Hewlett Packard Enterprise

PARALLEL PROGRAMMING IN CHAPEL: OVERVIEW AND OOKAMI

Brad Chamberlain

Webinar for the Ookami user community

January 13, 2022

TEASER FOR THIS TALK

Imagine having a programming language for HPC that was as…

…**programmable** as Python

…yet also as…

…**fast** as Fortran

…**scalable** as MPI or SHMEM

…**portable** as C

…**flexible** as C++

…**type-safe** as Fortran, C, C++, …

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

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."

SPEAKING OF THE CHAPEL TEAM…

Chapel is truly a team effort—we're currently at 19 full-time employees (+ a director), and we are hiring

Chapel Development Team at HPE

see: https://chapel-lang.org/contributors.html

OUTLINE

I. What is Chapel?

II. Chapel Motivation and Uses

III. Introduction to Chapel

IV. Live Demo?

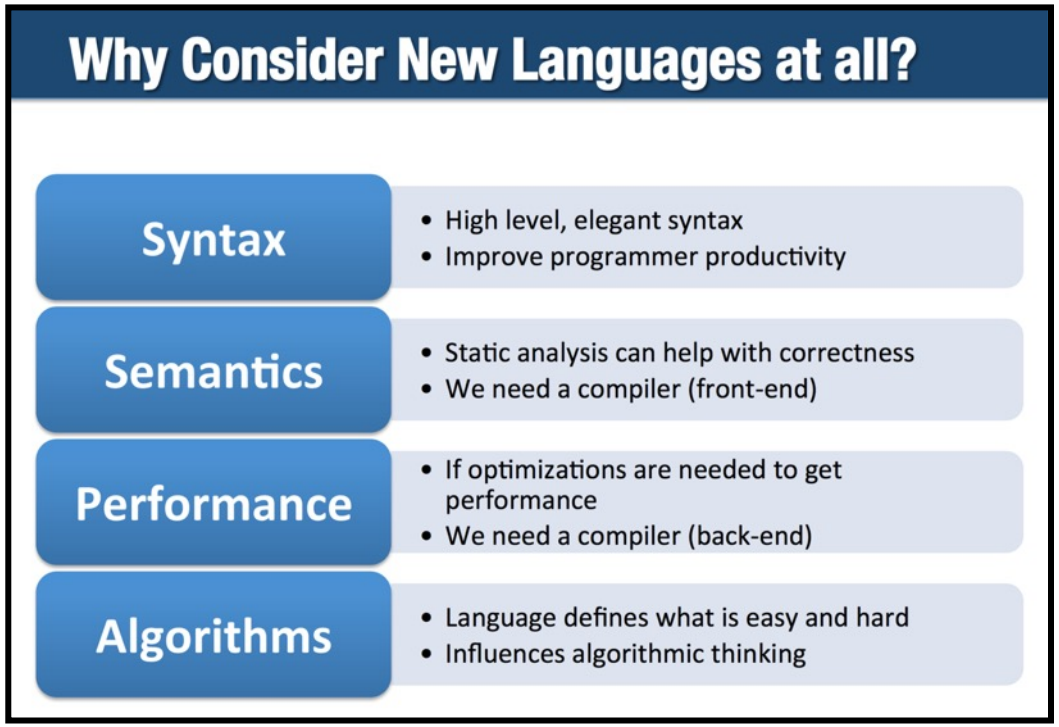
V. Chapel on Ookami

V. Summary and Resources

VI. Hands-On?

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 meet our needs…**

WHAT SHOULD A PRODUCTIVE LANGUAGE FOR HPC SUPPORT?

Traditional Language Characteristics

- programmability
- portability
- performance
- abstraction
- interoperability
- …

Features Unique to HPC

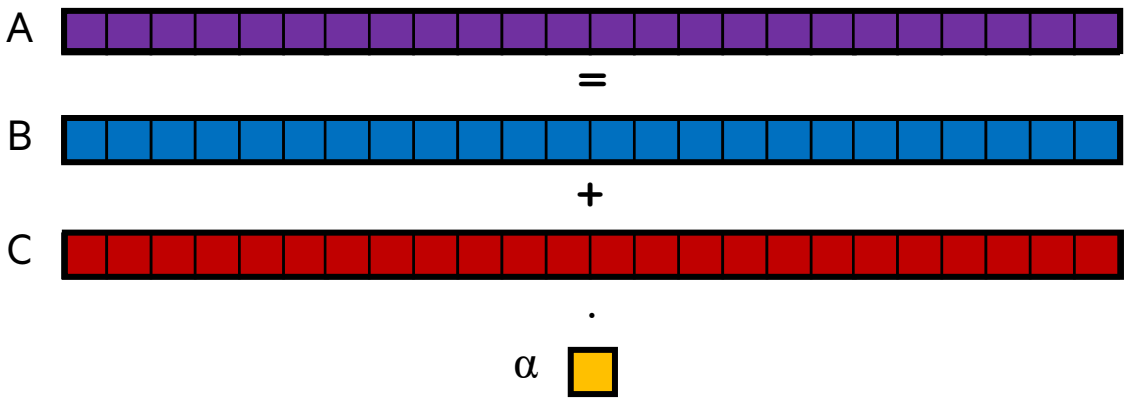
- ability to express **parallelism**
- ability to control and reason about **locality**

STREAM TRIAD: A TRIVIAL CASE OF PARALLELISM + LOCALITY

Given: m-element vectors A, B, C

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

In pictures:

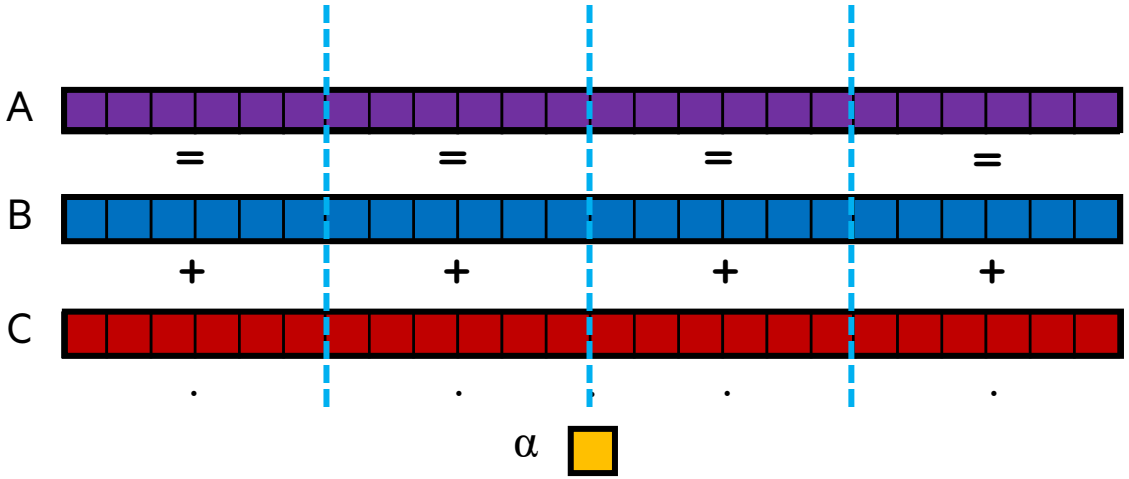


STREAM TRIAD: A TRIVIAL CASE OF PARALLELISM + LOCALITY

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 (shared memory / multicore)**:**

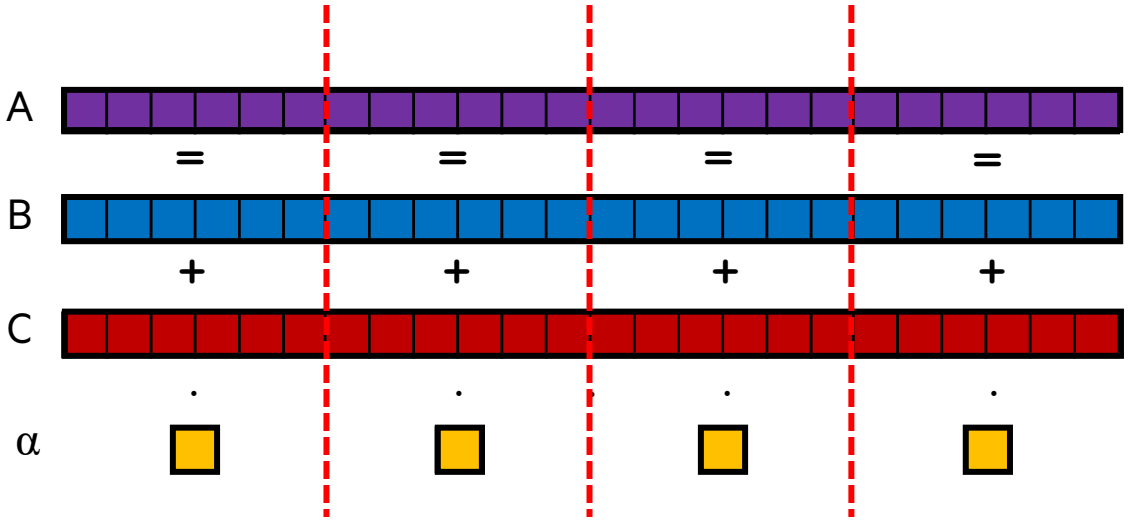


STREAM TRIAD: A TRIVIAL CASE OF PARALLELISM + LOCALITY

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)**:**

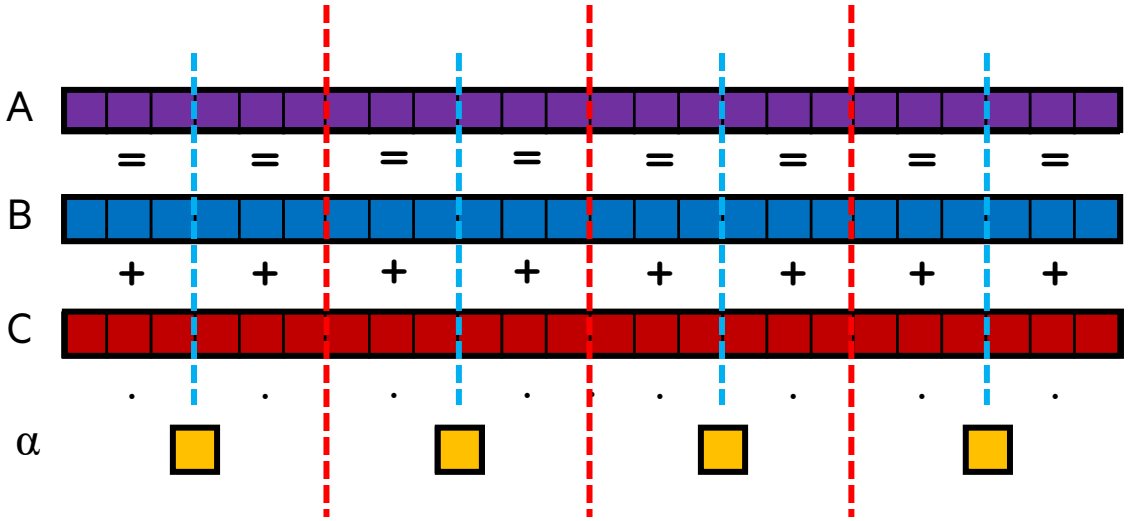


STREAM TRIAD: A TRIVIAL CASE OF PARALLELISM + LOCALITY

Given: m-element vectors A, B, C

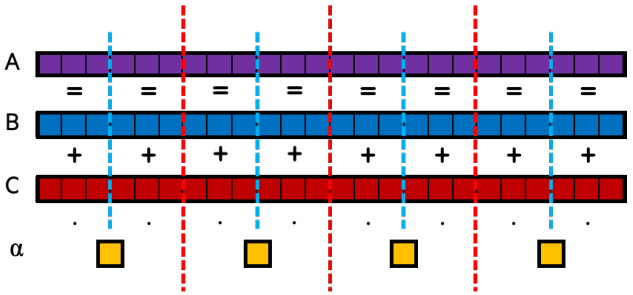
Compute: $\forall i \in 1..m,\ A_i = B_i + \alpha \cdot C_i$

In pictures, in parallel (distributed memory multicore)**:**



STREAM TRIAD IN CONVENTIONAL HPC PROGRAMMING MODELS

Many Disparate Notations for Expressing Parallelism + Locality



MPI

```
#include <hpcc.h>

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 );
```

```
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 );
  }

for (j=0; j<VectorSize; j++) {
  b[j] = 2.0;
  c[j] = 1.0;
}
scalar = 3.0;

for (j=0; j<VectorSize; j++)
  a[j] = b[j]+scalar*c[j];

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

return 0;  }
```

Note: This is a very trivial parallel computation—imagine the additional differences for something more complex!

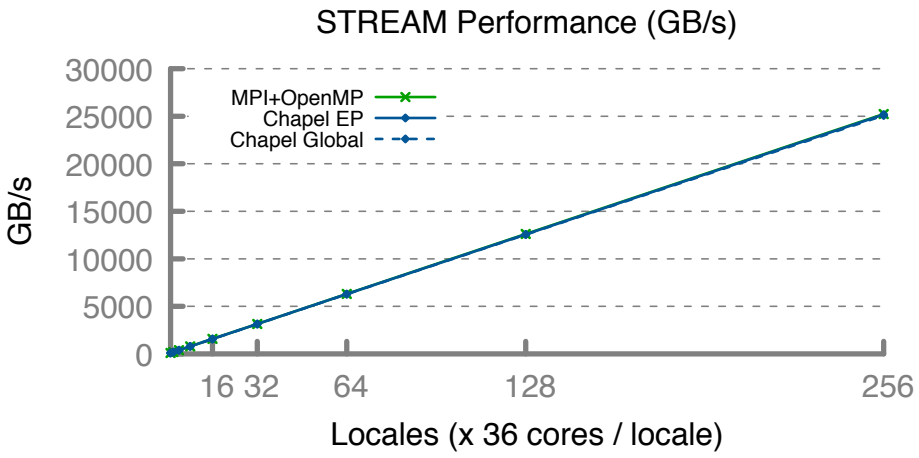
Challenge: Can we do better?

CHAPEL BENCHMARKS TEND TO BE CONCISE, CLEAR, AND COMPETITIVE

STREAM TRIAD: C + MPI + OPENMP

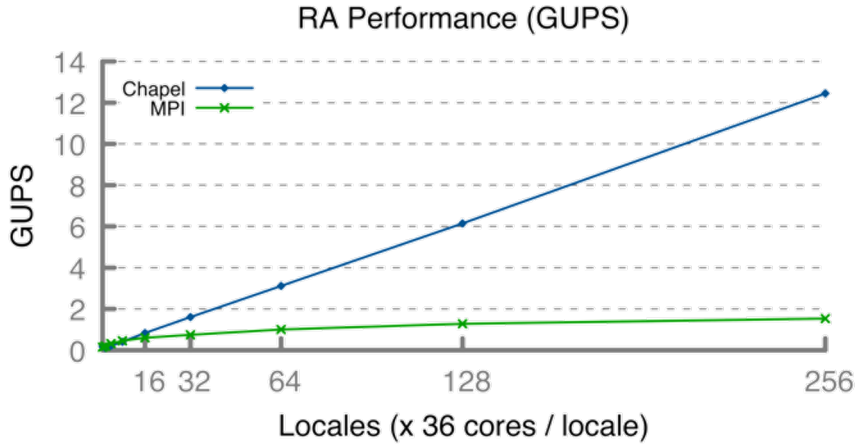
```c
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

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 );

  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
      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;
}
```

```chapel
use BlockDist;

config const m = 1000,
             alpha = 3.0;
const Dom = {1..m} dmapped …;
var A, B, C: [Dom] real;

B = 2.0;
C = 1.0;

A = B + alpha * C;
```



HPCC RA: MPI KERNEL

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



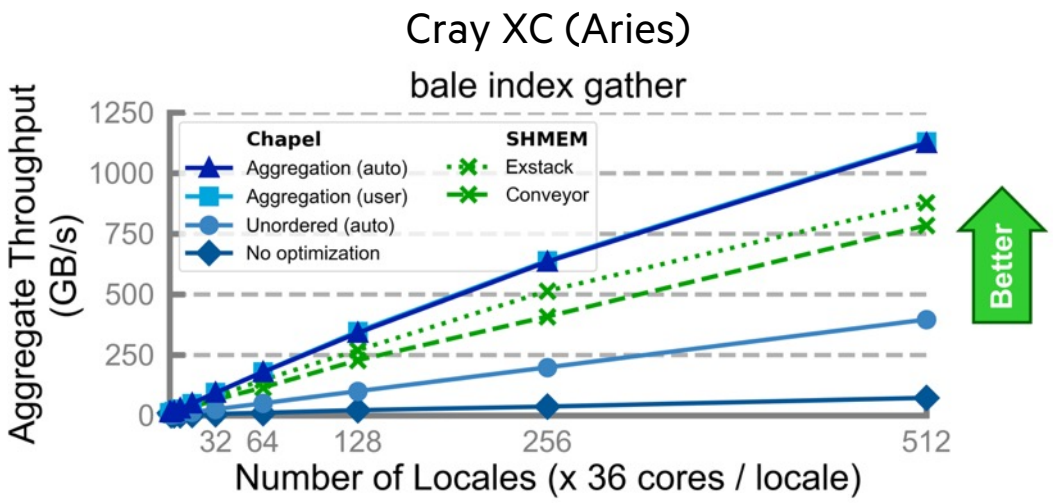
BALE INDEX GATHER: CHAPEL VS. EXSTACK VS. CONVEYORS

Exstack version

```
i=0;
while( exstack_proceed(ex, (i==l_num_req)) ) {
  i0 = i;
  while(i < l_num_req) {
    l_indx = pckindx[i] >> 16;
    pe  = pckindx[i] & 0xffff;
    if(!exstack_push(ex, &l_indx, pe))
      break;
    i++;
  }

  exstack_exchange(ex);

  while(exstack_pop(ex, &idx , &fromth)) {
    idx  = ltable[idx];
    exstack_push(ex, &idx, fromth);
  }
  lgp_barrier();
  exstack_exchange(ex);

  for(j=i0; j<i; j++) {
    fromth = pckindx[j] & 0xffff;
    exstack_pop_thread(ex, &idx, (uint64_t)fromth);
    tgt[j] = idx;
  }
  lgp_barrier();
}
```

Conveyors version

```
i = 0;
while (more = convey_advance(requests, (i == l_num_req)),
       more | convey_advance(replies, !more)) {

  for (; i < l_num_req; i++) {
    pkg.idx = i;
    pkg.val = pckindx[i] >> 16;
    pe = pckindx[i] & 0xffff;
    if (! convey_push(requests, &pkg, pe))
      break;
  }

  while (convey_pull(requests, ptr, &from) == convey_OK) {
    pkg.idx = ptr->idx;
    pkg.val = ltable[ptr->val];
    if (! convey_push(replies, &pkg, from)) {
      convey_unpull(requests);
      break;
    }
  }

  while (convey_pull(replies, ptr, NULL) == convey_OK)
    tgt[ptr->idx] = ptr->val;
}
```



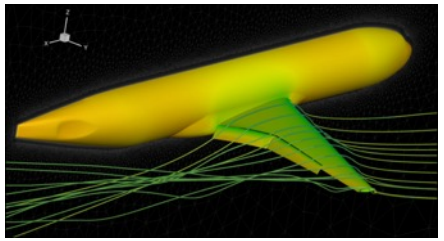
Elegant Chapel version (compiler-optimized w/ '--auto-aggregation')

```
forall (d, i) in zip(Dst, Inds) do
  d = Src[i];
```

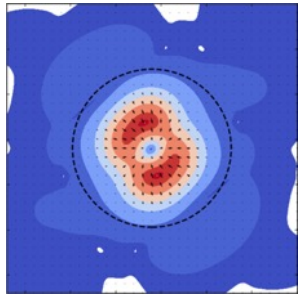
Manually Tuned Chapel version (using aggregator abstraction)

```
forall (d, i) in zip(Dst, Inds) with (var agg = new SrcAggregator(int)) do
  agg.copy(d, Src[i]);
```

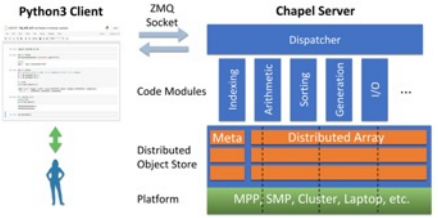
CURRENT FLAGSHIP CHAPEL APPLICATIONS



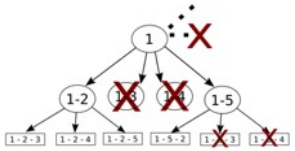
CHAMPS: 3D Unstructured CFD
Éric Laurendeau, Simon Bourgault-Côté, Matthieu Parenteau, et al.
École Polytechnique Montréal



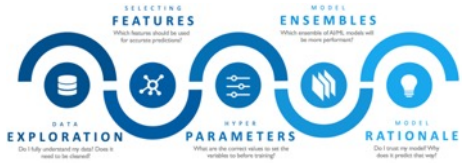
ChplUltra: Simulating Ultralight Dark Matter
Nikhil Padmanabhan, J. Luna Zagorac, *et al.*
Yale University / University of Auckland



Arkouda: NumPy at Massive Scale
Mike Merrill, Bill Reus, et al.
US DoD



ChOp: Chapel-based Optimization
Tiago Carneiro, Nouredine Melab, *et al.*
INRIA Lille, France

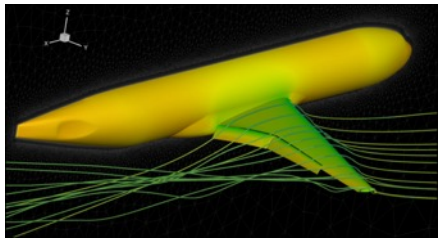


CrayAI: Distributed Machine Learning
Hewlett Packard Enterprise

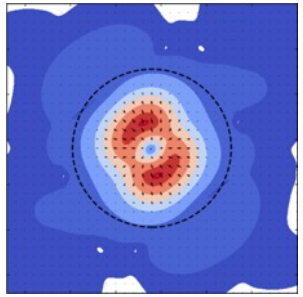
?

Your application here?

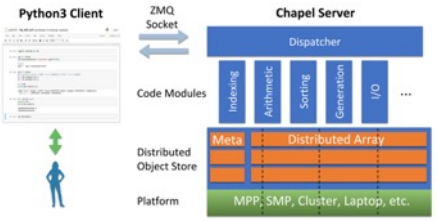
CURRENT FLAGSHIP CHAPEL APPLICATIONS



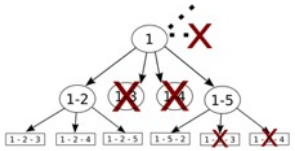
CHAMPS: 3D Unstructured CFD
Éric Laurendeau, Simon Bourgault-Côté, Matthieu Parenteau, et al.
École Polytechnique Montréal



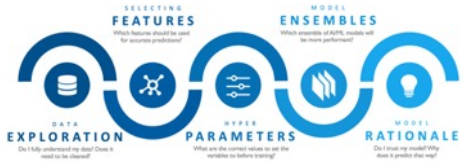
ChplUltra: Simulating Ultralight Dark Matter
Nikhil Padmanabhan, J. Luna Zagorac, *et al.*
Yale University / University of Auckland



Arkouda: NumPy at Massive Scale
Mike Merrill, Bill Reus, et al.
US DoD



ChOp: Chapel-based Optimization
Tiago Carneiro, Nouredine Melab, *et al.*
INRIA Lille, France



CrayAI: Distributed Machine Learning
Hewlett Packard Enterprise

?

Your application here?

ARKOUDA IN ONE SLIDE

What is it?

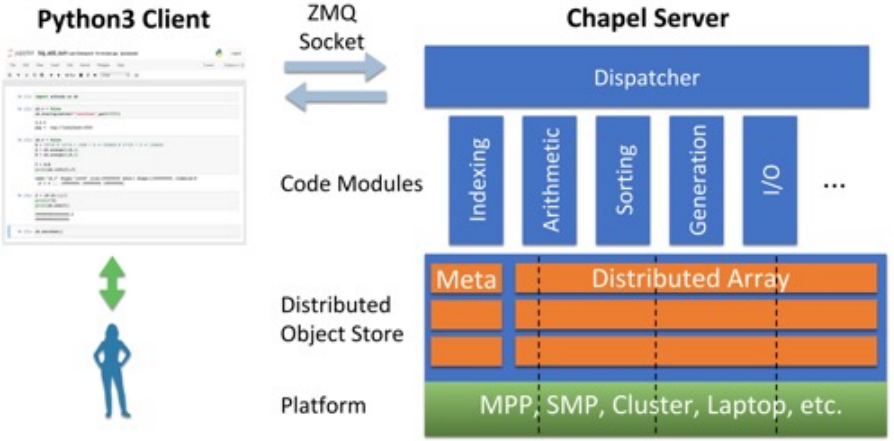
- A Python library supporting a key subset of NumPy and Pandas for Data Science
  - Computes massive-scale results within the human thought loop (seconds to minutes on multi-TB-scale arrays)
  - Uses a Python-client/Chapel-server model to get scalability and performance
- ~16k lines of Chapel, largely written in 2019, continually improved since then

Who wrote it?

- Mike Merrill, Bill Reus, et al., US DoD
- Open-source: https://github.com/Bears-R-Us/arkouda

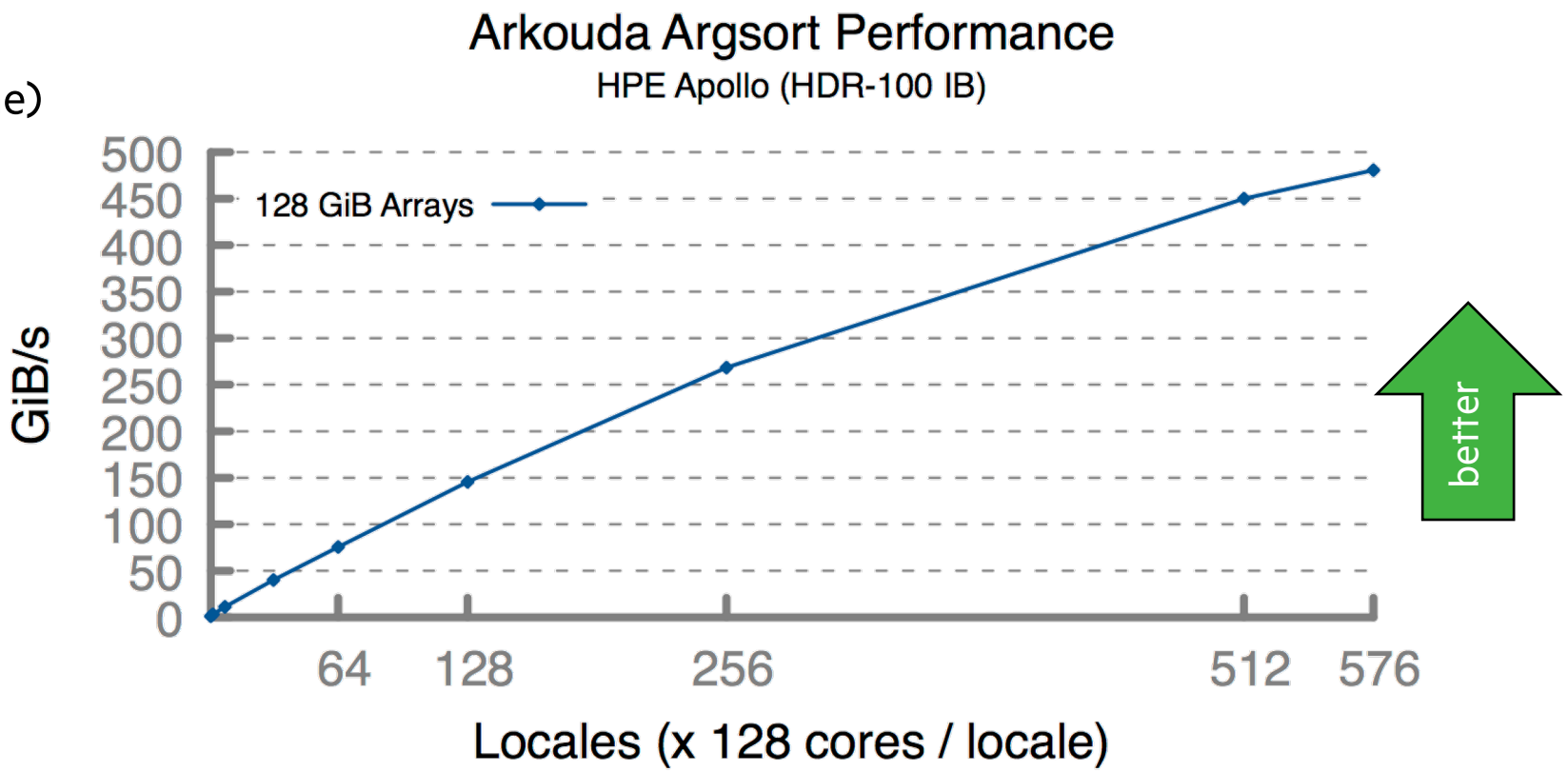
Why Chapel?

- high-level language with performance and scalability
  - close to Pythonic—doesn't repel Python users who look under the hood
- great distributed array support
- ports from laptop to supercomputer



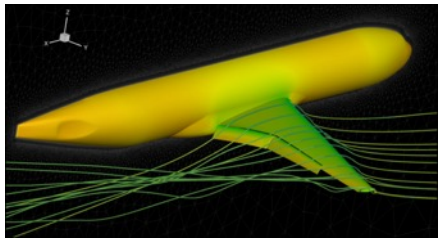
ARKOUDA ARGSORT: HERO RUN

- Recent run performed on a large Apollo system
  - 72 TiB of 8-byte values
  - 480 GiB/s (2.5 minutes elapsed time)
  - used 73,728 cores of AMD Rome
  - ~100 lines of Chapel code

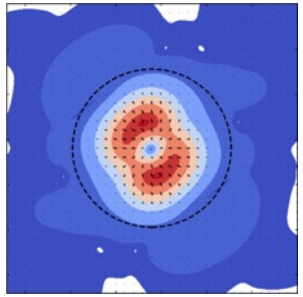


Close to world-record performance—Quite likely a record for performance::lines of code

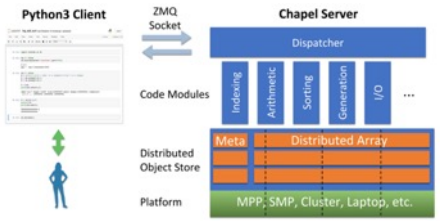
CURRENT FLAGSHIP CHAPEL APPLICATIONS



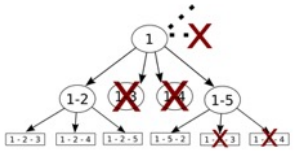
CHAMPS: 3D Unstructured CFD
Éric Laurendeau, Simon Bourgault-Côté, Matthieu Parenteau, et al.
École Polytechnique Montréal



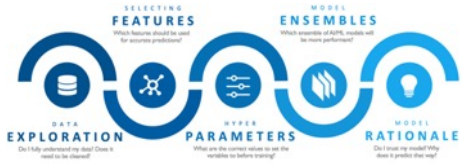
ChplUltra: Simulating Ultralight Dark Matter
Nikhil Padmanabhan, J. Luna Zagorac, *et al.*
Yale University / University of Auckland



Arkouda: NumPy at Massive Scale
Mike Merrill, Bill Reus, et al.
US DoD



ChOp: Chapel-based Optimization
Tiago Carneiro, Nouredine Melab, *et al.*
INRIA Lille, France



CrayAI: Distributed Machine Learning
Hewlett Packard Enterprise

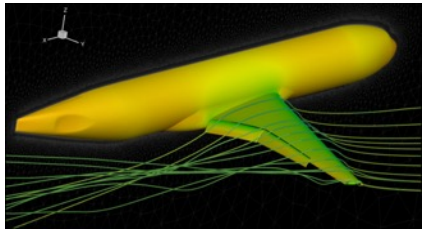
?

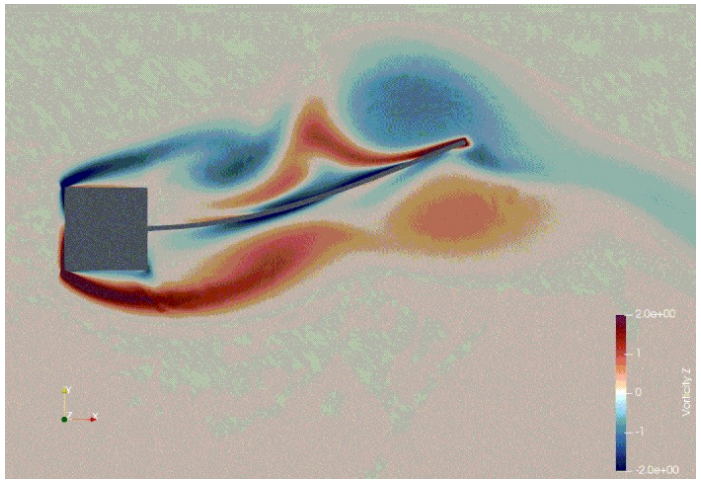
Your application here?

CHAMPS SUMMARY

What is it?

- 3D unstructured CFD framework for airplane simulation
- ~48k lines of Chapel written from scratch in ~2 years





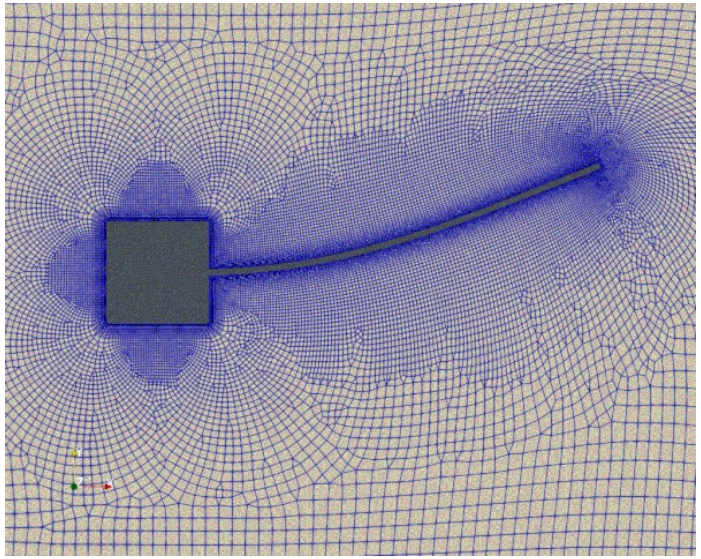
Who wrote it?

- Professor Éric Laurendeau's team at Polytechnique Montreal

POLYTECHNIQUE MONTRÉAL

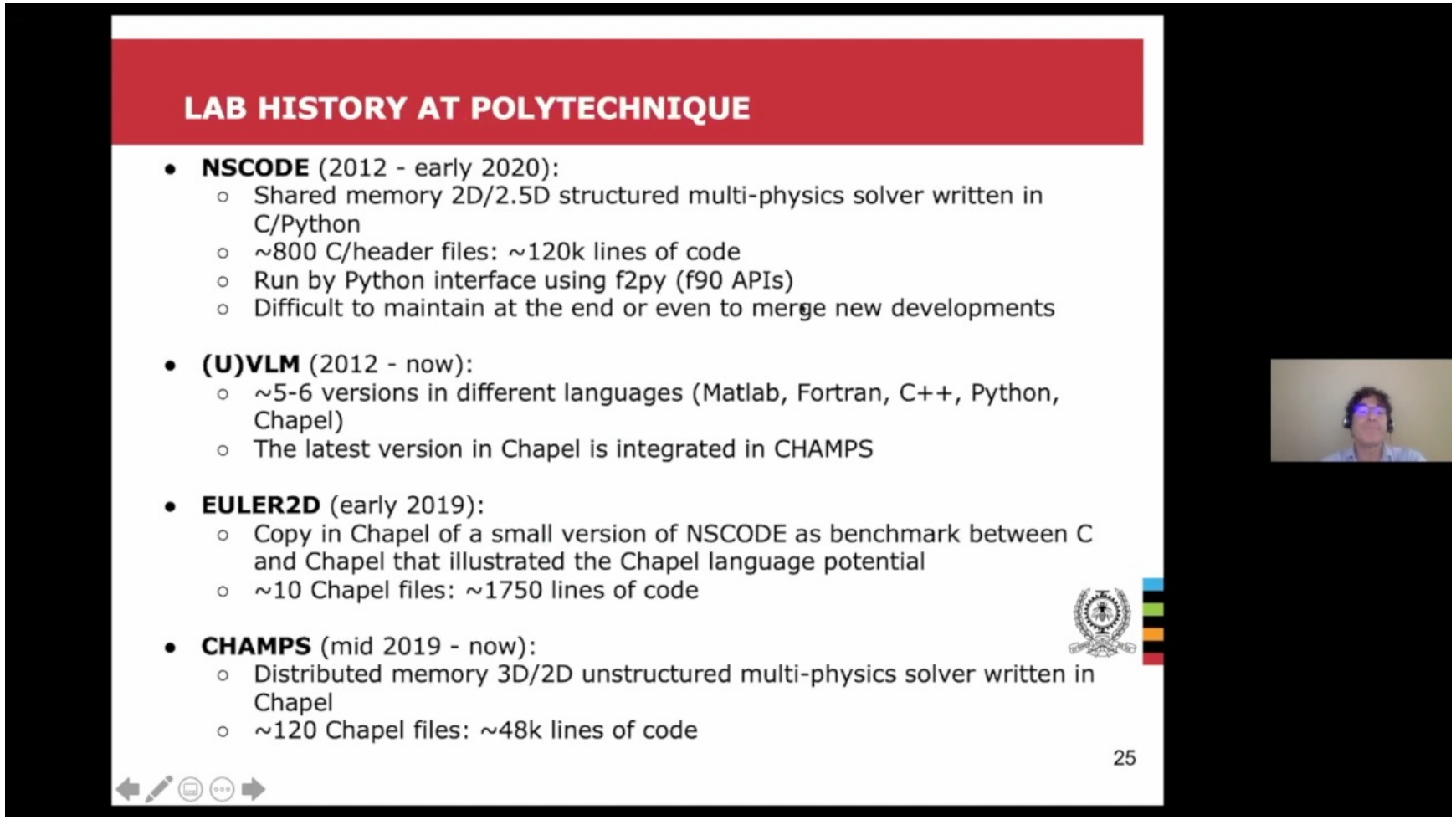
Why Chapel?

- performance and scalability competitive with MPI + C++
- students found it far more productive to use



CHAMPS: EXCERPT FROM ERIC'S CHIUW 2021 KEYNOTE (VIDEO)

HPC Lessons From 30 Years of Practice in CFD Towards Aircraft Design and Analysis



POLYTECHNIQUE MONTRÉAL

https://youtu.be/wD-a_KyB8al?t=1904

CHAMPS: EXCERPT FROM ERIC'S CHIUW 2021 KEYNOTE

HPC Lessons From 30 Years of Practice in CFD Towards Aircraft Design and Analysis

"To show you what Chapel did in our lab… [NSCODE, our previous framework] ended up 120k lines. And my students said, 'We can't handle it anymore. It's too complex, we lost track of everything.' And today, they went ***from 120k lines to 48k lines, so 3x less****.*

But the code is not 2D, it's 3D. And it's not structured, it's unstructured, which is way more complex. And it's multi-physics: aeroelastic, aero-icing. ***So, I've got industrial-type code in 48k lines.***

So, for me, this is like the proof of the benefit of Chapel, ***plus the smiles I have on my students everyday in the lab because they love Chapel as well.*** *So that's the key, that's the takeaway.*

POLYTECHNIQUE MONTRÉAL

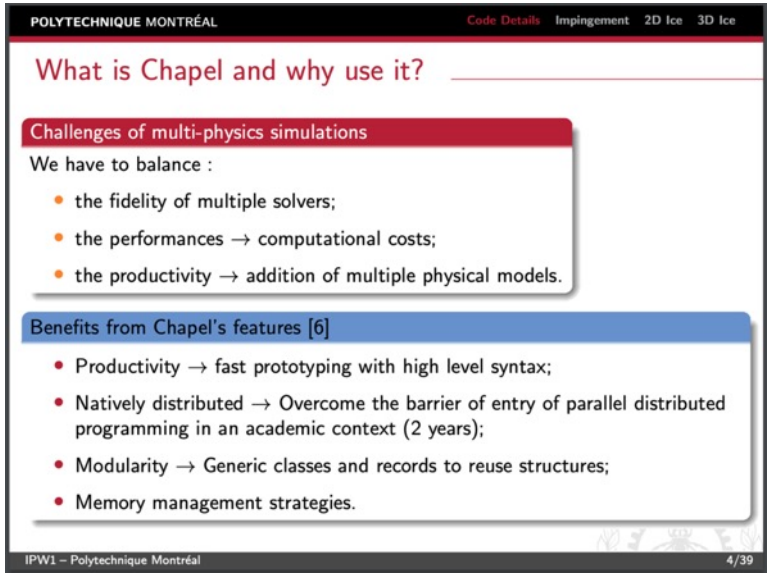
*[Chapel] promotes the programming efficiency … **We ask students at the master's degree to do stuff that would take 2 years and they do it in 3 months.** So, if you want to take a summer internship and you say, 'program a new turbulence model,' well they manage. And before, it was impossible to do."*

- Talk available online: https://youtu.be/wD-a_KyB8aI?t=1904 (hyperlink jumps to the section quoted here)

CHAMPS 2021 HIGHLIGHTS

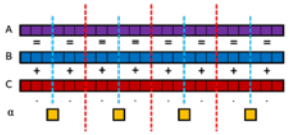
- Presented at CASI/IASC Aero 21 Conference
- Participated in 1st AIAA Ice Prediction Workshop
- Participating in 4th AIAA CFD High-lift Prediction Workshop
- Student presentation to CFD Society of Canada (CFDSC)

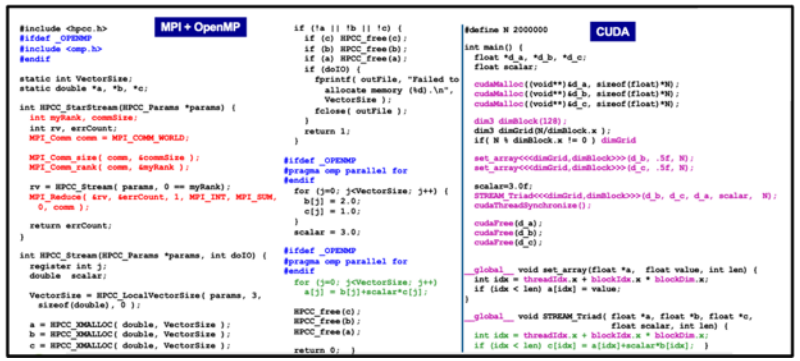
- **Achieving large-scale, high-quality results comparable to other major players in industry, government, academia:**
  - e.g., Boeing, Lockheed Martin, NASA, JAXA, Georgia Tech, …



SUMMARY OF THIS SECTION

- Conventional HPC programming notations are not particularly productive
  - they utilize too many distinct ways of specifying locality and parallelism
  - they are too specific to certain flavors of locality or parallelism

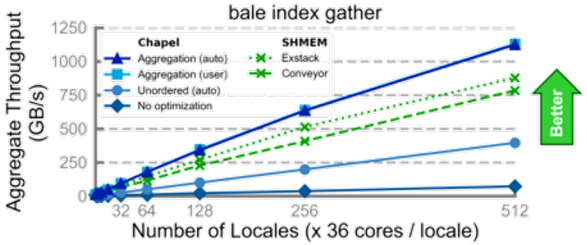


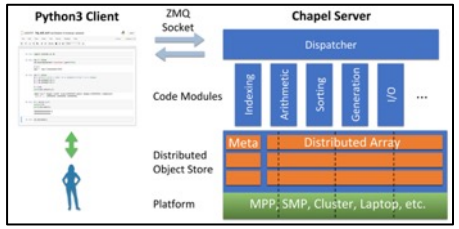


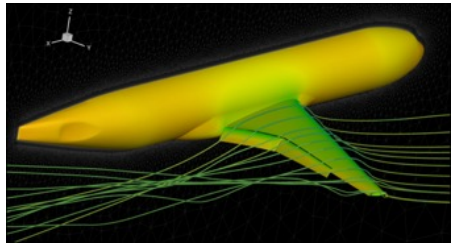
- Chapel's support for parallelism and locality supports…

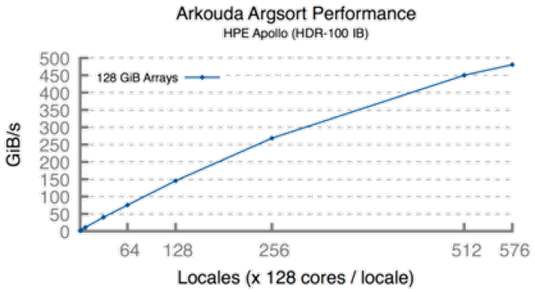
  …concise, clear, yet portable benchmarks

  …user applications that are productive and scalable





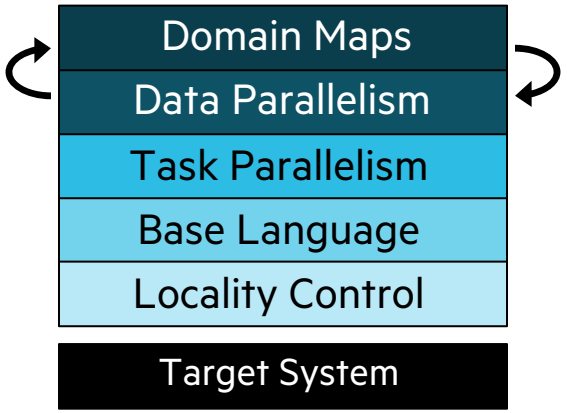




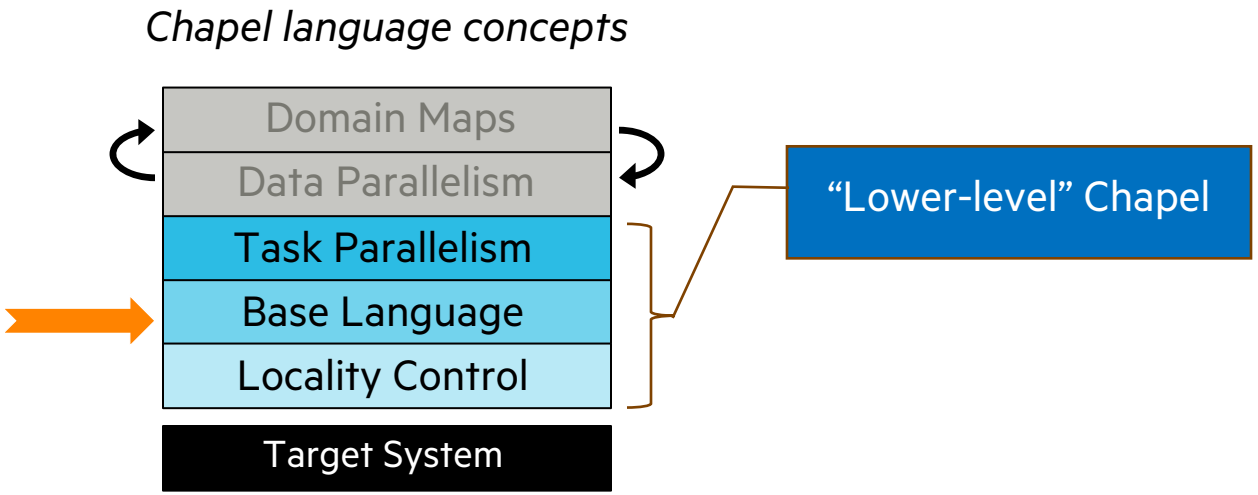
INTRODUCTION TO CHAPEL

CHAPEL FEATURE AREAS

Chapel language concepts



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;
  }
}
```

```
prompt> chpl fib.chpl
prompt>
```

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)

```
prompt> chpl fib.chpl
prompt> ./fib
```

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;
  }
}
```

Execute the loop's body for that value

'yield' this expression back to the loop's index variable

```
prompt> chpl fib.chpl
prompt> ./fib
0
```

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;
  }
}
```

Execute the loop's body for that value

Then continue the iterator from where it left off

Repeating until we fall out of it (or return)

```
prompt> chpl fib.chpl
prompt> ./fib
0
1
1
2
3
5
8
13
21
34
```

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;
  }
}
```

Config[urable] declarations support command-line overrides

```
prompt> chpl fib.chpl
prompt> ./fib --n=1000
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;
  }
}
```

Static type inference for:

- constants / variables
- arguments
- return types

Explicit typing also supported

```
prompt> chpl fib.chpl
prompt> ./fib --n=1000
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
…
```

FIBONACCI ITERATION

fib.chpl

```
config const n: int = 10;

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

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

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

Explicit typing also supported

```
prompt> chpl fib.chpl
prompt> ./fib --n=1000
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
…
```

FIBONACCI ITERATION

fib.chpl

```
config const n = 10;

for (i,f) in zip(0..<n, fib(n)) do
  writeln("fib #", i, " is ", f);

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

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

Zippered iteration

```
prompt> chpl fib.chpl
prompt> ./fib --n=1000
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
fib #7 is 13
fib #8 is 21
fib #9 is 34
fib #10 is 55
fib #11 is 89
fib #12 is 144
fib #13 is 233
fib #14 is 377
…
```

FIBONACCI ITERATION

fib.chpl

```
config const n = 10;

for (i,f) in zip(0..<n, fib(n)) do
  writeln("fib #", i, " is ", f);

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

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

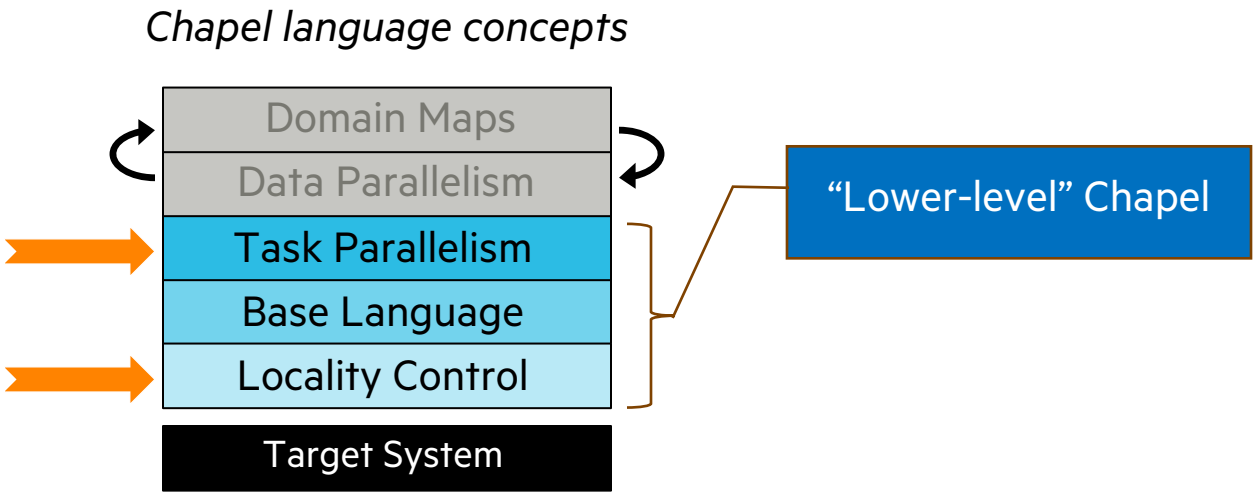
Range types and operators

```
prompt> chpl fib.chpl
prompt> ./fib --n=1000
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
fib #7 is 13
fib #8 is 21
fib #9 is 34
fib #10 is 55
fib #11 is 89
fib #12 is 144
fib #13 is 233
fib #14 is 377
…
```

OTHER BASE LANGUAGE FEATURES

- **Various basic types:** bool(w), int(w), uint(w), real(w), imag(w), complex(w), enums, tuples
- **Object-oriented programming**
  - Value-based records (like C structs supporting methods, generic fields, etc.)
  - Reference-based classes (somewhat like Java classes or C++ pointers-to-classes)
    - Nilable vs. non-nilable variants
    - Memory-management strategies (shared, owned, borrowed, unmanaged)
    - Lifetime checking
- **Error-handling**
- **Generic programming / polymorphism**
- **Compile-time meta-programming**
- **Modules** (supporting namespaces)
- **Procedure overloading / filtering**
- **Arguments:** default values, intents, name-based matching, type queries
- and more…

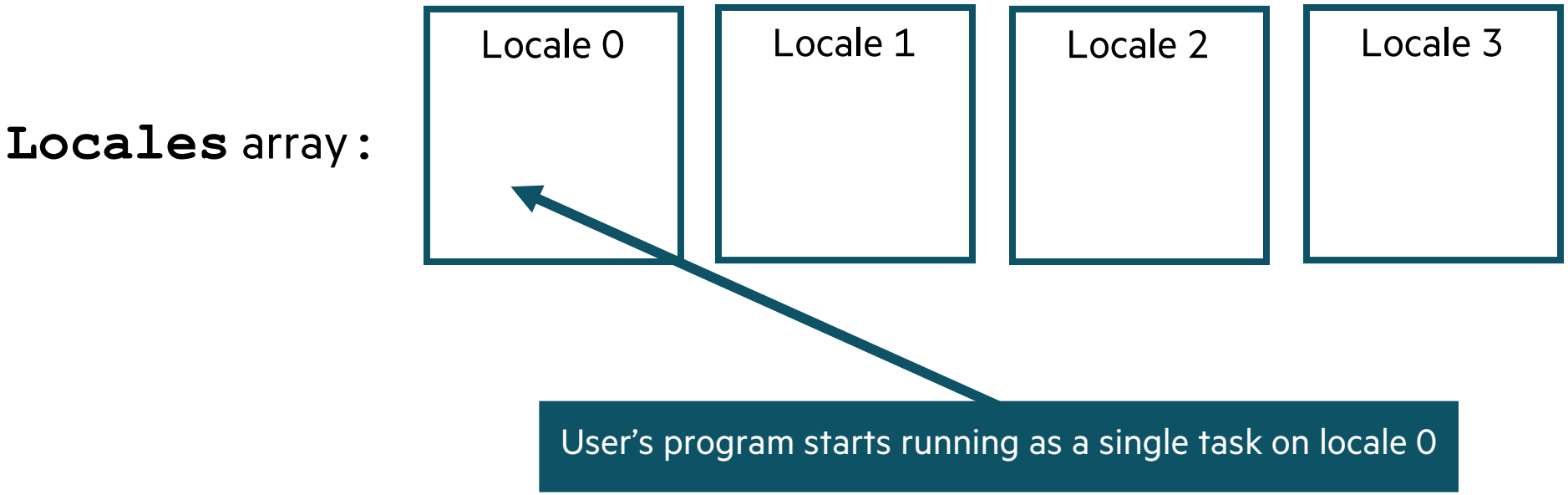
TASK PARALLELISM AND LOCALITY CONTROL



THE LOCALE: CHAPEL'S KEY FEATURE FOR LOCALITY

- *locale*: a unit of the target architecture that can run tasks and store variables
  - Think "compute node" on a typical HPC system

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



TASK-PARALLEL “HELLO WORLD”

helloTaskPar.chpl

```
const numTasks = here.maxTaskPar;
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.maxTaskPar;
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 this code is currently running

how many parallel tasks can my locale run at once?

what's my locale's name?

TASK-PARALLEL "HELLO WORLD"

helloTaskPar.chpl

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

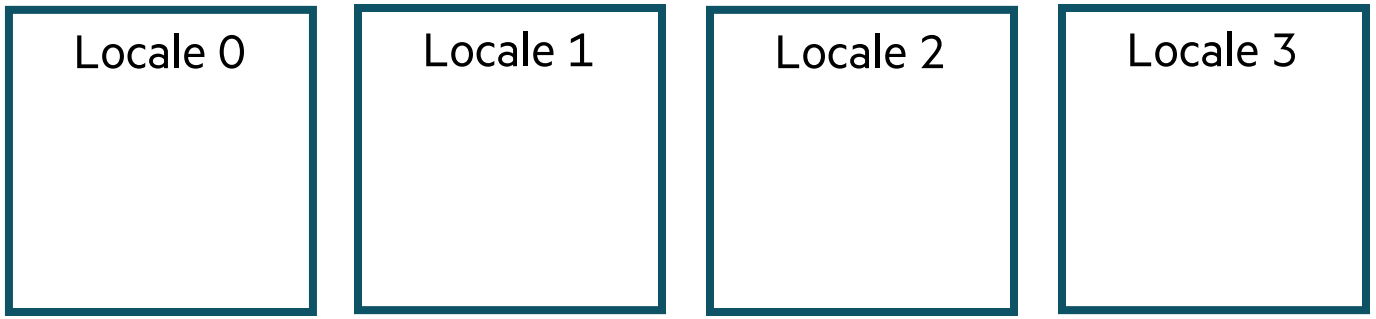
TASK-PARALLEL "HELLO WORLD" (DISTRIBUTED VERSION)

helloTaskPar.chpl

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

the array of locales we're running on (introduced a few slides back)

Locales array:



TASK-PARALLEL "HELLO WORLD" (DISTRIBUTED VERSION)

helloTaskPar.chpl

```
coforall loc in Locales {
  on loc {
    const numTasks = here.maxTaskPar;
    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
…
```

TASK-PARALLEL "HELLO WORLD" (DISTRIBUTED VERSION)

helloTaskPar.chpl

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

PARALLELISM AND LOCALITY ARE ORTHOGONAL IN CHAPEL

- This is a parallel, but local program:

```
coforall i in 1..msgs do
  writeln("Hello from task ", i);
```

- This is a distributed, but serial program:

```
writeln("Hello from locale 0!");
on Locales[1] do writeln("Hello from locale 1!");
on Locales[2] {
  writeln("Hello from locale 2!");
  on Locales[0] do writeln("Hello from locale 0!");
}
writeln("Back on locale 0");
```

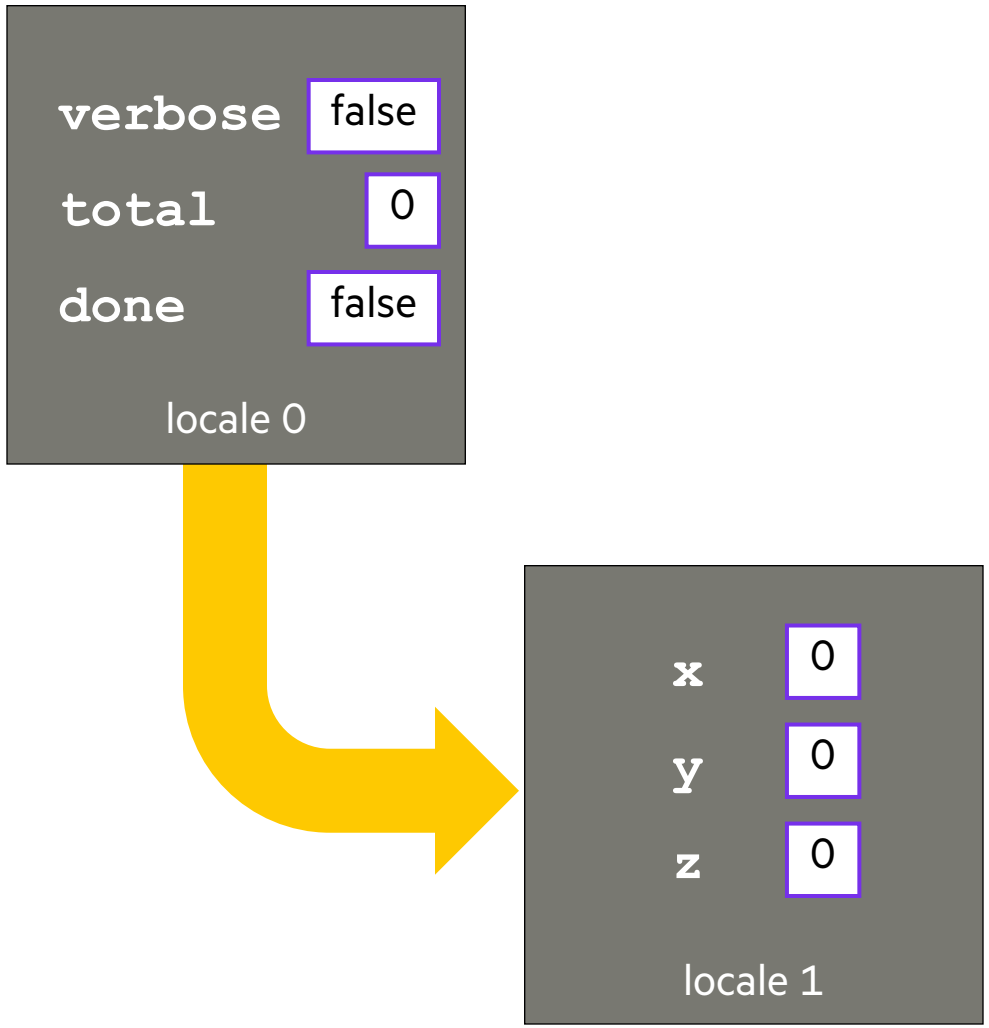
- This is a distributed parallel program:

```
coforall i in 1..msgs do
  on Locales[i%numLocales] do
    writeln("Hello from task ", i, " running on locale ", here.id);
```

VARIABLES ARE ALLOCATED LOCALLY TO WHERE THE TASK IS RUNNING

onClause.chpl

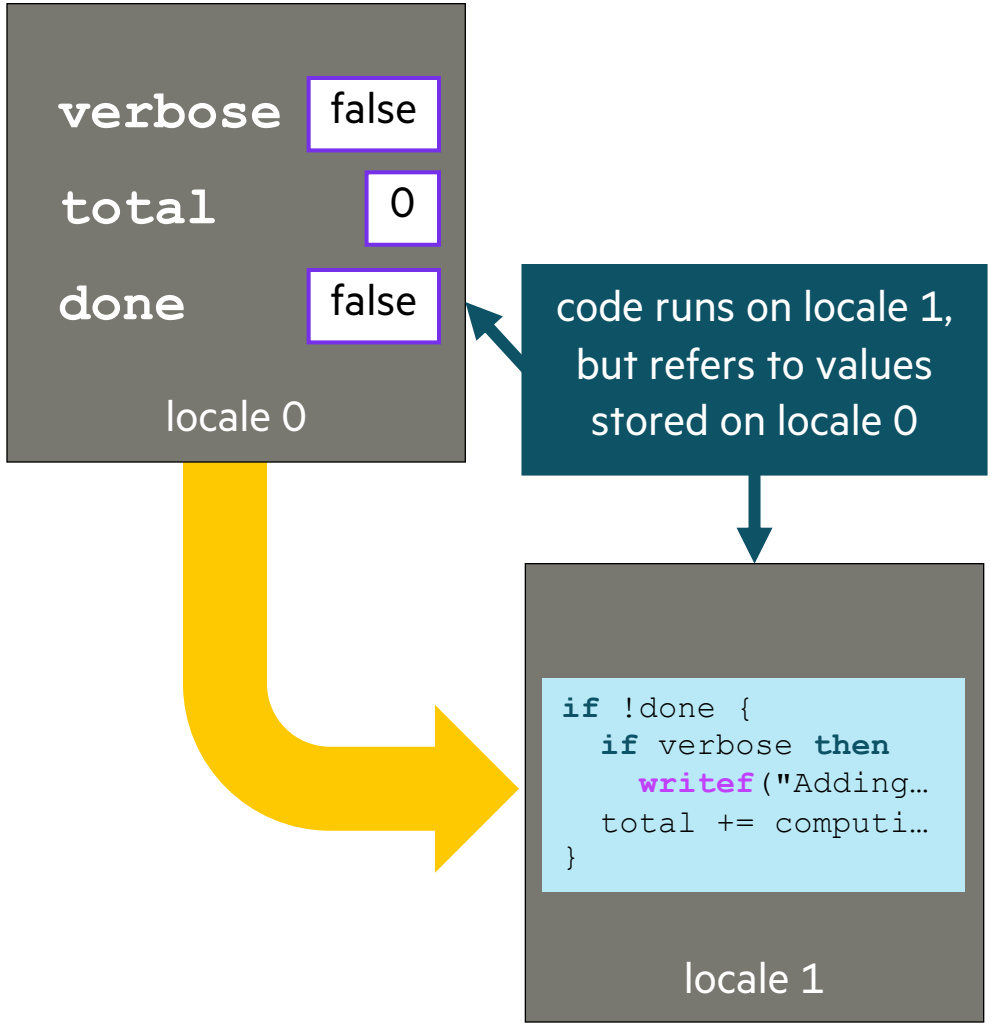
```
config const verbose = false;
var total = 0,
    done = false;

…

on Locales[1] {
  var x, y, z: int;
  …



}
```



CODE CAN REFER TO VISIBLE VARIABLES, EVEN WHEN THEY'RE REMOTE

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();
  }
}
```



SIDEBAR: CHAPEL'S DECEPTIVE SIMPLICITY

Chapel resembles traditional programming enough that it's easy to forget how roundabout SPMD can be:

- Need to do something on just one node?
  - **Chapel:** OK, just do it
  - **SPMD:** Make sure you're only doing it in one process

toy.chpl

```
proc main() {
  var x = stdin.read(int);
  writeln("Hello!");

  coforall loc in Locales do
    on loc do
      writeln(loc.id * x);

  writeln("Bye!");
}
```

toy-SPMD.chpl

```
proc main() {
  var x: int;
  if myProc() == 0 {
    x = stdin.read(int);
    writeln("Hello!");
  }

  broadcastAll(x, fromLocale=0);

  writeln(myProc() * x);

  barrierAll();

  if myProc() == 0 then
    writeln("Bye!");
}
```

SIDEBAR: CHAPEL'S DECEPTIVE SIMPLICITY

Chapel resembles traditional programming enough that it's easy to forget how roundabout SPMD can be:

- Want to ensure one thing finishes before the next?
  - **Chapel:** Typically happens through sequential ordering
  - **SPMD:** Defensively ensure nobody gets too far ahead

toy.chpl

```
proc main() {
  var x = stdin.read(int);
  writeln("Hello!");

  coforall loc in Locales do
    on loc do
      writeln(loc.id * x);

  writeln("Bye!");
}
```

toy-SPMD.chpl

```
proc main() {
  var x: int;
  if myProc() == 0 {
    x = stdin.read(int);
    writeln("Hello!");
  }

  broadcastAll(x, fromLocale=0);

  writeln(myProc() * x);

  barrierAll();

  if myProc() == 0 then
    writeln("Bye!");
}
```

SIDEBAR: CHAPEL'S DECEPTIVE SIMPLICITY

Chapel resembles traditional programming enough that it's easy to forget how roundabout SPMD can be:

- Want to refer to a remote variable?
  - **Chapel:** Is it in your lexical scope? Just name it!
  - **SPMD:** Insert communication, potentially in both the source and destination processes

toy.chpl

```
proc main() {
  var x = stdin.read(int);
  writeln("Hello!");

  coforall loc in Locales do
    on loc do
      writeln(loc.id * x);

  writeln("Bye!");
}
```

toy-SPMD.chpl

```
proc main() {
  var x: int;
  if myProc() == 0 {
    x = stdin.read(int);
    writeln("Hello!");
  }

  broadcastAll(x, fromLocale=0);

  writeln(myProc() * x);

  barrierAll();

  if myProc() == 0 then
    writeln("Bye!");
}
```

SIDEBAR: CHAPEL'S DECEPTIVE SIMPLICITY

Chapel resembles traditional programming enough that it's easy to forget how roundabout SPMD can be:

- Need some additional parallelism?
  - **Chapel:** we have features for that, like coforall, logically independent of hardware resources
  - **SPMD:** Umm… Well, I suppose you could mix in OpenMP, Pthreads, or CUDA…

toy.chpl

```
proc main() {
  var x = stdin.read(int);
  writeln("Hello!");

  coforall loc in Locales do
    on loc do
      writeln(loc.id * x);

  writeln("Bye!");
}
```

toy-SPMD.chpl

```
proc main() {
  var x: int;
  if myProc() == 0 {
    x = stdin.read(int);
    writeln("Hello!");
  }

  broadcastAll(x, fromLocale=0);

  writeln(myProc() * x);

  barrierAll();

  if myProc() == 0 then
    writeln("Bye!");
}
```

SIDEBAR: CHAPEL'S DECEPTIVE SIMPLICITY

Chapel resembles traditional programming enough that it's easy to forget how roundabout SPMD can be:

- And of course, if what you really want is SPMD, Chapel can do that as well…

toy.chpl

```
proc main() {
  var x = stdin.read(int);
  writeln("Hello!");

  coforall loc in Locales do
    on loc do
      writeln(loc.id * x);

  writeln("Bye!");
}
```

toy-SPMD.chpl

```
proc main() {
  var x: int;
  if myProc() == 0 {
    x = stdin.read(int);
    writeln("Hello!");
  }

  broadcastAll(x, fromLocale=0);

  writeln(myProc() * x);

  barrierAll();

  if myProc() == 0 then
    writeln("Bye!");
}
```

OTHER TASK PARALLEL FEATURES

- **begin / cobegin statements:** the two other ways of creating tasks

```
begin stmt;      // fire off an asynchronous task to run 'stmt'
```

```
cobegin {        // fire off a task for each of 'stmt1', 'stmt2', …
  stmt1;
  stmt2;
  stmt3;
  …
}                // wait here for these tasks to complete before proceeding
```

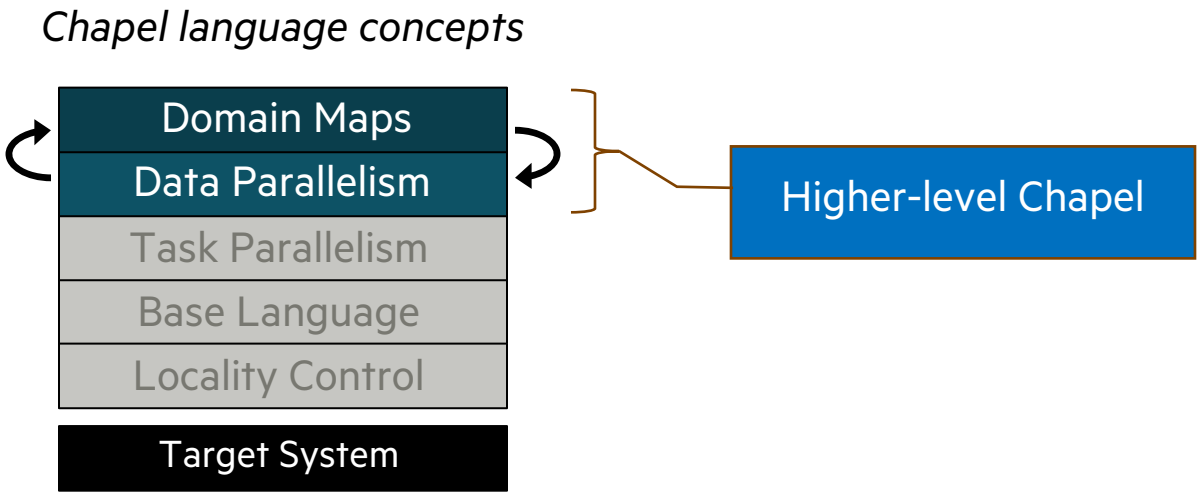
- **atomic / synchronized variables:** types for safe data sharing & coordination between tasks

```
var sum: atomic int;     // supports various atomic methods like .add(), .compareExchange(), …
var cursor: sync int;    // stores a full/empty bit governing reads/writes, supporting .readEF(), .writeEF()
```

- **task intents / task-private variables:** control how variables and tasks relate

```
coforall i in 1..niters with (ref x, + reduce y, var z: int) { … }
```

DATA PARALLELISM AND DOMAIN MAPS



DATA-PARALLEL ARRAY FILL

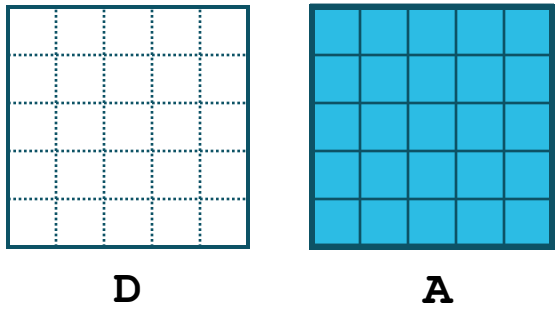
fillArray.chpl

```
config const n = 1000;

const 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;

const 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);
```



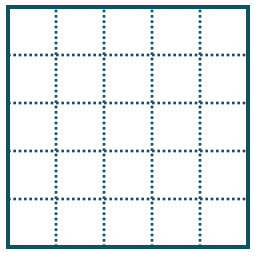
declare a domain, a first-class index set

declare an array over that domain

DATA-PARALLEL ARRAY FILL

fillArray.chpl

```
config const n = 1000;

const 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

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

declare a domain, a first-class index set

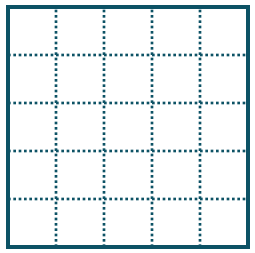
declare an array over that domain

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

DATA-PARALLEL ARRAY FILL

fillArray.chpl

```
config const n = 1000;

const 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

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

```
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
```

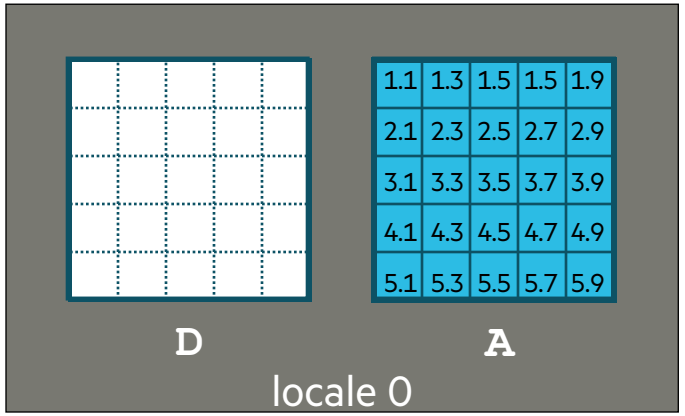
So far, this is a shared-memory program

Nothing refers to remote locales, explicitly or implicitly

DATA-PARALLEL ARRAY FILL

fillArray.chpl

```
config const n = 1000;

const 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

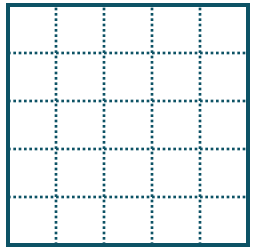
fillArray.chpl

```
config const n = 1000;

const 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;

const 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

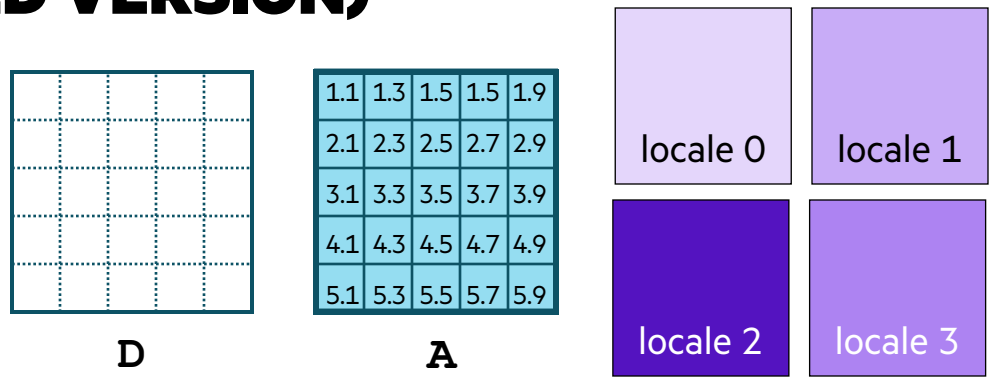
1.1	1.3	1.5	1.5	1.9
2.1	2.3	2.5	2.7	2.9
3.1	3.3	3.5	3.7	3.9
4.1	4.3	4.5	4.7	4.9
5.1	5.3	5.5	5.7	5.9

A

DATA-PARALLEL ARRAY FILL (DISTRIBUTED VERSION)

fillArray.chpl

```
use CyclicDist;

config const n = 1000;

const 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);
```

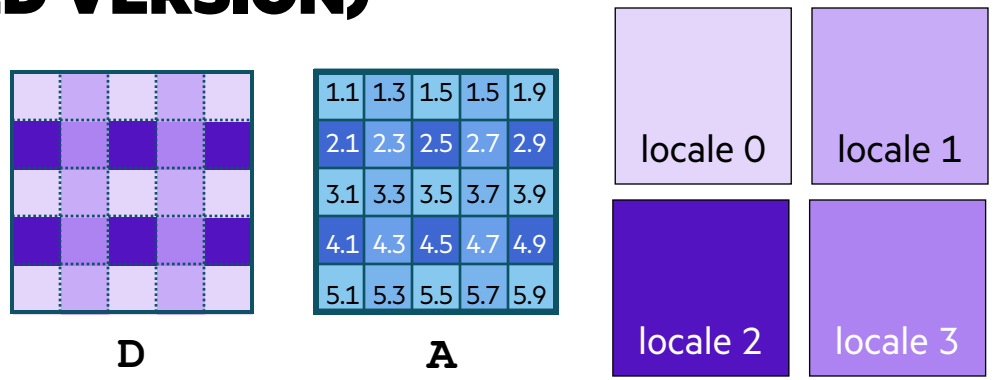


apply a domain map, specifying how to implement…
…the domain's indices,
…the array's elements,
…the loop's iterations,
…on the program's locales

DATA-PARALLEL ARRAY FILL (DISTRIBUTED VERSION)

fillArray.chpl

```
use CyclicDist;

config const n = 1000;

const 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);
```

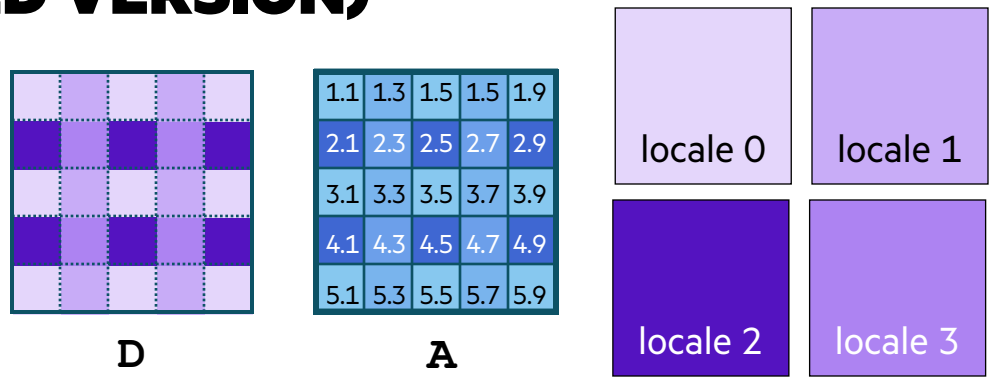


apply a domain map, specifying how to implement…
…the domain's indices,
…the array's elements,
…the loop's iterations,
…on the program's locales

DATA-PARALLEL ARRAY FILL (DISTRIBUTED VERSION)

fillArray.chpl

```
use CyclicDist;

config const n = 1000;

const 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);
```

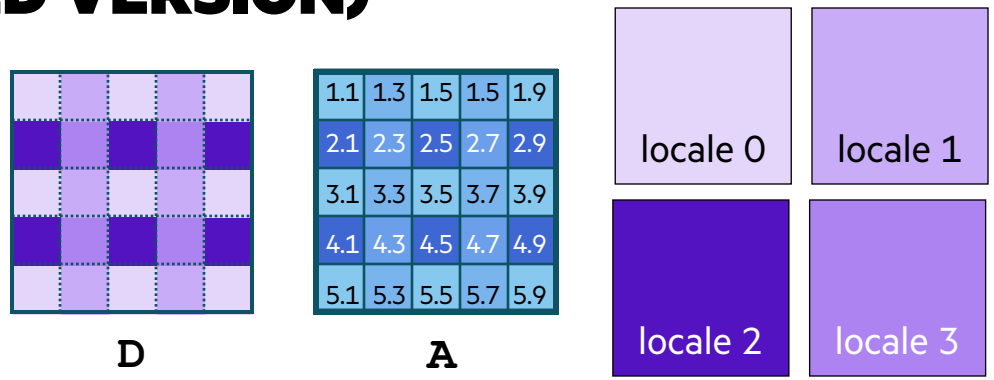



```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

DATA-PARALLEL ARRAY FILL (DISTRIBUTED VERSION)

fillArray.chpl

```
use CyclicDist;

config const n = 1000;

const 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;

const 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);
```

SPECTRUM OF CHAPEL FOR-LOOP STYLES

for loop: each iteration is executed serially by the current task

- predictable execution order, similar to conventional languages

foreach loop: all iterations executed by the current task, but in no specific order

- a candidate for vectorization, SIMD execution on GPUs

forall loop: all iterations are executed by one or more tasks in no specific order

- implemented using one or more tasks, locally or distributed, as determined by the iterand expression

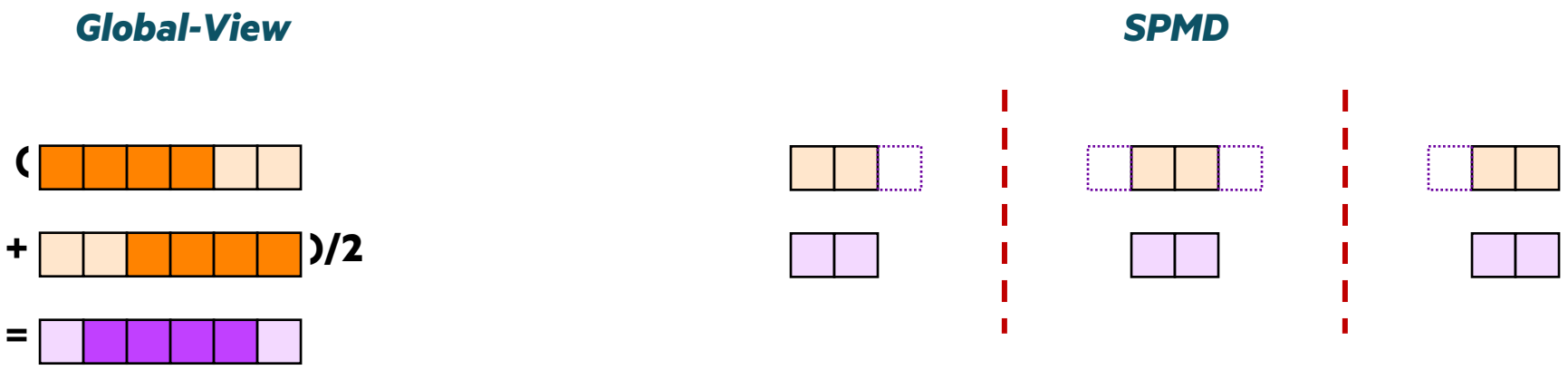
```
forall i in 1..n do …                    // forall loops over ranges use local tasks only
forall (i,j) in {1..n, 1..n} do …        // ditto for local domains…
forall elem in myLocArr do …             // …and local arrays
forall elem in myDistArr do …            // distributed arrays use tasks on each locale owning part of the array
forall i in myParIter(…) do …            // you can also write your own iterators that use the policy you want
```

coforall loop: each iteration is executed concurrently by a distinct task

- explicit parallelism; supports synchronization between iterations (tasks)

CHAPEL'S GLOBAL-VIEW OF DATA-PARALLELISM VS. SPMD

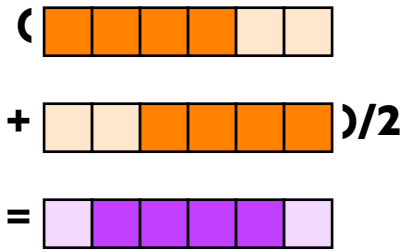
- "Apply a 3-point stencil to a vector"



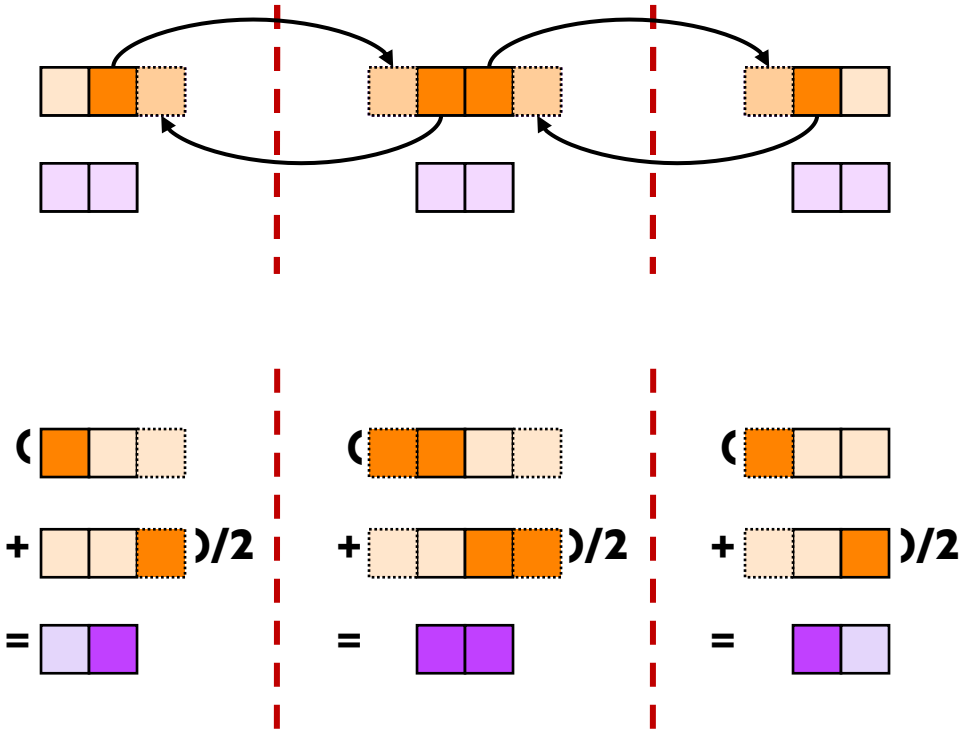
CHAPEL'S GLOBAL-VIEW OF DATA-PARALLELISM VS. SPMD

- "Apply a 3-point stencil to a vector"

Global-View



SPMD



CHAPEL'S GLOBAL-VIEW VS. SPMD

- "Apply a 3-point stencil to a vector"

Global-View

```
proc main() {
  var n = 1000;
  var A, B: [1..n] real;

  forall i in 2..n-1 do
    B[i] = (A[i-1] + A[i+1])/2;
}
```

SPMD (MPI-style)

```
proc main() {
  var n = 1000;
  var p = numProcs(),
      me = myProc(),
      myN = n/p,
  var A, B: [0..myN+1] real;

  if (me < p-1) {
    send(me+1, A[myN]);
    recv(me+1, A[myN+1]);
  }
  if (me > 0) {
    send(me-1, A[1]);
    recv(me-1, A[0]);
  }
  forall i in 1..myN do
    B[i] = (A[i-1] + A[i+1])/2;
}
```

Bug: Refers to uninitialized values at ends of A

CHAPEL'S GLOBAL-VIEW VS. SPMD

- "Apply a 3-point stencil to a vector"

Global-View

```
proc main() {
  var n = 1000;
  var A, B: [1..n] real;

  forall i in 2..n-1 do
    B[i] = (A[i-1] + A[i+1])/2;
}
```

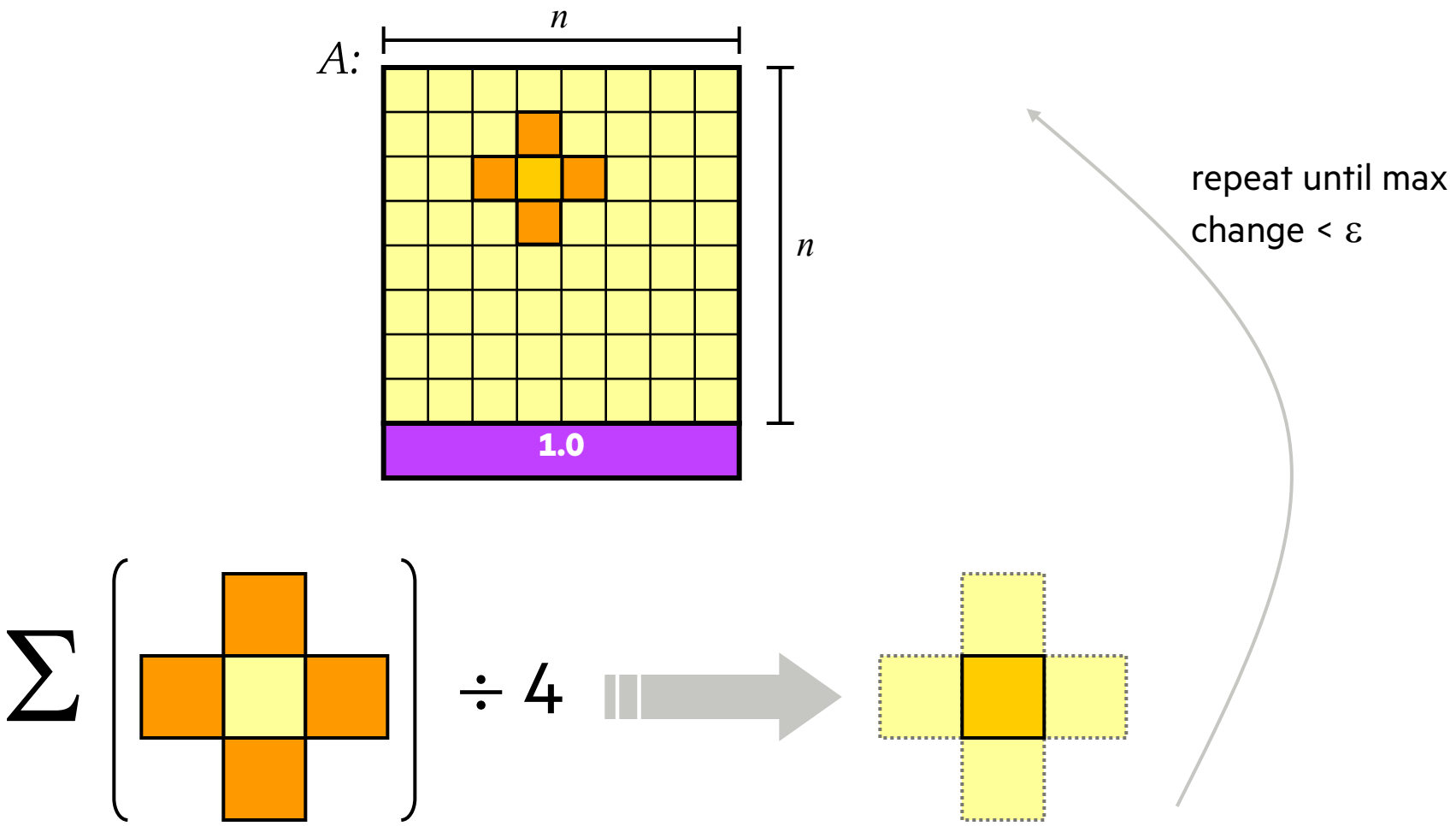
SPMD (MPI-style)

```
proc main() {
  var n = 1000;
  var p = numProcs(),
      me = myProc(),
      myN = n/p,
      myLo = 1,
      myHi = myN;
  var A, B: [0..myN+1] real;

  if (me < p-1) {
    send(me+1, A[myN]);
    recv(me+1, A[myN+1]);
  } else
    myHi = myN-1;
  if (me > 0) {
    send(me-1, A[1]);
    recv(me-1, A[0]);
  } else
    myLo = 2;
  forall i in myLo..myHi do
    B[i] = (A[i-1] + A[i+1])/2;
}
```

Assumes p evenly divides n

Communication becomes geometrically more complex for higher-dimensional arrays

JACOBI ITERATION IN PICTURES



JACOBI ITERATION IN CHAPEL

```
config const n = 6, epsilon = 0.01;

const AllInds = {0..n+1, 0..n+1},
            D = AllInds[1..n, 1..n],
      LastRow = AllInds[n+1..n+1, ..];

var A, Temp: [AllInds] real;



A[LastRow] = 1.0;

do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while delta > epsilon;

writeln(A);
```

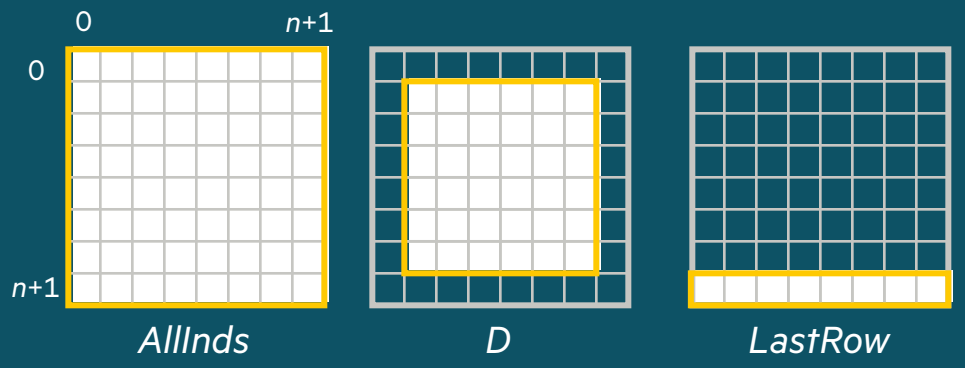
JACOBI ITERATION IN CHAPEL

```
config const n = 6, epsilon = 0.01;

const AllInds = {0..n+1, 0..n+1},
            D = AllInds[1..n, 1..n],
      LastRow = AllInds[n+1..n+1, ..];

var A, Temp: [AllInds] real;



A[LastRow] = 1.0;

do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] +

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while delta > epsilon;

writeln(A);
```

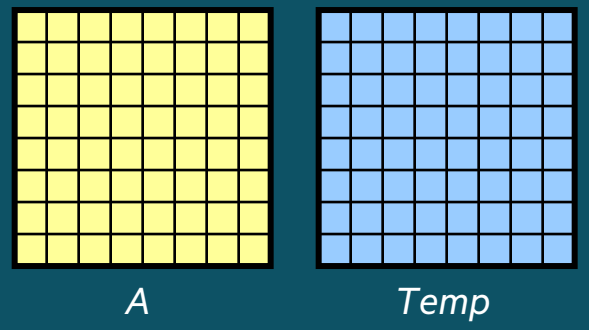
Declare configurable problem size and termination condition

Declare domains

- Slicing one domain with another computes the intersection



Declare arrays

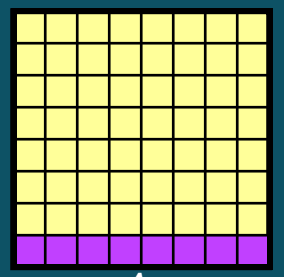


JACOBI ITERATION IN CHAPEL

```
config const n = 6, epsilon = 0.01;

const AllInds = {0..n+1, 0..n+1},
            D = AllInds[1..n, 1..n],
      LastRow = AllInds[n+1..n+1, ..];

var A, Temp: [AllInds] real;



A[LastRow] = 1.0;

do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while delta > epsilon;

writeln(A);
```

Set Explicit Boundary Condition

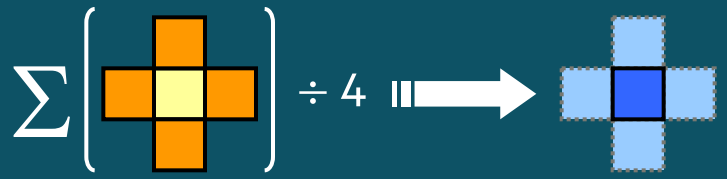
- indexing by a domain refers to the subarray in question
- scalar values are “promoted” when assigned to arrays
- “whole-array” operations like this are implicitly parallel



JACOBI ITERATION IN CHAPEL

```
config const n = 6, epsilon = 0.01;

const AllInds = {0..n+1, 0..n+1},
            D = AllInds[1..n, 1..n],
      LastRow = AllInds[n+1..n+1, ..];

var A, Temp: [AllInds] real;



A[LastRow] = 1.0;

do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while delta > epsilon;

writeln(A);
```

Compute 5-point stencil



JACOBI ITERATION IN CHAPEL

```
config const n = 6, epsilon = 0.01;

const AllInds = {0..n+1, 0..n+1},
            D = AllInds[1..n, 1..n],
      LastRow = AllInds[n+1..n+1, ..];

var A, Temp: [AllInds] real;

A[LastRow] = 1.0;

do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while delta > epsilon;

writeln(A);
```

Compute maximum change

- *op* **reduce** *expr* ⇒ collapse aggregate expression '*expr*' to a scalar using '*op*'
- abs() and '–' are scalar operations; calling them with array arguments results in parallel evaluation
- no temporary arrays are created when evaluating this statement

SIDEBAR: PROMOTION OF SCALAR SUBROUTINES

- Any function or operator that takes scalar arguments can be called with array expressions instead

```
proc foo(x: real, y: real, z: real) {
  return x**y + 10*c;
}
```

- Interpretation is similar to that of a zippered forall loop, thus:

```
C = foo(A, 2, B);
```

is equivalent to:

```
forall (c, a, b) in zip(C, A, B) do
  c = foo(a, 2, b);
```

as is:

```
C = A**2 + 10*c;
```

- So, in the Jacobi computation,

```
abs(A[D] - Temp[D]);
```

==

```
forall (a,t) in zip(A[D], Temp[D]) do abs(a - t);
```

JACOBI ITERATION IN CHAPEL

```
config const n = 6, epsilon = 0.01;

const AllInds = {0..n+1, 0..n+1},
            D = AllInds[1..n, 1..n],
      LastRow = AllInds[n+1..n+1, ..];

var A, Temp: [AllInds] real;



A[LastRow] = 1.0;

do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while delta > epsilon;

writeln(A);
```

Wrap up

- assign Temp back to A for next iteration
- see whether we terminate using normal do…while loop
- print out final array once we're done

JACOBI ITERATION IN CHAPEL (NAMED, TUPLE-INDEXED VARIANT)

```
config const n = 6, epsilon = 0.01;

const AllInds = {0..n+1, 0..n+1},
            D = AllInds[1..n, 1..n],
      LastRow = AllInds[n+1..n+1, ..];

var A, Temp: [AllInds] real;

const north = (-1,0), south = (1,0), east = (0,1), west = (0,-1);

A[LastRow] = 1.0;

do {
  forall ij in D do
    Temp[ij] = (A[ij+north] + A[ij+south] + A[ij+east] + A[ij+west]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while delta > epsilon;

writeln(A);
```

JACOBI ITERATION IN CHAPEL (SLICE-BASED VARIANT)

```
config const n = 6, epsilon = 0.01;

const AllInds = {0..n+1, 0..n+1},
            D = AllInds[1..n, 1..n],
      LastRow = AllInds[n+1..n+1, ..];

var A, Temp: [AllInds] real;

const north = (-1,0), south = (1,0), east = (0,1), west = (0,-1);

A[LastRow] = 1.0;

do {
  Temp[D] = (A[D.translate(north)] + A[D.translate(south)] +
             A[D.translate(east)]  + A[D.translate(west)]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while delta > epsilon;

writeln(A);
```

JACOBI ITERATION IN CHAPEL (BACK TO THE SIMPLE, LOCAL VERSION)

```
config const n = 6, epsilon = 0.01;

const AllInds = {0..n+1, 0..n+1},
            D = AllInds[1..n, 1..n],
      LastRow = AllInds[n+1..n+1, ..];

var A, Temp: [AllInds] real;



A[LastRow] = 1.0;

do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while delta > epsilon;

writeln(A);
```

JACOBI ITERATION IN CHAPEL (DISTRIBUTED VERSION)

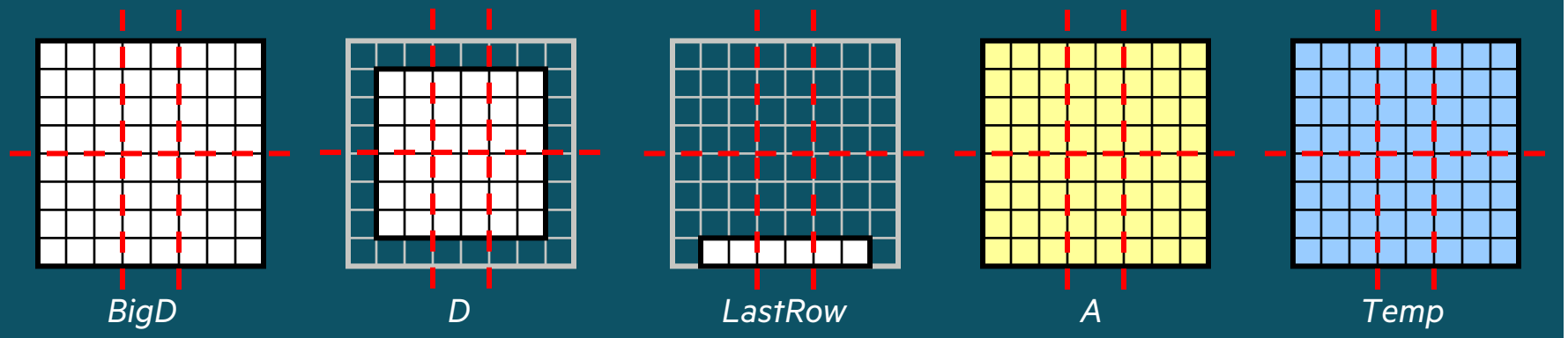
```
use BlockDist;

config const n = 6, epsilon = 0.01;

const AllInds = {0..n+1, 0..n+1} dmapped Block({1..n, 1..n}),
            D = AllInds[1..n, 1..n],
      LastRow = AllInds[n+1..n+1, ..];

var A, Temp: [AllInds] real;



A[LastRow] = 1.0;

do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while delta > epsilon;

writeln(A);
```

JACOBI ITERATION IN CHAPEL (DISTRIBUTED VERSION)

```
use BlockDist;

config const n = 6, epsilon = 0.01;

const AllInds = {0..n+1, 0..n+1} dmapped Block({1..n, 1..n}),
            D = AllInds[1..n, 1..n],
      LastRow = AllInds[n+1..n+1, ..];

var A, Temp: [AllInds] real;



A[LastRow] = 1.0;

do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,

  const delta = max red
  A[D] = Temp[D];
} while delta > epsilon

writeln(A);
```

With these two changes, we distribute our domains, and therefore our computation

- Domain slices inherit parent domain's distribution



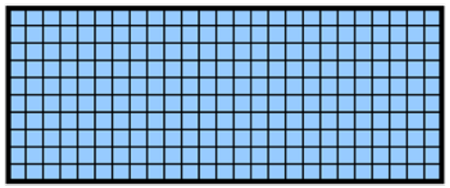
A FINAL NOTE ON THESE JACOBI EXAMPLES

- The previous slides were developed primarily to demonstrate data parallel features in Chapel
  - not necessarily to suggest “this is the best way to do Jacobi in Chapel”
  - specifically, we haven’t done any benchmarking or tuning of Jacobi as it hasn’t been of deep interest to our users

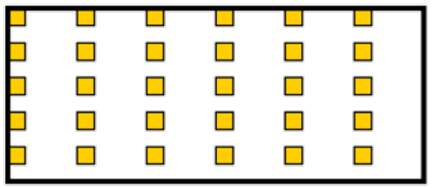
- If one wanted to do Jacobi in Chapel, there are a few other approaches to consider:
  - there’s a 'Stencil’ distribution that is similar to ‘Block’ yet with a notion of ghost cells for caching neighbor values
  - and if one were to do a comparison, it’d be good to compare with a more manual SPMD version in Chapel as well

OTHER DATA PARALLEL FEATURES

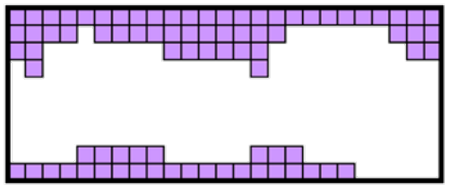
- **Scans:** parallel prefix operations
- **User-defined Parallel Iterators, Reduce/Scan Operations**
- **Several Domain/Array Types:**



dense



strided

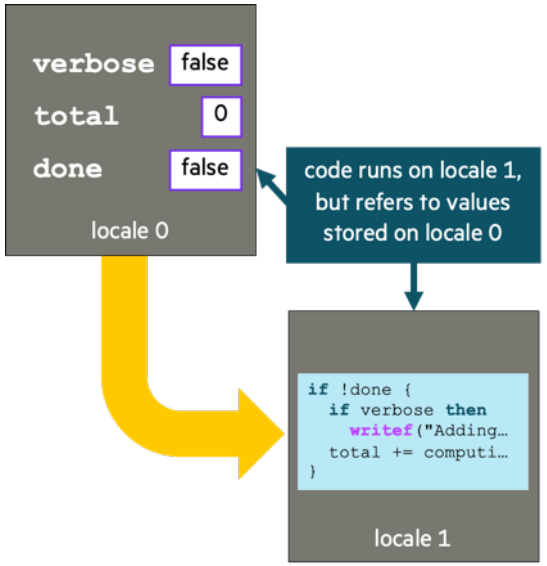


sparse

associative

SUMMARY OF THIS SECTION

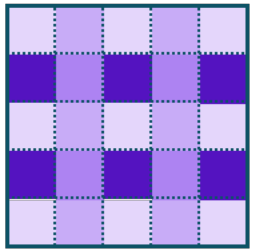
- Chapel supports a rich set of language features
  - a modern, productive set of base language features
  - “low-level” features for creating tasks and placing them on a system
  - a global namespace for referring to data lexically, whether local or remote
  - high-level data-parallel features such as forall loops and promotion
  - a rich set of domains and arrays, including global-view distributed arrays



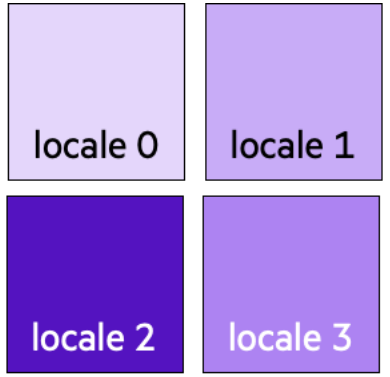
fib.chpl

```
config const n = 10;

for (i,f) in zip(0..<n, fib(n)) do
  writeln("fib #", i, " is ", 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=1000
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
fib #7 is 13
fib #8 is 21
fib #9 is 34
fib #10 is 55
fib #11 is 89
fib #12 is 144
fib #13 is 233
fib #14 is 377
…
```



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



CHAPEL SOUNDS GREAT…THERE MUST BE A CATCH?

- We think Chapel is great, yet at times, it can admittedly also be frustrating
  - Compile times can feel sluggish (but we're currently working on improving them)
  - Error messages can be confusing or poor (ditto)
  - Chapel code doesn't support GPUs very well yet (ditto)
  - Sometimes reasonable code performs poorly (ditto)
  - Tools are lacking (not receiving much attention at present)
- Essentially, Chapel is a continually improving work-in-progress
  - Depending on your needs and personality, it may be perfect for you today, or it could make sense to wait
  - We have a reputation for being very responsive to users' questions and needs
- Another catch is that any language, however great, must overcome social challenges to become adopted
  - The HPC community is particularly skeptical of new languages
    - in part due to being performance- and HW-centric; in part due to having been burnt by past language attempts
- All that said, we think that the number of HPC-focused programming languages should be > 0
  - And that Chapel is as strong a contender as any

LIVE DEMO?
(TIME AND INTEREST PERMITTING)

CHAPEL ON OOKAMI

CHAPEL ON OOKAMI

- Chapel runs on Ookami
  - Our December release (1.25.1) is pre-installed as a module for users' convenience
    - We'll talk more about this in the hands-on section

- We have performed some baseline performance measurements

  …but let's cover some disclaimers first

CHAPEL ON OOKAMI: IMPORTANT DISCLAIMERS

- We have not put *any* effort into tuning Chapel for A64FX processors
  - Our team spends a lot of effort tuning and optimizing Chapel for Cray and InfiniBand networks
    - And, to a lesser extent, optimizing for recent Intel and AMD processor designs
  - To date, A64FX has not been a priority for us

- We also haven't focused much on optimizing vectorization in Chapel
  - Our approach is to expose opportunities for vectorization to the back-end compiler, relying on it
  - recently, we have been focused on code generation for GPUs which is related, but different

- A64FX is, in many respects, the opposite of what we've been most focused on in recent years:
  - **Our recent focus:** massive data sets on systems with lots of memory and bandwidth
  - **A64FX:** memory capacity limited, vectorization-focused

BASELINE OOKAMI PERFORMANCE COMPARISONS

16-node Chapel results:

Benchmark	Apollo-CL	Ookami	Ratio
Stream Triad	2579 GB/s	7808 GB/s	3.03x
PRK Stencil	1335 GFlops/s	949 GFlops/s	0.71x
ISx	2.55 s	6.86 s	0.37x
Bale IndexGather	4.8 GB/s/node	0.4 GB/s/node	0.08x
HPCC RA (RMA)	0.0016 GUPS	0.0009 GUPS	0.56x
HPCC RA (AM)	0.0084 GUPS	0.0024 GUPS	0.29x

Ookami's HBM greatly benefits highly localized computations

System Characteristics:

System	Network	Cores per node (locale)	Processor Type
Apollo-CL	HDR IB	48	Cascade Lake
Ookami	HDR IB	48	A64FX

(see the disclaimers on the preceding slide before drawing any conclusions from these results)

BASELINE OOKAMI PERFORMANCE COMPARISONS

16-node Chapel results:

Benchmark	Apollo-CL	Ookami	Ratio
Stream Triad	2579 GB/s	7808 GB/s	3.03x
PRK Stencil	1335 GFlops/s	949 GFlops/s	0.71x
ISx	2.55 s	6.86 s	0.37x
Bale IndexGather	4.8 GB/s/node	0.4 GB/s/node	0.08x
HPCC RA (RMA)	0.0016 GUPS	0.0009 GUPS	0.56x
HPCC RA (AM)	0.0084 GUPS	0.0024 GUPS	0.29x

Possible explanations for the poor results for more complex benchmarks:

- Chapel and its communication optimizations may require more powerful scalar cores than those on A64FX
- Chapel's heuristics for NUMA affinity may be less effective on A64FX than on CascadeLake
- Lack of vectorization / CPU specialization may hurt Chapel more on A64FX than on CascadeLake
- Chapel's tasking library (Qthreads) may not perform as well on A64FX as on CascadeLake

(see the disclaimers on the preceding slide before drawing any conclusions from these results)

POTENTIAL OOKAMI IMPROVEMENTS: MULTIPLE PROCESSES PER NODE

- Traditionally, Chapel runs a single process per locale per compute node
  - Parallelism is typically implemented via user-level tasks
    - executed using worker threads that are pinned 1:1 to the compute node's cores
  - NUMA affinity is dealt with heuristically by Chapel's implementation
    - not perfect, but has typically worked "well enough" in practice
  - This approach has had various benefits for us, including:
    - simple execution model for users
    - single communication mechanism for cross-locale accesses
    - good surface-to-volume properties, particularly as core counts have increased significantly
- For various reasons, we have discussed enabling a slightly coarser execution model
  - e.g., running using a process/locale per…

    …NUMA domain / CMG?

    …NIC?

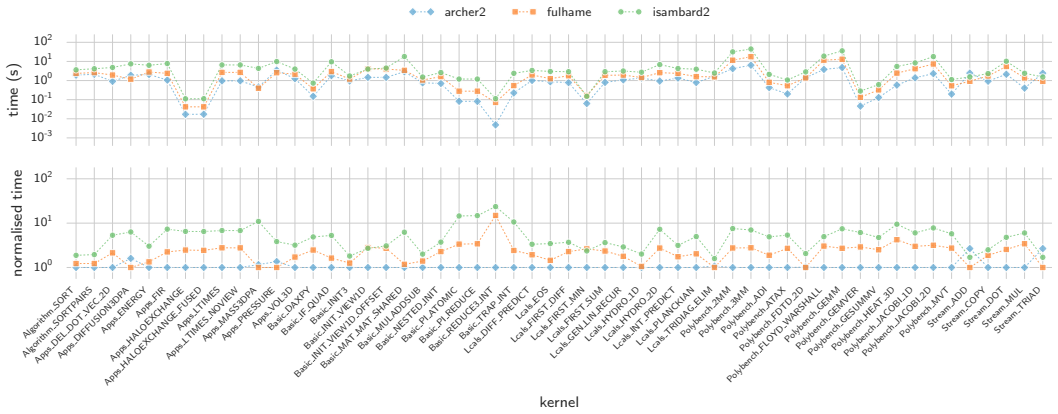
    …GPU?
- If NUMA effects are hurting Chapel on A64FX more than conventional processors, this could help

POTENTIAL OOKAMI IMPROVEMENTS: BETTER VECTORIZATION

- As mentioned earlier, vectorization optimizations have not been a big focus for our group to date
- Three promising directions:
  1. Simon Moll et al.'s LLVM Region Vectorizer has demonstrated benefits for Chapel via outer-loop vectorization
     - we'd like to explore this more and potentially enable it by default in Chapel
  2. ARM has been upstreaming SVE contributions to LLVM that could also improve our performance on A64FX
     - thanks to Tony Curtis for bringing this to our attention
  3. colleagues at EPCC have recently started a study of vectorization on ARM-based HPC systems
     - (see next slide)

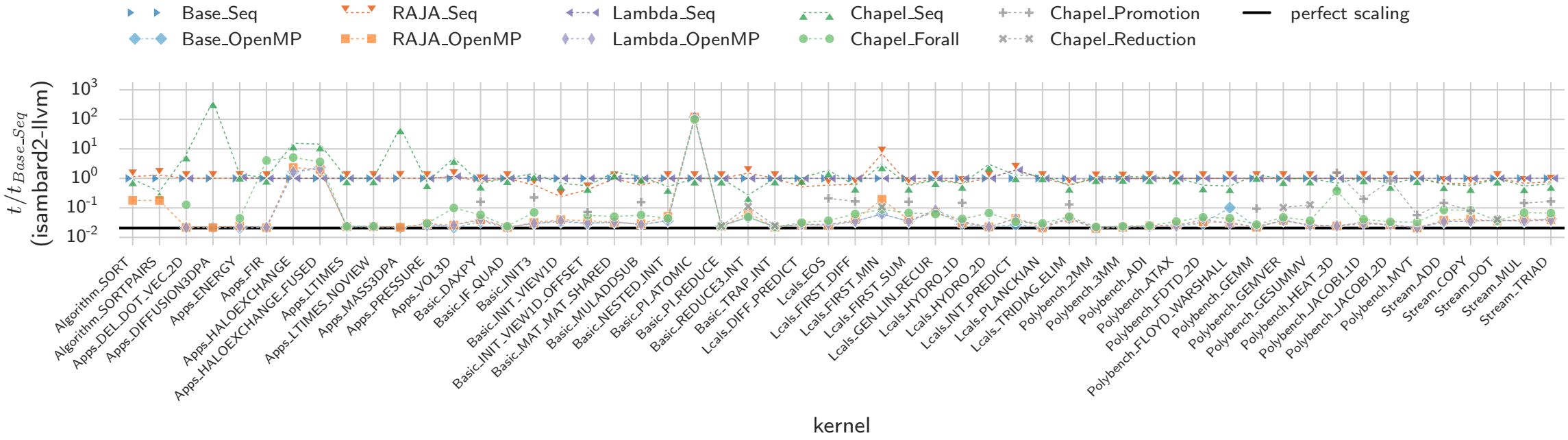
POTENTIAL MITIGATION STRATEGIES: BETTER VECTORIZATION

- Michele Weiland & Ricardo Jesus (EPCC): studying how well languages target ARM-based HPC systems
  - Using RAJAPerf as the basis for their study
    - Have created a Chapel port using a few different computational styles
  - Running on ARCHER2 (AMD EPYC 7742), Fulhame (Marvell ThunderX2), and Isambard 2 (Fujitsu A64FX)
  - Starting to generate preliminary results:



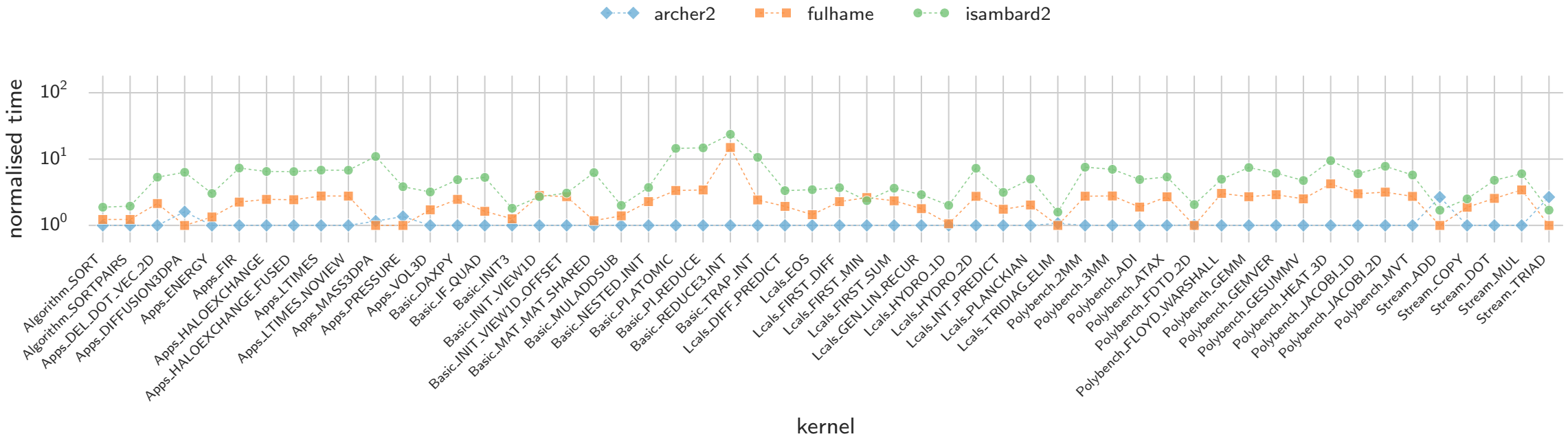
POTENTIAL MITIGATION STRATEGIES: BETTER VECTORIZATION

- Michele Weiland & Ricardo Jesus (EPCC): studying how well languages target ARM-based HPC systems
  - Using RAJAPerf as the basis for their study
    - Have created a Chapel port using a few different computational styles
  - Running on ARCHER2 (AMD EPYC 7742), Fulhame (Marvell ThunderX2), and Isambard 2 (Fujitsu A64FX)
  - Starting to generate preliminary results:



POTENTIAL MITIGATION STRATEGIES: BETTER VECTORIZATION

- Michele Weiland & Ricardo Jesus (EPCC): studying how well languages target ARM-based HPC systems
  - Using RAJAPerf as the basis for their study
    - Have created a Chapel port using a few different computational styles
  - Running on ARCHER2 (AMD EPYC 7742), Fulhame (Marvell ThunderX2), and Isambard 2 (Fujitsu A64FX)
  - Starting to generate preliminary results:



BASELINE OOKAMI PERFORMANCE COMPARISONS

16-node Chapel results:

Benchmark	Apollo-CL	Ookami	Ratio	Ookami-SHMEM
Stream Triad	2579 GB/s	7808 GB/s	3.03x	
PRK Stencil	1335 GFlops/s	949 GFlops/s	0.71x	
ISx	2.55 s	6.86 s	0.37x	5.76 s
Bale IndexGather	4.8 GB/s/node	0.4 GB/s/node	0.08x	0.47 GB/s/node
HPCC RA (RMA)	0.0016 GUPS	0.0009 GUPS	0.56x	
HPCC RA (AM)	0.0084 GUPS	0.0024 GUPS	0.29x	

SHMEM reference versions perform similarly to Chapel

- these are run with a process per core, so don't suffer NUMA effects
- may suggest that scalar processor / communication overheads are the likely big difference

(see the disclaimers on the preceding slide before drawing any conclusions from these results)

CHAPEL ON OOKAMI PERFORMANCE SUMMARY

- Stating for re-emphasis: We've invested no effort to date in making Chapel perform well on Ookami

- That said, we see several potential avenues for improvements:
  - better vectorization / specialization for A64FX
  - improve our understanding of how Chapel's communication code paths behave on A64FX
  - running a process/locale per CMG ; better NUMA heuristics
  - studying and tuning the performance of the Qthreads tasking library on A64FX
  - …something else we haven't learned yet due to lack of study?

SUMMARY & RESOURCES

SUMMARY

Chapel cleanly supports…

…expression of parallelism and locality

…a diverse set of parallel features, at various levels

…specifying how to map computations to the system

Chapel is powerful:

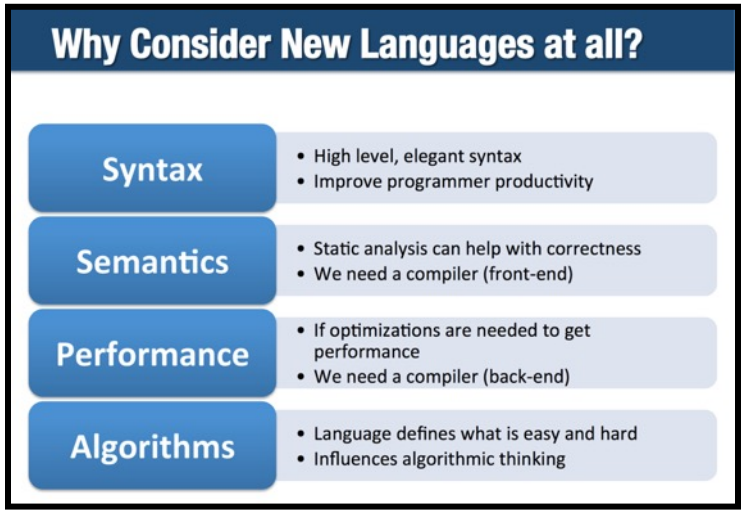
- it supports succinct, straightforward code
- it can result in performance that competes with or beats C+MPI[+OpenMP]

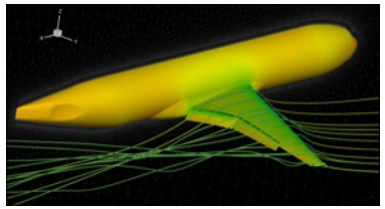
Chapel is being used for productive parallel applications at scale

- recent users have reaped its benefits in 10k–48k-line applications

Chapel is available and working on Ookami

- further study is required to understand opportunities for improved performance





Benchmark	Apollo-CL	Ookami	Ratio
Stream Triad	2579 GB/s	7808 GB/s	3.03x
PRK Stencil	1335 GFlops/s	949 GFlops/s	0.71x
ISx	2.55 s	6.86 s	0.37x
Bale IndexGather	4.8 GB/s/node	0.4 GB/s/node	0.08x
HPCC RA (RMA)	0.0016 GUPS	0.0009 GUPS	0.56x
HPCC RA (AM)	0.0084 GUPS	0.0024 GUPS	0.29x

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

AGAIN, WE ARE HIRING

Chapel Development Team at HPE

see: https://chapel-lang.org/jobs.html

CHAPEL RESOURCES

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

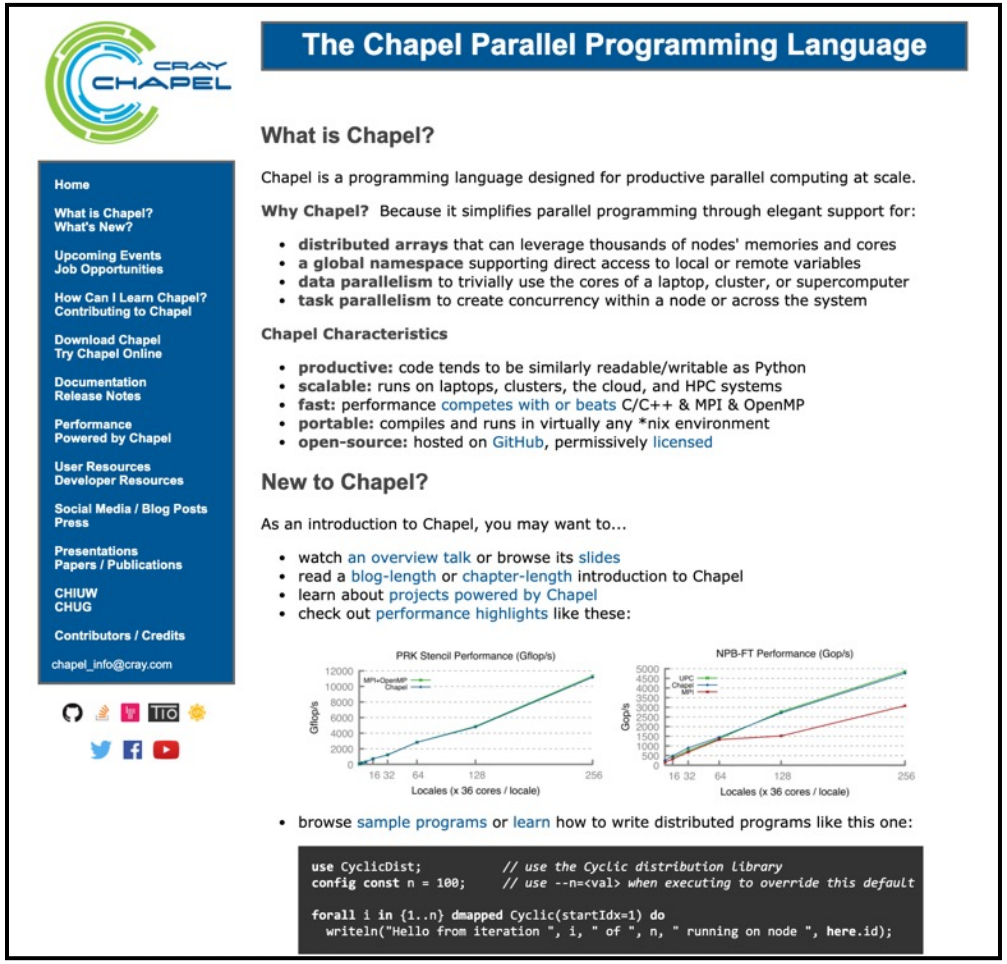
- (points to all other resources)

Social Media:

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

Community Discussion / Support:

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



SUGGESTED READING / VIEWING

Chapel Overviews / History (in chronological order):

- *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 *8th Annual Chapel Implementers and Users Workshop* (CHIUW 2021), June 2021
- Chapel Release Notes — current version 1.25, October 2021

Arkouda:

- Bill Reus's CHIUW 2020 keynote talk: https://chapel-lang.org/CHIUW2020.html#keynote
- Arkouda GitHub repo and pointers to other resources: https://github.com/Bears-R-Us/arkouda

CHAMPS:

- Eric Laurendeau's CHIUW 2021 keynote talk: https://chapel-lang.org/CHIUW2021.html#keynote
  – two of his students also gave presentations at CHIUW 2021, also available from the URL above
- Another paper/presentation by his students at https://chapel-lang.org/papers.html (search "Laurendeau")

HANDS-ON
(TIME AND INTEREST PERMITTING)

USING CHAPEL ON OOKAMI

- Chapel is pre-installed on Ookami, thanks to Eva Siegmann and Tony Curtis
  - Installed at `/lustre/software/chapel/apollo/chapel-1.25.1`
  - Available via normal module commands:

```
prompt> module load chapel
```

  - Sample programs available:

```
prompt> ls $CHPL_HOME/examples/*.chpl
hello.chpl                    hello4-datapar-dist.chpl
hello2-module.chpl            hello5-taskpar.chpl
hello3-datapar.chpl           hello6-taskpar-dist.chpl
```

(see also the 'primers/' and 'benchmarks/' subdirectories)

  - Compile and run as shown in previous examples:

```
prompt> chpl $CHPL_HOME/examples/hello6-taskpar-dist.chpl
prompt> ./hello6-taskpar-dist -nl 4
```

INSTALLING AND USING CHAPEL ON YOUR OWN SYSTEM

- Often, getting started with Chapel on a supercomputer can be annoying
  - Environment not as set up to your liking as your primary machine
  - Shared resource, queueing times, etc.

- You're welcome to install Chapel on your laptop or favorite system if you're able to
  - **Mac homebrew** (Catalina or later) / **Linuxbrew**: 'brew install chapel' (supports single-locale runs only)
  - **Mac / Linux / *nix:** Install and build from source, see https://chapel-lang.org/download.html
  - **Windows:** Use Linux bash shell / Windows Subsystem for Linux and see previous line
  - **Docker:** See https://chapel-lang.org/install-docker.html

- Developing a Chapel program on a laptop and then running it on a supercomputer is a common practice
  - And Chapel's global view tends to make it easy:
    - almost always runs correctly
    - typically not too difficult to get using multiple locales, particularly for data-parallel codes
    - optimizing it can take more effort…

"WHAT SHOULD I DO DURING THE HANDS-ON SESSION?"

- You're welcome to do whatever appeals to you, but here are some possibilities:
  - Try compiling, running, modifying examples from this talk
    - I've made most of them available on Ookami at:

      `/lustre/projects/global/Chapel/ookami-webinar/slideExamples/`

  - Try one of the hands-on exercises
    - There are four exercises prepared, two that are simpler and two that are more involved (see next slide)
    - Instructions and starting files are also on Ookami:

      `/lustre/projects/global/Chapel/ookami-webinar/handsOn/`

  - Try coding up a computation or parallel pattern of interest to you

- Note that you will probably want to create your own local, writeable copy of the materials above, e.g.:

```
prompt> cp -r /lustre/projects/global/Chapel/ookami-webinar .
prompt> chmod -R u+w ookami-webinar
```

PREPARED HANDS-ON EXERCISES

- **Advent of Code 2021 Day 1:** given an array of numbers, compute some simple statistics on it
  - This is a trivial computation that you'd have no trouble doing in a language you're familiar with
  - Goal is to use it to get familiar with Chapel
  - Opportunities for array operations, data parallelism, reductions

- **Advent of Code 2021 Day 4:** simulate a bingo game with an octopus
  - This is slightly more involved and interesting, but still straightforward
  - More opportunities for array operations, data parallelism, 2D arrays

- **Ray Tracer:** given a ray tracing framework, fill in some missing details to make it work
  - Exercises 2D arrays, data parallelism

- **Bounded Buffer:** use Chapel's sync and atomic variables
  - Exercises task parallelism and synchronization—relies on sync/atomic variables, not really covered today

(Note that none of these runs are large enough to *require* multiple locales, though most are amenable to them)

NEED HELP?

- We'll be handling Q&A today both live and via #chapel-webinar on the IACS Slack channel
  - Members of the Chapel team besides myself will be on Slack to answer questions, screen share, etc.

- After today, help is available via:
  - **Stack Overflow:** for questions that will likely be valuable to others (tag your question with 'chapel')
  - **Discourse:** our community web forum / mailing list technology
  - **Gitter:** our community real-time chat technology
  - **GitHub issues:** for filing bug reports, feature requests, etc.

GENERAL TIPS WHEN GETTING STARTED WITH CHAPEL (ALSO IN README)

- Online **documentation** is here: https://chapel-lang.org/docs/
  - The primers can be particularly valuable for learning a concept: https://chapel-lang.org/docs/primers/index.html
    - These are also available from a Chapel release in '$CHPL_HOME/examples/primers/'
      - or '$CHPL_HOME/test/release/examples/primers/' if you clone from GitHub

- When debugging, **almost anything in Chapel can be printed out** with 'writeln(expr1, expr2, expr3);'
  - Types can be printed after being cast to strings, e.g. 'writeln("Type of ", expr, " is ", expr.type:string);'
  - A quick way to print a bunch of values out clearly is to print a tuple made up of them 'writeln((x, y, z));'

- Once your code is correct, before doing any performance timings, be sure to re-compile with **'--fast'**
  - Turns on optimizations, turns off safety checks, slows down compilation, speeds up execution significantly
  - Then, when you go back to making modifications, be sure to stop using `--fast` in order to turn checks back on

- For vim / emacs users, **syntax highlighters** are in $CHPL_HOME/highlight
  - Imperfect, but typically better than nothing
  - Emacs MELPA users may want to use the chapel-mode available there (better in many ways, weird in others)

THANK YOU

https://chapel-lang.org
@ChapelLanguage