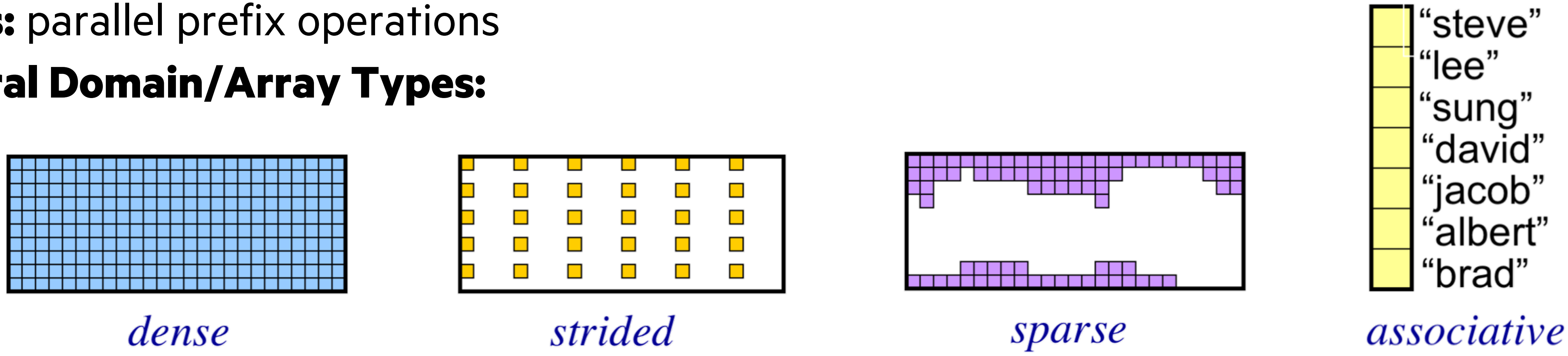
Domain maps are written in Chapel using "lower-level" features

- objects, methods, tasks, on-clauses, …

```
iter CyclicDom.defaultParIterator() {
  coforall loc in this.targetLocales do
    on loc do
      coforall tid in 1..here.numPUs() do
        for idx in myInds(loc, tid, …) do
          yield idx;
}
```

OTHER DATA PARALLEL FEATURES

- **Parallel Iterators and Zippering**
- **Slicing:** refer to subarrays using ranges / domains
- **Promotion:** execute scalar functions in parallel using array arguments
- **Reductions:** collapse arrays to scalars or subarrays
- **Scans:** parallel prefix operations
- **Several Domain/Array Types:**



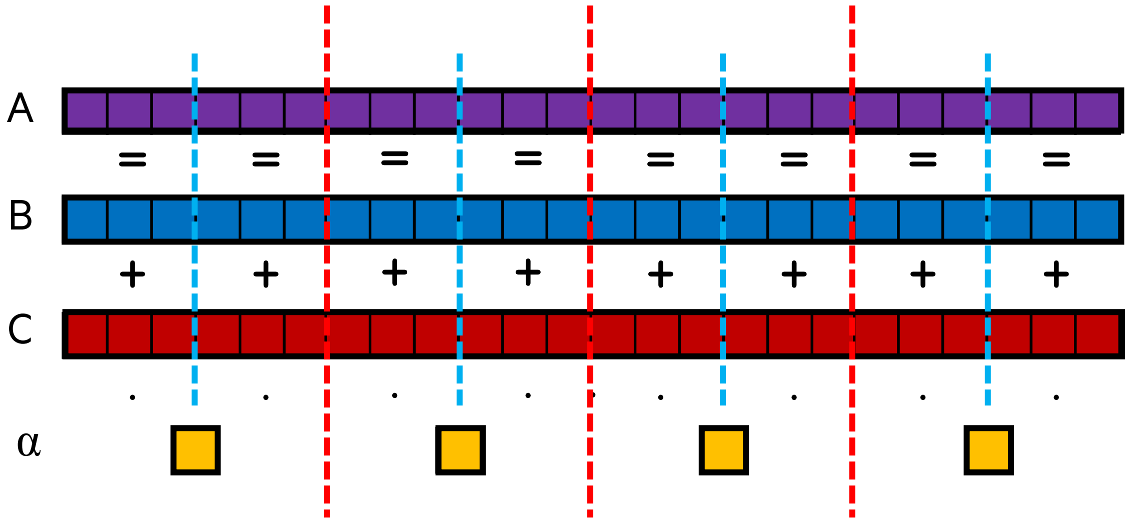
CHAPEL-ENABLED OPTIMIZATIONS

STREAM TRIAD: A TRIVIAL PARALLEL COMPUTATION

Given: m-element vectors A, B, C

Compute: $\forall i \in 1..m,\ A_i = B_i + \alpha \cdot C_i$

In pictures, in parallel (distributed memory multicore):



STREAM TRIAD: C + MPI + OPENMP

```c
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Comm_size( comm, &commSize );
  MPI_Comm_rank( comm, &myRank );

  rv = HPCC_Stream( params, 0 == myRank);
  MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm );

  return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
  register int j;
  double  scalar;

  VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(double), 0 );

  a = HPCC_XMALLOC( double, VectorSize );
  b = HPCC_XMALLOC( double, VectorSize );
  c = HPCC_XMALLOC( double, VectorSize );
```

```c
  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
      fprintf( outFile, "Failed to allocate memory (%d).\n", VectorSize );
      fclose( outFile );
    }
    return 1;
  }

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 1.0;
  }
  scalar = 3.0;

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];

  HPCC_free(c);
  HPCC_free(b);
  HPCC_free(a);

  return 0;
}
```

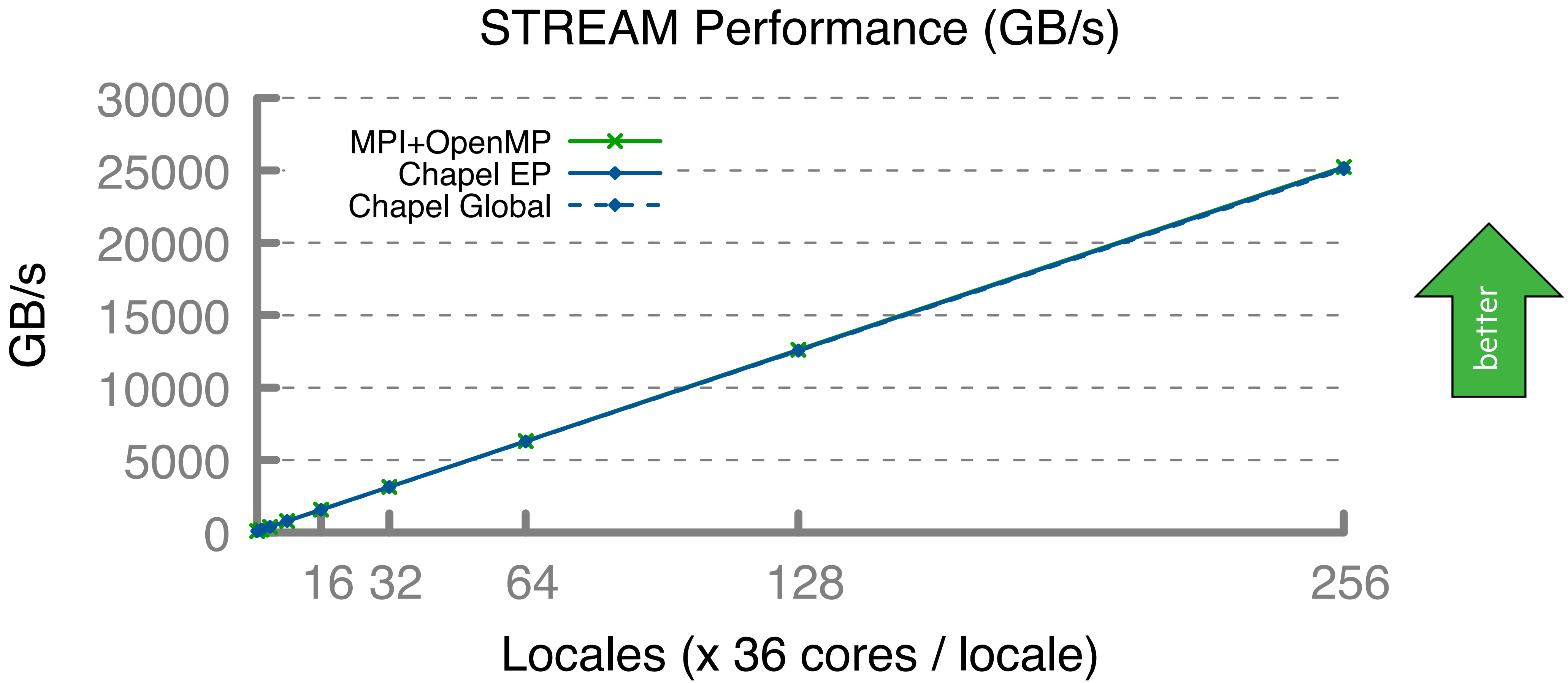
STREAM TRIAD: CHAPEL (GLOBAL, PROMOTED VERSION)

```
use BlockDist;

config const m = 1000,
             alpha = 3.0;

const Dom = {1..m} dmapped Block({1..m});

var A, B, C: [Dom] real;

B = 2.0;
C = 1.0;

A = B + alpha * C;
```

```
int HPCC_Stream(HPCC_Params *params, int doIO) {
  register int j;
  double  scalar;

  VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(double), 0 );

  a = HPCC_XMALLOC( double, VectorSize );
  b = HPCC_XMALLOC( double, VectorSize );
  c = HPCC_XMALLOC( double, VectorSize );
```

```
  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
      fprintf( outFile, "Failed to allocate memory (%d).\n", VectorSize );
      fclose( outFile );
    }
    return 1;
  }

ifdef _OPENMP
pragma omp parallel for
endif
  for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 1.0;
  }
  scalar = 3.0;

ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];

  HPCC_free(c);
  HPCC_free(b);
  HPCC_free(a);

  return 0;
}
```

STREAM TRIAD: CHAPEL VS. C+MPI+OPENMP



STREAM TRIAD: CHAPEL (GLOBAL, RANDOM ACCESS VERSION)

```
use BlockDist;

config const m = 1000,
             alpha = 3.0;

const Dom = {1..m} dmapped Block({1..m});

var A, B, C: [Dom] real;

B = 2.0;
C = 1.0;

// was: A = B + alpha * C;
forall i in Dom do
  A[i] = B[i] + alpha * C[i];
```

```
if (!a || !b || !c) {
  if (c) HPCC_free(c);
  if (b) HPCC_free(b);
  if (a) HPCC_free(a);
  if (doIO) {
    fprintf( outFile, "Failed to allocate memory (%d).\n", VectorSize );
    fclose( outFile );
  }
  return 1;
}

ifdef _OPENMP
pragma omp parallel for
endif
for (j=0; j<VectorSize; j++) {
  b[j] = 2.0;
  c[j] = 1.0;
```

```
proc BlockArr.randomAccess(idx) ref {
  if this.domain.isLocal(idx) then
    return this.localAccess(idx);
  }
  const remoteLoc = this.domain.idxToLocale(idx);
  return computeRefToRemoteElt(remoteLoc, idx);
}
```

```
VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(

a = HPCC_XMALLOC( double, VectorSize );
b = HPCC_XMALLOC( double, VectorSize );
c = HPCC_XMALLOC( double, VectorSize );
```

Lots of overhead, given that we know all accesses are local

STREAM TRIAD: CHAPEL (GLOBAL, LOCAL ACCESS VERSION)

```
use BlockDist;

config const m = 1000,
             alpha = 3.0;

const Dom = {1..m} dmapped Block({1..m});

var A, B, C: [Dom] real;

B = 2.0;
C = 1.0;

// was: A = B + alpha * C;
forall i in Dom do
  A.localAccess[i] = B.localAccess[i] + alpha * C.localAccess[i];
```

As fast as the promoted version, but… annoying

```
memory (%d).\n", VectorSize );
VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(double

a = HPCC_XMALLOC( double, VectorSize );
b = HPCC_XMALLOC( double, VectorSize );
c = HPCC_XMALLOC( double, VectorSize );

HPCC_free(b);
HPCC_free(a);

return 0;
}
```

AUTOMATIC LOCAL ACCESS OPTIMIZATION

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

forall i in Dom do
  A[i] = B[i] + alpha * C[i];
```

The compiler knows that…

…*Dom* is the domain of *A*, *B*, and *C*

…*i* is an index from *Dom*

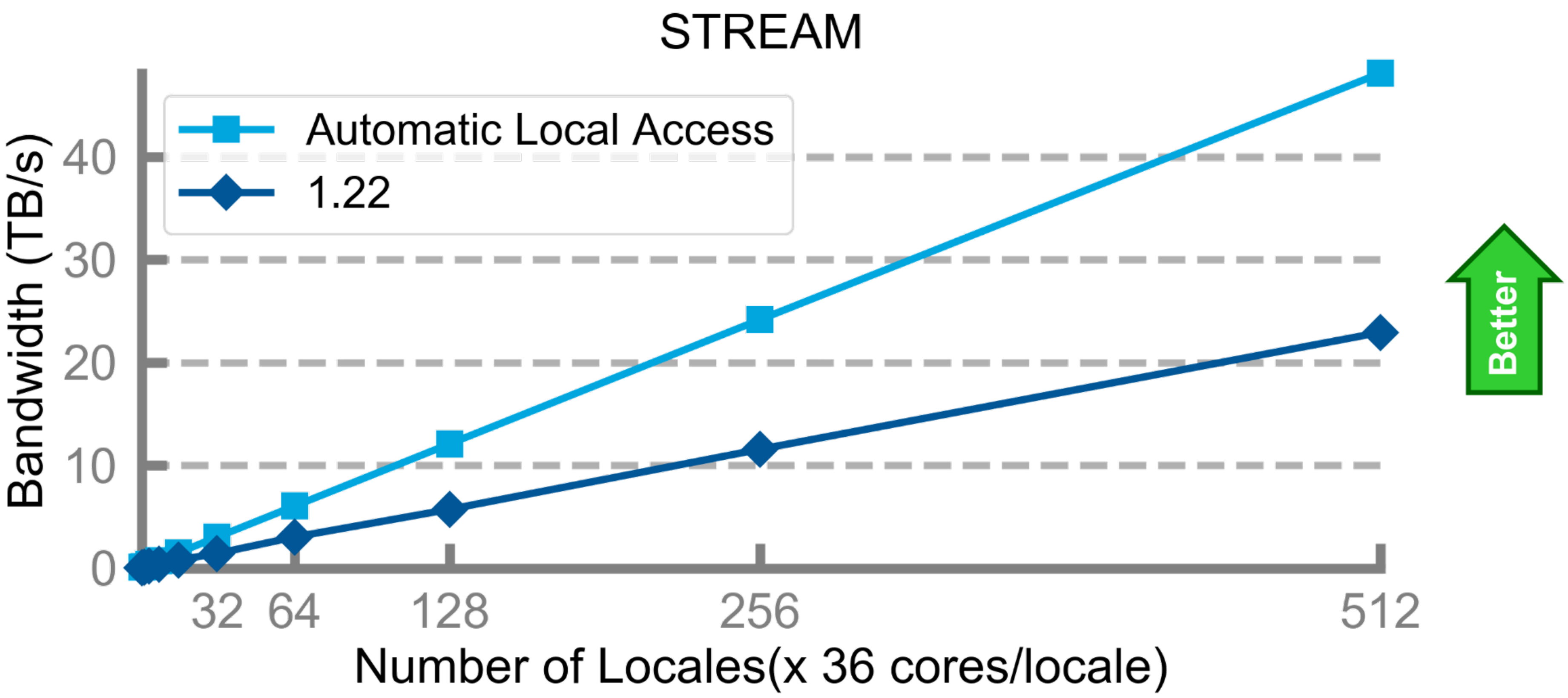
…so, it knows A[i], B[i], C[i] are local to the task that owns iteration *i*

…and can therefore strength-reduce the loop to:

```
forall i in Dom do
  A.localAccess[i] = B.localAccess[i] + alpha * C.localAccess[i];
```

- This optimization has been planned since the dawn of Chapel in the 2000's
  - motivated by work on ZPL in the 1990's
- Implemented by Engin Kayraklioglu in June 2020
  - also supports several less-obvious variations

PERFORMANCE DUE TO AUTO-LOCAL-ACCESS OPTIMIZATION



AUTOMATIC LOCAL ACCESS OPTIMIZATION

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

A = B + alpha * C;
```

=

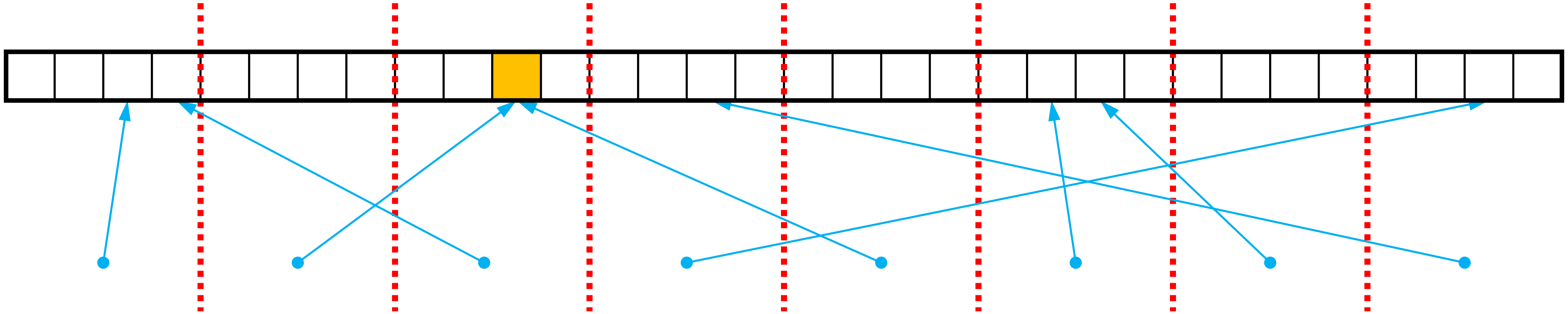
```
var A, B, C: [Dom] real;

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

=

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

forall i in Dom do
  A[i] = B[i] + alpha * C[i];
```

HPCC RANDOM ACCESS (RA)

Data Structure: distributed table



Computation: update random table locations in parallel

HPCC RA: MPI KERNEL

```
/* Perform updates to main table.  The scalar equivalent is:
 *
 *   for (i=0; i<NUPDATE; i++) {
 *     Ran = (Ran << 1) ^ (((s64Int) Ran < 0) ? POLY : 0);
 *     Table[Ran & (TABSIZE-1)] ^= Ran;
 *   }
 */

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
          MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
 while (i < SendCnt) {
   /* receive messages */
   do {
     MPI_Test(&inreq, &have_done, &status);
     if (have_done) {
       if (status.MPI_TAG == UPDATE_TAG) {
         MPI_Get_count(&status, tparams.dtype64, &recvUpdates);
         bufferBase = 0;
         for (j=0; j < recvUpdates; j ++) {
           inmsg = LocalRecvBuffer[bufferBase+j];
           LocalOffset = (inmsg & (tparams.TableSize - 1)) -
                  tparams.GlobalStartMyProc;
           HPCC_Table[LocalOffset] ^= inmsg;
         }
       } else if (status.MPI_TAG == FINISHED_TAG) {
         NumberReceiving--;
       } else
         MPI_Abort( MPI_COMM_WORLD, -1 );
       MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
                 MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
     }
   } while (have_done && NumberReceiving > 0);
   if (pendingUpdates < maxPendingUpdates) {
     Ran = (Ran << 1) ^ ((s64Int) Ran < ZERO64B ? POLY : ZERO64B);
     GlobalOffset = Ran & (tparams.TableSize-1);
     if ( GlobalOffset < tparams.Top)
       WhichPe = ( GlobalOffset / (tparams.MinLocalTableSize + 1) );
     else
       WhichPe = ( (GlobalOffset - tparams.Remainder) /
                  tparams.MinLocalTableSize );
     if (WhichPe == tparams.MyProc) {
       LocalOffset = (Ran & (tparams.TableSize - 1)) -
                  tparams.GlobalStartMyProc;
       HPCC_Table[LocalOffset] ^= Ran;
     }  else {
       HPCC_InsertUpdate(Ran, WhichPe, Buckets);
        pendingUpdates++;
      }
      i++;
    }
    else {
      MPI_Test(&outreq, &have_done, MPI_STATUS_IGNORE);
      if (have_done) {
        outreq = MPI_REQUEST_NULL;
        pe = HPCC_GetUpdates(Buckets, LocalSendBuffer, localBufferSize,
                             &peUpdates);
        MPI_Isend(&LocalSendBuffer, peUpdates, tparams.dtype64, (int)pe,
                  UPDATE_TAG, MPI_COMM_WORLD, &outreq);
        pendingUpdates -= peUpdates;
      }
    }
  }
/* send remaining updates in buckets */
while (pendingUpdates > 0) {
  /* receive messages */
  do {
    MPI_Test(&inreq, &have_done, &status);
    if (have_done) {
      if (status.MPI_TAG == UPDATE_TAG) {
        MPI_Get_count(&status, tparams.dtype64, &recvUpdates);
        bufferBase = 0;
        for (j=0; j < recvUpdates; j ++) {
          inmsg = LocalRecvBuffer[bufferBase+j];
          LocalOffset = (inmsg & (tparams.TableSize - 1)) -
                 tparams.GlobalStartMyProc;
          HPCC_Table[LocalOffset] ^= inmsg;
        }
      } else if (status.MPI_TAG == FINISHED_TAG) {
        /* we got a done message.  Thanks for playing... */
        NumberReceiving--;
      } else {
        MPI_Abort( MPI_COMM_WORLD, -1 );
      }
      MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
                MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
    }
  } while (have_done && NumberReceiving > 0);

  MPI_Test(&outreq, &have_done, MPI_STATUS_IGNORE);
  if (have_done) {
    outreq = MPI_REQUEST_NULL;
    pe = HPCC_GetUpdates(Buckets, LocalSendBuffer, localBufferSize,
                         &peUpdates);
    MPI_Isend(&LocalSendBuffer, peUpdates, tparams.dtype64, (int)pe,
              UPDATE_TAG, MPI_COMM_WORLD, &outreq);
    pendingUpdates -= peUpdates;
  }
}
/* send our done messages */
for (proc_count = 0 ; proc_count < tparams.NumProcs ; ++proc_count) {
  if (proc_count == tparams.MyProc) { tparams.finish_req[tparams.MyProc] =
                                     MPI_REQUEST_NULL; continue; }
  /* send garbage - who cares, no one will look at it */
  MPI_Isend(&Ran, 0, tparams.dtype64, proc_count, FINISHED_TAG,
            MPI_COMM_WORLD, tparams.finish_req + proc_count);
}
/* Finish everyone else up... */
while (NumberReceiving > 0) {
  MPI_Wait(&inreq, &status);
  if (status.MPI_TAG == UPDATE_TAG) {
    MPI_Get_count(&status, tparams.dtype64, &recvUpdates);
    bufferBase = 0;
    for (j=0; j < recvUpdates; j ++) {
      inmsg = LocalRecvBuffer[bufferBase+j];
      LocalOffset = (inmsg & (tparams.TableSize - 1)) -
             tparams.GlobalStartMyProc;
      HPCC_Table[LocalOffset] ^= inmsg;
    }
  } else if (status.MPI_TAG == FINISHED_TAG) {
    /* we got a done message.  Thanks for playing... */
    NumberReceiving--;
  } else {
    MPI_Abort( MPI_COMM_WORLD, -1 );
  }
  MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
            MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
}

MPI_Waitall( tparams.NumProcs, tparams.finish_req, tparams.finish_statuses);
```

HPCC RA: MPI KERNEL COMMENT VS. CHAPEL

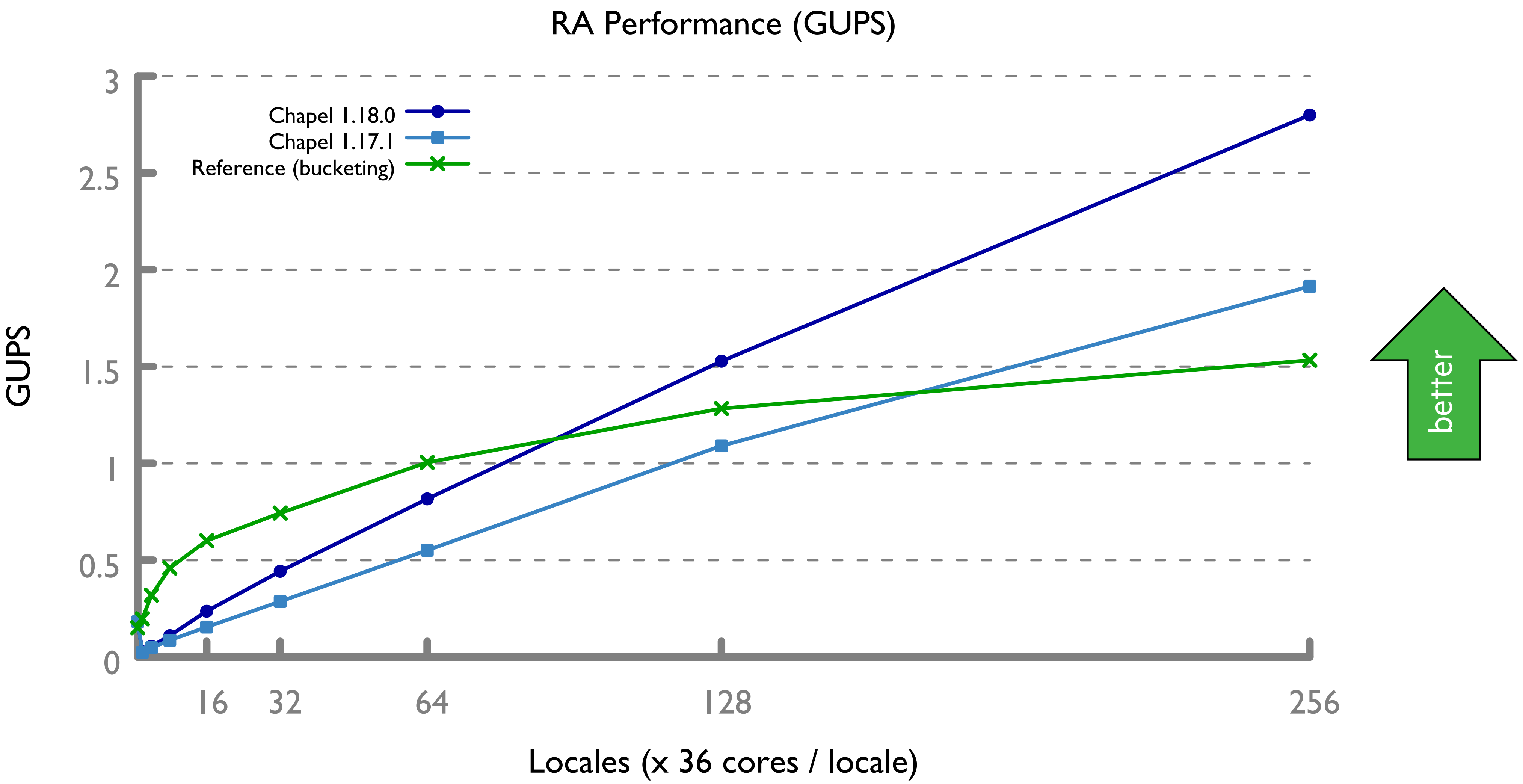
MPI Comment

```
/* Perform updates to main table.  The scalar equivalent is:
 *
 *     for (i=0; i<NUPDATE; i++) {
 *       Ran = (Ran << 1) ^ (((s64Int) Ran < 0) ? POLY : 0);
 *       Table[Ran & (TABSIZE-1)] ^= Ran;
 *     }
 */
```

Chapel Kernel

```
forall (_, r) in zip(Updates, RAStream()) do
  T[r & indexMask].xor(r);
```

HPCC RA: CHAPEL VS. C+MPI (SEPTEMBER 2018)



UNORDERED OPERATION OPTIMIZATION

```
/* Perform updates to main table.  The scalar equivalent is:
 *
 *   for (i=0; i<NUPDATE; i++) {
 *     Ran = (Ran << 1) ^ (((s64Int) Ran < 0) ? POLY : 0);
 *     Table[Ran & (TABSIZE-1)] ^= Ran;
 *   }
 */
```

Chapel Kernel

```
forall (_, r) in zip(Updates, RAStream()) do
  T[r & indexMask].xor(r);
```

Gets lowered roughly to…

```
coforall loc in Updates.targetLocales do
  on loc do
    coforall tid in 1..here.numPUs() do
      for idx in myInds(loc, tid, …) do
        T[idx & indexMask].xor(idx);
}
```

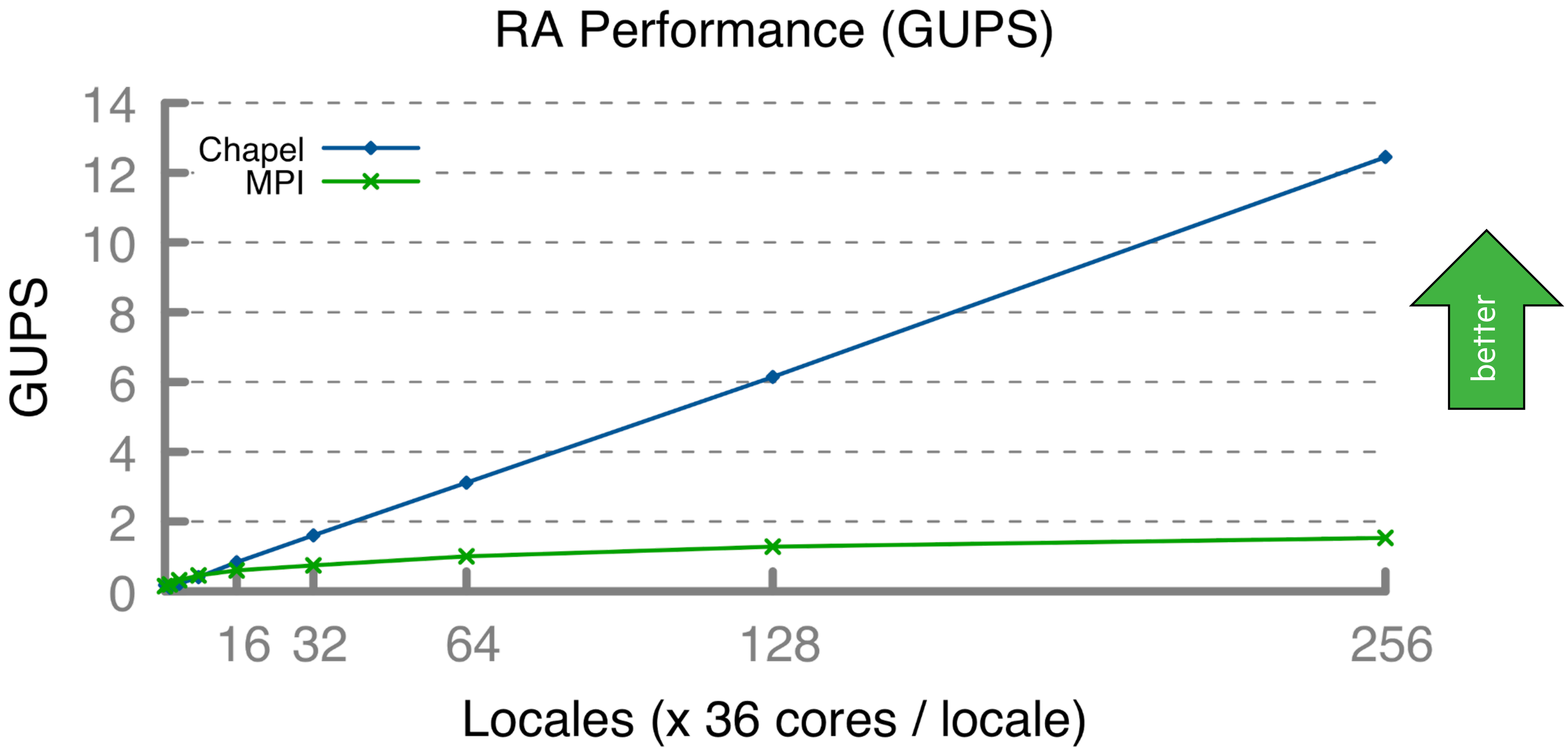
But, for a parallel loop with no data dependencies, why perform these high-latency operations serially?

So, our compiler rewrites the inner loop to perform the ops asynchronously

```
for idx in myInds(loc, tid, …) do
  T[idx & indexMask].unorderedXor(idx);
unorderedFence();
```

- Implemented by Michael Ferguson and Elliot Ronaghan, 2019

HPCC RA: CHAPEL VS. C+MPI



HPCC RA: CHAPEL VS. C+MPI

```
/* Perform updates to main table.  The scalar equivalent is:
 *
 *   for (i=0; i<NUPDATE; i++) {
 *     Ran = (Ran << 1) ^ (((s64Int) Ran < 0) ? POLY : 0);
 *     Table[Ran & (TABSIZE-1)] ^= Ran;
 *   }
 */

MPI_Irecv(&LocalRecvBuffer, loca
          MPI_ANY_SOURCE, MPI_A
 while (i < SendCnt) {
   /* receive messages */
   do {
     MPI_Test(&inreq, &have_done, &status);
     if (have_done) {
       if (status.MPI_TAG == UPDATE_TAG) {
         MPI_Get_count(&status, tparams.dtype64, &recvUpdates);
         bufferBase = 0;
         for (j=0; j < recvUpdates; j
           inmsg = LocalRecvBuffer[bu
           LocalOffset = (inmsg & (tp
                         tparams.Glob
           HPCC_Table[LocalOffset] ^=
         }
       } else if (status.MPI_TAG == FINISHED_TAG) {
         NumberReceiving--;
       } else
         MPI_Abort( MPI_COMM_WORLD, -1 );
       MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
                 MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
     }
   } while (have_done && NumberReceiving > 0);
   if (pendingUpdates < maxPendingUpdates) {
     Ran = (Ran << 1) ^ ((s64Int) Ran < ZERO64B ? POLY : ZERO64B);
     GlobalOffset = Ran & (tparams.TableSize-1);
     if ( GlobalOffset < tparams.Top)
       WhichPe = ( GlobalOffset / (tparams.MinLocalTableSize + 1) );
     else
       WhichPe = ( (GlobalOffset - tparams.Remainder) /
                   tparams.MinLocalTableSize );
     if (WhichPe == tparams.MyProc) {
       LocalOffset = (Ran & (tparams.TableSize - 1)) -
                     tparams.GlobalStartMyProc;
       HPCC_Table[LocalOffset] ^= Ran;
```

```
     } else {
       HPCC_InsertUpdate(Ran, WhichPe, Buckets);
       pendingUpdates++;
     }
     i++;
   }
 ...
              UPDATE_TAG, MPI_COMM_WORLD, &outreq);
     pendingUpdates -= peUpdates;
   }
 }
 ...
     if (have_done) {
       if (status.MPI_TAG == UPDATE_TAG) {
         MPI_Get_count(&status, tparams.dtype64, &recvUpdates);
         bufferBase = 0;
         for (j=0; j < recvUpdates; j ++) {
           inmsg = LocalRecvBuffer[bufferBase+j];
           LocalOffset = (inmsg & (tparams.TableSize - 1)) -
                         tparams.GlobalStartMyProc;
           HPCC_Table[LocalOffset] ^= inmsg;
         }
       } else if (status.MPI_TAG == FINISHED_TAG) {
         /* we got a done message.  Thanks for playing... */
         NumberReceiving--;
       } else {
         MPI_Abort( MPI_COMM_WORLD, -1 );
       }
       MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
                 MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
     }
   } while (have_done && NumberReceiving > 0);
```

```
   MPI_Test(&outreq, &have_done, MPI_STATUS_IGNORE);
   if (have_done) {
     outreq = MPI_REQUEST_NULL;
     ...                                           ndBuffer, localBufferSize,
 ...                                          , tparams.dtype64, (int)pe,
                                                &outreq);
 ...
 for (proc_count = 0 ; proc_count < tparams.NumProcs ; ++proc_count) {
   if (proc_count == tparams.MyProc) { tparams.finish_req[tparams.MyProc] =
                                       MPI_REQUEST_NULL; continue; }
   /* send garbage - who cares, no one will look at it */
 ...                                 pe64, proc_count, FINISHED_TAG,
 ...                                 ams.finish_req + proc_count);
 ...
   if (status.MPI_TAG == UPDATE_TAG) {
     MPI_Get_count(&status, tparams.dtype64, &recvUpdates);
     bufferBase = 0;
     for (j=0; j < recvUpdates; j ++) {
       inmsg = LocalRecvBuffer[bufferBase+j];
       LocalOffset = (inmsg & (tparams.TableSize - 1)) -
                     tparams.GlobalStartMyProc;
       HPCC_Table[LocalOffset] ^= inmsg;
     }
   } else if (status.MPI_TAG == FINISHED_TAG) {
     /* we got a done message.  Thanks for playing... */
     NumberReceiving--;
   } else {
     MPI_Abort( MPI_COMM_WORLD, -1 );
   }
   MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
             MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
 }

MPI_Waitall( tparams.NumProcs, tparams.finish_req, tparams.finish_statuses);
```

Chapel Kernel

```
forall (_, r) in zip(Updates, RAStream()) do
  T[r & indexMask].xor(r);
```

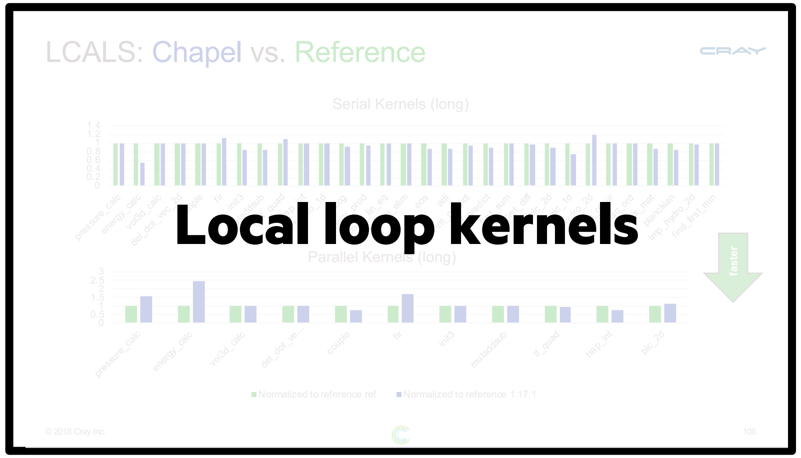
Now, imagine what it would take for a compiler to optimize the C+MPI code…

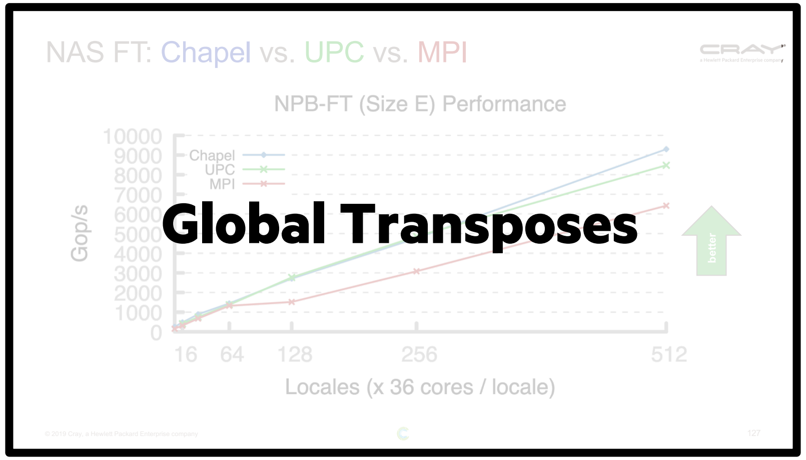
HPC PATTERNS & BENCHMARKS

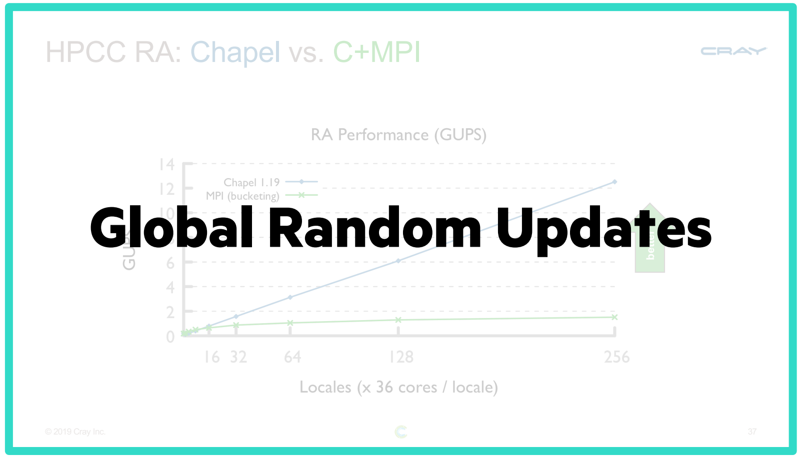
LCALS

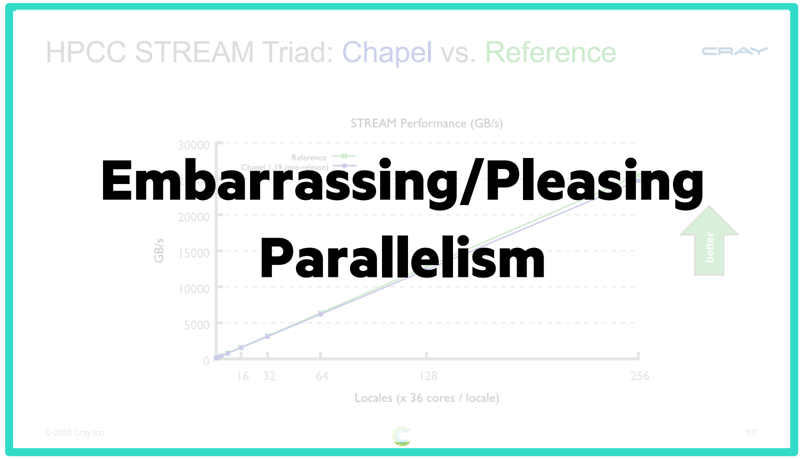
NAS FT

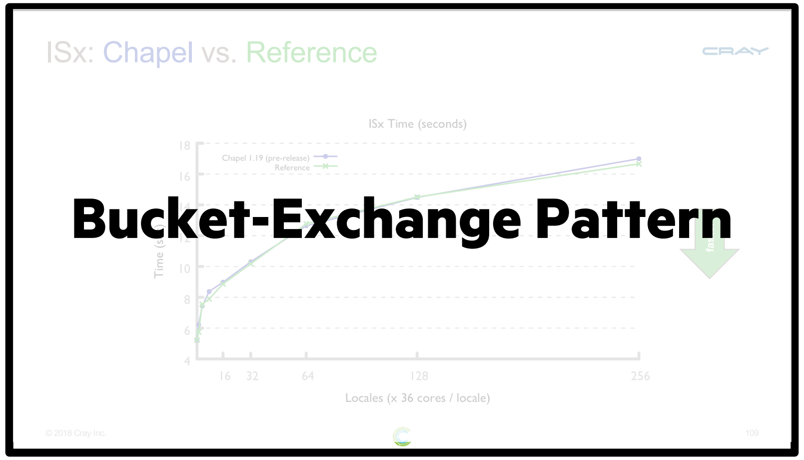
HPCC RA

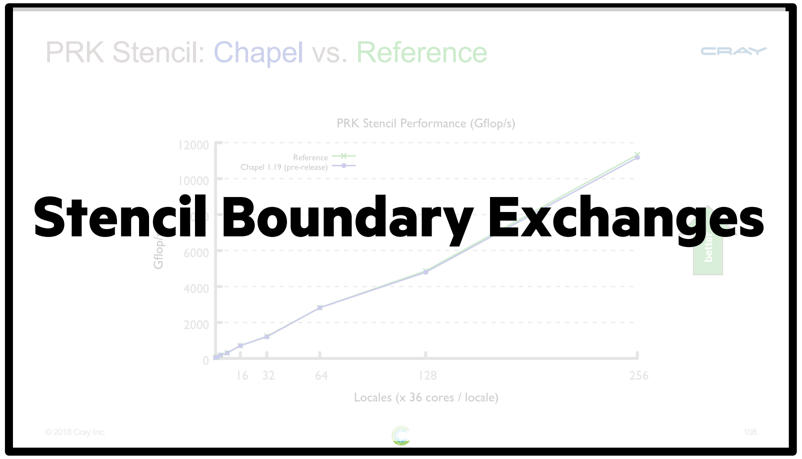












STREAM Triad

ISx

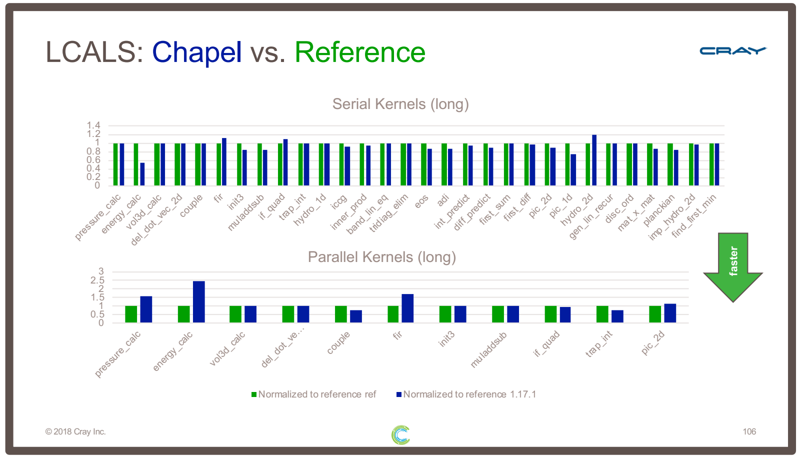
PRK Stencil

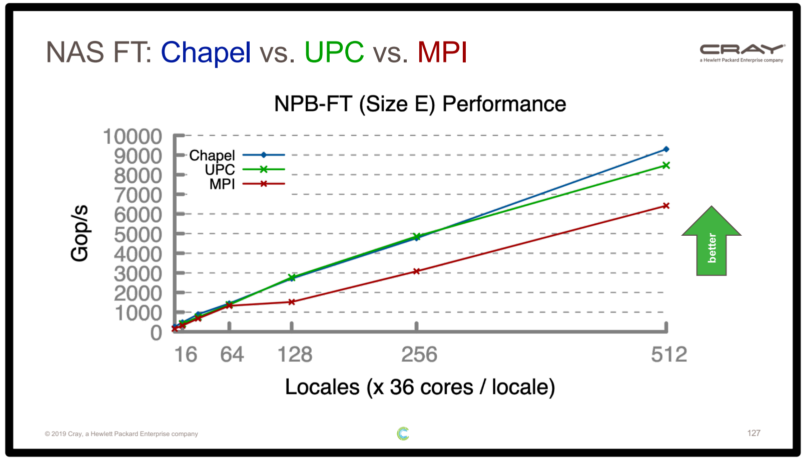
HPC PATTERNS & BENCHMARKS: CHAPEL VS. REFERENCE

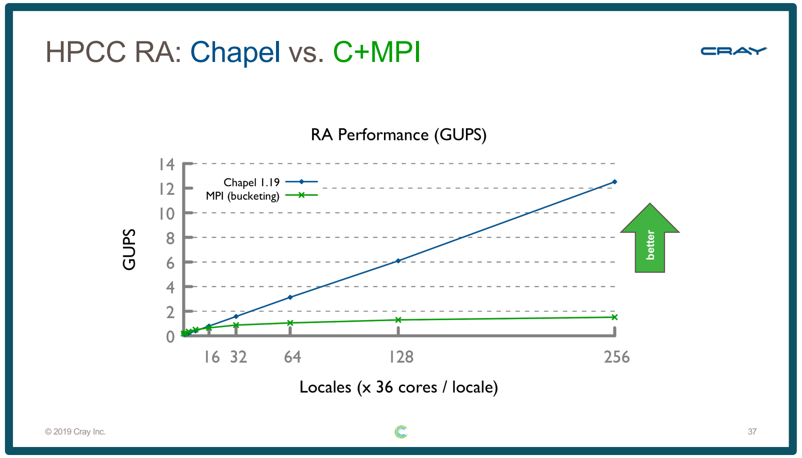
LCALS

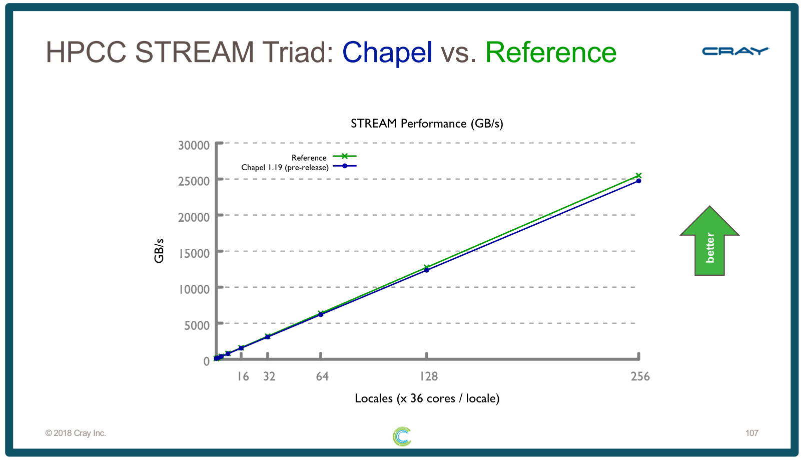
NAS FT

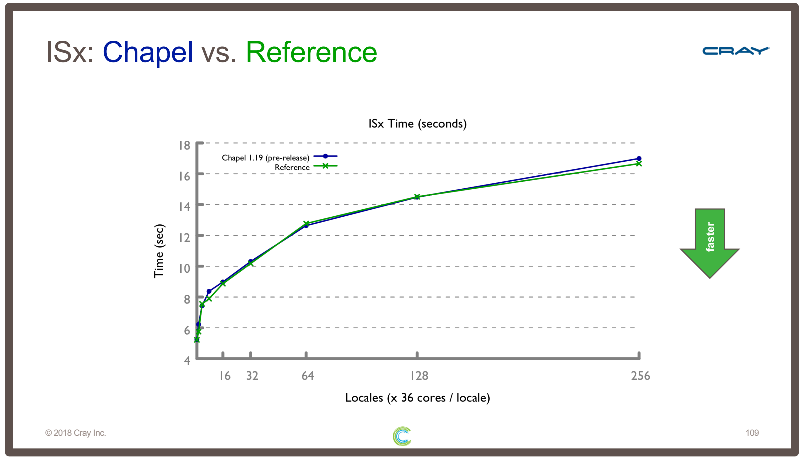
HPCC RA

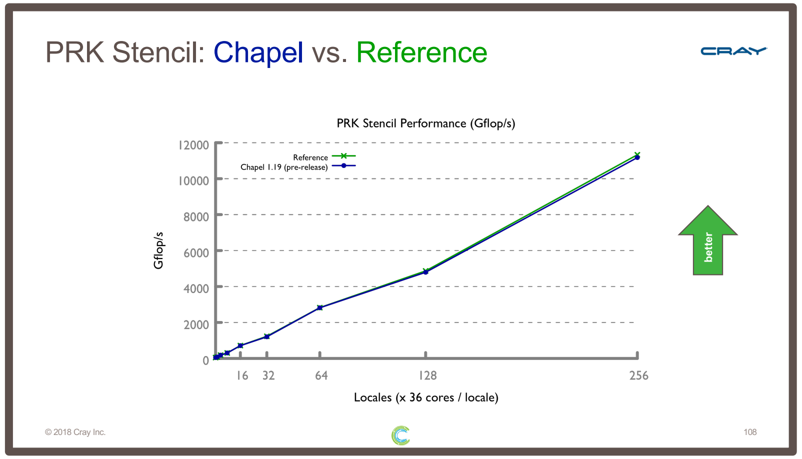












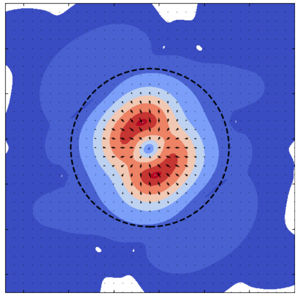
STREAM Triad

ISx

PRK Stencil

More on Chapel performance online at: https://chapel-lang.org/performance.html

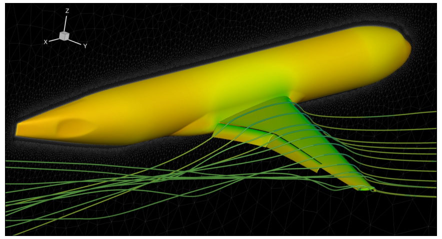
NOTABLE APPLICATIONS OF CHAPEL



ChplUltra: Simulating Ultralight Dark Matter

Nikhil Padmanabhan, J. Luna Zagorac, Richard Easther, *et al.*

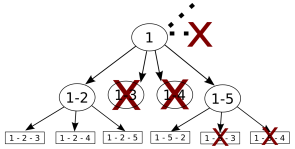
Yale University / University of Auckland



CHAMPS: 3D Computational Fluid Dynamics

Eric Laurendeau, Simon Bourgault-Côté, Matthieu Parenteau, et al.

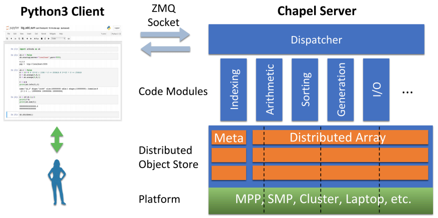
École Polytechnique Montréal



ChOp: Chapel-based Optimization

Nouredine Melab, Tiago Carneiro, *et al.*

INRIA Lille, France



Arkouda: NumPy at Massive Scale

Mike Merrill, Bill Reus, et al.

US DOD

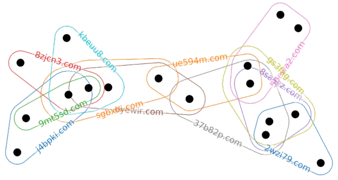
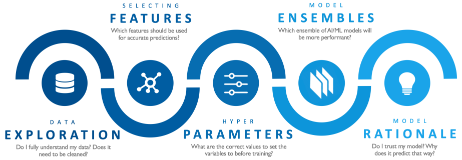


Image courtesy of Cliff Joslyn, cliff.joslyn@pnnl.gov

CHGL: Chapel Hypergraph Library

Cliff Joslyn, Jesun Firoz, Louis Jenkins, *et al.*

PNNL



CrayAI: Distributed Machine Learning

Hewlett Packard Enterprise

For more information, see: https://chapel-lang.org/poweredby.html

SUMMARY & RESOURCES

SUMMARY

Chapel cleanly and orthogonally supports…

…expression of parallelism and locality

…specifying how to map computations to the system

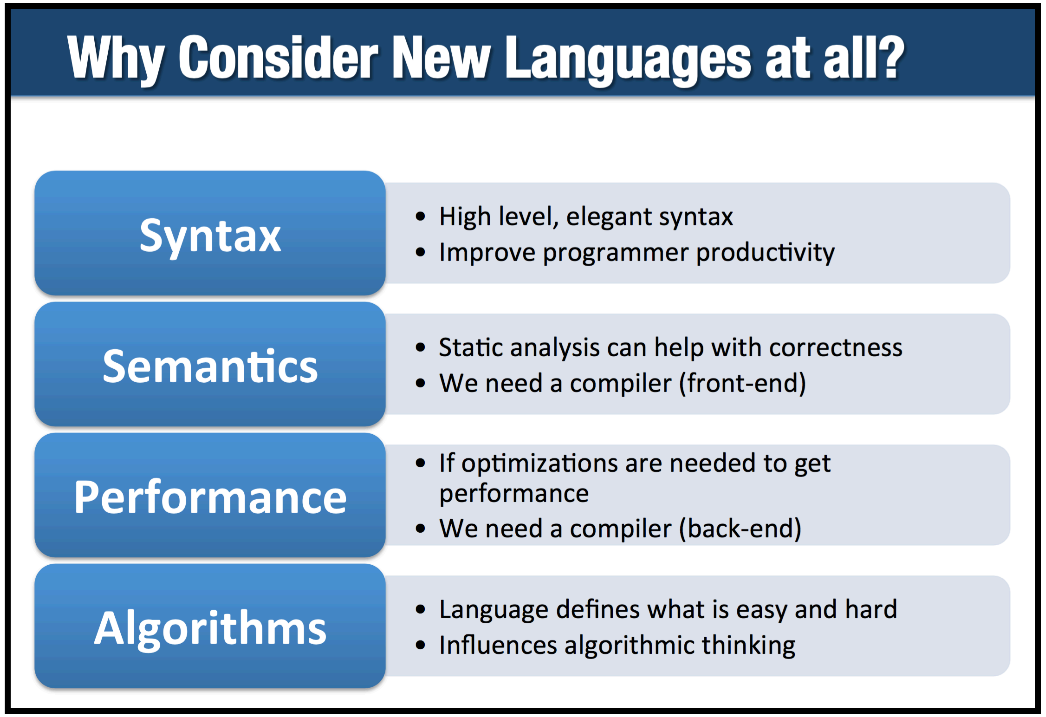
Chapel's compilation is neither magical nor heroic

- lower-level features have straightforward compilation path
- higher-level features built in terms of lower-level features
- language designed with parallel optimization in mind

Chapel is powerful:

- supports succinct, straightforward code
- can result in performance that competes with, or beats, C+MPI+OpenMP

Chapel is attractive to computational scientists and Python programmers

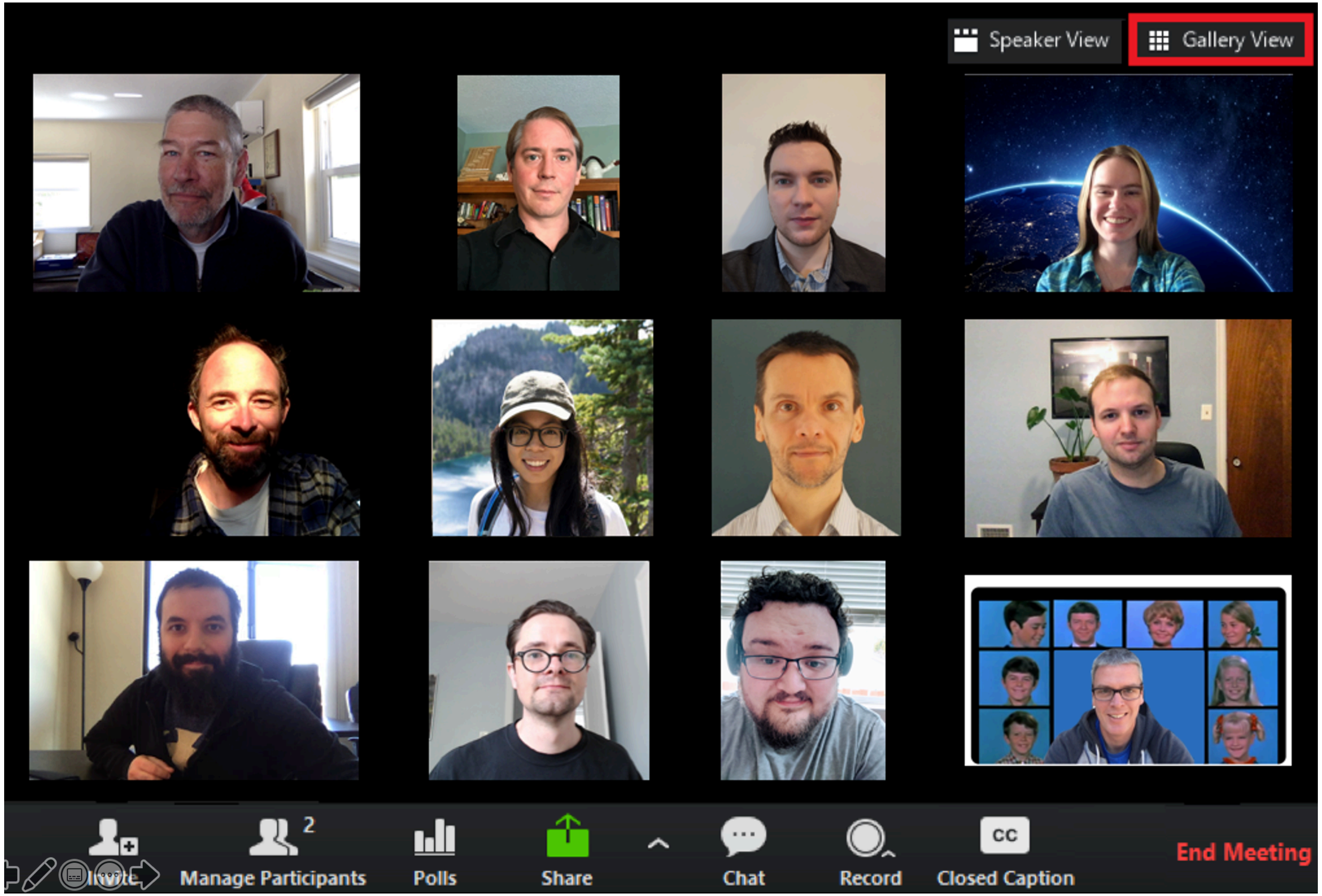


[Image source: Kathy Yelick's, CHIUW 2018 keynote: *Why Languages Matter More Than Ever*, used with permission]

CHALLENGES AND NEXT STEPS

- **Generate code for GPUs**
  - How will the compiler need to evolve? Will the language need to?
- **Rearchitect the compiler**
  - Shed cruft from research prototype days to harden the compiler
  - Reduce compile times
    - potentially via separate compilation / incremental recompilation?
  - Support interpreted / interactive Chapel programming
- **Continue to optimize performance**
- **Release Chapel 2.0**
  - guarantee backwards-compatibility for core language and library
- **Continue to grow the Chapel community**

WE ARE HIRING



Full-time:

- Keep an eye on our jobs site
  - (I need to update it this week)

Summers:

- HPE internships
- Google Summer of Code

CHAPEL RESOURCES

Chapel homepage: https://chapel-lang.org

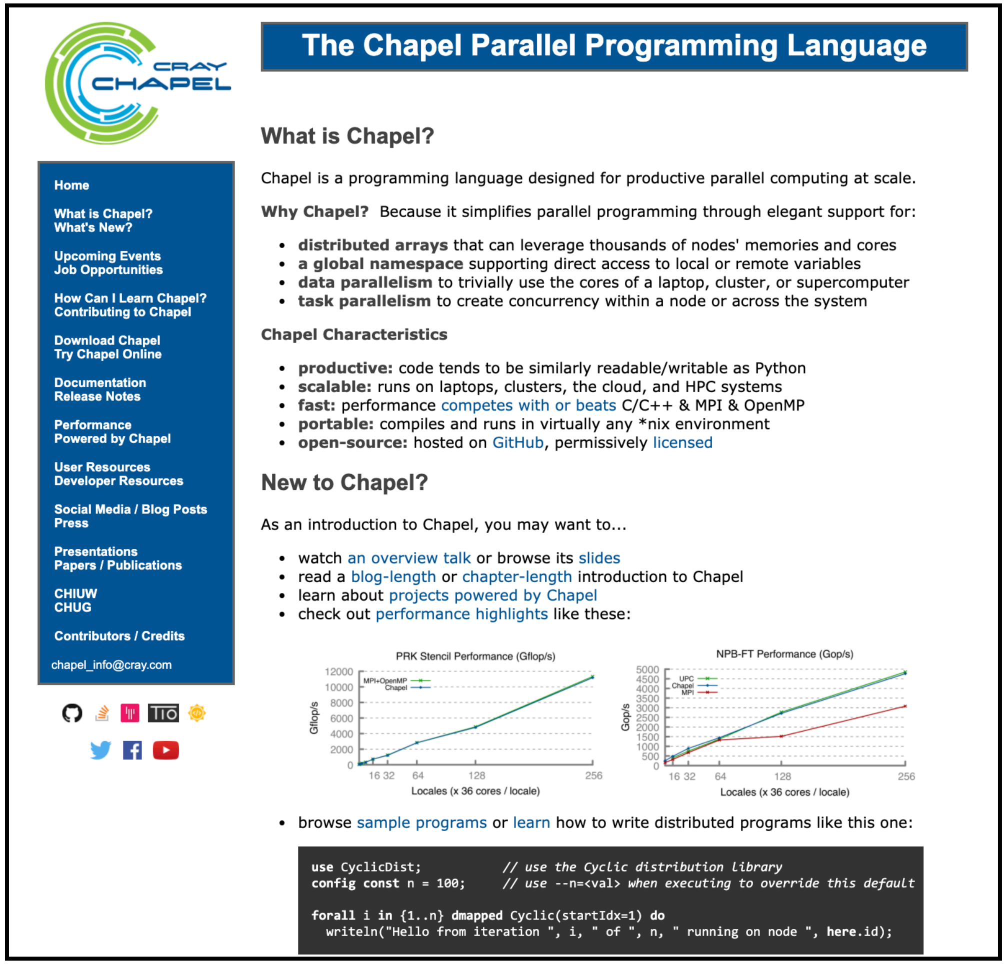
- (points to all other resources)

Social Media:

- Twitter: @ChapelLanguage
- Facebook: @ChapelLanguage
- YouTube: http://www.youtube.com/c/ChapelParallelProgrammingLanguage

Community / Support:

- Gitter: https://gitter.im/chapel-lang/chapel
- Discourse: https://chapel.discourse.group/
- Stack Overflow: https://stackoverflow.com/questions/tagged/chapel
- GitHub Issues: https://github.com/chapel-lang/chapel/issues



SUGGESTED READING

Chapel Overviews / History:

- *Chapel* chapter from *Programming Models for Parallel Computing*, MIT Press, edited by Pavan Balaji, November 2015
- *Chapel Comes of Age: Making Scalable Programming Productive*, Chamberlain et al., *CUG 2018*, May 2018
- Proceedings of the *7th Annual Chapel Implementers and Users Workshop* (CHIUW 2020), May 2020
- Chapel Release Notes — current version 1.22, April 2020

Implementation of Chapel's High-Level Features:

- *User-Defined Distributions and Layouts in Chapel: Philosophy and Framework*, Chamberlain et al., *HotPar'10*, June 2010
- *Authoring User-Defined Domain Maps in Chapel*, Chamberlain et al., *CUG 2011*, May 2011
- *User-Defined Parallel Zippered iterators in Chapel*, Chamberlain et al., *PGAS 2011*, October 2011

THESIS STATEMENT, REDUX

Well-designed languages can improve user productivity while also enabling new optimizations.

We believe Chapel to be such a language.

THANK YOU

https://chapel-lang.org
@ChapelLanguage