Hewlett Packard Enterprise

PRODUCTIVE PARALLEL PROGRAMMING USING CHAPEL

Brad Chamberlain & Michelle Strout

August 4, 2022

WHAT IS CHAPEL?

Chapel: A modern parallel programming language

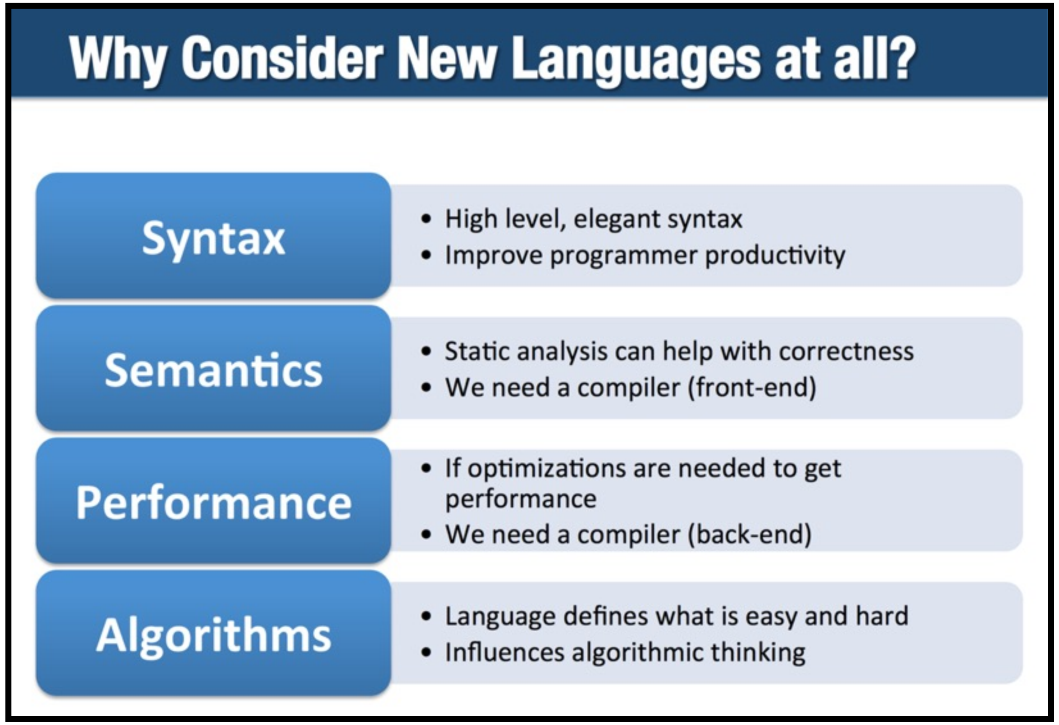
- portable & scalable
- open-source & collaborative

Goals:

- Support general parallel programming
- Make parallel programming at scale far more productive
  - Python-like support for rapid prototyping
  - yet with the performance, scalability, portability of Fortran/C/C++, MPI, OpenMP, CUDA, …

WHY CREATE A NEW LANGUAGE?

- **Because parallel programmers deserve better**
  - the state of the art for HPC programming is a mash-up of libraries, pragmas, and extensions
  - SPMD-based models are restrictive compared to having a global namespace and asynchrony
  - 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]

SCALABLE PARALLEL COMPUTING THAT'S AS EASY AS PYTHON?

Imagine having a programming language for parallel computing that was as…

…**programmable** as Python

…yet also as…

…**fast** as Fortran

…**scalable** as MPI

…**GPU-ready** as CUDA/OpenMP/OpenCL/OpenACC/…

…**portable** as C

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

This is our motivation for Chapel

OUTLINE

- Introductory Content
  - What is Chapel?
  - Chapel Characteristics
  - Chapel Benchmarks & Apps
  - Chapel vs. Standard Practice
- Further Details: Chapel Features
  - Base Language Features
  - Task-Parallelism & Locality
  - Data-Parallelism
- Wrap-up

CHAPEL CHARACTERISTICS

KEY CHARACTERISTICS OF CHAPEL

- **compiled:** to generate the best performance possible
- **statically typed:** to avoid simple errors after hours of execution
- **interoperable:** with C, Fortran, Python, ...
- **portable:** runs on laptops, clusters, the cloud, supercomputers
- **open-source:** to reduce barriers to adoption and leverage community contributions

WHAT DO CHAPEL PROGRAMS LOOK LIKE?

helloTaskPar.chpl: print a message from each core in the system

```
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);
  }
}
```

```
> chpl helloTaskPar.chpl
> ./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
…
```

fillArray.chpl: declare and parallel-initialize a distributed array

```
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);
```

```
> chpl fillArray.chpl
> ./fillArray --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
```

CHAPEL RELEASES

Q: What is provided in a Chapel release?

A: Chapel releases contain…

…**the Chapel compiler** (‘chpl’)**:** translates Chapel source code into optimized executables

…**runtime libraries:** help map Chapel programs to a system’s capabilities (e.g., processors, network, memory, …)

…**library modules:** provide standard algorithms, data types, capabilities, …

…**documentation:** also available online at: https://chapel-lang.org/docs/

…**sample programs:** primers, benchmarks, etc.

Q: How often is Chapel released? When is the next one?

A: Chapel is released every 3 months

- version 1.27.0 was released June 30, 2022
- version 1.28.0 is scheduled for September 17, 2022

THE CHAPEL TEAM AT HPE

Our team consists of:

- 19 full-time employees
- 3 summer interns
- our director

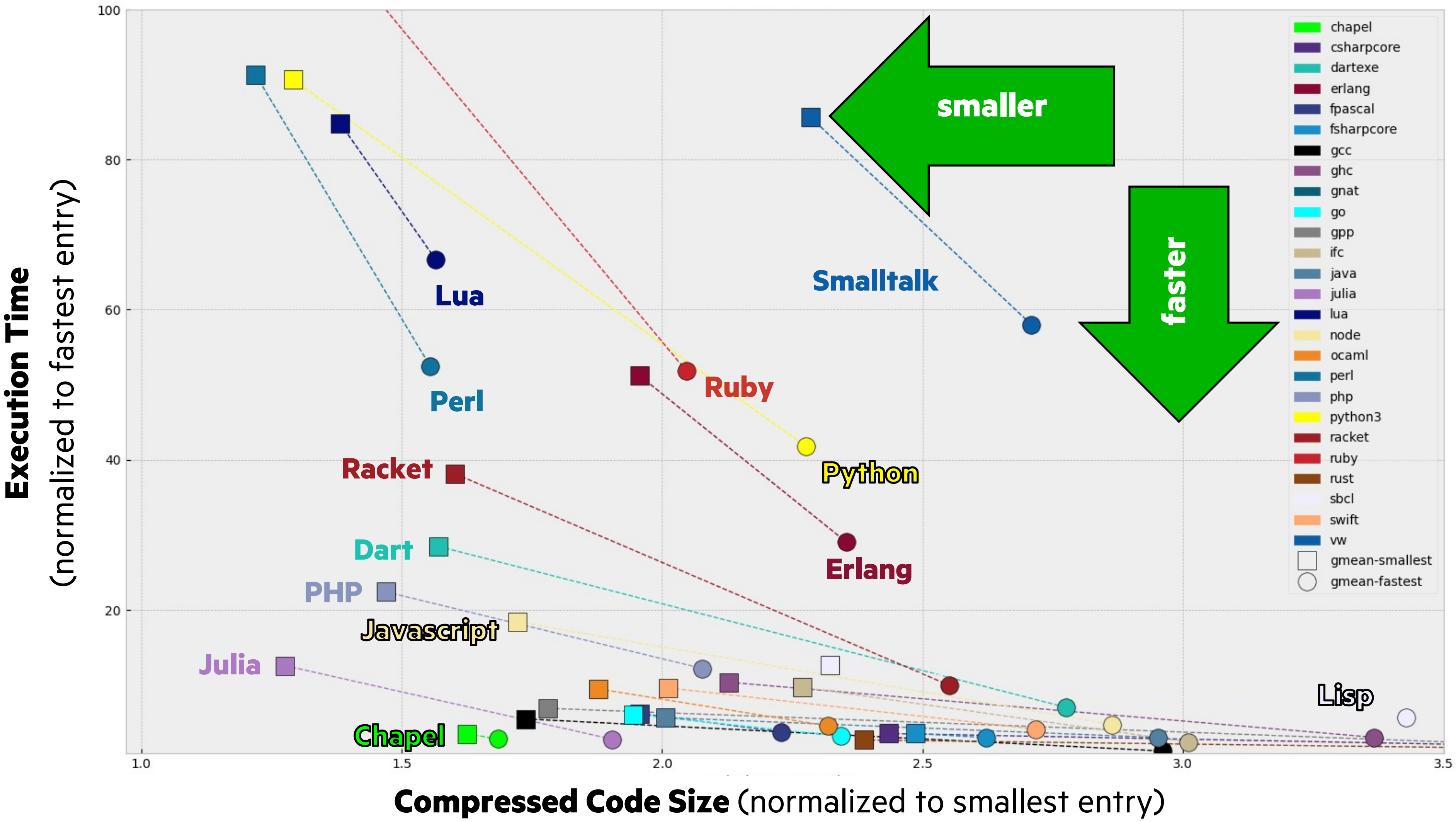
We also have:

- a visiting scholar joining soon
- an open position

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

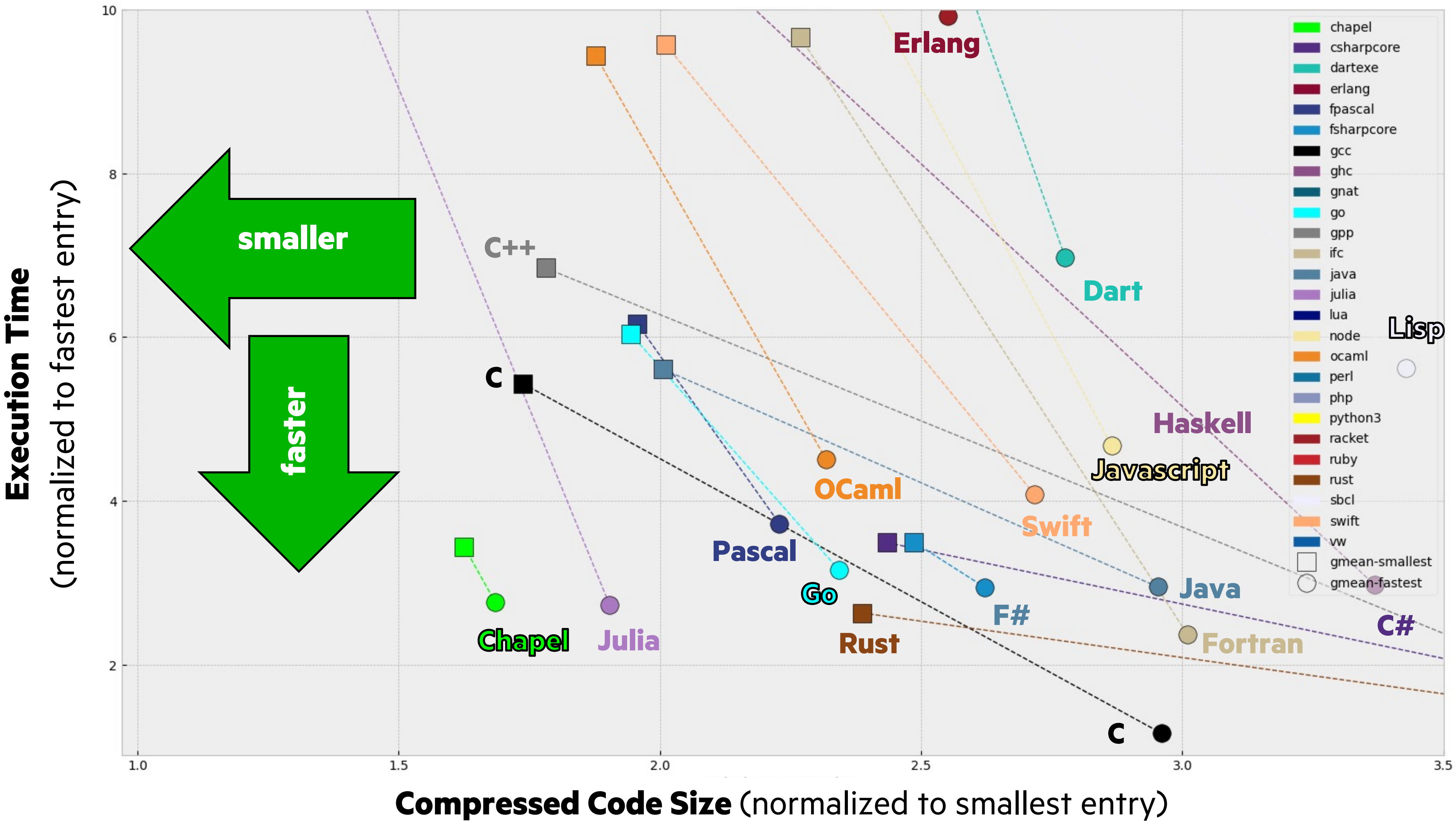
CHAPEL BENCHMARKS AND APPLICATIONS

FOR DESKTOP BENCHMARKS, CHAPEL IS COMPACT AND FAST



[plot generated by summarizing data from https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html as of May 10, 2022]

FOR DESKTOP BENCHMARKS, CHAPEL IS COMPACT AND FAST (ZOOMED)



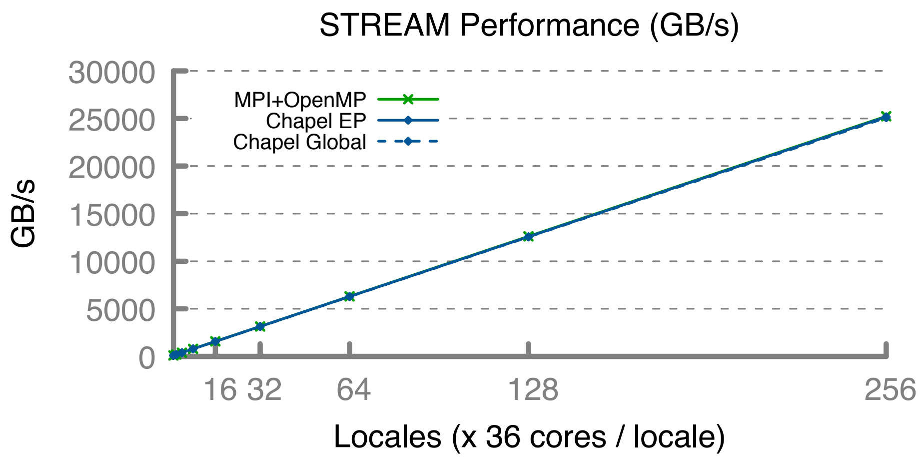
[plot generated by summarizing data from https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html as of May 10, 2022]

FOR HPC BENCHMARKS, CHAPEL TENDS TO BE CONCISE, CLEAR, AND COMPETITIVE

STREAM TRIAD: C + MPI + OPENMP

```
#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;
}
```

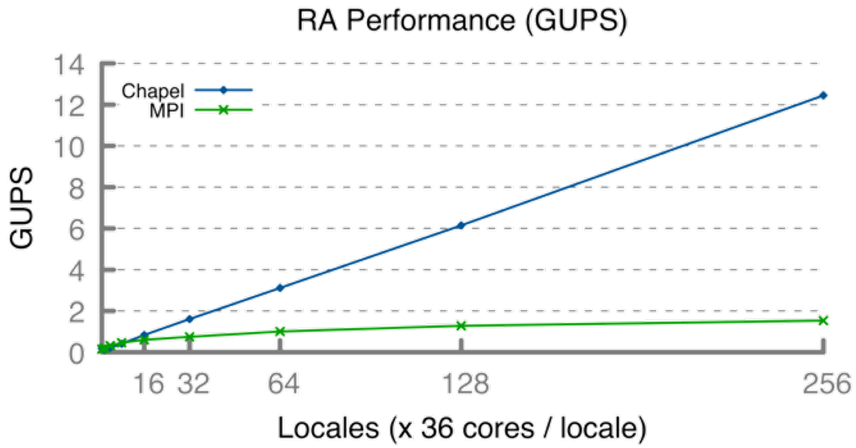
```
use BlockDist;

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;
```



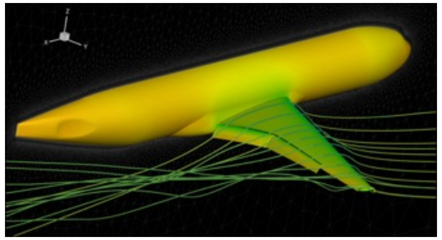
HPCC RA: MPI KERNEL

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

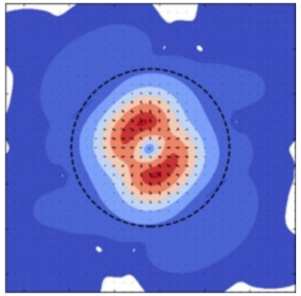
72



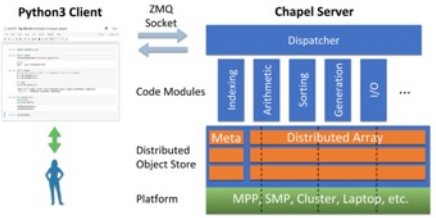
FLAGSHIP CHAPEL APPLICATIONS



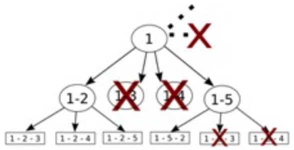
CHAMPS: 3D Unstructured CFD
Éric Laurendeau, Simon Bourgault-Côté, Matthieu Parenteau, et al.
École Polytechnique Montréal



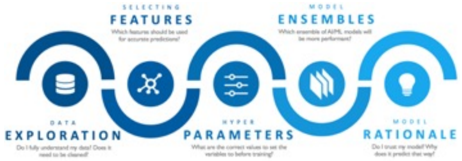
ChplUltra: Simulating Ultralight Dark Matter
Nikhil Padmanabhan, J. Luna Zagorac, *et al.*
Yale University / University of Auckland



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

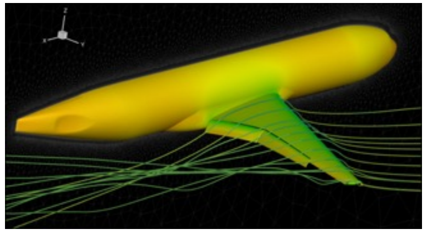
?

Your application here?

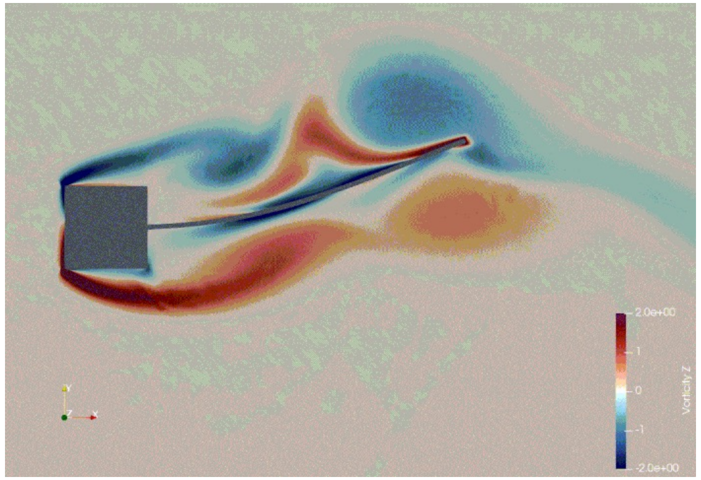
(images provided by their respective teams and used with permission)

CHAMPS SUMMARY

What is it?



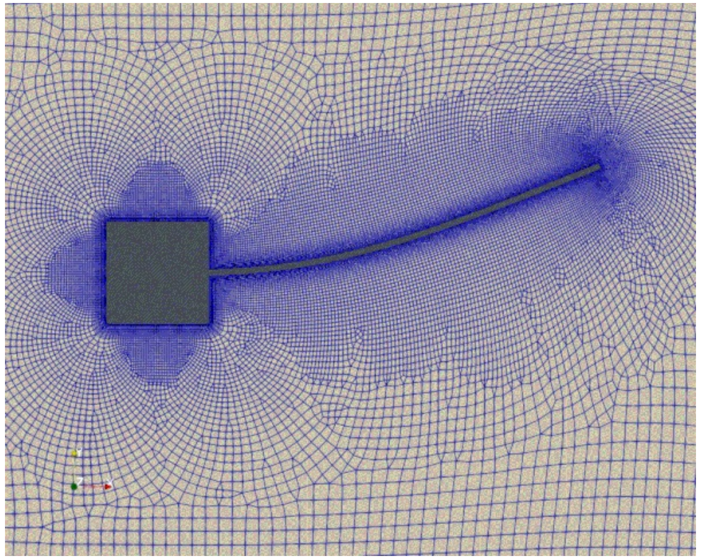
- 3D unstructured CFD framework for airplane simulation
- ~120k lines of Chapel written from scratch in ~3 years



Who wrote it?

- Professor Éric Laurendeau's students + postdocs at Polytechnique Montreal

POLYTECHNIQUE MONTRÉAL



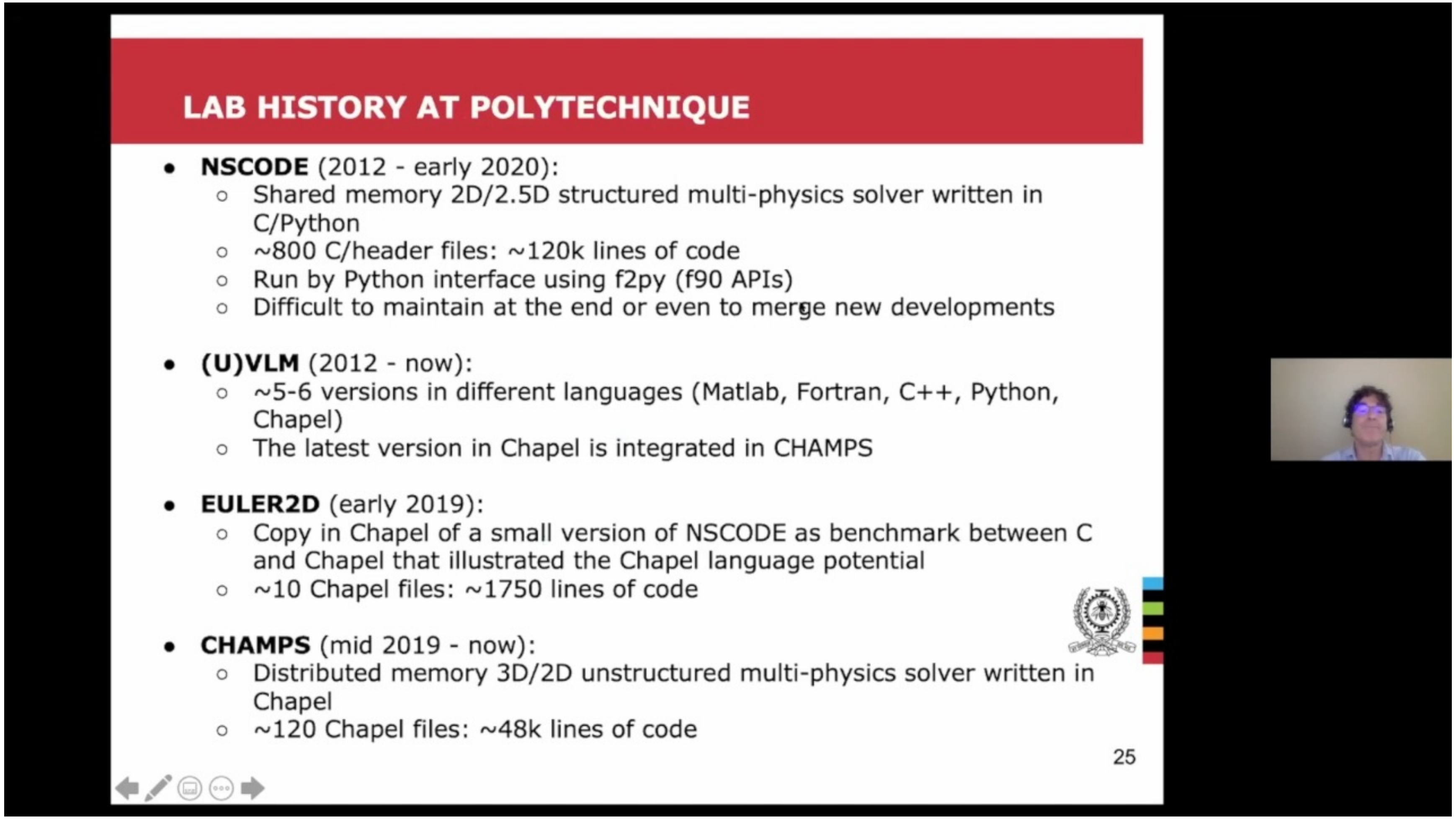
Why Chapel?

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

(images provided by the CHAMPS team and used with permission)

CHAMPS: EXCERPT FROM ÉRIC'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_KyB8aI?t=1904

(images provided by the CHAMPS team and used with permission)

CHAMPS: EXCERPT FROM ÉRIC'S CHIUW 2021 KEYNOTE (TRANSCRIPT)

HPC Lessons From 30 Years of Practice in CFD Towards Aircraft Design and Analysis (June 4, 2021)

"To show you what Chapel did in our lab… [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… ***So, I've got industrial-type code in 48k lines.****"*

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."*

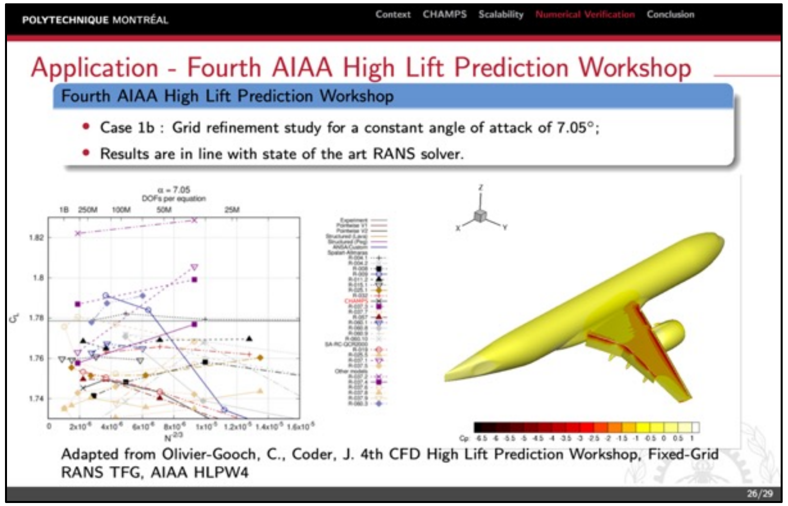
"So, for me, this is like the proof of the benefit of Chapel, ***plus the smiles I have on my students everyday in the lab because they love Chapel as well.*** *So that's the key, that's the takeaway."*

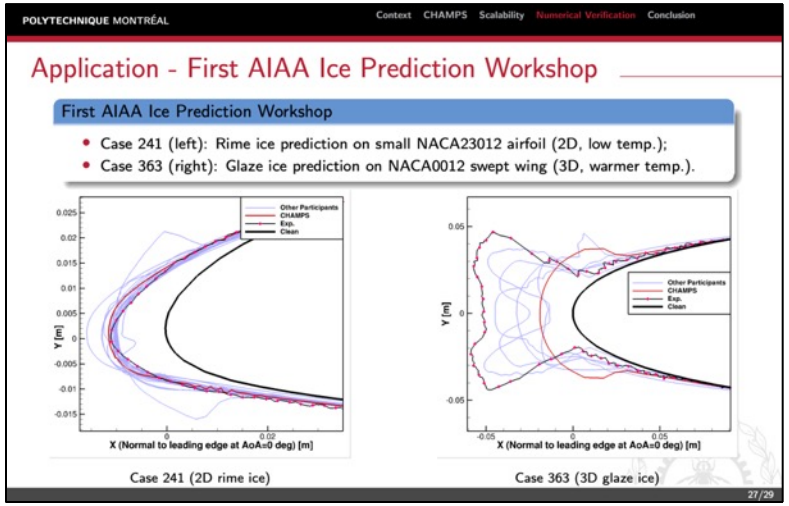
- Talk available online: https://youtu.be/wD-a_KyB8aI?t=1904 (hyperlink jumps to the section quoted here)

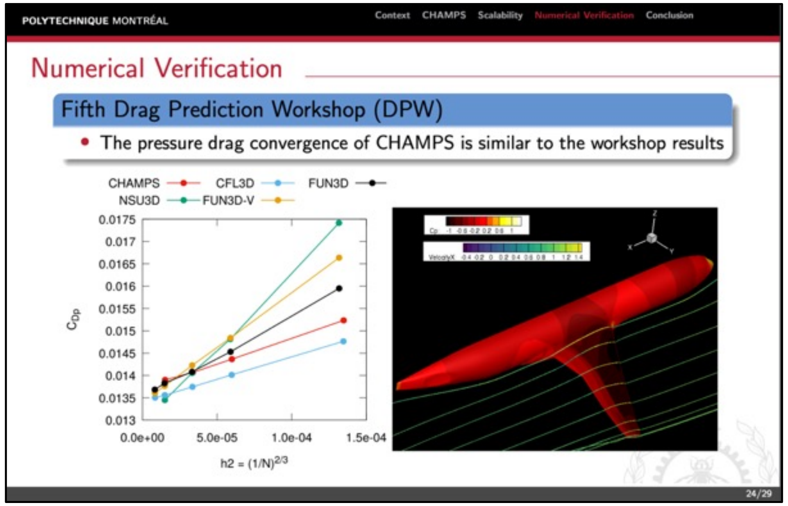
(images provided by the CHAMPS team and used with permission)

RECENT CHAMPS HIGHLIGHTS

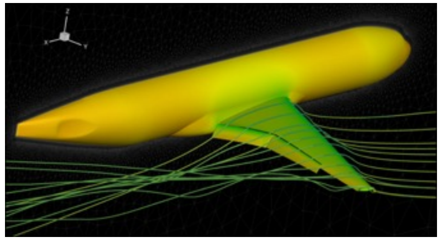
- **CHAMPS 2.0** was released this year
  - added many new capabilities and improvements
  - grew from ~48k to ~120k lines
- Team gave 5–6 talks at **2022 AIAA AVIATION** in June
- While on sabbatical this year, Éric presented at **ONERA**, **DLR**, **U. de Strasbourg**, **T. U. Braunschweig**
- Participated in the **4th AIAA High-lift Prediction Workshop** and **1st AIAA Ice Prediction Workshop**
  - Generating comparable results to high-profile sites: Boeing, Lockheed Martin, NASA, JAXA, Georgia Tech, …



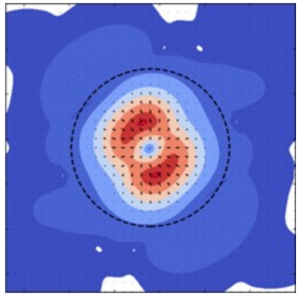




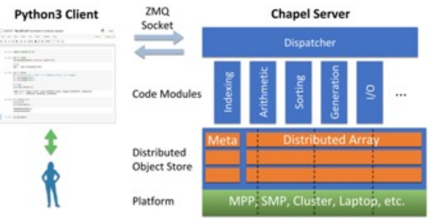
CURRENT FLAGSHIP CHAPEL APPLICATIONS



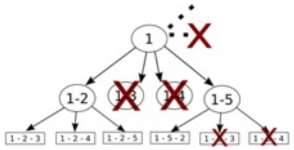
CHAMPS: 3D Unstructured CFD
Éric Laurendeau, Simon Bourgault-Côté, Matthieu Parenteau, et al.
École Polytechnique Montréal



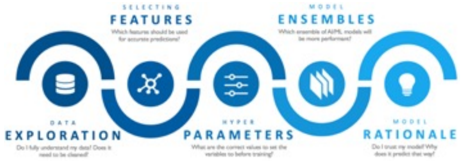
ChplUltra: Simulating Ultralight Dark Matter
Nikhil Padmanabhan, J. Luna Zagorac, *et al.*
Yale University / University of Auckland



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?

(images provided by their respective teams and used with permission)

DATA SCIENCE IN PYTHON AT SCALE?

Motivation: Say you've got…

…HPC-scale data science problems to solve

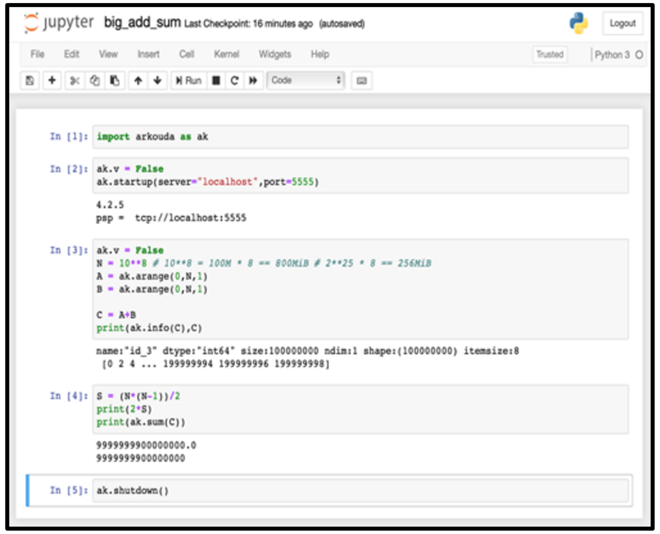
…a bunch of Python programmers

…access to HPC systems

How will you leverage your Python programmers to get your work done?

ARKOUDA'S HIGH-LEVEL APPROACH

Arkouda Client
(written in Python)



Arkouda Server
(written in Chapel)

User writes Python code in Jupyter, making familiar NumPy/Pandas calls

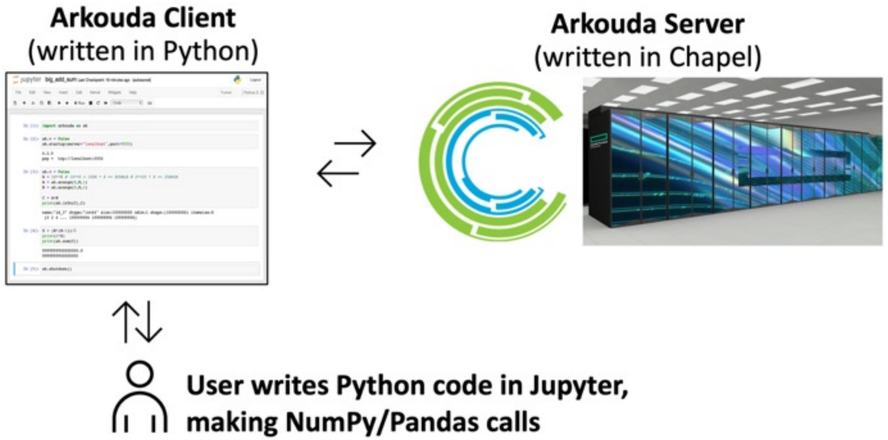
ARKOUDA SUMMARY

What is it?

- A Python library supporting a key subset of NumPy and Pandas for Data Science
  - Uses a Python-client/Chapel-server model to get scalability and performance
  - Computes massive-scale results (multi-TB-scale arrays) within the human thought loop (seconds to a few minutes)
- ~22k 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?

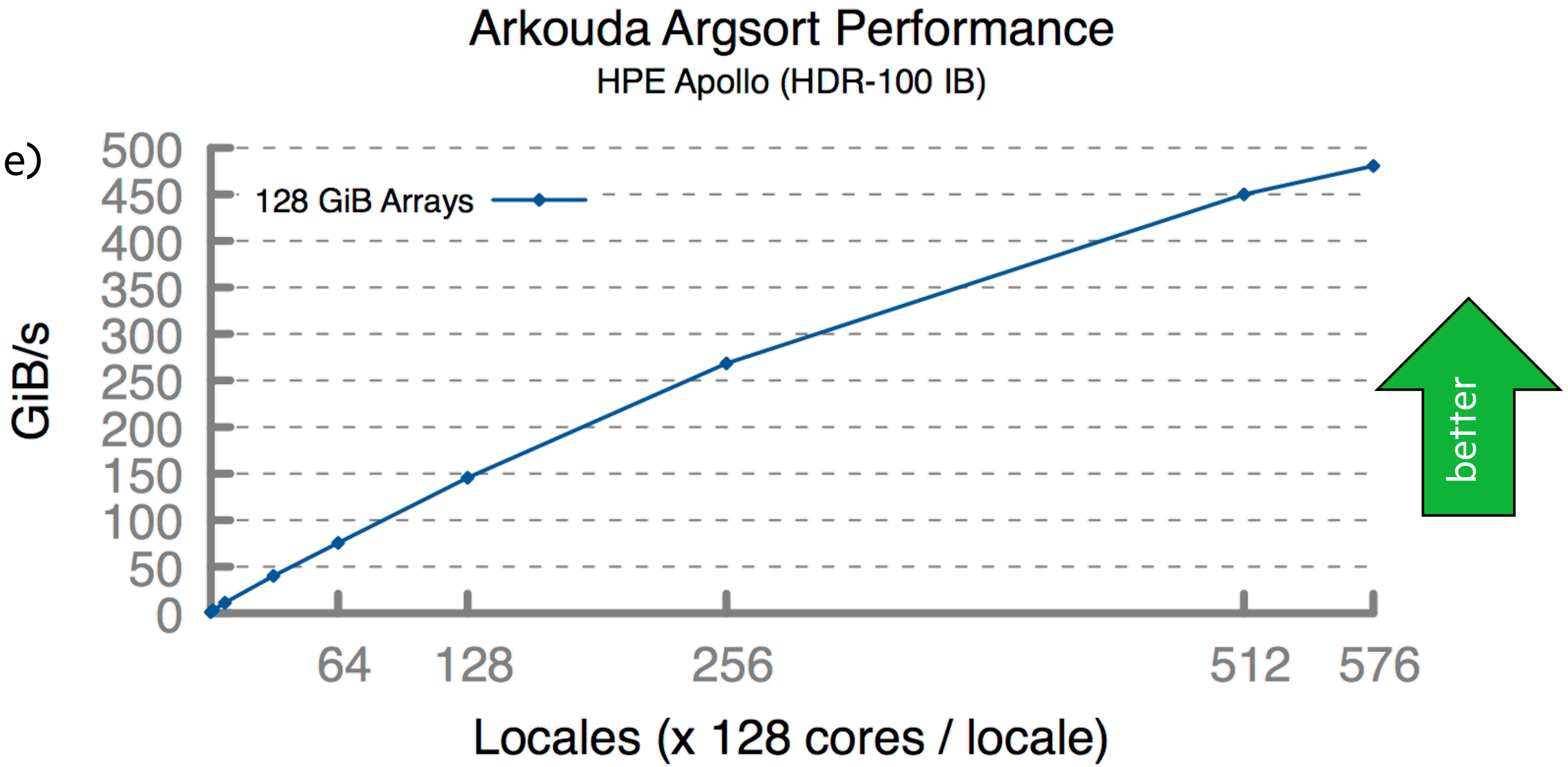
- high-level language with performance and scalability
- close to Pythonic
  - enabled writing Arkouda rapidly
  - doesn't repel Python users who look under the hood
- ports from laptop to supercomputer

ARKOUDA PERFORMANCE COMPARED TO NUMPY

benchmark	NumPy 0.75 GB	Arkouda (serial) 0.75 GB 1 core, 1 node	Arkouda (parallel) 0.75 GB 36 cores x 1 node	Arkouda (distributed) 384 GB 36 cores x 512 nodes
argsort	0.03 GiB/s --	0.05 GiB/s **1.66x**	0.50 GiB/s **16.7x**	55.12 GiB/s **1837.3x**
coargsort	0.03 GiB/s --	0.07 GiB/s **2.3x**	0.50 GiB/s **16.7x**	29.54 GiB/s **984.7x**
gather	1.15 GiB/s --	0.45 GiB/s 0.4x	13.45 GiB/s **11.7x**	539.52 GiB/s **469.1x**
reduce	9.90 GiB/s --	11.66 GiB/s **1.2x**	118.57 GiB/s **12.0x**	43683.00 GiB/s **4412.4x**
scan	2.78 GiB/s --	2.12 GiB/s 0.8x	8.90 GiB/s **3.2x**	741.14 GiB/s **266.6x**
scatter	1.17 GiB/s --	1.12 GiB/s **1.0x**	13.77 GiB/s **11.8x**	914.67 GiB/s **781.8x**
stream	3.94 GiB/s --	2.92 GiB/s 0.7x	24.58 GiB/s **6.2x**	6266.22 GiB/s **1590.4x**

ARKOUDA ARGSORT AT MASSIVE SCALE

- Ran on a large Apollo system, summer 2021
  - 73,728 cores of AMD Rome
  - 72 TiB of 8-byte values
  - 480 GiB/s (2.5 minutes elapsed time)
  - ~100 lines of Chapel code



Close to world-record performance—quite likely a record for performance/SLOC

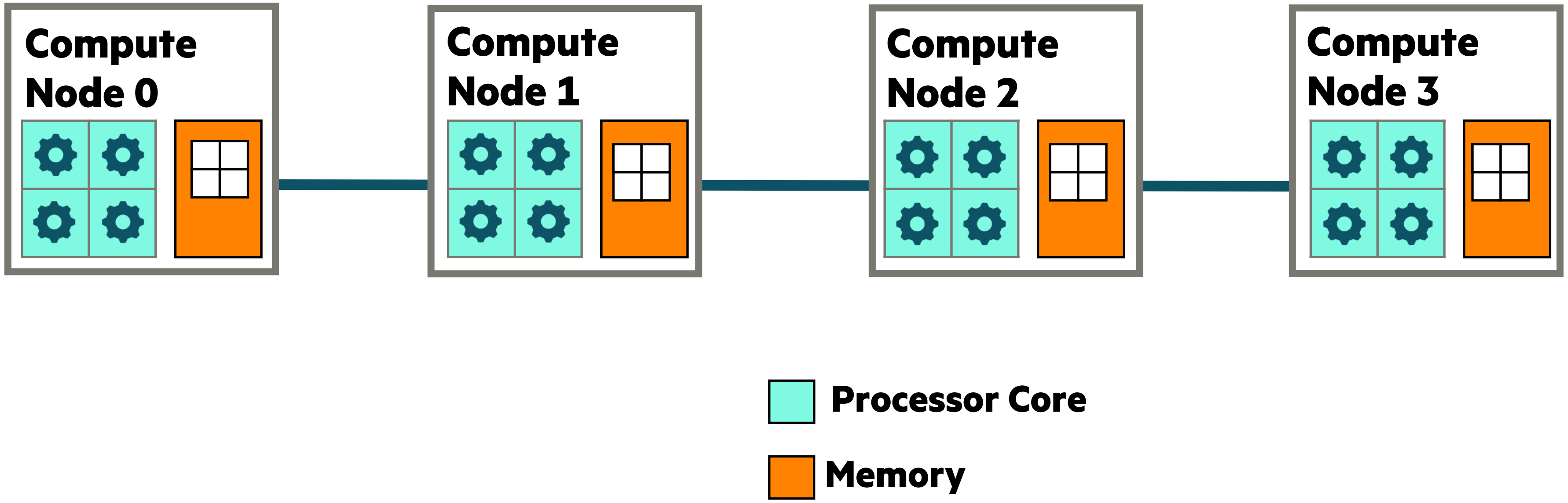
OUTLINE/TIME CHECK

- Introductory Content
  - What is Chapel?
  - Chapel Characteristics
  - Chapel Benchmarks & Apps
  - Chapel vs. Standard Practice
- Further Details: Chapel Features
  - Base Language Features
  - Task-Parallelism & Locality
  - Data-Parallelism
- Wrap-up

CHAPEL VS. STANDARD PRACTICE: PARALLELISM + LOCALITY, SPMD VS. GLOBAL-VIEW

KEY CONCERNS FOR SCALABLE PARALLEL COMPUTING

1. **parallelism:** What tasks should run simultaneously?
2. **locality:** Where should tasks run? Where should data be allocated?

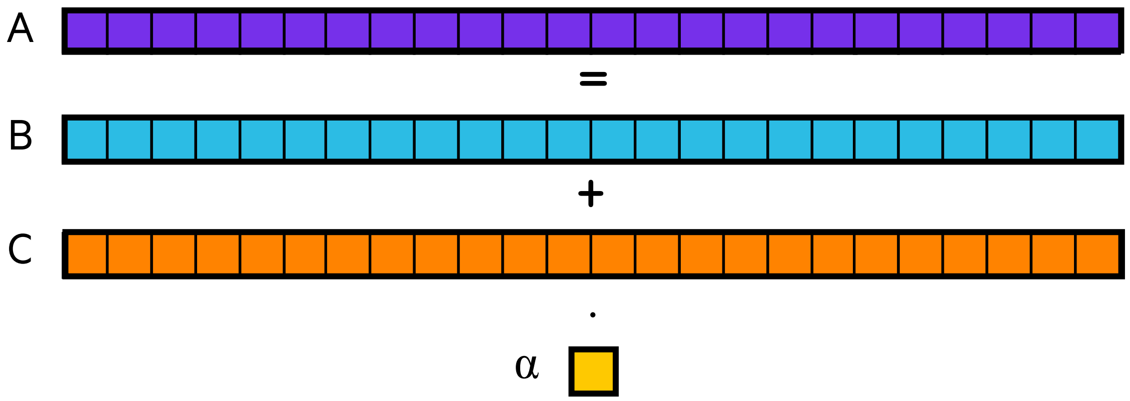


STREAM TRIAD: A TRIVIAL CASE OF PARALLELISM + LOCALITY

Given: *m*-element vectors *A*, *B*, *C*

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

In pictures:

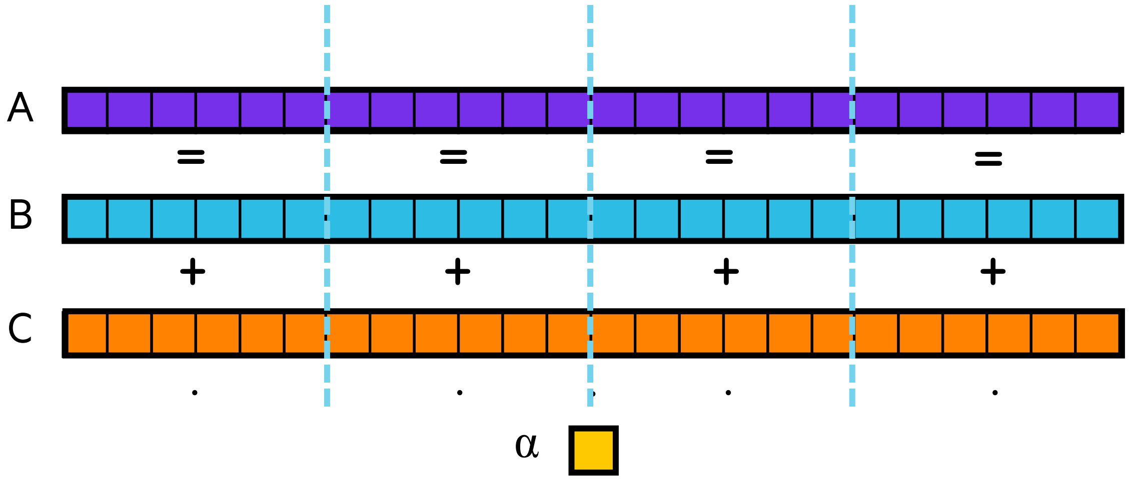


STREAM TRIAD: A TRIVIAL CASE OF PARALLELISM + LOCALITY

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)**:**

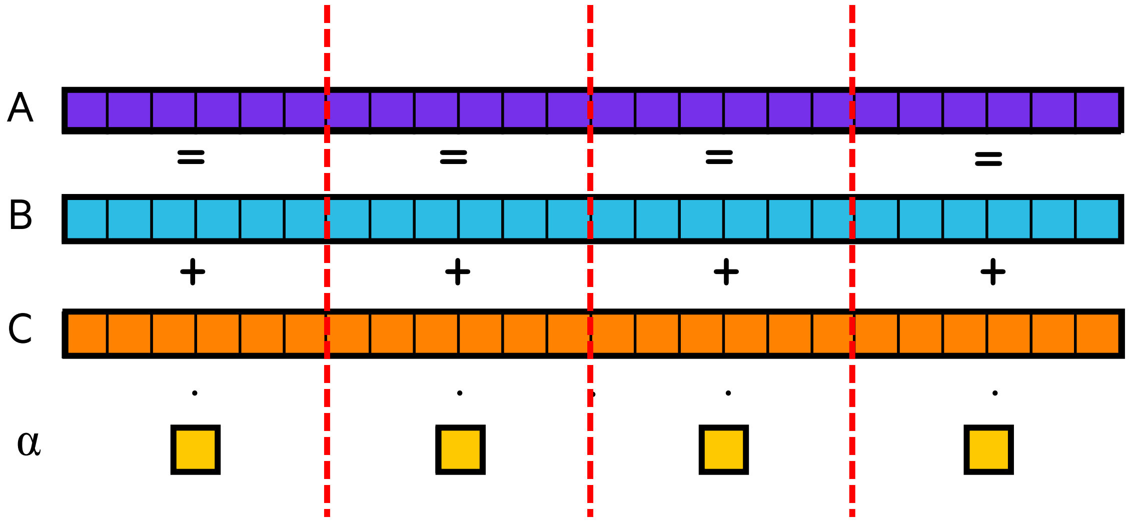


STREAM TRIAD: A TRIVIAL CASE OF PARALLELISM + LOCALITY

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)**:**

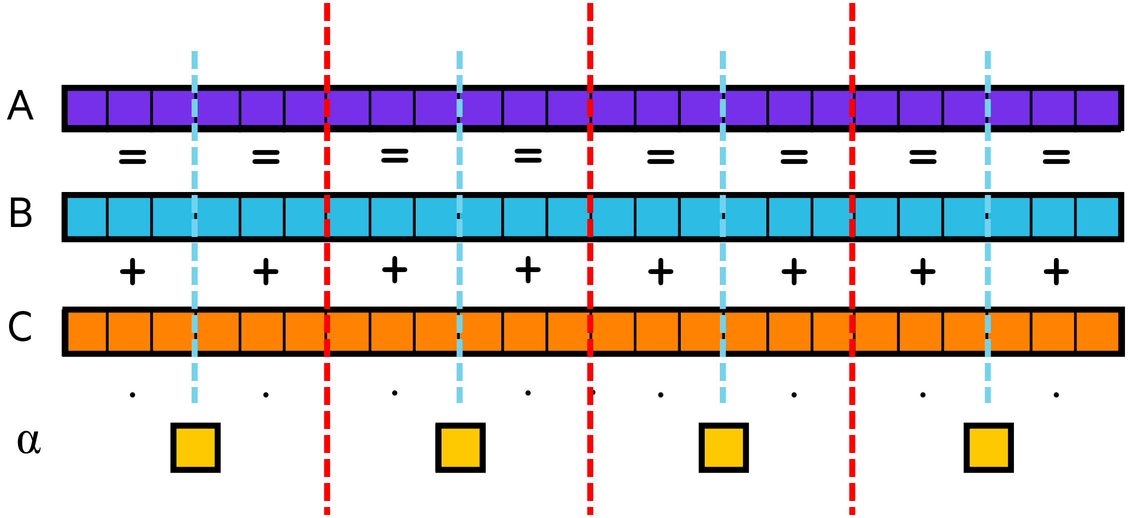


STREAM TRIAD: A TRIVIAL CASE OF PARALLELISM + LOCALITY

Given: *m*-element vectors *A*, *B*, *C*

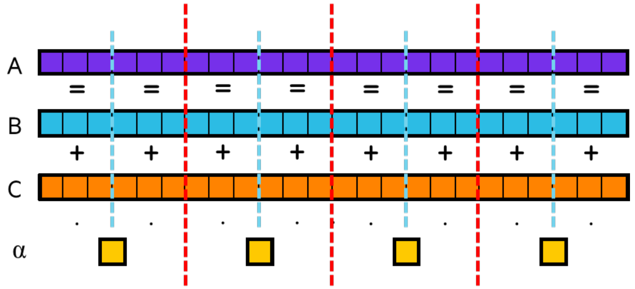
Compute: $\forall i \in 1..m, A_i = B_i + \alpha \cdot C_i$

In pictures, in parallel (distributed memory multicore):



STREAM TRIAD IN CONVENTIONAL HPC PROGRAMMING MODELS

Many Disparate Notations for Expressing Parallelism + Locality



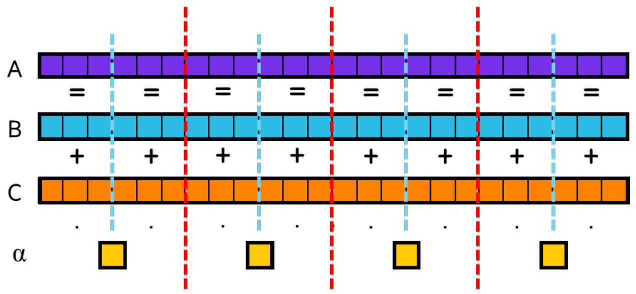
MPI

```
#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 );
  }
  return 1;
}


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;  }
```

STREAM TRIAD IN CONVENTIONAL HPC PROGRAMMING MODELS

Many Disparate Notations for Expressing Parallelism + Locality



MPI + OpenMP

```
#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 (%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;  }
```

CUDA

```
#define N 2000000

int main() {
  float *d_a, *d_b, *d_c;
  float scalar;

  cudaMalloc((void**)&d_a, sizeof(float)*N);
  cudaMalloc((void**)&d_b, sizeof(float)*N);
  cudaMalloc((void**)&d_c, sizeof(float)*N);

  dim3 dimBlock(128);
  dim3 dimGrid(N/dimBlock.x );
  if( N % dimBlock.x != 0 ) dimGrid

  set_array<<<dimGrid,dimBlock>>>(d_b, .5f, N);
  set_array<<<dimGrid,dimBlock>>>(d_c, .5f, N);

  scalar=3.0f;
  STREAM_Triad<<<dimGrid,dimBlock>>>(d_b, d_c, d_a, scalar,  N);
  cudaThreadSynchronize();

  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);

__global__ void set_array(float *a,  float value, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) a[idx] = value;
}

__global__ void STREAM_Triad( float *a, float *b, float *c,
                              float scalar, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) c[idx] = a[idx]+scalar*b[idx];  }
```

Note: This is a trivial parallel computation—imagine the additional complexity for something more realistic…

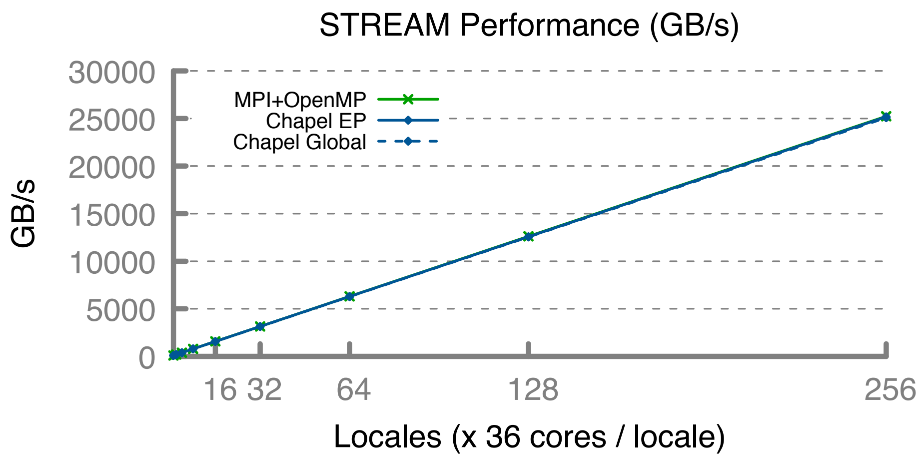
Challenge: Can we do better?

SPMD VS. GLOBAL-VIEW ACCOUNTS FOR MUCH OF CODE SIZE DIFFERENCES HERE

STREAM TRIAD: C + MPI + OPENMP

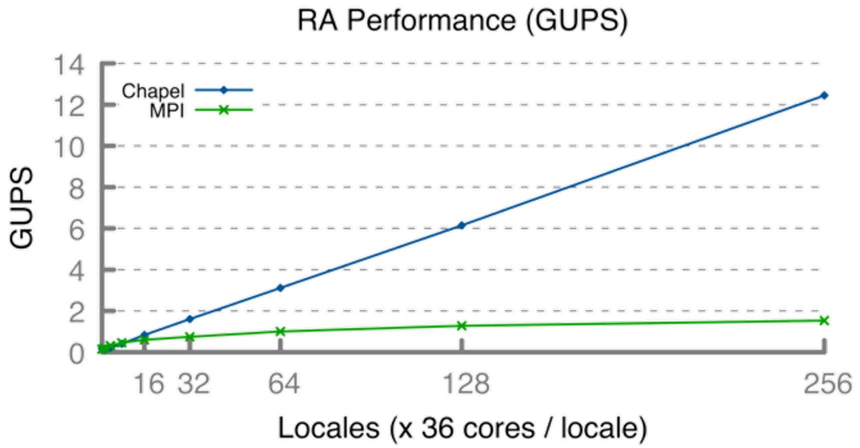
```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

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;
}
```

```
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;
```



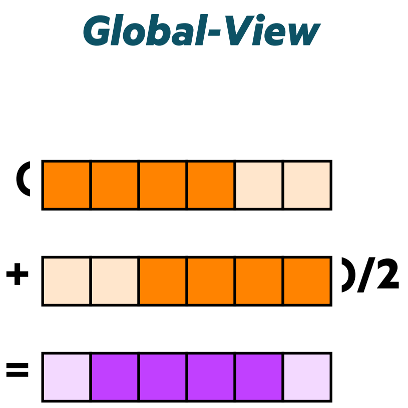
HPCC RA: MPI KERNEL

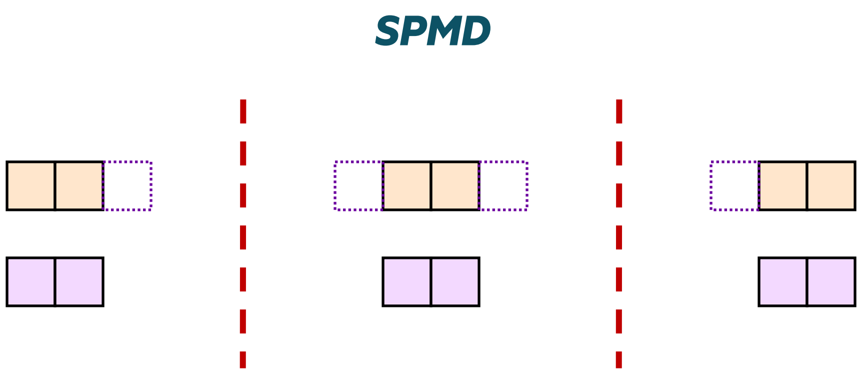
```
…
forall (_, r) in zip(Updates, RAStream()) do
  T[r & indexMask].xor(r);
…
```



CHAPEL SUPPORTS GLOBAL-VIEW / POST-SPMD PROGRAMMING

- “Apply a 3-point stencil to a vector”

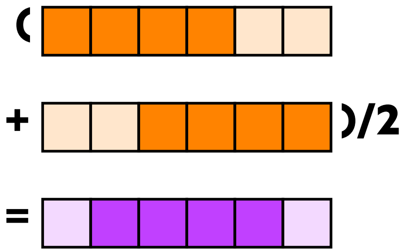




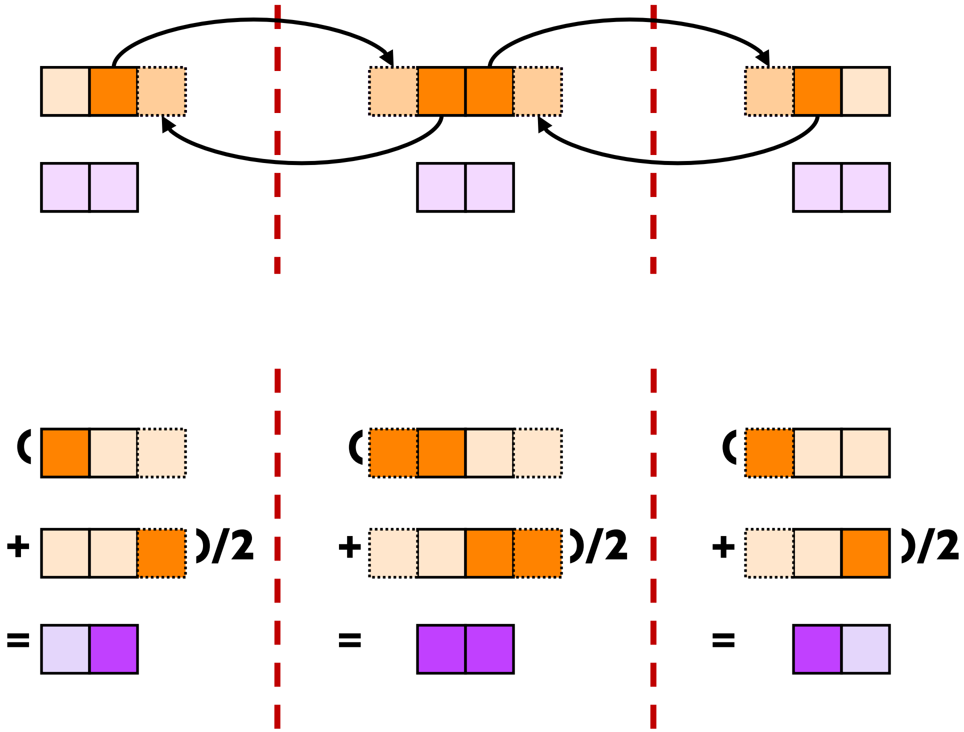
CHAPEL SUPPORTS GLOBAL-VIEW / POST-SPMD PROGRAMMING

- “Apply a 3-point stencil to a vector”

Global-View



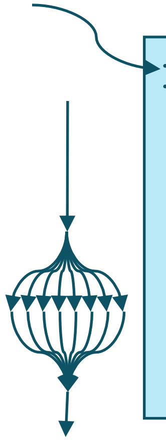
SPMD



CHAPEL SUPPORTS GLOBAL-VIEW / POST-SPMD PROGRAMMING

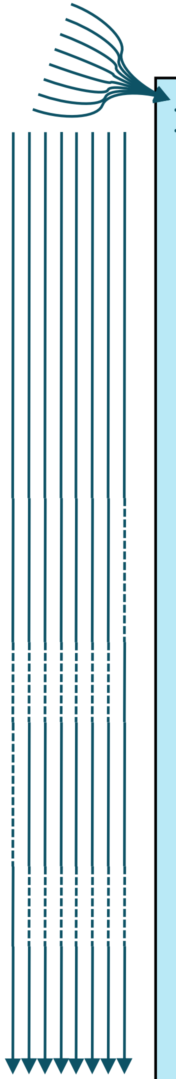
- "Apply a 3-point stencil to a vector"

Global-View Chapel code



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

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

SPMD pseudocode (MPI-esque)



```
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;
}
```

TWO QUICK SIDEBARS TO ROUND OUT THIS SECTION

1. Doing SPMD programming in Chapel
2. Illustrating Chapel's global namespace

SIDEBAR 1: CHAPEL SUPPORTS SPMD PROGRAMMING AS WELL

- Being a general-purpose language, Chapel doesn't preclude you from writing SPMD patterns in Chapel:

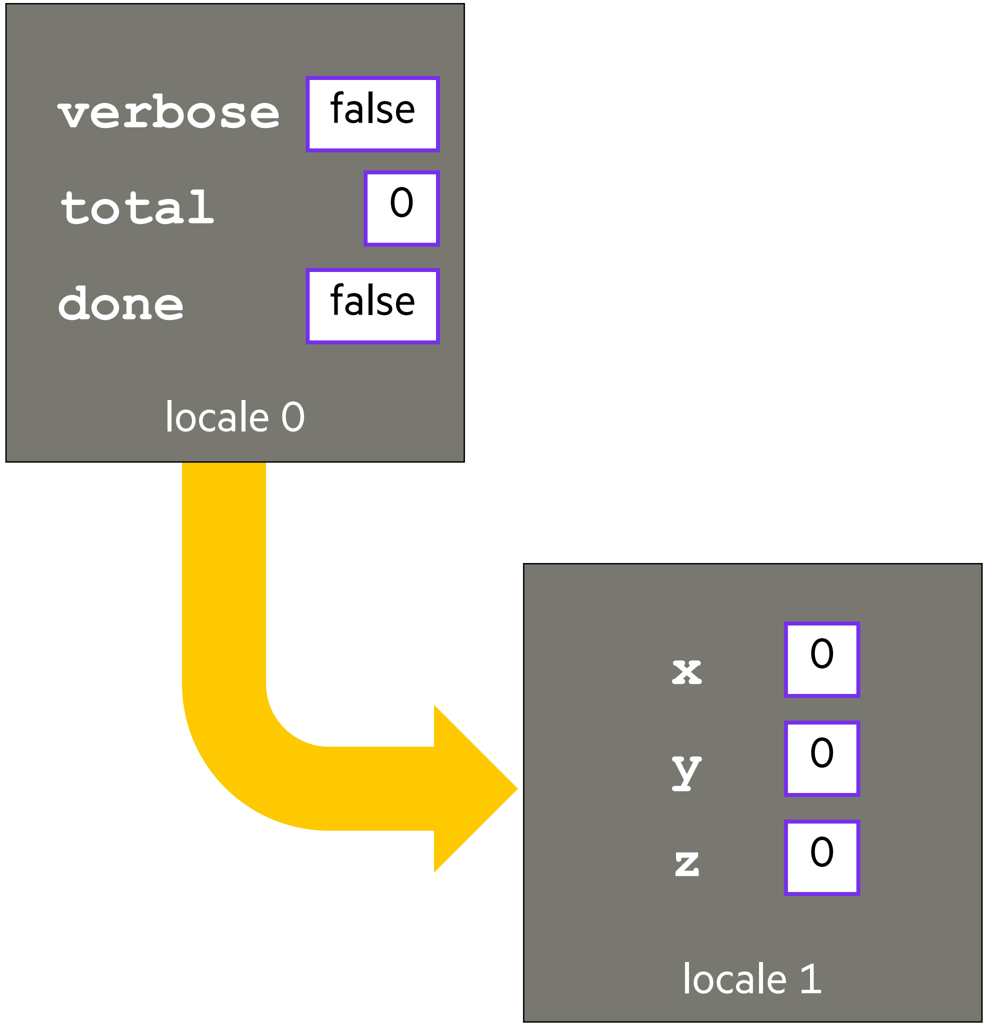
```
coforall loc in Locales do
  on loc do
    myMain();

proc myMain() {
  // ... write your SPMD computation here ...
}
```

SIDEBAR 2: CHAPEL'S GLOBAL NAMESPACE

Note 1: Variables are allocated on the locale where the task is running

onClause.chpl

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

…

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


}
```

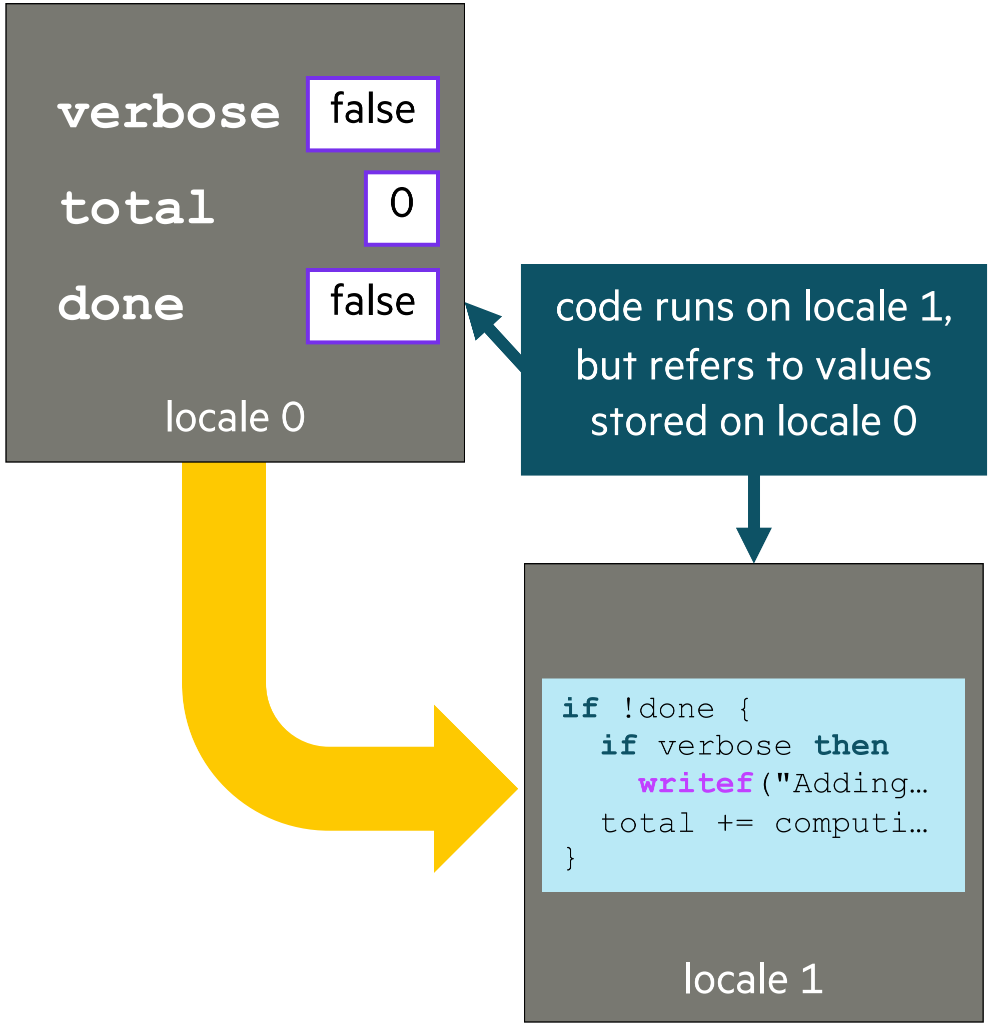


SIDEBAR 2: CHAPEL'S GLOBAL NAMESPACE

Note 2: Tasks can refer to visible variables, whether local or 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();
  }
}
```

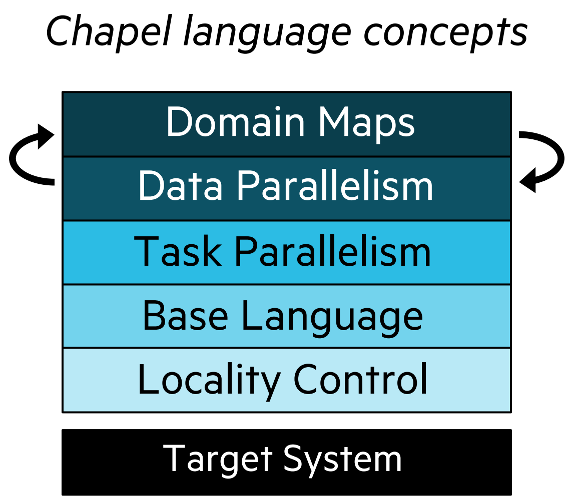


OUTLINE/TIME CHECK

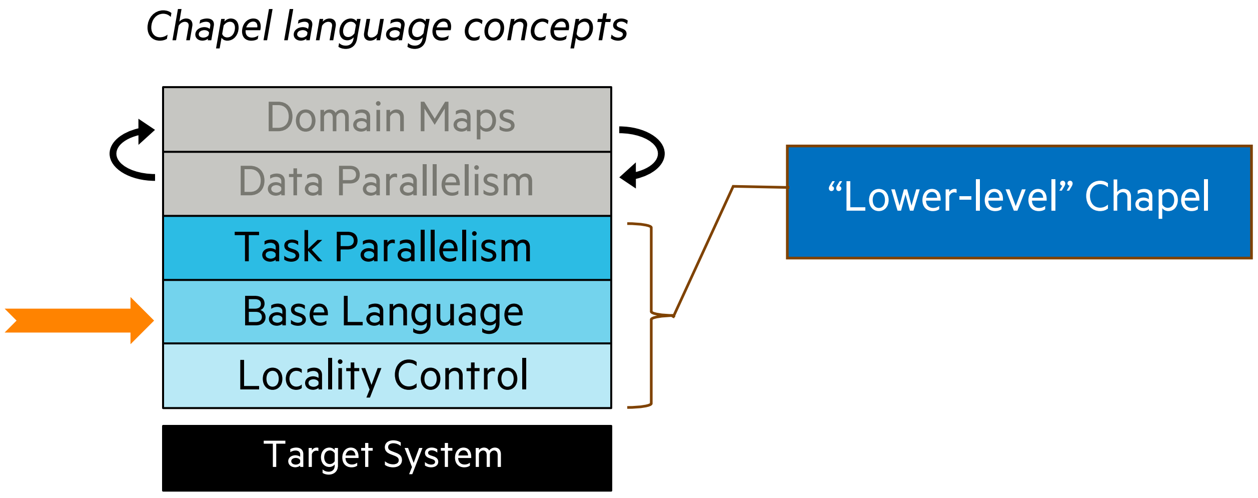
- Introductory Content
  - What is Chapel?
  - Chapel Characteristics
  - Chapel Benchmarks & Apps
  - Chapel vs. Standard Practice
- Further Details: Chapel Features
  - Base Language Features
  - Task-Parallelism & Locality
  - Data-Parallelism
- Wrap-up

FURTHER DETAILS:
OVERVIEW OF CHAPEL FEATURES

CHAPEL FEATURE AREAS



BASE LANGUAGE



A TOY COMPUTATION: THE FIBONACCI SEQUENCE

- Our first program shows a stylized way of computing *n* values of the Fibonacci sequence in Chapel…
  - This is admittedly an artificial example, but you might imagine replacing it with the code required to…

    …traverse your data structure

    …iterate in a tiled manner over your array

    …or any other iteration pattern that you'd like to paramterize, reuse, or abstract away from your primary computations

- The Fibonacci Sequence:
  - First two items:

    0

    1
  - Successive terms found by adding the previous two terms

    1 (0 + 1)

    2 (1 + 1)

    3 (1 + 2)

    5 (2 + 3)

    8 (3 + 5)

    …

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;
  }
}
```

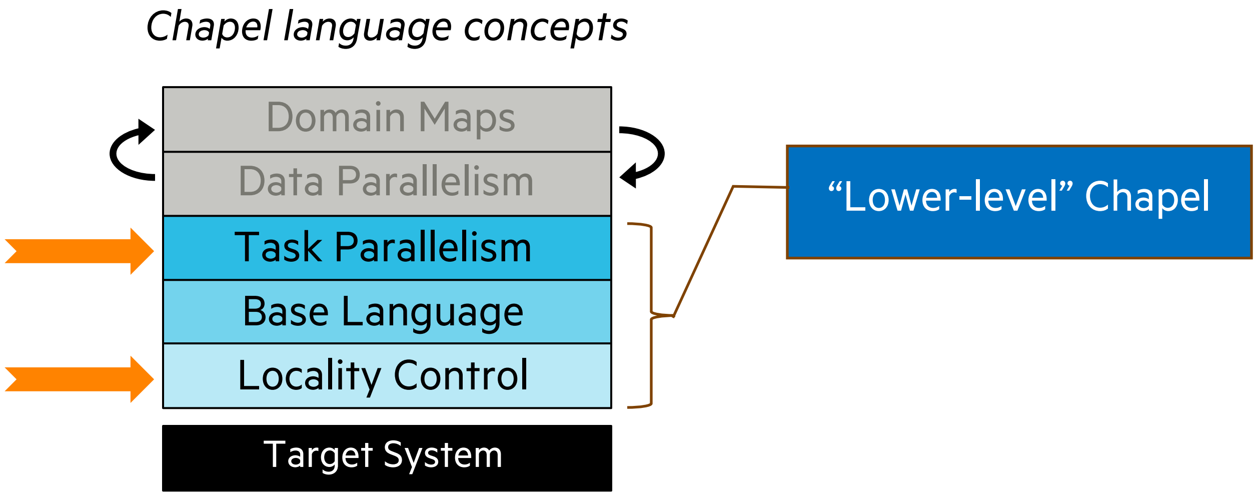
Range types and operators

```
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

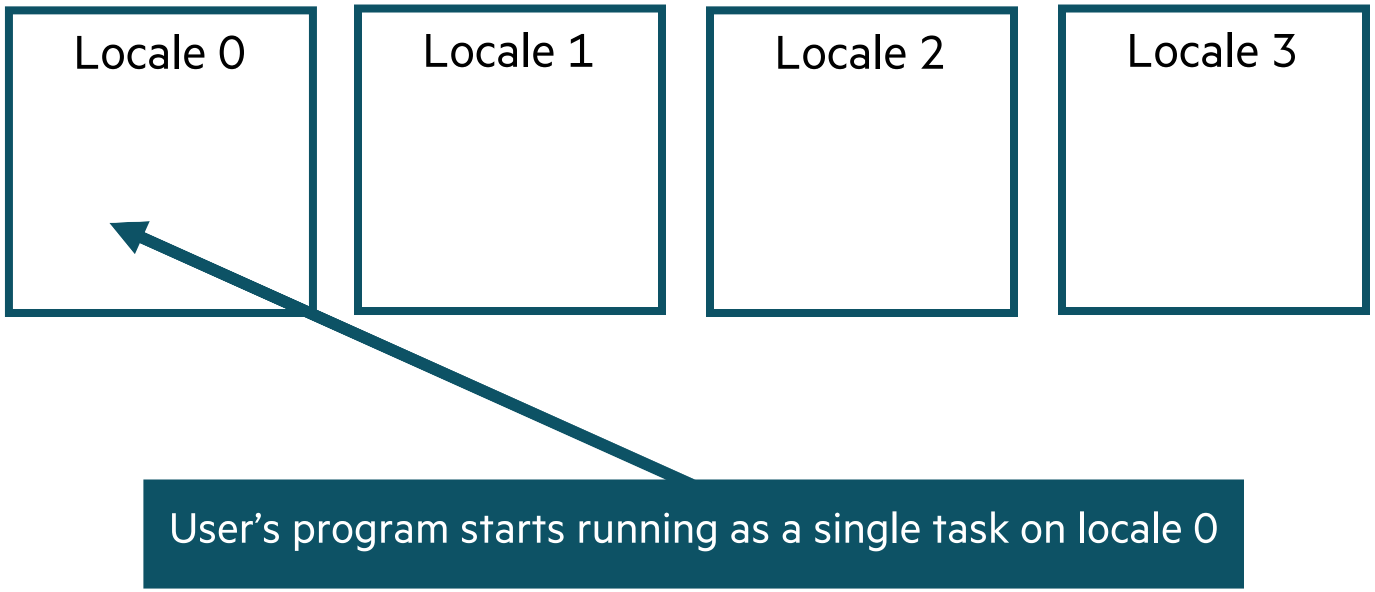


THE LOCALE: CHAPEL'S KEY FEATURE FOR LOCALITY

- *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'
```

`Locales` array:



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

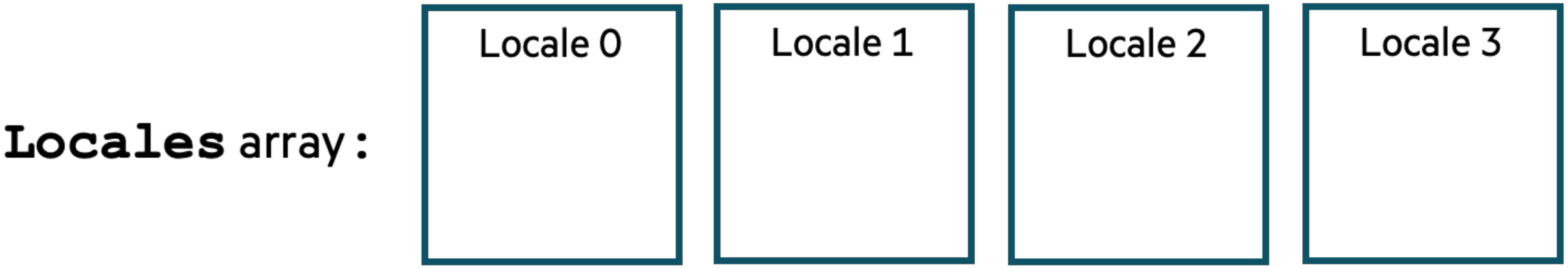
```
coforall loc in Locales {
  on loc {
    const numTasks = here.maxTaskPar;
    coforall tid in 1..numTasks do
      writef("Hello from task %n of %n on %s\n",
              tid, numTasks, here.name);
  }
}
```

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)



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
…
```

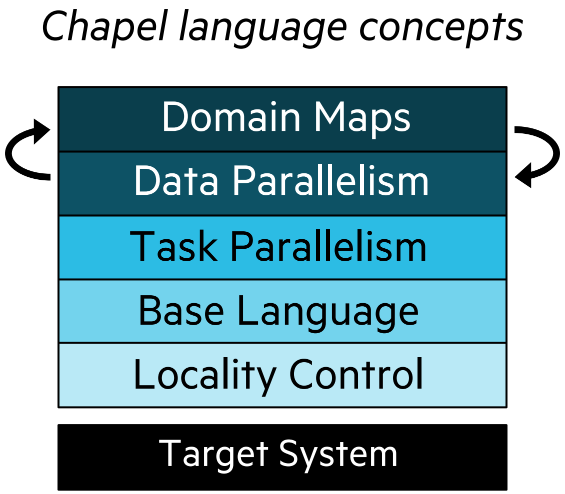
TASK-PARALLEL "HELLO WORLD" (DISTRIBUTED VERSION)

helloTaskPar.chpl

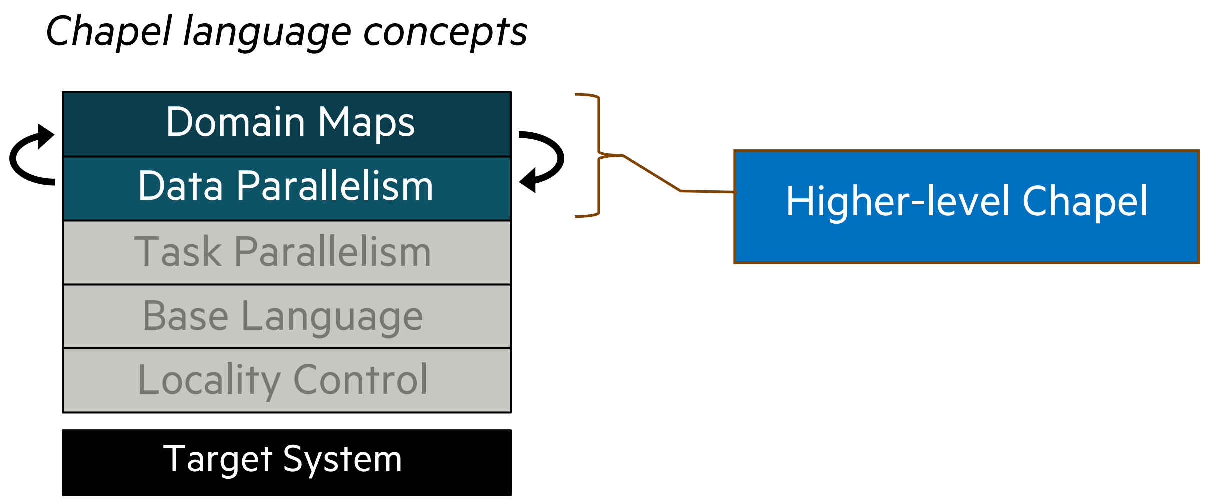
```
coforall loc in Locales {
  on loc {
    const numTasks = here.maxTaskPar;
    coforall tid in 1..numTasks do
      writef("Hello from task %n of %n on %s\n",
             tid, numTasks, here.name);
  }
}
```

```
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
…
```

CHAPEL FEATURE AREAS



DATA PARALLELISM AND DOMAIN MAPS



DATA-PARALLEL ARRAY FILL

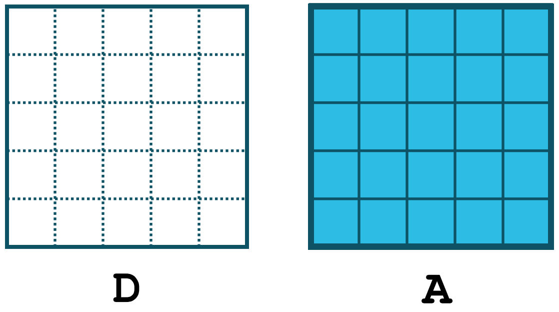
fillArray.chpl

```
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);
```



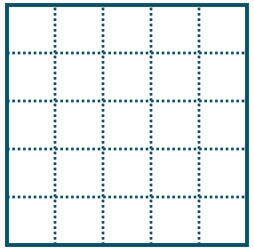
declare a domain, a first-class index set

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

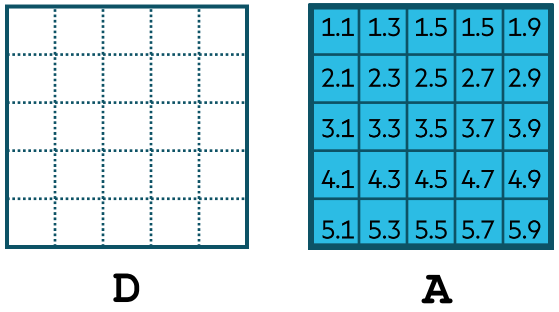
declare an array over that domain

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);
```




```
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
```

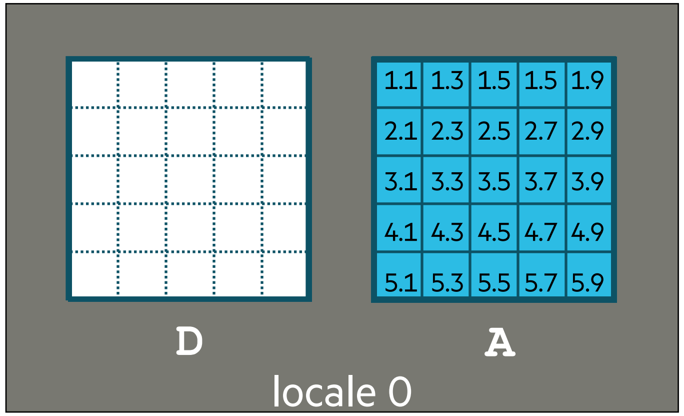
So far, this is a shared-memory program

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

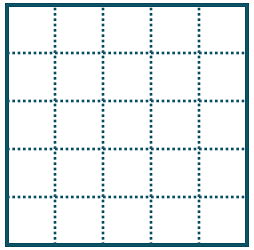
fillArray.chpl

```
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

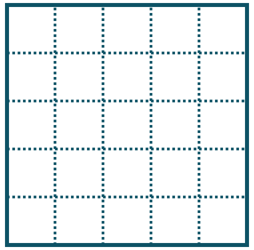
1.1	1.3	1.5	1.5	1.9
2.1	2.3	2.5	2.7	2.9
3.1	3.3	3.5	3.7	3.9
4.1	4.3	4.5	4.7	4.9
5.1	5.3	5.5	5.7	5.9

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);
```



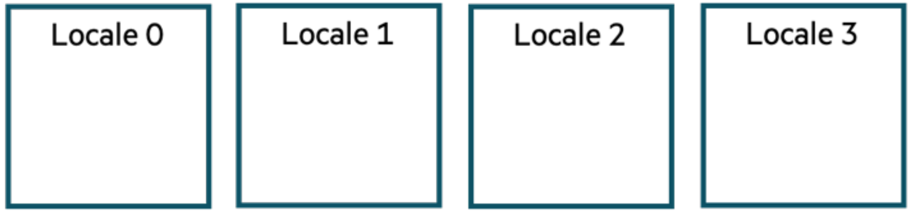
D

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

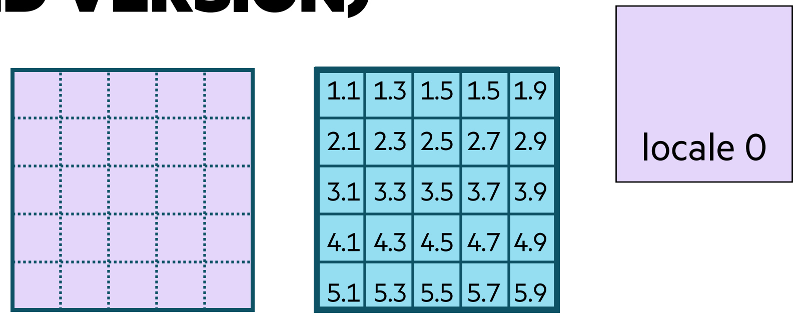
A

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

Locales array:



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);
```

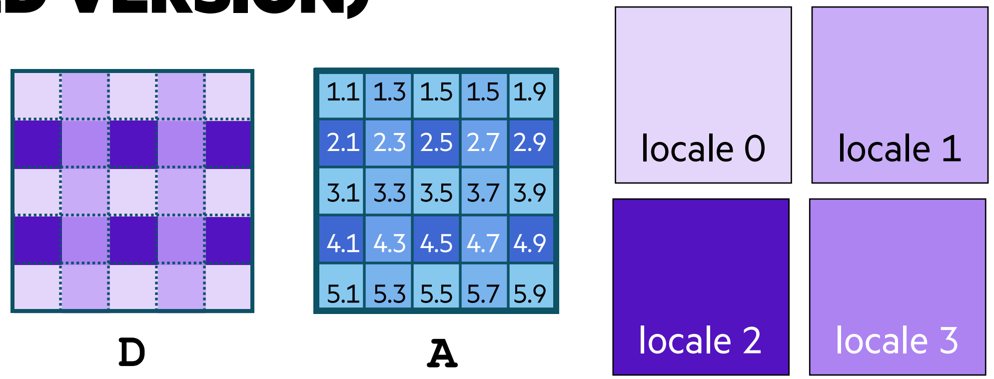
```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5 --numLocales=1
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

Because this computation is independent of the locales,
changing the number of locales or distribution doesn’t affect the output

DATA-PARALLEL ARRAY FILL (DISTRIBUTED VERSION)

fillArray.chpl

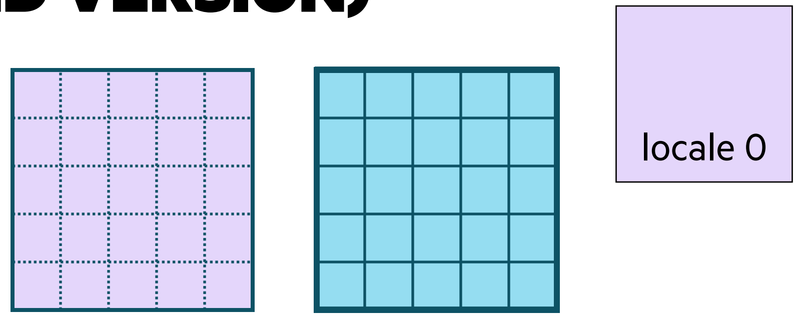
```
use CyclicDist;

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
```

Because this computation is independent of the locales,
changing the number of locales or distribution doesn’t affect the output

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*10 + j + (here.id+1)/10.0;

writeln(A);
```

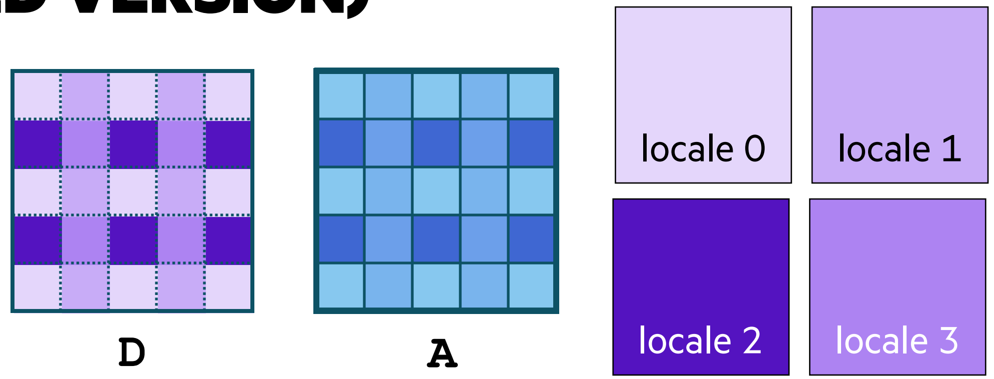
```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5 --numLocales=1
11.1 12.1 13.1 14.1 15.1
21.1 22.1 23.1 24.1 25.1
31.1 32.1 33.1 34.1 35.1
41.1 42.1 43.1 44.1 45.1
51.1 52.1 53.1 54.1 55.1
```

If we make it sensitive to the locales,
the output varies with the distribution details

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*10 + j + (here.id+1)/10.0;

writeln(A);
```




```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5 --numLocales=4
11.1 12.2 13.1 14.2 15.1
21.3 22.4 23.3 24.4 25.3
31.1 32.2 33.1 34.2 35.1
41.3 42.4 43.3 44.4 45.3
51.1 52.2 53.1 54.2 55.1
```

If we make it sensitive to the locales,
the output varies with the distribution details

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*10 + j + (here.id+1)/10.0;

writeln(A);
```

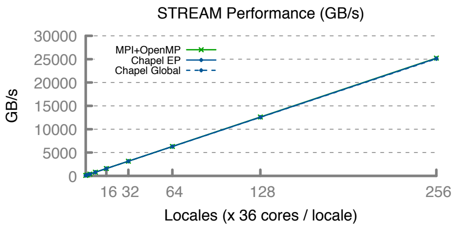
WRAP-UP

SUMMARY

Chapel is unique among programming languages

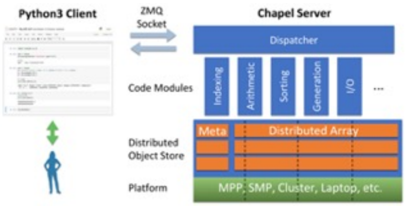
- built-in features for scalable parallel computing make it HPC-ready
- supports clean, concise code relative to conventional approaches
- ports and scales from laptops to supercomputers

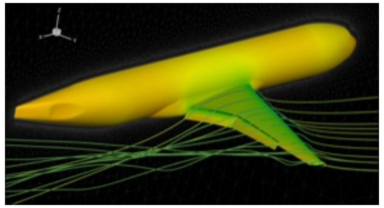
```
use BlockDist;

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;
```



Chapel is being used for productive parallel computing at scale

- users are reaping its benefits in practical, cutting-edge applications
- applicable to domains as diverse as physical simulations and data science





If you're interested in taking Chapel for a spin, let us know!

- we're happy to work with users and user groups to help ease the learning curve
- we're discussing holding a day-long tutorial for you with hands-on, pending interest

CHAPEL RESOURCES

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

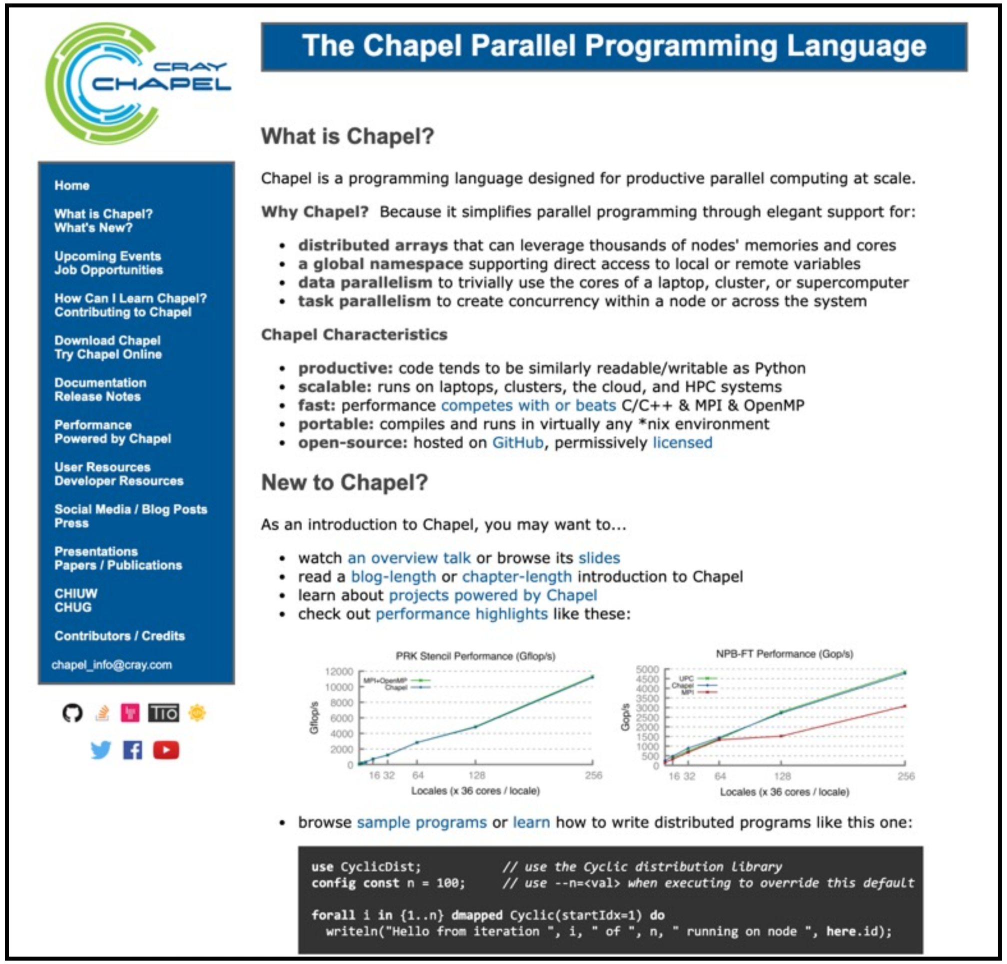
- (points to all other resources)

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



THANK YOU

https://chapel-lang.org
@ChapelLanguage


BACKUP SLIDES: CHAPEL ON GPUS

CHAPEL ON GPUS

Background:

- GPUs have become a key feature in many HPC systems
- We have long described Chapel's goal as being "any parallel algorithm on any parallel hardware"
- Yet, historically, Chapel releases have only supported GPUs via interoperability
  - i.e., call GPU code written in CUDA, OpenCL, OpenMP, … as an extern routine

What's New?

- Lots of progress in the past year…

CHAPEL FOR GPUS: CHAPEL 1.24.0

Targeting GPUs with Chapel was possible for the first time, but very low-level:

```
pragma "codegen for GPU"
export proc add_nums(A: c_ptr(real(64))){
  A[0] = A[0]+5;
}

var funcPtr = createFunction();
var A = [1, 2, 3, 4, 5];
__primitive("gpu kernel launch", funcPtr,
            <grid and block size>,…,
            c_ptrTo(A), …);
writeln(A);
```

```
extern {
  #define FATBIN_FILE "chpl__gpu.fatbin"
  double createFunction(){
    fatbinBuffer = <read FATBIN_FILE into buffer>
    cuModuleLoadData(&cudaModule, fatbinBuffer);
    cuModuleGetFunction(&function, cudaModule,
                        "add_nums");}

}
```

CHAPEL FOR GPUS: CHAPEL 1.25.0

Raised the level of abstraction significantly, yet with significant restrictions:

- only relatively simple computations
- single GPU only
- single locale only

```
on here.gpus[0] {
  var A: [0..<n] int;
  foreach a in A do
    a += 1;
}
```

‘on’ statement controls the execution/allocation policy

‘A’ will be allocated in the Unified Virtual Memory

‘foreach’ will turn into a GPU kernel

CHAPEL FOR GPUS: CHAPEL 1.26.0

Improved generality: computational styles, multiple GPUs, CPU+GPU parallelism

Two concurrent tasks

```
cobegin {
  A[0..<cpuSize] += 1;
  coforall subloc in 1..numGPUs do on here.getChild(subloc) {
    const myShare = cpuSize+gpuSize*(subloc-1)..#gpuSize;

    var AonThisGPU = A[myShare];
    AonThisGPU += 1;
    A[myShare] = AonThisGPU;
  }
}
```

CPU works on its part

GPUs work on their part and copy the result back

CHAPEL FOR GPUS: CHAPEL 1.27.0

Added support for using the GPUs of multiple locales simultaneously, improved sublocale abstractions

```
config const n=1000, alpha=0.5;

coforall loc in Locales do on loc {
  coforall gpu in here.gpus do on gpu {
    var A, B, C: [1..n] real;
    A = B + alpha * C;
  }
}
```

CHAPEL FOR GPUS: WHAT'S NEXT?

- Performance Analysis & Improvements
- Portability to additional vendors
- GPU participation in inter-node communication