



Hewlett Packard
Enterprise

ONE-DAY CHAPEL TUTORIAL



Chapel Team
October 16, 2023

ONE DAY CHAPEL TUTORIAL

- 9-10:30: Getting started using Chapel for parallel programming
- 10:30-10:45: break
- 10:45-12:15: Chapel basics in the context of the n-body example code
- 12:15-1:15: lunch
- 1:15-2:45: Distributed and shared-memory parallelism especially w/arrays (data parallelism)
- 2:45-3:00: break
- 3:00-4:30: More parallelism including for asynchronous parallelism (task parallelism)
- 4:30-5:00: Wrap-up including gathering further questions from attendees



OUTLINE: OVERVIEW OF PROGRAMMING IN CHAPEL

- Chapel Goals, Usage, and Comparison with other Tools
- Hello World (Hands On)
- Chapel Execution Model and Parallel Hello World (Hands On)
- kmer counting using file IO, config consts, strings, maps (Hands On)
- Parallelizing a program that processes files (Hands On)
- GPU programming support
- Learning goals for rest of tutorial



CHAPEL GOALS, USAGE, AND COMPARISON WITH OTHER TOOLS

CHAPEL PROGRAMMING LANGUAGE

Chapel is a general-purpose programming language that provides **ease of parallel programming, high performance, and portability.**

And is being used in applications in various ways:

refactoring existing codes,

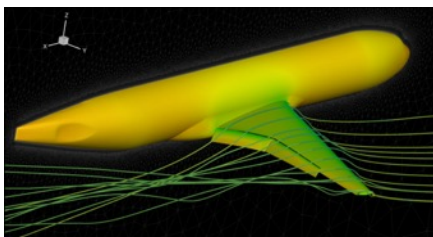
developing new codes,

serving high performance to Python codes **(Chapel server with Python client)**, and

providing distributed and shared memory parallelism for existing codes.

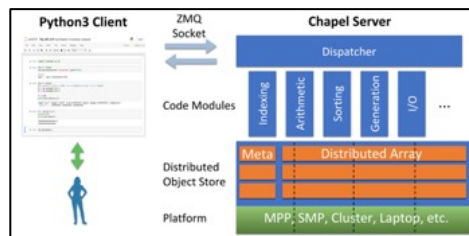


APPLICATIONS OF CHAPEL: LINKS TO USERS' TALKS (SLIDES + VIDEO)



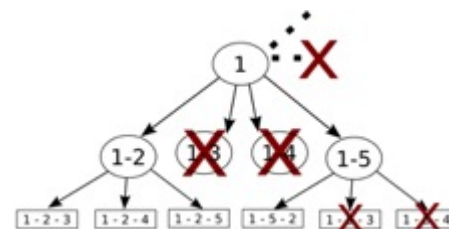
CHAMPS: 3D Unstructured CFD

[CHIUW 2021](#) [CHIUW 2022](#)



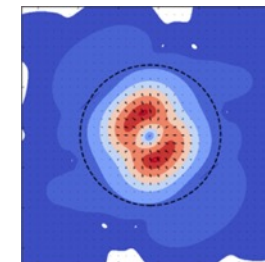
Arkouda: Interactive Data Science at Massive Scale

[CHIUW 2020](#) [CHIUW 2023](#)



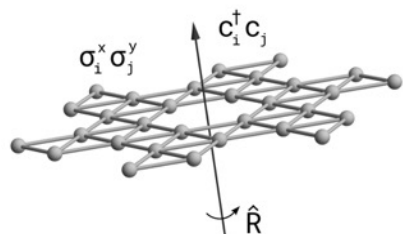
ChOp: Chapel-based Optimization

[CHIUW 2021](#) [CHIUW 2023](#)



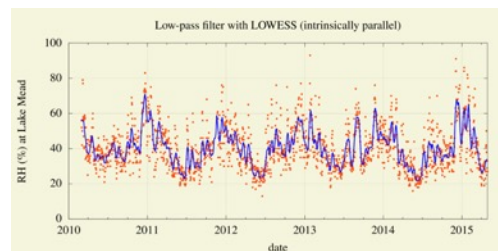
ChplUltra: Simulating Ultralight Dark Matter

[CHIUW 2020](#) [CHIUW 2022](#)



Lattice-Symmetries: a Quantum Many-Body Toolbox

[CHIUW 2022](#)



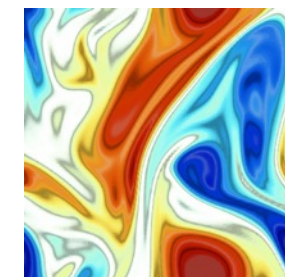
Desk dot chpl: Utilities for Environmental Eng.

[CHIUW 2022](#)

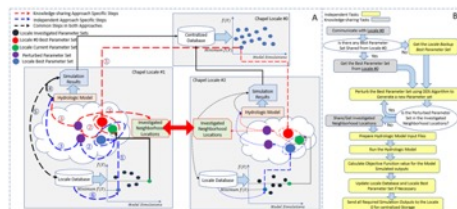


RapidQ: Mapping Coral Biodiversity

[CHIUW 2023](#)

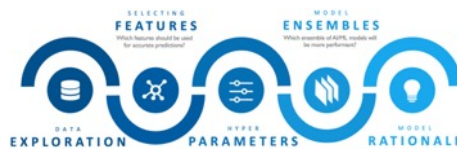


ChapQG: Layered Quasigeostrophic CFD



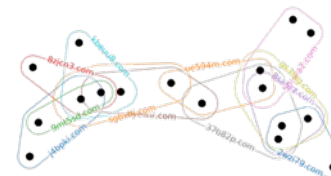
Chapel-based Hydrological Model Calibration

[CHIUW 2023](#)



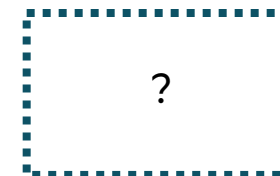
CrayAI HyperParameter Optimization (HPO)

[CHIUW 2021](#)



CHGL: Chapel Hypergraph Library

[CHIUW 2020](#)

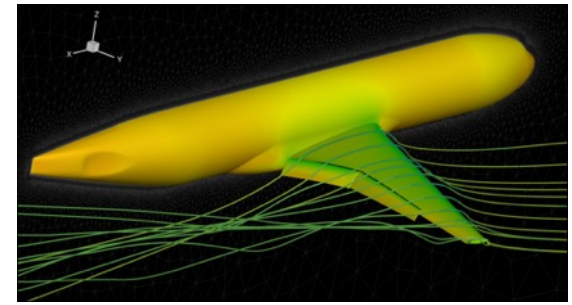


Your Application Here?

HIGHLIGHTS OF CHAPEL USAGE

CHAMPS: Computational Fluid Dynamics framework for airplane simulation

- Professor Eric Laurendeau's team at Polytechnique Montreal
- Performance: achieves competitive results w.r.t. established, world-class frameworks from Stanford, MIT, etc.
- Programmability: *"We ask students at the master's degree to do stuff that would take 2 years and they do it in 3 months."*



Arkouda: data analytics framework (<https://github.com/Bears-R-Us/arkouda>)

- Mike Merrill, Bill Reus, et al., US DOD
- Python front end client, Chapel server that processes dozens of terabytes in seconds
- April 2023: 1200 GiB/s for argsort on an HPE EX system



Other recent users

- Marjan Asgari et al, "Development of a knowledge-sharing parallel computing approach for calibrating distributed watershed hydrologic models", Environmental Modeling and Software.
- Scott Bachman has written some coral reef image analysis applications in Chapel.



CHAPEL IS HIGHLY PERFORMANT AND SCALABLE

HPE Apollo (May 2021)



- HDR-100 Infiniband network (100 Gb/s)
- 576 compute nodes
- 72 TiB of 8-byte values
- ~480 GiB/s (~150 seconds)

HPE Cray EX (April 2023)



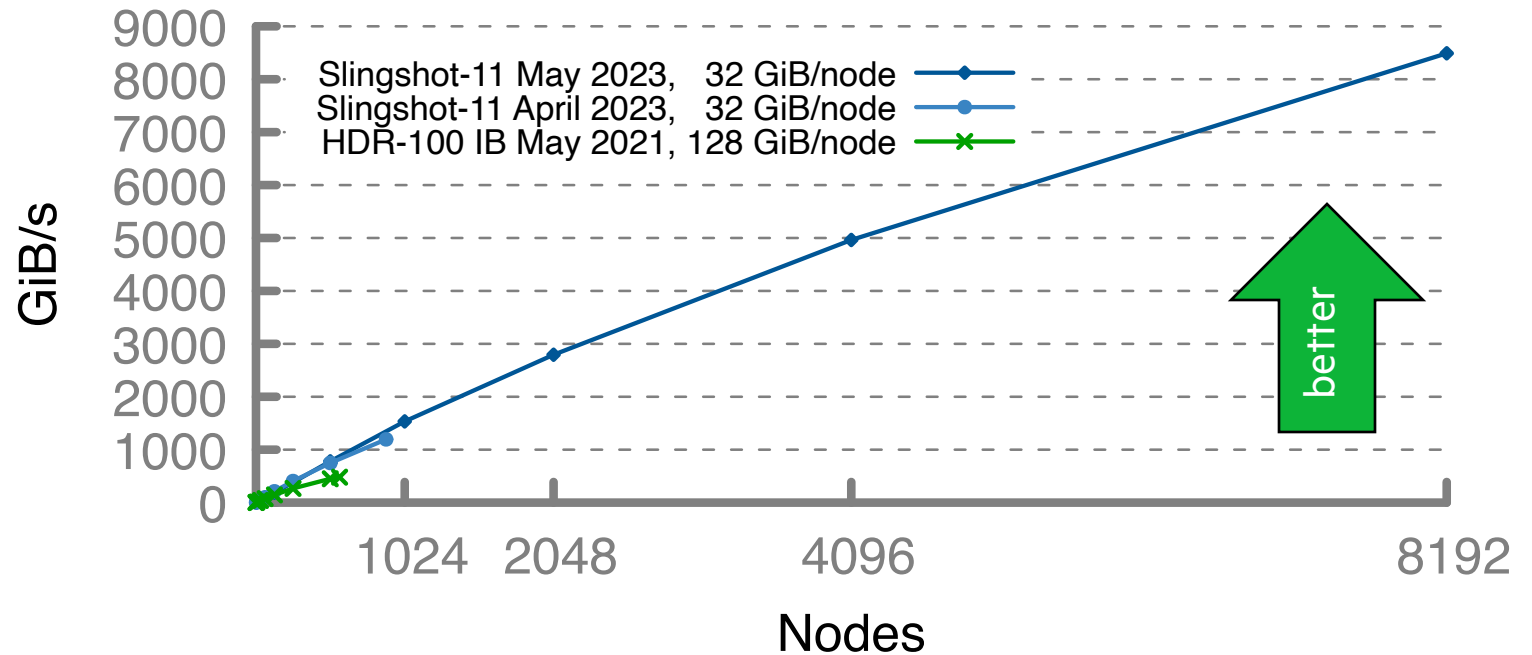
- Slingshot-11 network (200 Gb/s)
- 896 compute nodes
- 28 TiB of 8-byte values
- ~1200 GiB/s (~24 seconds)

HPE Cray EX (May 2023)



- Slingshot-11 network (200 Gb/s)
- 8192 compute nodes
- 256 TiB of 8-byte values
- ~8500 GiB/s (~31 seconds)

Arkouda Argosort Performance



A notable performance achievement in ~100 lines of Chapel

COMPARE WITH OTHER PARALLEL PROGRAMMING MODELS

- Shared-memory parallelism
 - Pthreads: low-level library for creating and managing threads of
 - OpenMP: pragmas added before loops and other statements in C
 - Rust, Julia: programming languages with some threaded parallel
 - RAJA, Kokkos: C++ libraries that use template metaprogramming
- Distributed-memory parallelism and shared-memory parallelism
 - MPI+X:
 - MPI stands for message passing interface
 - MPI is a library for sending and receiving messages between processes
 - All processes allocate their own memory and run the same program, S
 - There are many options for X: OpenMP, Pthreads, Python, Julia, RAJA
 - OpenSHMEM: library for implementing a partitioned global address
 - Spark: Python, Scala, and Java accessible library for especially th
 - Regent and Legion: programming language and runtime that implements implicit task parallelism
 - Kokkos Remote Spaces: extends Kokkos C++ template views to distributed views

Chapel:

- shared memory parallelism,
- distributed-memory parallelism,
- data parallelism,
- task parallelism,
- map-reduce parallelism,
- vector parallelism,
- GPU parallelism, ...

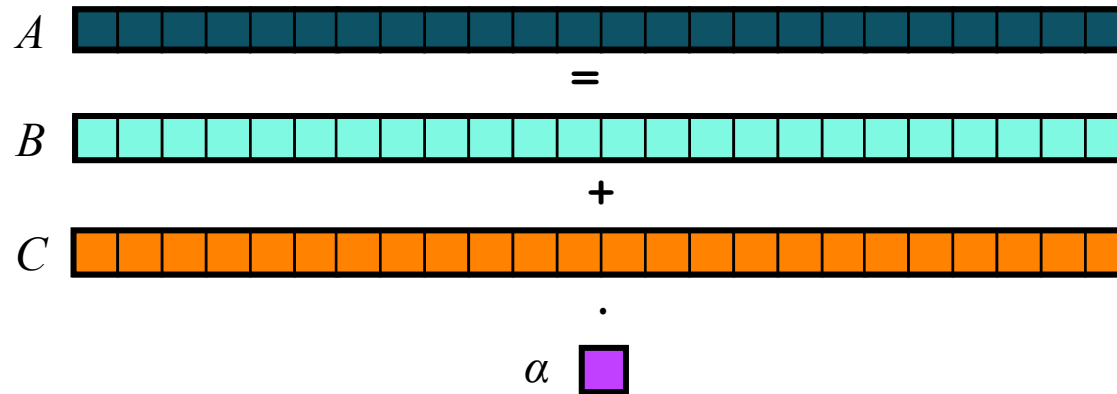
All can be expressed in the same programming language.

LET'S COMPARE WITH STREAM TRIAD: A PARALLEL COMPUTATION

Given: m -element vectors A, B, C

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

In pictures:

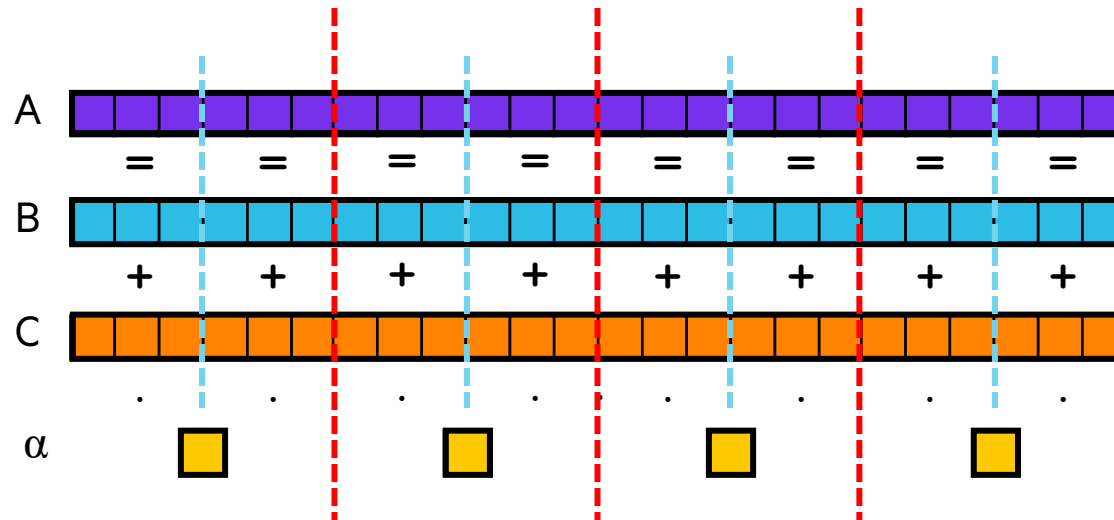


LET'S COMPARE WITH STREAM TRIAD: A PARALLEL COMPUTATION

Given: n -element vectors A, B, C

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

In pictures, in parallel (distributed memory multicore, global-view):

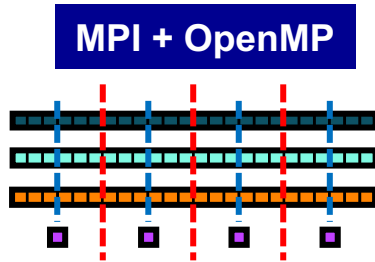


STREAM TRIAD: IN MPI+OPENMP

```
#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;
```



```
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 );
    }
}
```

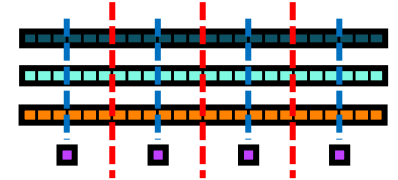
```
#define N 2000000
```

CUDA

```
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);
```



*HPC suffers from too many distinct notations for expressing parallelism and locality.
This tends to be a result of bottom-up language design.*

```
rv = HPCC_StarStream(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 );
```

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

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

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

return 0; }
```

```
STREAM_Triad(<<<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]; }
```

WHY SO MANY PROGRAMMING MODELS?

HPC tends to approach programming models bottom-up:

Given a system and its core capabilities...

...provide features that permit users to access the available performance.

Type of HW Parallelism	Programming Model	Unit of Parallelism
Inter-node	MPI	executable
Intra-node/multicore	OpenMP / pthreads	iteration/task
Instruction-level vectors/threads	pragmas	iteration
GPU/accelerator	CUDA / Open[MP CL ACC]	SIMD function/task

benefits: lots of control; decent generality; easy to implement

downsides: lots of user-managed detail; brittle to changes



STREAM TRIAD: IN CHAPEL

The special sauce:

How should this index set—and any arrays and computations over it—be mapped to the system?

```
use BlockDist;

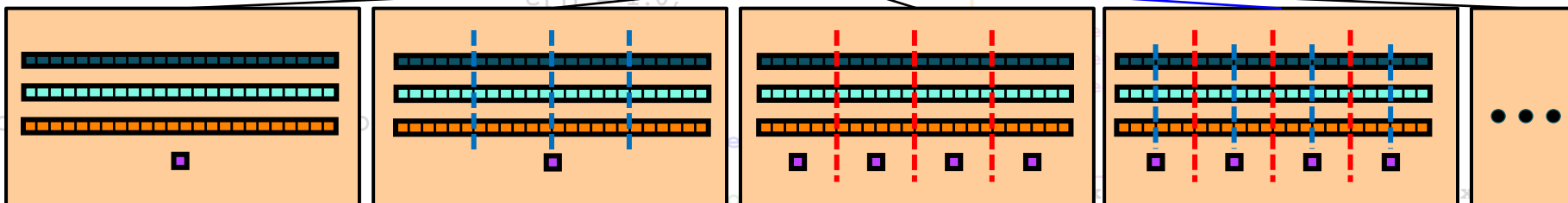
config const m = 1000,
           alpha = 3.0;

const ProblemSpace = blockDist.createDomain({1..m});

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

B = 2.0;
C = 1.0;

A = B + alpha * C;
```



Philosophy: Top-down language design can tease system-specific implementation details away from an algorithm, permitting the compiler, runtime, applied scientist, and HPC expert to each focus on their strengths.

HELLO WORLD (HANDS ON)

HANDS ON: HOW TO DO THE HANDS ON



01-hello.chpl

Zip file with example codes and slides

- <https://chapel-lang.org/tutorials/Oct2023/ChapelExamplesFromOct2023Tutorial.zip>

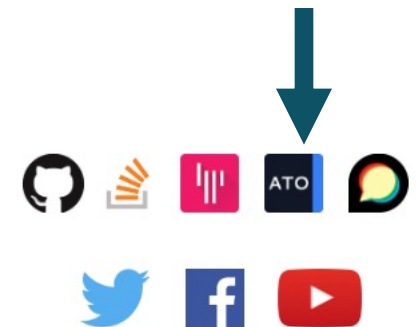
Using a container on your laptop

- First, install docker for your machine and start it up (see the README.md for more info)
- Then, use the chapel-gasnet docker container

```
docker pull docker.io/chapel/chapel-gasnet      # takes about 5 minutes
cd ChapelTutorialSlidesAndCodes                # assuming zip file has been unzipped
docker run --rm -it -v "$PWD":/myapp -w /myapp chapel/chapel-gasnet /bin/bash
root@xxxxxxxxxx:/myapp# chpl 01-hello.chpl
root@xxxxxxxxxx:/myapp# ./01-hello -nl 1
```

Attempt this Online website for running Chapel code

- Go to main Chapel webpage at <https://chapel-lang.org/>
- Click on the little ATO icon on the lower left that is above the YouTube icon



"HELLO WORLD" IN CHAPEL: TWO VERSIONS

- Fast prototyping

```
writeln("Hello, world!");
```



01-hello.chpl

- “Production-grade”

```
module Hello {  
  
    proc main() {  
        writeln("Hello, world!");  
    }  
  
}
```



01-hello-production.chpl



"HELLO WORLD" IN CHAPEL: TWO VERSIONS

- Fast prototyping (configurable)

```
config const audience = "world";  
writeln("Hello, ", audience, "!");
```



01-hello-configurable.chpl

- “Production-grade” (configurable)

```
module Hello {  
    config const audience = "world";  
  
    proc main() {  
        writeln("Hello, ", audience, "!");  
    }  
}
```



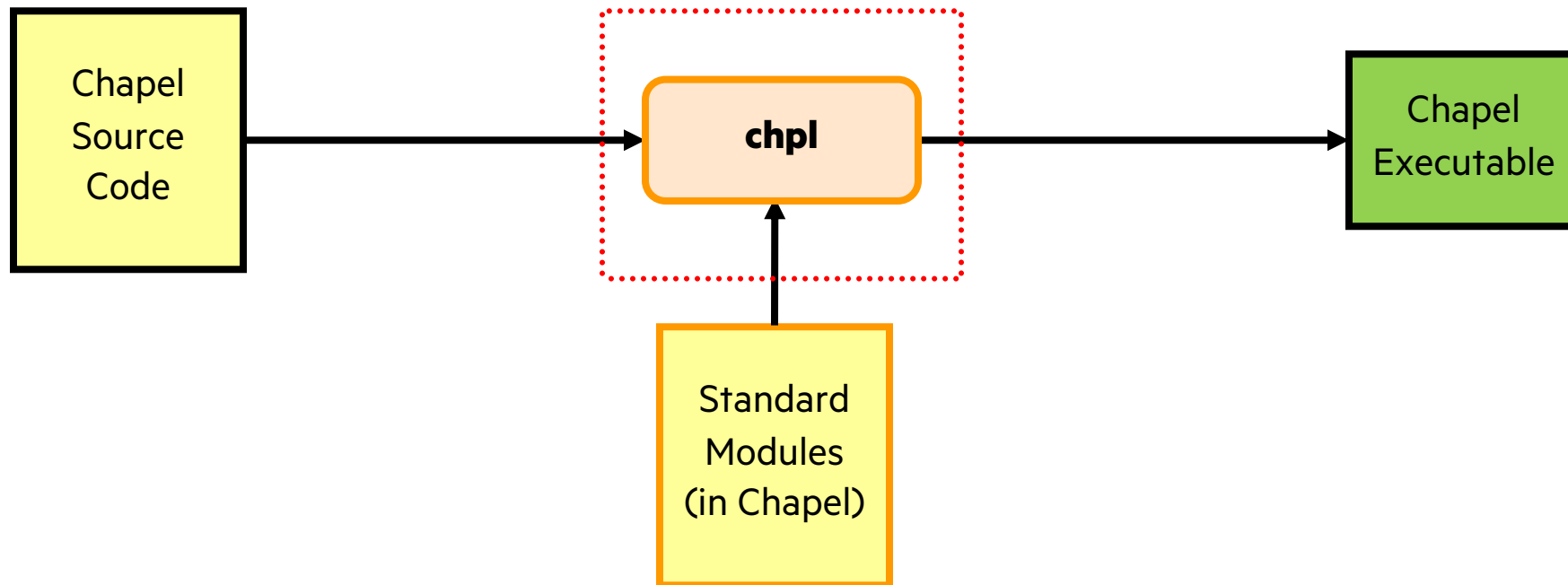
01-hello-production-configurable.chpl

- To change ‘audience’ for a given run:

```
./01-hello-configurable -nl 1 --audience="y'all"
```



COMPILING CHAPEL



CHAPEL EXECUTION MODEL AND PARALLEL HELLO WORLD (HANDS ON)

CHAPEL EXECUTION MODEL AND TERMINOLOGY: LOCALES

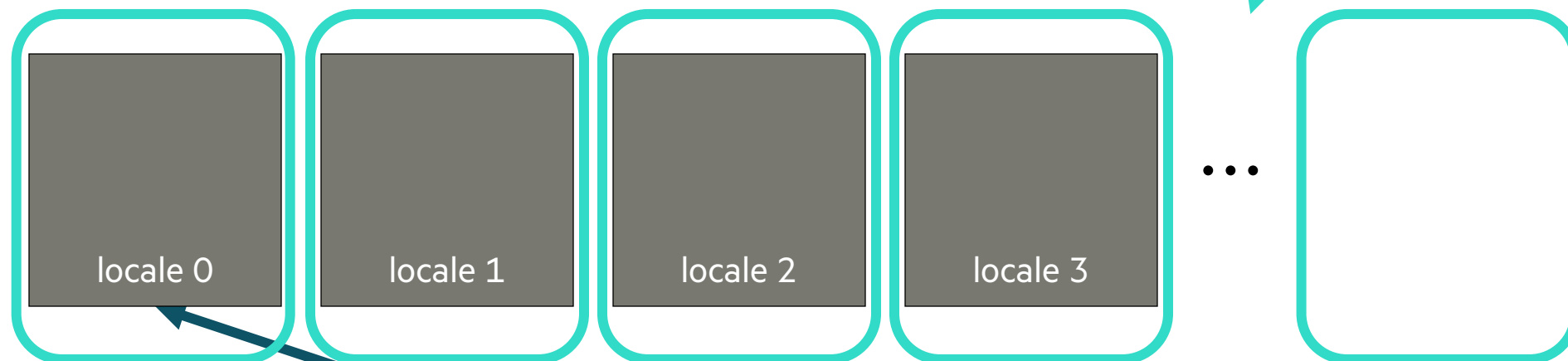
Locales can run tasks and store variables

- Each locale executes on a “compute node” on a parallel system
- User specifies number of locales on executable’s command-line

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

System has many nodes

Locales array :



User's code starts running as a single task on locale 0

TASK-PARALLEL “HELLO WORLD”

01-hello-dist-node-names.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”

01-hello-dist-node-names.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 we’re currently running

how many concurrent tasks does this node support (typically the number of processor cores)?

what’s my locale’s name?

TASK-PARALLEL “HELLO WORLD”

01-hello-dist-node-names.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

```
> chpl 01-hello-dist-node-names.chpl  
> ./01-hello-dist-node-names -nl 1  
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”

01-hello-dist-node-names.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);
```

```
> chpl 01-hello-dist-node-names.chpl  
> ./01-hello-dist-node-names -nl 1  
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” (DISTRIBUTED VERSION)

01-hello-dist-node-names.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

Locales array:

locale 0

locale 1

locale 2

locale 3

TASK-PARALLEL “HELLO WORLD” (DISTRIBUTED VERSION)

01-hello-dist-node-names.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

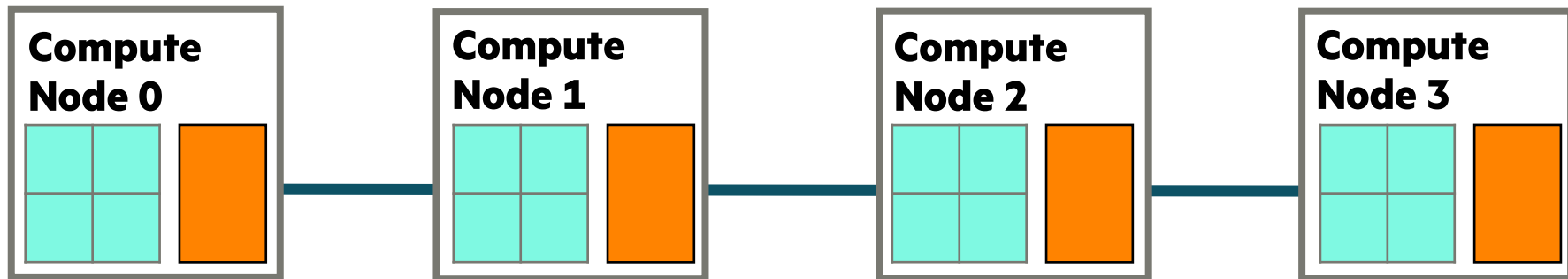
```
> chpl 01-hello-dist-node-names.chpl  
> ./01-hello-dist-node-names -nl=4  
Hello from task 1 of 4 on n1032  
Hello from task 4 of 4 on n1032  
Hello from task 1 of 4 on n1034  
Hello from task 2 of 4 on n1032  
Hello from task 1 of 4 on n1033  
Hello from task 3 of 4 on n1034  
Hello from task 1 of 4 on n1035  
...
```

LOCALES AND EXECUTION MODEL IN CHAPEL

In Chapel, a locale refers to a compute resource with...

- processors, so it can run tasks
- memory, so it can store variables

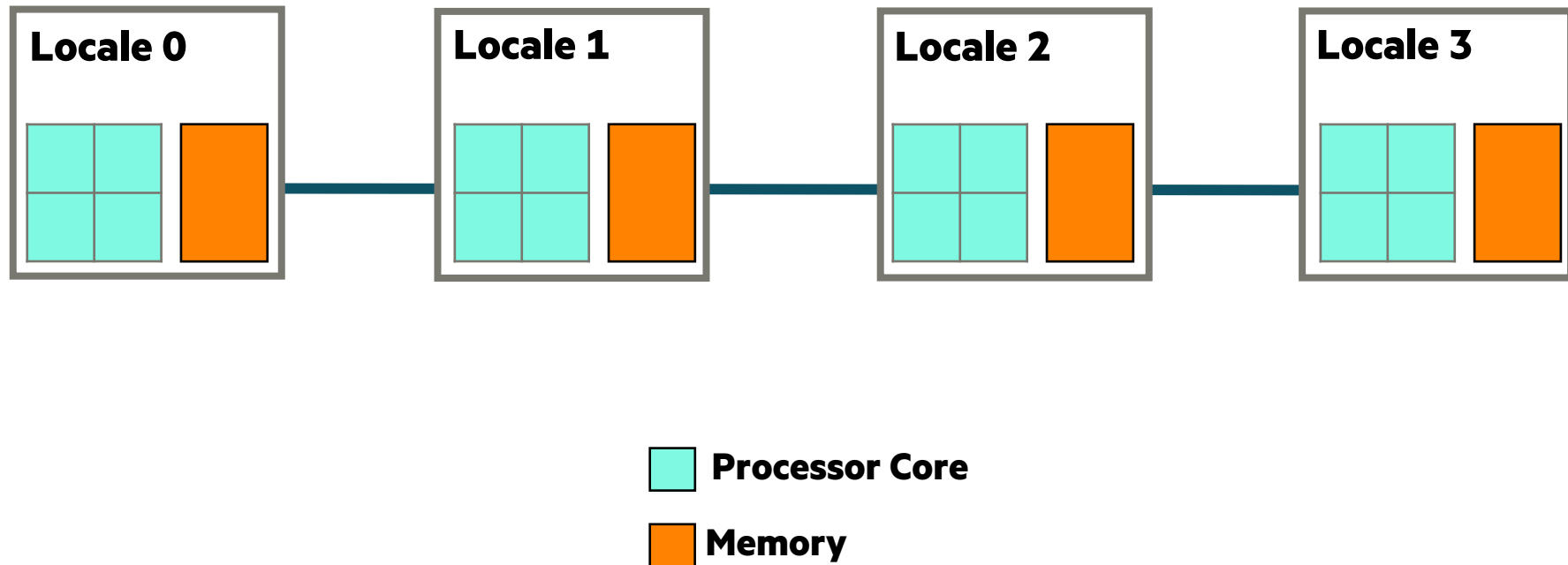
For now, think of each compute node as having one locale run on it



LOCALES AND EXECUTION MODEL IN CHAPEL

Two key built-in variables for referring to locales in Chapel programs:

- **Locales:** An array of locale values representing the system resources on which the program is running
- **here:** The locale on which the current task is executing



GETTING STARTED WITH LOCALES

- Users specify # of locales when running Chapel programs

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

```
% a.out -nl 8
```

- Chapel provides built-in locale variables

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

- User's `main()` begins executing on locale #0, i.e. 'Locales[0]'



LOCALE OPERATIONS

- Locale methods support queries about the target system:

```
proc locale.physicalMemory(...) { ... }  
proc locale.maxTaskPar { ... }  
proc locale.id { ... }  
proc locale.name { ... }
```

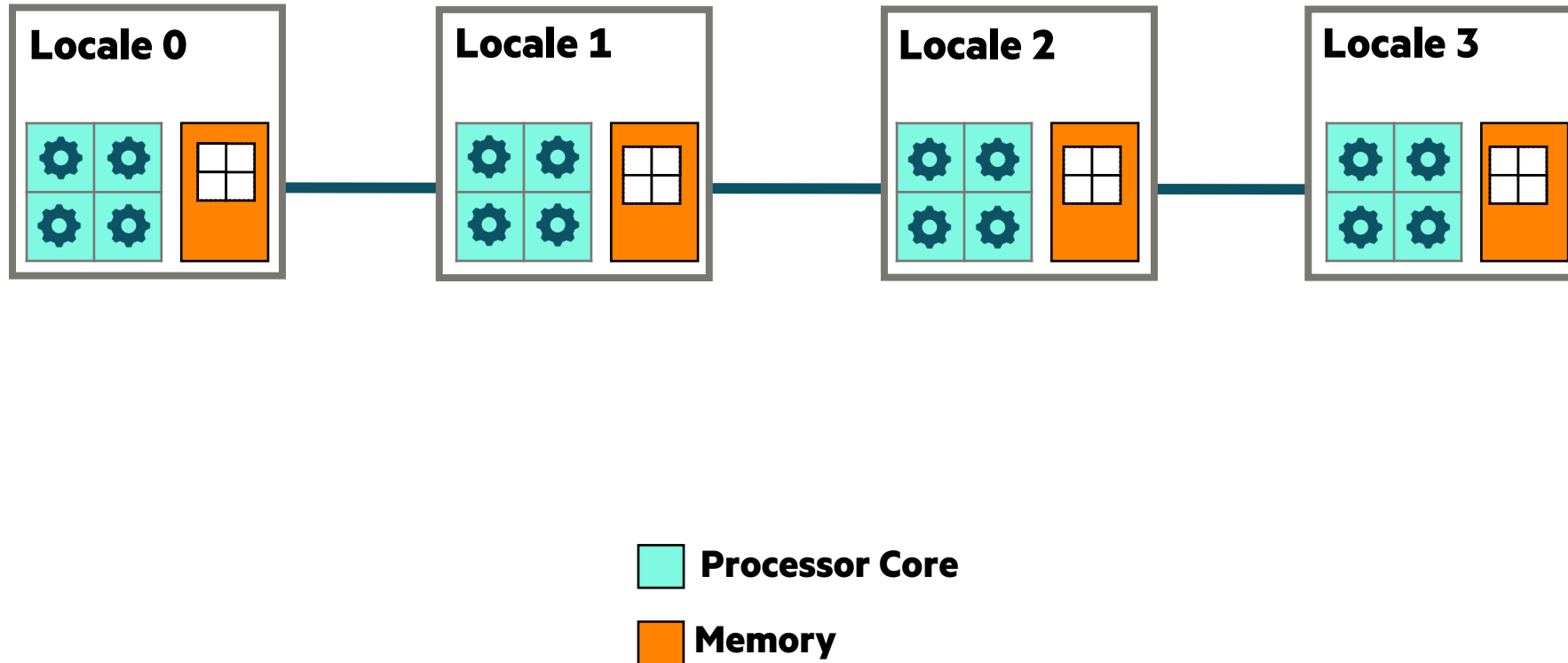
- *On-clauses* support placement of computations:

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

```
on A[i,j] do  
    bigComputation(A);  
  
on node.left do  
    search(node.left);
```


KEY CONCERNS FOR SCALABLE PARALLEL COMPUTING

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



BASIC FEATURES FOR LOCALITY



01-basics-on.chpl

01-basics-on.chpl

```
writeln("Hello from locale ", here.id);  
  
var A: [1..2, 1..2] real;  
  
on Locales[1] {  
    var B: [1..2, 1..2] real;  
  
    B = 2 * A;  
}
```

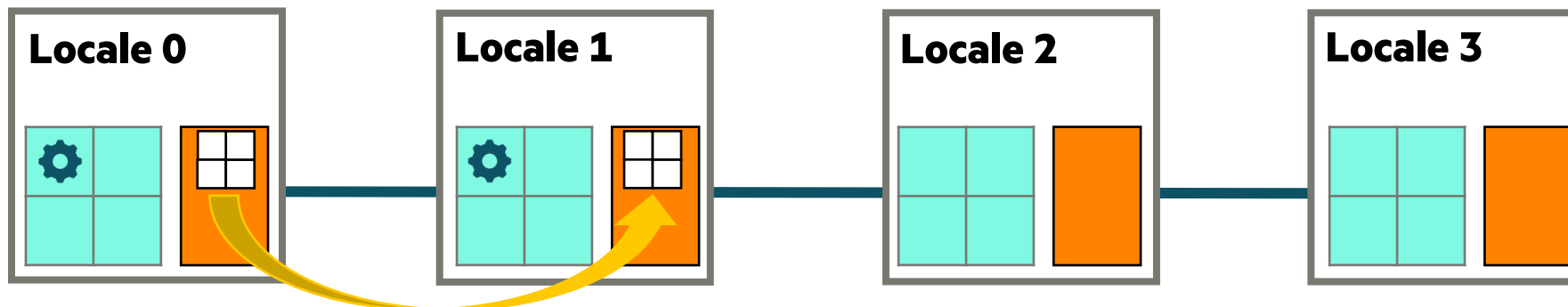
All Chapel programs begin running as a single task on locale 0

Variables are stored using the memory local to the current task

on-clauses move tasks to other locales

remote variables can be accessed directly

This is a serial, but distributed computation



BASIC FEATURES FOR LOCALITY



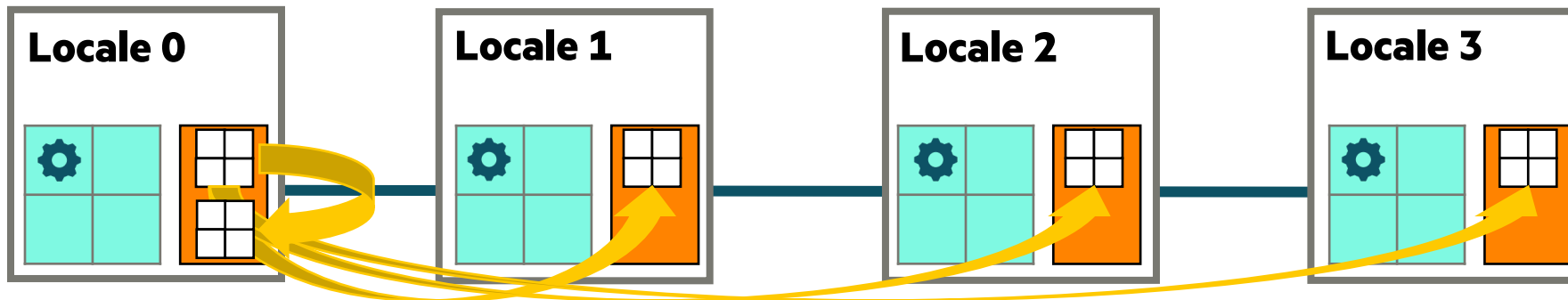
01-basics-for.chpl

01-basics-for.chpl

```
writeln("Hello from locale ", here.id);  
  
var A: [1..2, 1..2] real;  
  
for loc in Locales {  
  on loc {  
    var B = A;  
  }  
}
```

This loop will serially iterate over the program's locales

This is also a serial, but distributed computation



MIXING LOCALITY WITH TASK PARALLELISM



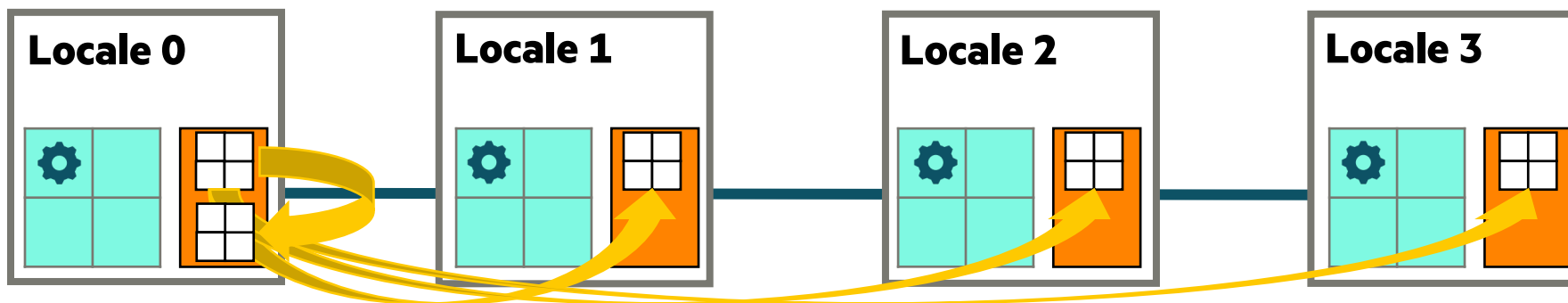
01-basics-coforall.chpl

01-basics-coforall.chpl

```
writeln("Hello from locale ", here.id);  
  
var A: [1..2, 1..2] real;  
  
coforall loc in Llocales {  
  on loc {  
    var B = A;  
  }  
}
```

The coforall loop creates
a parallel task per iteration

This results in a parallel distributed computation



ARRAY-BASED PARALLELISM AND LOCALITY



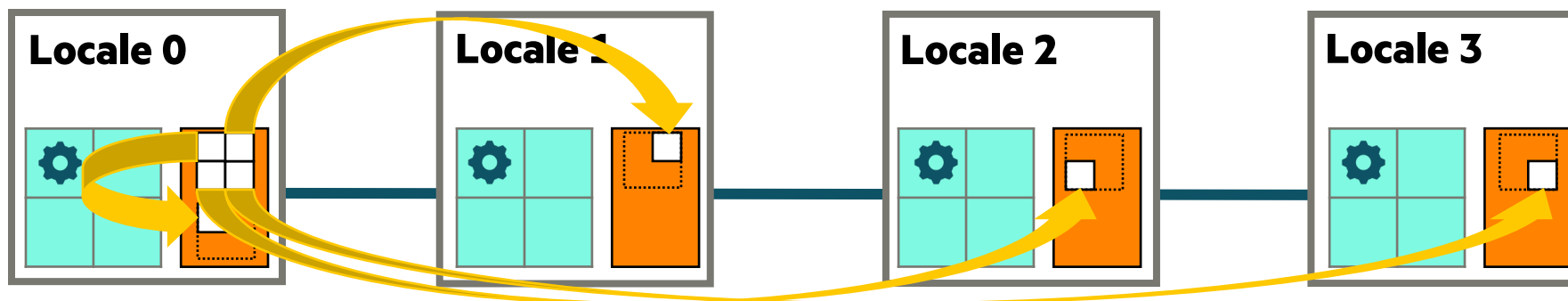
01-basics-distarr.chpl

01-basics-distarr.chpl

```
writeln("Hello from locale ", here.id);  
  
var A: [1..2, 1..2] real;  
  
use BlockDist;  
  
var D = blockDist.createDomain({1..2, 1..2});  
var B: [D] real;  
B = A;
```

Chapel also supports distributed domains (index sets) and arrays

They also result in parallel distributed computation



HANDS ON: PARALLELISM ACROSS AND WITHIN LOCALES

 01-hellopar.chpl

Parallel hello world

- 01-hellopar.chpl

Things to try

```
chpl 01-hellopar.chpl
./01-hellopar -nl 1 --tasksPerLocale=3
./01-hellopar -nl 2 --tasksPerLocale=3
```

Key concepts

- 'coforall' over the `Locales` array with an `on` statement
- 'coforall' creating some number of tasks per locale
- configuration constants, 'config const'
- range expression, '0..<tasksPerLocale'
- 'writeln'
- inline comments start with '//'

```
// can be set on the command line with --tasksPerLocale=2
config const tasksPerLocale = 1;

// parallel loops over nodes and then over threads
coforall loc in Locales do on loc {
    coforall tid in 0..<tasksPerLocale {

        writeln("Hello world! ",
                "(from task ", tid,
                " of ", tasksPerLocale,
                " on locale ", here.id,
                " of ", numLocales, ")");
    }
}
```

PARALLELISM AND LOCALITY ARE ORTHOGONAL IN CHAPEL



01-parallelism-and-locality.chpl

- 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);
```

HANDS ON: PARALLELISM AND LOCALITY IN CHAPEL

Goals

- Compile and run some of the examples from the last section
- Experiment some with '01-basics-distarr.chpl'

Compile and run some of the other examples from the last section

```
chpl 01-parallelism-and-locality.chpl  
./01-parallelism-and-locality -nl 1  
./01-parallelism-and-locality -nl 4
```

Experiment some with '01-basics-distarr.chpl'

1. what happens when you add a 'writeln(D)' to write out the domain 'D'?
2. what happens when you change 'D's initial value to '{0..3,0..3}'?
3. where does the computation on locales other than locale 0 happen?



OUTLINE: OVERVIEW OF PROGRAMMING IN CHAPEL

- Chapel Goals, Usage, and Comparison with other Tools
- Hello World (Hands On)
- Chapel Execution Model and Parallel Hello World (Hands On)
- kmer counting using file IO, config consts, strings, maps (Hands On)
- Parallelizing a program that processes files (Hands On)
- GPU programming support
- Learning goals for rest of tutorial



KMER COUNTING USING FILE IO, CONFIG CONSTS, AND STRINGS (HANDS ON)

SERIAL CODE USING MAP/Dictionary: K-MER COUNTING



kmer.chpl

kmer.chpl

```
use Map, IO;

config const infilename = "kmer_large_input.txt";
config const k = 4;

var sequence, line : string;
var f = open(infilename, ioMode.r);
var infile = f.reader();
while infile.readLine(line) {
    sequence += line.strip();
}

var nkmerCounts : map(string, int);

for ind in 0..<(sequence.size-k) {
    nkmerCounts[sequence[ind..#k]] += 1;
}
```

'Map' and 'IO' are two of the standard libraries provided in Chapel. A 'map' is like a dictionary in python.

'config const' indicates a configuration constant, which result in built-in command-line parsing

Reading all of the lines from the input file into the string 'sequence'.

The variable 'nkmerCounts' is being declared as a dictionary mapping strings to ints

Counting up each kmer in the sequence

HANDS ON: EXPERIMENTING WITH THE K-MER EXAMPLE



kmer.chpl

Some things to try out with 'kmer.chpl'

```
chpl kmer.chpl
./kmer -nl 1

./kmer -nl 1 --k=10 # can change k
./kmer -nl 1 --infilename="kmer.chpl" # changing infilename
./kmer -nl 1 --k=10 --infilename="kmer.chpl" # can change both
```

Key concepts

- 'use' command for including modules
- configuration constants, 'config const'
- reading from a file
- 'map' data structure



PARALLELIZING A PROGRAM THAT PROCESSES FILES (HANDS ON)

ANALYZING MULTIPLE FILES USING PARALLELISM

 parfilekmer.chpl

parfilekmer.chpl

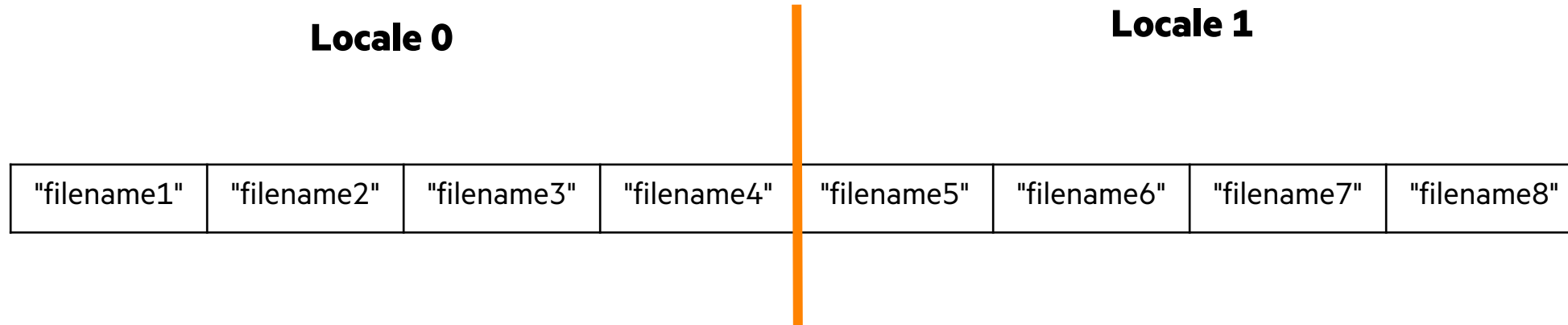
```
use FileSystem, BlockDist;
config const dir = "DataDir";
var fList = findFiles(dir);
var filenames =
    blockDist.createArray(0..<fList.size, string);
filenames = fList;

// per file word count
forall f in filenames {
    ...
    // code from kmer.chpl
    ...
}
```

```
prompt> chpl --fast parfilekmer.chpl
prompt> ./parfilekmer -nl 1
prompt> ./parfilekmer -nl 4
```

- shared and distributed-memory parallelism using 'forall'
 - in other words, parallelism within the locale/node and across locales/nodes
- a distributed array
- command line options to indicate number of locales

BLOCK DISTRIBUTION OF ARRAY OF STRINGS



- Array of strings for filenames is distributed across locales
- 'forall' will do parallelism across locales and then within each locale to take advantage of multicore



HANDS ON: PROCESSING FILES IN PARALLEL

 parfilekmer.chpl

Some things to try out with 'parfilekmer.chpl'

```
chpl parfilekmer.chpl --fast
./parfilekmer -nl 2 --dir="SomethingElse/"      # change dir with inputs files

./parfilekmer -nl 2 --k=10                       # can also change k
```

Concepts illustrated

- 'forall' provides distributed and shared memory parallelism when do a 'forall' over the Block distributed array
- No remote puts and gets



GPU PROGRAMMING SUPPORT

GPU SUPPORT IN CHAPEL

Generate code for GPUs

- Support for NVIDIA and AMD GPUs
- Exploring Intel support

Key concepts

- Using the 'locale' concept to indicate execution and data allocation on GPUs
- 'forall' and 'foreach' loops are converted to kernels
- Arrays declared within GPU sublocale code blocks are allocated on the GPU

Chapel code calling CUDA examples

- <https://github.com/chapel-lang/chapel/blob/main/test/gpu/interop/stream/streamChpl.chpl>
- <https://github.com/chapel-lang/chapel/blob/main/test/gpu/interop/cuBLAS/cuBLAS.chpl>

For more info...

- <https://chapel-lang.org/docs/technotes/gpu.html>

gpuExample.chpl

```
use GpuDiagnostics;
startGpuDiagnostics();

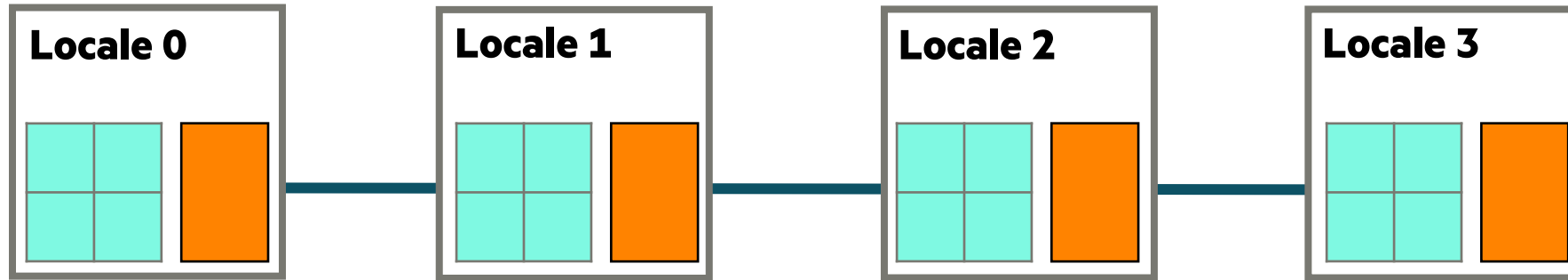
var operateOn =
if here.gpus.size>0 then here.gpus
    else [here,];

// Same code can run on GPU or CPU
coforall loc in operateOn do on loc {
    var A : [1..10] int;
    foreach a in A do a+=1;
    writeln(A);
}

stopGpuDiagnostics();
writeln(getGpuDiagnostics());
```

KEY CONCERNS FOR SCALABLE PARALLEL COMPUTING

1. **parallelism:** Which tasks should run simultaneously?
2. **locality:** Where should tasks run? Where should data be allocated?
 - complicating matters, compute nodes now often have GPUs with their own processors and memory

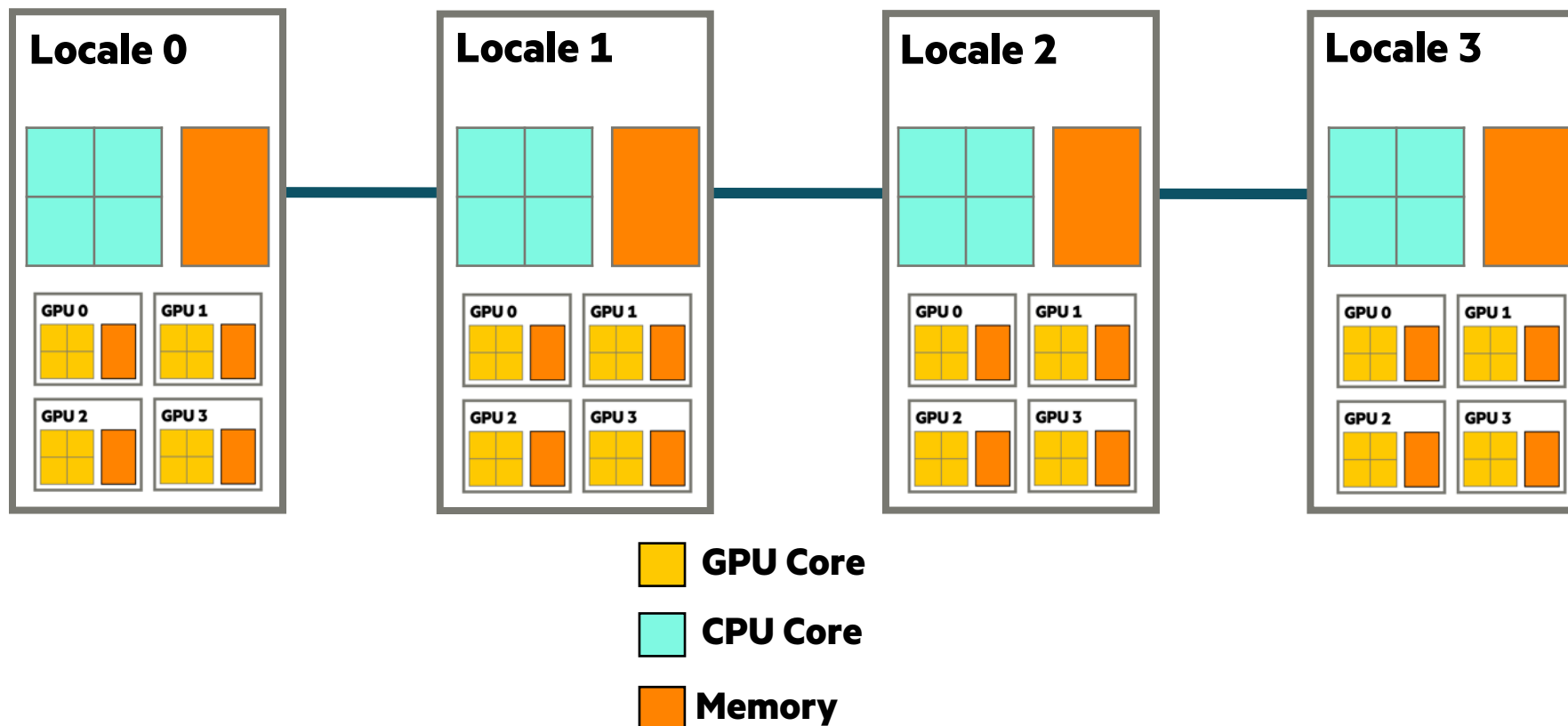


KEY CONCERNS FOR SCALABLE PARALLEL COMPUTING

1. parallelism: Which tasks should run simultaneously?

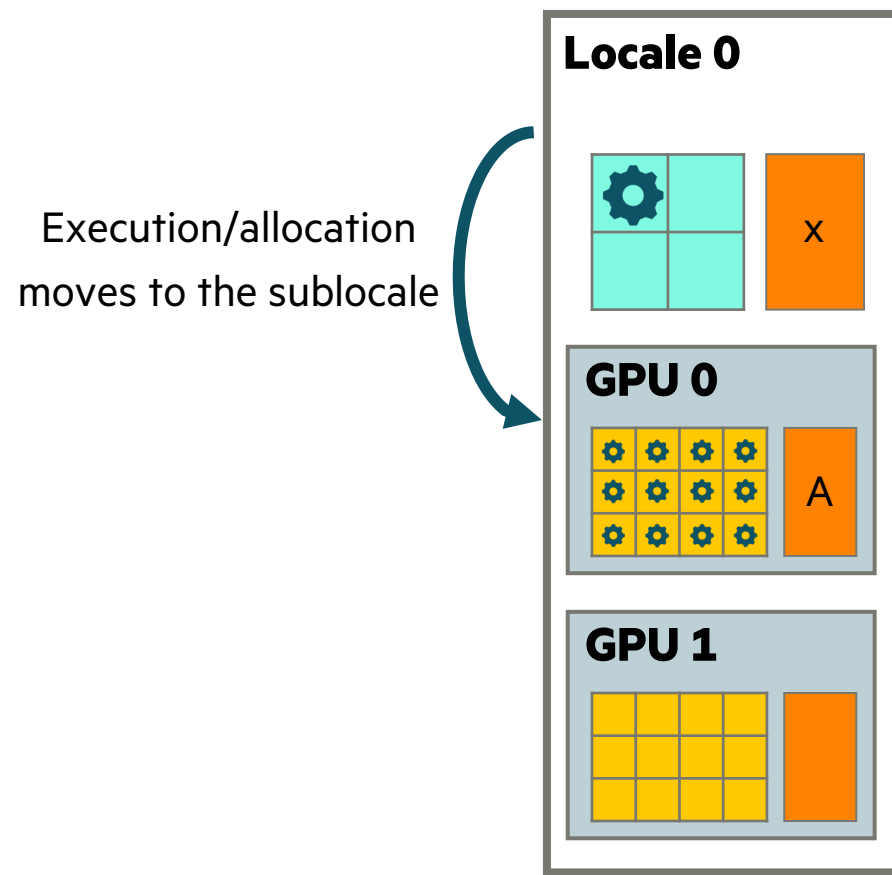
2. locality: Where should tasks run? Where should data be allocated?


- complicating matters, compute nodes now often have GPUs with their own processors and memory
- we represent these as *sub-locales* in Chapel






PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS

 CPU Core  GPU Core  Memory



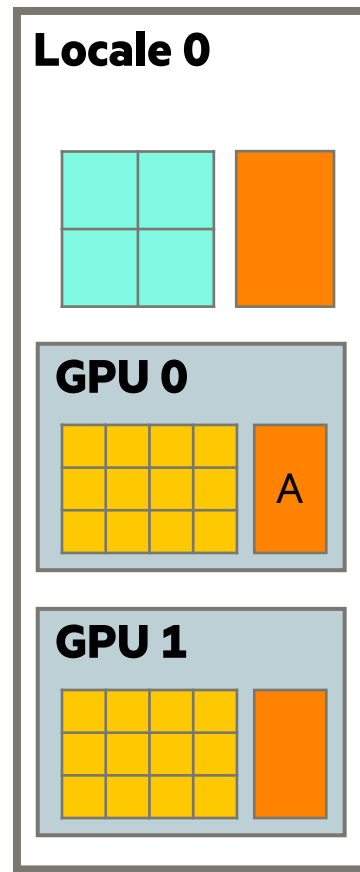
 `var x = 10;`



 `on here.gpus[0] {`
 `var A = [1, 2, 3, 4, 5, ...];`
 `foreach a in A do a += 1;`
}

 `writeln(x);`

PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS

 CPU Core  GPU Core  Memory



```
var x = 10;
```

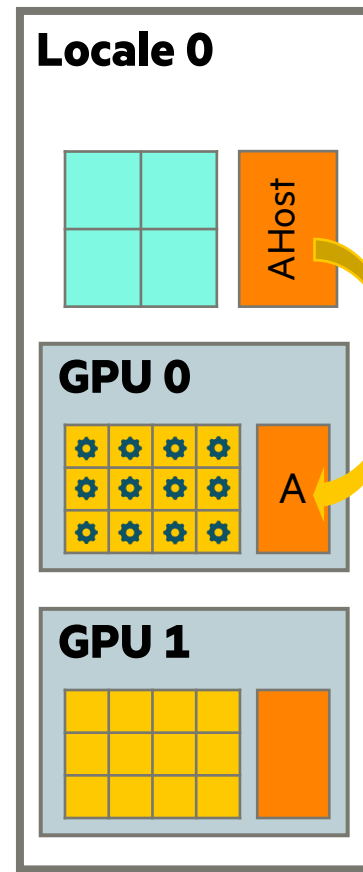


```
on here.gpus[0] {  
  var A = [1, 2, 3, 4, 5, ...];  
  foreach a in A do a += 1;  
}
```

```
writeln(x);
```

PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS

 CPU Core  GPU Core  Memory



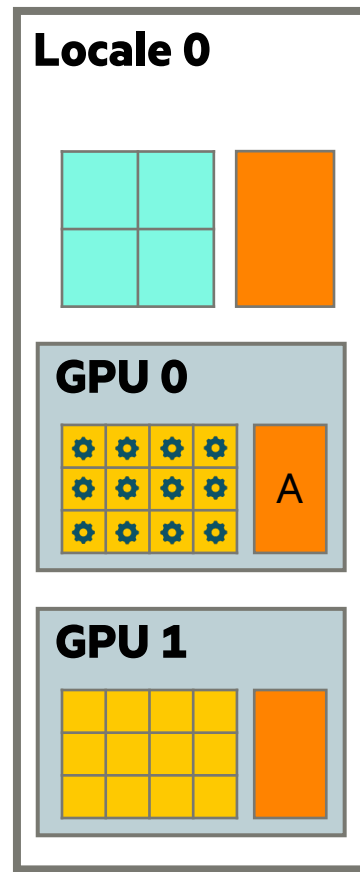
```
var x = 10;  
var AHost = [1, 2, 3, 4, 5, ...];
```

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

```
writeln(x);
```

PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS

 CPU Core  GPU Core  Memory



```
var x = 10;
```

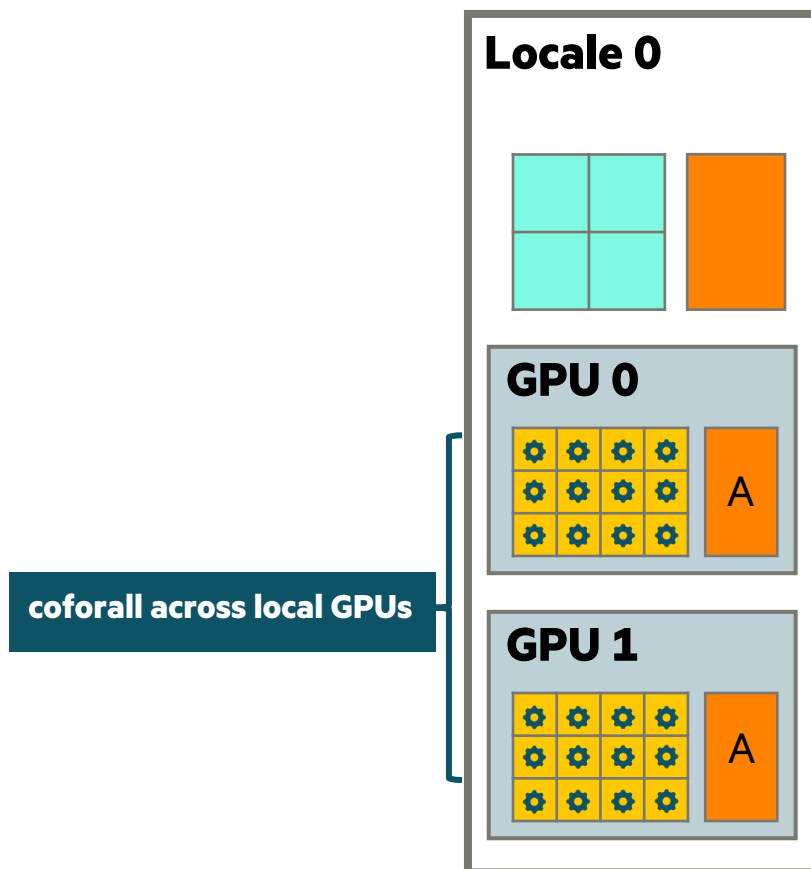


```
on here.gpus[0] {  
  var A = [1, 2, 3, 4, 5, ...];  
  foreach a in A do a += 1;  
}
```

```
writeln(x);
```


PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS

 CPU Core  GPU Core  Memory



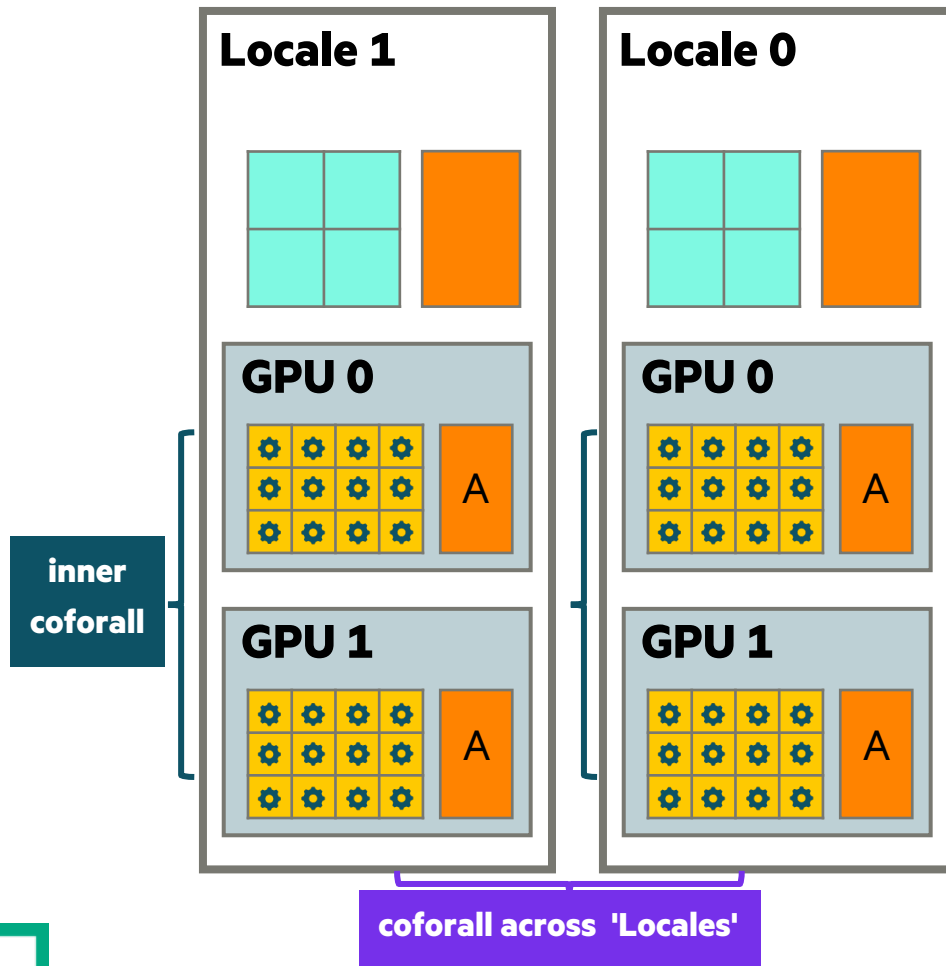
```
var x = 10;
```

```
coforall g in here.gpus do on g {  
  var A = [1, 2, 3, 4, 5, ...];  
  foreach a in A do a += 1;  
}
```

```
writeln(x);
```

PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS

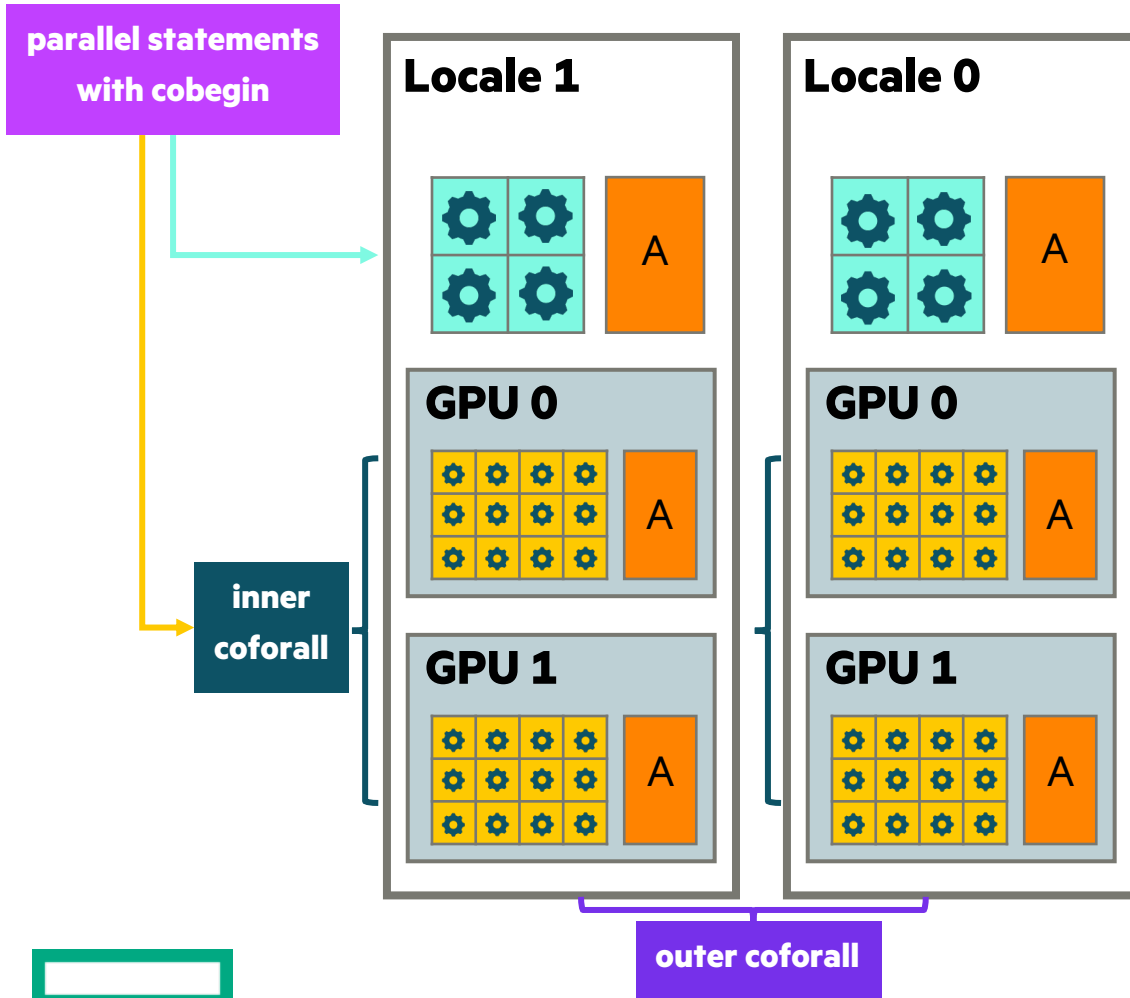
 CPU Core  GPU Core  Memory



```
var x = 10;  
coforall l in Locales do on l {  
  
  coforall g in here.gpus do on g {  
    var A = [1, 2, 3, 4, 5, ...];  
    foreach a in A do a += 1;  
  }  
  
  writeln(x);  
}
```

PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS

 CPU Core  GPU Core  Memory



```
var x = 10;  
coforall l in Locales do on l {  
  cobegin {  
    coforall g in here.gpus do on g {  
      var A = [1, 2, 3, 4, 5, ...];  
      foreach a in A do a += 1;  
    }  
    {  
      var A = [1, 2, 3, 4, 5, ...];  
      foreach a in A do a += 1;  
    }  
  }  
}  
writeln(x);
```

LEARNING OBJECTIVES FOR THE REST OF THE TUTORIAL

LEARNING OBJECTIVES FOR TODAY'S CHAPEL TUTORIAL

- Familiarity with the Chapel execution model including how to run codes in parallel on a single node, across nodes, and both
- Learn Chapel concepts by compiling and running provided code examples
 - ✓ Serial code using map/dictionary, (k-mer counting from bioinformatics)
 - ✓ Parallelism and locality in Chapel
 - ✓ Distributed parallelism and 1D arrays, (processing files in parallel)
- Chapel basics in the context of an n-body code
- Distributed parallelism and 2D arrays, (heat diffusion problem)
- How to parallelize histogram
- Using CommDiagnostics for counting remote reads and writes
- Chapel and Arkouda best practices including avoiding races and performance gotchas
- Where to get help and how you can participate in the Chapel community



OTHER CHAPEL EXAMPLES & PRESENTATIONS

Primers

- <https://chapel-lang.org/docs/primers/index.html>

Blog posts for Advent of Code

- <https://chapel-lang.org/blog/index.html>

Test directory in main repository

- <https://github.com/chapel-lang/chapel/tree/main/test>

Presentations

- <https://chapel-lang.org/presentations.html>



ONE DAY CHAPEL TUTORIAL

- 9-10:30: Getting started using Chapel for parallel programming
- 10:30-10:45: break
- 10:45-12:15: Chapel basics in the context of the n-body example code
- 12:15-1:15: lunch
- 1:15-2:45: Distributed and shared-memory parallelism especially w/arrays (data parallelism)
- 2:45-3:00: break
- 3:00-4:30: More parallelism including for asynchronous parallelism (task parallelism)
- 4:30-5:00: Wrap-up including gathering further questions from attendees



CHAPEL RESOURCES

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

- (points to all other resources)

Social Media:

- Twitter: [@ChapelLanguage](https://twitter.com/ChapelLanguage)
- Facebook: [@ChapelLanguage](https://facebook.com/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>



The Chapel Parallel Programming Language

What is Chapel?

Chapel is a programming language designed for productive parallel computing at scale.

Why Chapel? Because it simplifies parallel programming through elegant support for:

- **distributed arrays** that can leverage thousands of nodes' memories and cores
- **a global namespace** supporting direct access to local or remote variables
- **data parallelism** to trivially use the cores of a laptop, cluster, or supercomputer
- **task parallelism** to create concurrency within a node or across the system

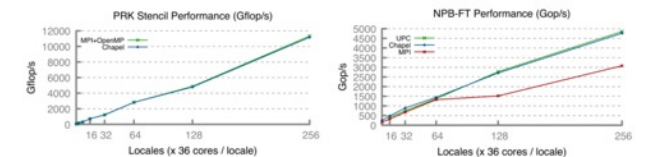
Chapel Characteristics

- **productive:** code tends to be similarly readable/writable as Python
- **scalable:** runs on laptops, clusters, the cloud, and HPC systems
- **fast:** performance *competes with or beats* C/C++ & MPI & OpenMP
- **portable:** compiles and runs in virtually any *nix environment
- **open-source:** hosted on GitHub, permissively *licensed*

New to Chapel?

As an introduction to Chapel, you may want to...

- watch an [overview talk](#) or browse its [slides](#)
- read a [blog-length](#) or [chapter-length](#) introduction to Chapel
- learn about [projects powered by Chapel](#)
- check out [performance highlights](#) like these:



- browse [sample programs](#) or learn how to write distributed programs like this one:

```
use CyclicDist;           // use the Cyclic distribution library
config const n = 100;     // use --n=<val> when executing to override this default

forall i in {1..n} dmapped Cyclic(startIdx=1) do
  writeln("Hello from iteration ", i, " of ", n, " running on node ", here.id);
```




Hewlett Packard
Enterprise

ONE-DAY CHAPEL TUTORIAL

SESSION 2: CHAPEL BASICS



Chapel Team
October 16, 2023

ONE DAY CHAPEL TUTORIAL

- 9-10:30: Getting started using Chapel for parallel programming
- 10:30-10:45: break
- 10:45-12:15: Chapel basics in the context of the n-body example code
- 12:15-1:15: lunch
- 1:15-2:45: Distributed and shared-memory parallelism especially w/arrays (data parallelism)
- 2:45-3:00: break
- 3:00-4:30: More parallelism including for asynchronous parallelism (task parallelism)
- 4:30-5:00: Wrap-up including gathering further questions from attendees



OUTLINE: CHAPEL BASICS

- Running Example: n-body computation (Hands On)
- Variables, Constants, and Operators
- Records and Classes
- Tuples
- Arrays
- Writing out Tuples, Records, and Arrays (Hands On)
- Main() Procedure
- Ranges and basic control flow
- Procedures and iterators
- Where might we parallelize the n-body computation? (Hands On)



RUNNING EXAMPLE: N-BODY COMPUTATION (HANDS ON)

N-BODY IN CHAPEL (WHERE N == 5)

- A serial computation
- From the Computer Language Benchmarks Game
 - Chapel implementation in release under `examples/benchmarks/shootout/nbody.chpl`
- Computes the influence of 5 bodies on one another
 - The Sun, Jupiter, Saturn, Uranus, Neptune
- Executes for a user-specifiable number of timesteps

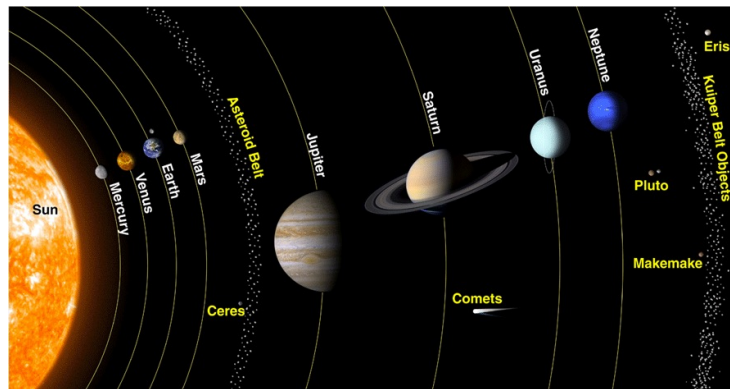


Image source: <http://spaceplace.nasa.gov/review/ice-dwarf/solar-system-lrg.png>

HANDS ON: COMPILING AND RUNNING N-BODY



nbody.chpl

Things to try

```
chpl nbody.chpl
time ./nbody -nl 1
time ./nbody -nl 1 -n=100000

chpl --fast nbody.chpl
time ./nbody -nl 1
time ./nbody -nl 1 -n=100000
```

```
// number of timesteps to simulate
config const n = 10000;
...
```

Key concepts

- *nix 'time' command is an easy way to see how long a program takes to run
- Compile with '--fast' to have 'chpl' compiler generate faster code



VARIABLES, CONSTANTS, AND OPERATORS

5-BODY IN CHAPEL: VARIABLE AND RECORD DECLARATIONS



nbody.chpl

```
const pi = 3.141592653589793,  
      solarMass = 4 * pi**2,  
      daysPerYear = 365.24;
```

Variable declarations

```
config const numsteps = 10000;
```

```
record body {  
  var pos: 3*real;  
  var v: 3*real;  
  var mass: real;  
}
```

```
...
```



VARIABLES, CONSTANTS, AND PARAMETERS

Basic syntax

```
declaration:  
  var    identifier [: type] [= init-expr];  
  const  identifier [: type] [= init-expr];  
  param  identifier [: type] [= init-expr];
```

Examples

```
const pi: real = 3.14159;  
var count: int;           // initialized to 0  
param debug = true;      // inferred to be bool
```

Meaning

- var/const: execution-time variable/constant
- param: compile-time constant
- No init-expr \Rightarrow initial value is the type's default
- No type \Rightarrow type is taken from init-expr



PRIMITIVE TYPES

Syntax

Type	Description	Default Value	Currently-Supported Bit Widths	Default Bit Width
bool	logical value	false		impl. dep.
int	signed integer	0	8, 16, 32, 64	64
uint	unsigned integer	0	8, 16, 32, 64	64
real	real floating point	0.0	32, 64	64
imag	imaginary floating point	0.0i	32, 64	64
complex	complex floating points	0.0 + 0.0i	64, 128	128
string	character string	""	N/A	N/A

Examples

```
primitive-type:  
  type-name [( bit-width )]
```

```
int(16)    // 16-bit int  
real(32)   // 32-bit real  
uint      // 64-bit uint
```



CHAPEL'S STATIC TYPE INFERENCE



nbody.chpl

```
const pi = 3.14,           // pi is a real
      coord = 1.2 + 3.4i,  // coord is a complex...
      coord2 = pi*coord,   // ...as is coord2
      name = "brad",       // name is a string
      verbose = false;     // verbose is boolean

proc addem(x, y) {         // addem() has generic arguments
  return x + y;            // and an inferred return type
}

var sum = addem(1, pi),     // sum is a real
    fullname = addem(name, "ford"); // fullname is a string

writeln((sum, fullname));
```

(4.14, bradford)

BASIC OPERATORS AND PRECEDENCE

Operator	Description	Associativity	Overloadable
:	cast	left	yes
**	exponentiation	right	yes
! ~	logical and bitwise negation	right	yes
* / %	multiplication, division and modulus	left	yes
(unary) + -	positive identity and negation	right	yes
<< >>	shift left and shift right	left	yes
&	bitwise/logical and	left	yes
^	bitwise/logical xor	left	yes
	bitwise/logical or	left	yes
+ -	addition and subtraction	left	yes
<= >= < >	ordered comparison	left	yes
== !=	equality comparison	left	yes
&&	short-circuiting logical and	left	via isTrue
	short-circuiting logical or	left	via isTrue

5-BODY IN CHAPEL: DECLARATIONS



nbody.chpl

```
const pi = 3.141592653589793,  
      solarMass = 4 * pi**2,  
      daysPerYear = 365.24;
```

Variable declarations

```
config const numsteps = 10000;
```

```
record body {  
  var pos: 3*real;  
  var v: 3*real;  
  var mass: real;  
}
```

```
...
```

5-BODY IN CHAPEL: DECLARATIONS



nbody.chpl

```
const pi = 3.141592653589793,  
      solarMass = 4 * pi**2,  
      daysPerYear = 365.24;
```

```
config const numsteps = 10000;
```

```
record body {  
  var pos: 3*real;  
  var v: 3*real;  
  var mass: real;  
}
```

```
...
```

Configuration Variable

5-BODY IN CHAPEL: DECLARATIONS



nbody.chpl

```
const pi = 3.141592653589793,  
      solarMass = 4 * pi**2,  
      daysPerYear = 365.24;
```

```
config const numsteps = 10000;
```

```
record body {  
  var pos: 3*real;  
  var v: 3*real;  
  var mass: real;  
}
```

```
...
```

Configuration Variable

```
$ ./nbody --numsteps=100
```

CONFIGS

 02-configs.chpl

```
param intSize = 32;  
type elementType = real(32);  
const epsilon = 0.01:elementType;  
var start = 1:int(intSize);
```


CONFIGS

 02-configs.chpl

```
config param intSize = 32;  
config type elementType = real(32);  
config const epsilon = 0.01:elementType;  
config var start = 1:int(intSize);
```

```
$ chpl 02-configs.chpl -sintSize=64 -selementType=real  
$ ./02-configs-start=2 -nl 1 --epsilon=0.00001
```

5-BODY IN CHAPEL: DECLARATIONS



nbody.chpl

```
const pi = 3.141592653589793,  
      solarMass = 4 * pi**2,  
      daysPerYear = 365.24;
```

```
config const numsteps = 10000;
```

```
record body {  
  var pos: 3*real;  
  var v: 3*real;  
  var mass: real;  
}
```

```
...
```

Configuration Variable

5-BODY IN CHAPEL: DECLARATIONS



nbody.chpl

```
const pi = 3.141592653589793,  
      solarMass = 4 * pi**2,  
      daysPerYear = 365.24;  
  
config const numsteps = 10000;  
  
record body {  
  var pos: 3*real;  
  var v: 3*real;  
  var mass: real;  
}  
  
...
```

Record declaration

RECORDS AND CLASSES

RECORDS AND CLASSES

 02-records-and-classes.chpl

Chapel's object types

- Contain variable definitions (fields)
- Contain procedure & iterator definitions (methods)
- Records: value-based (e.g., assignment copies fields)
- Classes: reference-based (e.g., assignment aliases object)

Example

```
use Math;
record circle {
  var radius: real;
  proc area() {
    return pi*radius**2;
  }
}
```

```
var c1 = new circle(radius=1.0);
var c2 = c1;    // copies c1
c1.radius = 5.0;
writeln(c2.radius); // prints 1.0
```

RECORDS AND CLASSES

 02-records-and-classes.chpl

Chapel's object types

- Contain variable definitions (fields)
- Contain procedure & iterator definitions (methods)
- Records: value-based (e.g., assignment copies fields)
- Classes: reference-based (e.g., assignment aliases object)

Example

```
use Math;
class Circle {
  var radius: real;
  proc area() {
    return pi*radius**2;
  }
}
```

```
// c1 is a nilable class
var c1: Circle? = new shared Circle(radius=1.0);
var c2 = c1;           // aliases c1's circle
c1!.radius = 5.0;
writeln(c2!.radius); // prints 5.0
```

CLASSES VS. RECORDS

Classes

- heap-allocated
 - Variables point to objects
 - Support mem. mgmt. policies
- 'reference' semantics
 - compiler will only copy pointers
- support inheritance
- support dynamic dispatch
- identity matters most
- similar to Java classes

Records

- allocated in-place
 - Variables are the objects
 - Always freed at end of scope
- 'value' semantics
 - compiler may introduce copies
- no inheritance
- no dynamic dispatch
- value matters most
- similar to C++ structs
 - (sans pointers)



5-BODY IN CHAPEL: DECLARATIONS



nbody.chpl

```
const pi = 3.141592653589793,  
      solarMass = 4 * pi**2,  
      daysPerYear = 365.24;  
  
config const numsteps = 10000;  
  
record body {  
  var pos: 3*real;  
  var v: 3*real;  
  var mass: real;  
}  
  
...
```

Tuple type

OUTLINE: CHAPEL BASICS

- Running Example: n-body computation (Hands On)
- Variables, Constants, and Operators
- Records and Classes
- Tuples
- Arrays
- Writing out Tuples, Records, and Arrays (Hands On)
- Main() Procedure
- Ranges and basic control flow
- Procedures and iterators
- Where might we parallelize the n-body computation? (Hands On)



TUPLES (HANDS ON)

TUPLES

Use

- support lightweight grouping of values
 - e.g., passing/returning multiple procedure arguments at once
 - short vectors
 - multidimensional array indices
- support heterogeneous data types

Examples

```
var coord: (int, int, int) = (1, 2, 3);  
var coordCopy: 3*int = coord;  
var (i1, i2, i3) = coord;  
var triple: (int, string, real) = (7, "eight", 9.0);
```

5-BODY IN CHAPEL: DECLARATIONS



nbody.chpl

```
const pi = 3.141592653589793,  
      solarMass = 4 * pi**2,  
      daysPerYear = 365.24;
```

Variable declarations

```
config const numsteps = 10000;
```

Configuration Variable

```
record body {  
  var pos: 3*real;  
  var v: 3*real;  
  var mass: real;  
}
```

Record declaration

...

Tuple type

5-BODY IN CHAPEL: THE BODIES



nbody.chpl

```
var bodies =
  [ /* sun */
    new body(mass = solarMass),

    /* jupiter */
    new body(pos = ( 4.84143144246472090e+00,
                    -1.16032004402742839e+00,
                    -1.03622044471123109e-01),
              v = ( 1.66007664274403694e-03 * daysPerYear,
                    7.69901118419740425e-03 * daysPerYear,
                    -6.90460016972063023e-05 * daysPerYear),
              mass = 9.54791938424326609e-04 * solarMass),

    /* saturn */
    new body(...),

    /* uranus */
    new body(...),

    /* neptune */
    new body(...)
  ];
```

5-BODY IN CHAPEL: THE BODIES



nbody.chpl

```
var bodies =  
[ /* sun */  
  new body(mass = solarMass),  
  
  /* jupiter */  
  new body(pos = ( 4.84143144246472090e+00,  
                  -1.16032004402742839e+00,  
                  -1.03622044471123109e-01),  
            v = ( 1.66007664274403694e-03 * daysPerYear,  
                  7.69901118419740425e-03 * daysPerYear,  
                  -6.90460016972063023e-05 * daysPerYear),  
            mass = 9.54791938424326609e-04 * solarMass),  
  
  /* saturn */  
  new body(...),  
  
  /* uranus */  
  new body(...),  
  
  /* neptune */  
  new body(...)  
];
```

Create a record object

5-BODY IN CHAPEL: THE BODIES



nbody.chpl

```
var bodies =  
[ /* sun */  
  new body(mass = solarMass),  
  
  /* jupiter */  
  new body(pos = ( 4.84143144246472090e+00,  
                  -1.16032004402742839e+00,  
                  -1.03622044471123109e-01),  
            v = ( 1.66007664274403694e-03 * daysPerYear,  
                  7.69901118419740425e-03 * daysPerYear,  
                  -6.90460016972063023e-05 * daysPerYear),  
            mass = 9.54791938424326609e-04 * solarMass),  
  
  /* saturn */  
  new body(...),  
  
  /* uranus */  
  new body(...),  
  
  /* neptune */  
  new body(...)  
];
```

Tuple values



5-BODY IN CHAPEL: THE BODIES



nbody.chpl

```
var bodies =  
  [ /* sun */  
    new body(mass = solarMass),  
  
    /* jupiter */  
    new body(pos = ( 4.84143144246472090e+00,  
                    -1.16032004402742839e+00,  
                    -1.03622044471123109e-01),  
              v = ( 1.66007664274403694e-03 * daysPerYear,  
                    7.69901118419740425e-03 * daysPerYear,  
                    -6.90460016972063023e-05 * daysPerYear),  
              mass = 9.54791938424326609e-04 * solarMass),  
  
    /* saturn */  
    new body(...),  
  
    /* uranus */  
    new body(...),  
  
    /* neptune */  
    new body(...)  
  ];
```

Array
value

ARRAYS

ARRAY TYPES



02-array-examples.chpl

Syntax

```
array-type:  
  [ domain-expr ] elt-type  
array-value:  
  [elt1, elt2, elt3, ... eltn]
```

Meaning

- array-type: stores an element of elt-type for each index
- array-value: represent the array with these values

Examples

```
var A: [1..3] int,           // A stores 0, 0, 0  
    B = [5, 3, 9],          // B stores 5, 3, 9  
    C: [1..m, 1..n] real,    // 2D m by n array of reals  
    D: [1..m][1..n] real;    // array of arrays of reals
```

More on arrays in data parallelism section later...

5-BODY IN CHAPEL: THE BODIES



nbody.chpl

```
var bodies =  
[ /* sun */  
  new body(mass = solarMass),  
  
  /* jupiter */  
  new body(pos = ( 4.84143144246472090e+00,  
                  -1.16032004402742839e+00,  
                  -1.03622044471123109e-01),  
            v = ( 1.66007664274403694e-03 * daysPerYear,  
                  7.69901118419740425e-03 * daysPerYear,  
                  -6.90460016972063023e-05 * daysPerYear),  
            mass = 9.54791938424326609e-04 * solarMass),  
  
  /* saturn */  
  new body(...),  
  
  /* uranus */  
  new body(...),  
  
  /* neptune */  
  new body(...)  
];
```

Create a record object

Tuple values

Array
value

HANDS ON: WRITING TUPLES, RECORDS, AND ARRAYS



nbody.chpl

Put a 'writeln("bodies = ", bodies);' into program

```
chpl nbody.chpl
./nbody -nl 1
bodies =(pos = (0.0, 0.0, 0.0), vel = (0.0, 0.0, 0.0),
mass = 39.4784) (pos = (4.84143, -1.16032, -0.103622), vel
= (0.606326, 2.81199, -0.0252184), mass = 0.0376937) (pos
= (8.34337, 4.1248, -0.403523), vel = (-1.01077, 1.82566,
0.00841576), mass = 0.0112863) (pos = (12.8944, -15.1112,
-0.223308), vel = (1.08279, 0.868713, -0.0108326), mass =
0.00172372) (pos = (15.3797, -25.9193, 0.179259), vel =
(0.979091, 0.594699, -0.034756), mass = 0.00203369)
-0.169075164
-0.169016441
```

MAIN() PROCEDURE

5-BODY IN CHAPEL: MAIN()



nbody.chpl

...

```
proc main() {  
    initSun();  
  
    writef("%.9r\n", energy());  
    for 1..numsteps do  
        advance(0.01);  
    writef("%.9r\n", energy());  
}
```

...

5-BODY IN CHAPEL: MAIN()



nbody.chpl

...

```
proc main() {  
  initSun();  
  
  writef("%.9r\n", energy());  
  for 1..numsteps do  
    advance(0.01);  
  writef("%.9r\n", energy());  
}
```

...

Procedure Definition

5-BODY IN CHAPEL: MAIN()



nbody.chpl

...

```
proc main() {  
  initSun();  
  
  writef("%.9r\n", energy());  
  for 1..numsteps do  
    advance(0.01);  
  writef("%.9r\n", energy());  
}
```

...

Procedure Call

5-BODY IN CHAPEL: MAIN()



nbody.chpl

```
...  
  
proc main() {  
  initSun();  
  
  writef("%.9r\n", energy());  
  for 1..numsteps do  
    advance(0.01);  
  writef("%.9r\n", energy());  
}  
  
...
```

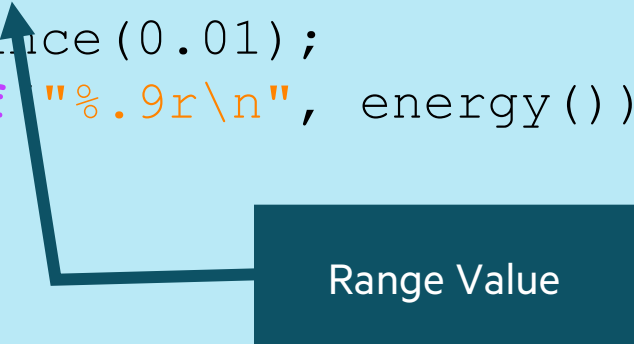
Formatted I/O

5-BODY IN CHAPEL: MAIN()



nbody.chpl

```
...  
  
proc main() {  
  initSun();  
  
  writef("%.9r\n", energy());  
  for 1..numsteps do  
    advance(0.01);  
  writef("%.9r\n", energy());  
}  
  
...
```



RANGES: INTEGER SEQUENCES

RANGE VALUES: INTEGER SEQUENCES

Syntax

```
range-expr:  
    [low] .. [high]
```

Definition

- Regular sequence of integers
 $\text{low} \leq \text{high}$: $\text{low}, \text{low}+1, \text{low}+2, \dots, \text{high}$
 $\text{low} > \text{high}$: degenerate (an empty range)
 low or high unspecified: unbounded in that direction

Examples

```
1..6           // 1, 2, 3, 4, 5, 6  
6..1           // empty  
3..            // 3, 4, 5, 6, 7, ...
```



RANGE OPERATORS

 02-range-operators.chpl

```
const r = 1..10;

printVals(r);
printVals(r # 3);
printVals(r by 2);
printVals(r by -2);
printVals(r by 2 # 3);
printVals(r # 3 by 2);
printVals(0.. #n);

proc printVals(r) {
  for i in r do
    write(i, " ");
  writeln();
}
```

```
1 2 3 4 5 6 7 8 9 10
1 2 3
1 3 5 7 9
10 8 6 4 2
1 3 5
1 3
0 1 2 3 4 ... n-1
```

5-BODY IN CHAPEL: MAIN()



nbody.chpl

```
...  
  
proc main() {  
    initSun();  
  
    writef("%.9r\n", energy());  
    for 1..numsteps do  
        advance(0.01);  
    writef("%.9r\n", energy());  
}  
  
...
```

Serial for loop

BASIC SERIAL CONTROL FLOW

FOR LOOPS

Syntax

```
for-loop:  
  for [index-expr in] iteratable-expr { stmt-list }
```

Meaning

- Executes loop body serially, once per loop iteration
- Declares new variables for identifiers in *index-expr*
 - type and const-ness determined by *iteratable-expr*
 - *iteratable-expr* could be a range, array, iterator, iterable object, ...

Examples

```
var A: [1..3] string = [ " DO", " RE", " MI" ];  
  
for i in 1..3 { write (A[i]); }           // DO RE MI  
for a in A { a += "LA"; } write (A);      // DOLA RELA MILA
```


CONTROL FLOW: OTHER FORMS

- Conditional statements

```
if cond { computeA(); } else { computeB(); }
```

- While loops

```
while cond {  
    compute();  
}
```

- For loops

```
for indices in iterable-expr {  
    compute();  
}
```

- Select statements

```
select key {  
    when value1 { compute1(); }  
    when value2 { compute2(); }  
    otherwise   { compute3(); }  
}
```

CONTROL FLOW: BRACES VS. KEYWORDS

Control flow statements specify bodies using curly brackets (compound statements)

- Conditional statements

```
if cond { computeA(); } else { computeB(); }
```

- While loops

```
while cond {  
    compute();  
}
```

- For loops

```
for indices in iterable-expr {  
    compute();  
}
```

- Select statements

```
select key {  
    when value1 { compute1(); }  
    when value2 { compute2(); }  
    otherwise   { compute3(); }  
}
```

CONTROL FLOW: BRACES VS. KEYWORDS

They also support keyword-based forms for single-statement cases

- Conditional statements

```
if cond then computeA(); else computeB();
```

- While loops

```
while cond do  
    compute();
```

- For loops

```
for indices in iterable-expr do  
    compute();
```

- Select statements

```
select key {  
    when value1 do compute1();  
    when value2 do compute2();  
    otherwise   do compute3();  
}
```

CONTROL FLOW: BRACES VS. KEYWORDS

Of course, since compound statements are single statements, the two forms can be mixed...

- Conditional statements

```
if cond then { computeA(); } else { computeB(); }
```

- While loops

```
while cond do {  
    compute();  
}
```

- For loops

```
for indices in iterable-expr do {  
    compute();  
}
```

- Select statements

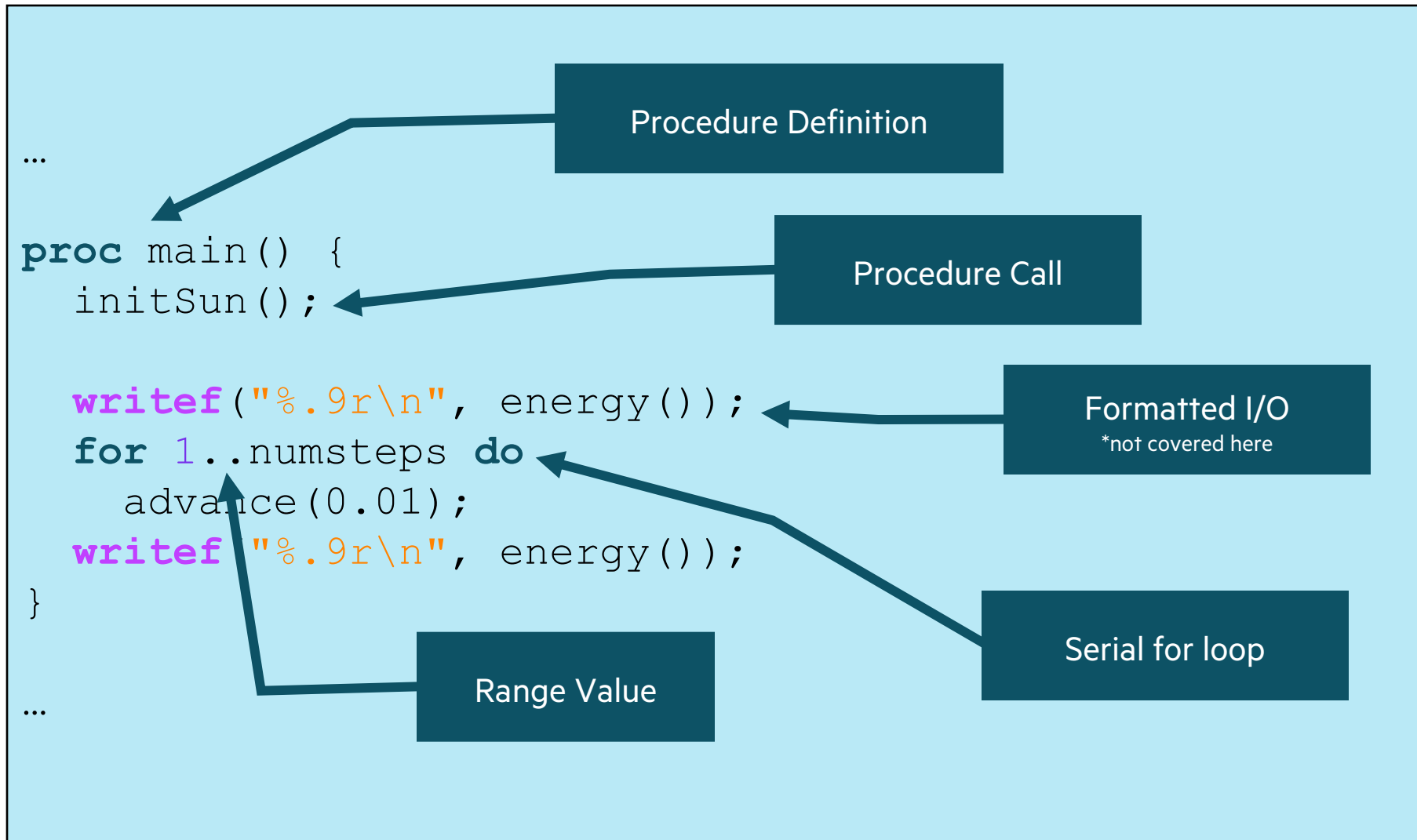
```
select key {  
    when value1 do { compute1(); }  
    when value2 do { compute2(); }  
    otherwise do { compute3(); }  
}
```

PROCEDURES AND ITERATORS

5-BODY IN CHAPEL: MAIN()



nbody.chpl



5-BODY IN CHAPEL: ADVANCE()



nbody.chpl

```
advance(0.01);  
...  
proc advance(dt) {  
  for i in 1..numbodies {  
    for j in i+1..numbodies {  
      const dpos = bodies[i].pos - bodies[j].pos,  
      mag = dt / sqrt(sumOfSquares(dpos))**3;  
  
      bodies[i].v -= dpos * bodies[j].mass * mag;  
      bodies[j].v += dpos * bodies[i].mass * mag;  
    }  
  }  
  
  for b in bodies do  
    b.pos += dt * b.v;  
}
```

5-BODY IN CHAPEL: ADVANCE()



nbody.chpl

```
advance(0.01);
```

```
...
```

```
proc advance(dt) {
```

```
  for i in 1..numbodies {
```

```
    for j in i+1..numbodies {
```

```
      const dpos = bodies[i].pos - bodies[j].pos,
```

```
      mag = dt / sqrt(sumOfSquares(dpos))**3;
```

```
      bodies[i].v -= dpos * bodies[j].mass * mag;
```

```
      bodies[j].v += dpos * bodies[i].mass * mag;
```

```
    }
```

```
  }
```

```
  for b in bodies do
```

```
    b.pos += dt * b.v;
```

```
}
```

$$m_1 \mathbf{a}_1 = \frac{G m_1 m_2}{r_{12}^3} (\mathbf{r}_2 - \mathbf{r}_1) \quad \text{Sun-Earth}$$

$$m_2 \mathbf{a}_2 = \frac{G m_1 m_2}{r_{21}^3} (\mathbf{r}_1 - \mathbf{r}_2) \quad \text{Earth-Sun}$$

5-BODY IN CHAPEL: ADVANCE()



nbody.chpl

```
advance(0.01);  
...  
proc advance(dt) {  
  for i in 1..numbodies {  
    for j in i+1..numbodies {  
      const dpos = bodies[i].pos - bodies[j].pos,  
          mag = dt / sqrt(sumOfSquares(dpos))**3;  
  
      bodies[i].v -= dpos * bodies[j].mass * mag;  
      bodies[j].v += dpos * bodies[i].mass * mag;  
    }  
  }  
  
  for b in bodies do  
    b.pos += dt * b.v;  
}
```

Procedure call

Procedure definition

PROCEDURES, BY EXAMPLE

- Example to compute the area of a circle

```
proc area(radius: real): real {  
    return 3.14 * radius**2;  
}
```

```
writeln(area(2.0)); // 12.56
```

```
proc area(radius) {  
    return 3.14 * radius**2;  
}
```

Argument and return
types can be omitted

- Example of argument default values, naming

```
proc writeCoord(x: real = 0.0, y: real = 0.0) {  
    writeln((x,y));  
}
```

```
writeCoord(2.0);           // (2.0, 0.0)  
writeCoord(y=2.0);         // (0.0, 2.0)  
writeCoord(y=2.0, 3.0);    // (3.0, 2.0)
```

ARGUMENT INTENTS

Arguments can optionally be given intents

- (blank): varies with type; follows principle of least surprise
 - most types: **const in** or **const ref**
 - sync/single vars, atomics: **ref**
- **ref**: formal is a reference back to the actual
- **const [ref | in]**: disallows modification of the formal
- **param/type**: actual must be a param/type
- **in**: initializes formal using actual; permits formal to be modified
- **out**: copies formal into actual at procedure return
- **inout**: does both of the above



ARGUMENT INTENTS, BY EXAMPLE

- For some types, argument intents are needed so as to avoid inadvertent races

```
proc foo(x: real, y: [] real) {  
    // x = 1.2;    // illegal: scalars are passed 'const in' by default  
    // y = 3.4;    // illegal: 'ref' by default for arrays is deprecated  
}  
  
var r: real,  
    A: [1..3] real;  
  
foo(r, A);  
  
writeln((r, A));
```

ARGUMENT INTENTS, BY EXAMPLE

- Arguments can optionally be given intents.
- 'ref' intent means the actual being passed in will be modified

```
proc foo(ref x: real, ref y: [] real) {  
    x = 1.2;    // OK: actual is modified  
    y = 3.4;    // OK: actual is modified  
}  
  
var r: real,  
    A: [1..3] real;  
  
foo(r, A);  
  
writeln( (r, A) );    // writes (1.2, [3.4, 3.4, 3.4])
```

ARGUMENT INTENTS, BY EXAMPLE

- Can't pass a 'const' to a 'ref' intent

```
proc foo(ref x: real, ref y: [] real) {  
    x = 1.2; // OK: actual is modified  
    y = 3.4; // OK: actual is modified  
}  
  
const r: real,  
      A: [1..3] real;  
  
// foo(r, A); // illegal, can't pass a constant to a 'ref' intent  
  
writeln((r, A)); // writes (0.0, [0.0, 0.0, 0.0])
```

ARGUMENT INTENTS, BY EXAMPLE

- Can pass a 'const' to a 'const ref' intent
- However, can't write to a formal coming in as 'const' intent

```
proc foo(const ref x: real, const ref y: [] real) {  
    // x = 1.2;    // illegal: can't modify constant arguments  
    // y = 3.4;    // illegal: can't modify constant arguments  
}  
  
const r: real,  
      A: [1..3] real;  
  
foo(r, A);    // OK to create constant references to constants  
  
writeln((r, A));    // writes (0.0, [0.0, 0.0, 0.0])
```

ARGUMENT INTENTS, BY EXAMPLE

- Can't pass 'const' and 'var' into 'param' intents

```
proc foo(param x: real, type t) {  
    ...  
    ...  
}  
  
const r: real,  
      A: [1..3] real;  
  
// foo(r, A); // illegal: can't pass vars and consts to params and types  
  
writeln((r, A)); // writes (0.0, [0.0, 0.0, 0.0])
```


ARGUMENT INTENTS, BY EXAMPLE

- Can pass a literal, param, or a type into 'param' intent

```
proc foo(param x: real, type t) {  
    ...  
    ...  
}  
  
const r: real,  
       A: [1..3] real;  
  
foo(1.2, r.type); // OK: passing a literal/param and a type  
  
writeln((r, A)); // writes (0.0, [0.0, 0.0, 0.0])
```

ARGUMENT INTENTS, BY EXAMPLE

- 'in' intents cause the actual argument value to be copied into the formal

```
proc foo(in x: real, in y: [] real) {  
  x = 1.2;  // OK: local copy is modified  
  y = 3.4;  // OK: local copy is modified  
}  
  
var r: real,  
    A: [1..3] real;  
  
foo(r, A);  
  
writeln((r, A));  // writes (0.0, [0.0, 0.0, 0.0])
```

ARGUMENT INTENTS, BY EXAMPLE

- 'out' intents cause the formal value to be copied into actual argument upon return from procedure

```
proc foo(out x: real, out y: [] real) {  
  x = 1.2;  // OK: local copy is modified  
  y = [3.4, 3.4, 3.4];  // OK: local copy is modified  
}  
  
var r: real,  
    A: [1..3] real;  
  
foo(r, A);  
  
writeln((r, A));  // writes (1.2, [3.4, 3.4, 3.4])
```

ARGUMENT INTENTS, BY EXAMPLE

- 'inout' intent is a combination of 'in' and 'out' intent

```
proc foo(inout x: real, inout y: [] real) {  
  x = 1.2;  // OK: local copy is modified  
  y = 3.4;  // OK: local copy is modified  
}  
  
var r: real,  
    A: [1..3] real;  
  
foo(r, A);  
  
writeln((r, A));  // writes (1.2, [3.4, 3.4, 3.4])
```

5-BODY IN CHAPEL: ADVANCE()



nbody.chpl

```
proc advance(dt) {  
  for i in 1..numbodies {  
    for j in i+1..numbodies {  
      const dpos = bodies[i].pos - bodies[j].pos,  
            mag = dt / sqrt(sumOfSquares(dpos))**3;  
  
      bodies[i].v -= dpos * bodies[j].mass * mag;  
      bodies[j].v += dpos * bodies[i].mass * mag;  
    }  
  }  
  
  for b in bodies do  
    b.pos += dt * b.v;  
}
```

5-BODY IN CHAPEL: ALTERNATIVE USING ITERATORS



nbody.chpl

Use of iterator

```
proc advance(dt) {  
  for (i,j) in triangle(numbodies) {  
    const dpos = bodies[i].pos - bodies[j].pos,  
        mag = dt / sqrt(sumOfSquares(dpos))**3;  
    ...  
  }  
  ...  
}
```

Definition of iterator

```
iter triangle(n) {  
  for i in 1..n do  
    for j in i+1..n do  
      yield (i,j);  
    }  
}
```

5-BODY IN CHAPEL: ADVANCE() USING ITERATORS



nbody.chpl

```
proc advance(dt) {  
  for (i,j) in triangle(numbodies) {  
  
    const dpos = bodies[i].pos - bodies[j].pos,  
          mag = dt / sqrt(sumOfSquares(dpos))**3;  
  
    bodies[i].v -= dpos * bodies[j].mass * mag;  
    bodies[j].v += dpos * bodies[i].mass * mag;  
  }  
  
  for b in bodies do  
    b.pos += dt * b.v;  
}
```

HANDS ON: WHERE MIGHT WE CONSIDER PARALLELIZING N-BODY



Look at 'nbody.chpl' and identify...

- 'for' loops that can be parallelized
- 'for' loops that need to stay serial to keep meaning
- 'for' loops that are "mostly" parallel but have something like +=

Can be parallelized

```
for b in bodies do  
    b.pos += dt * b.v;
```

Inherently serial loop

```
for 1..numsteps do  
    advance(0.01);
```

Can be parallelized but
have to avoid races when
adding into velocity field

```
for i in 1..numbodies {  
    for j in i+1..numbodies {  
        const dpos = bodies[i].pos - bodies[j].pos,  
              mag = dt / sqrt(sumOfSquares(dpos)) ** 3;  
        bodies[i].v -= dpos * bodies[j].mass * mag;  
        bodies[j].v += dpos * bodies[i].mass * mag;  
    }  
}
```


OUTLINE: CHAPEL BASICS

- Running Example: n-body computation (Hands On)
- Variables, Constants, and Operators
- Records and Classes
- Tuples
- Arrays
- Writing out Tuples, Records, and Arrays (Hands On)
- Main() Procedure
- Ranges and basic control flow
- Procedures and iterators
- Where might we parallelize the n-body computation? (Hands On)



LEARNING OBJECTIVES FOR TODAY'S CHAPEL TUTORIAL

- Familiarity with the Chapel execution model including how to run codes in parallel on a single node, across nodes, and both
- Learn Chapel concepts by compiling and running provided code examples
 - ✓ Serial code using map/dictionary, (k-mer counting from bioinformatics)
 - ✓ Parallelism and locality in Chapel
 - ✓ Distributed parallelism and 1D arrays, (processing files in parallel)
 - ✓ Chapel basics in the context of an n-body code
- Distributed parallelism and 2D arrays, (heat diffusion problem)
- How to parallelize histogram
- Using CommDiagnostics for counting remote reads and writes
- Chapel and Arkouda best practices including avoiding races and performance gotchas
- Where to get help and how you can participate in the Chapel community



ONE DAY CHAPEL TUTORIAL

- 9-10:30: Getting started using Chapel for parallel programming
- 10:30-10:45: break
- 10:45-12:15: Chapel basics in the context of the n-body example code
- 12:15-1:15: lunch
- 1:15-2:45: Distributed and shared-memory parallelism especially w/arrays (data parallelism)
- 2:45-3:00: break
- 3:00-4:30: More parallelism including for asynchronous parallelism (task parallelism)
- 4:30-5:00: Wrap-up including gathering further questions from attendees



CHAPEL RESOURCES

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


- (points to all other resources)

Social Media:

- Twitter: [@ChapelLanguage](https://twitter.com/ChapelLanguage)
- Facebook: [@ChapelLanguage](https://facebook.com/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>



The Chapel Parallel Programming Language

What is Chapel?

Chapel is a programming language designed for productive parallel computing at scale.

Why Chapel? Because it simplifies parallel programming through elegant support for:

- **distributed arrays** that can leverage thousands of nodes' memories and cores
- **a global namespace** supporting direct access to local or remote variables
- **data parallelism** to trivially use the cores of a laptop, cluster, or supercomputer
- **task parallelism** to create concurrency within a node or across the system

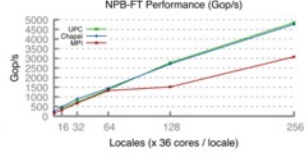
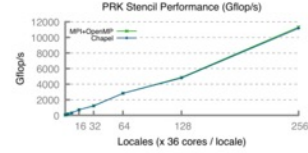
Chapel Characteristics

- **productive:** code tends to be similarly readable/writable as Python
- **scalable:** runs on laptops, clusters, the cloud, and HPC systems
- **fast:** performance *competes with or beats* C/C++ & MPI & OpenMP
- **portable:** compiles and runs in virtually any *nix environment
- **open-source:** hosted on [GitHub](#), permissively [licensed](#)

New to Chapel?

As an introduction to Chapel, you may want to...

- watch an [overview talk](#) or browse its [slides](#)
- read a [blog-length](#) or [chapter-length](#) introduction to Chapel
- learn about [projects powered by Chapel](#)
- check out [performance highlights](#) like these:



PRK Stencil Performance (Gflop/s)

NPB-FT Performance (Gop/s)

- browse [sample programs](#) or [learn](#) how to write distributed programs like this one:

```
use CyclicDist;           // use the Cyclic distribution library
config const n = 100;     // use --n=cval when executing to override this default

forall i in {1..n} dmapped Cyclic(startIdx=1) do
  writeln("Hello from iteration ", i, " of ", n, " running on node ", here.id);
```



Hewlett Packard
Enterprise

ONE-DAY CHAPEL TUTORIAL

SESSION 3: PARALLELISM IN CHAPEL



Chapel Team
October 16, 2023

ONE DAY CHAPEL TUTORIAL

- 9-10:30: Getting started using Chapel for parallel programming
- 10:30-10:45: break
- 10:45-12:15: Chapel basics in the context of the n-body example code
- 12:15-1:15: lunch
- 1:15-2:45: Distributed and shared-memory parallelism especially w/arrays (data parallelism)
- 2:45-3:00: break
- 3:00-4:30: More parallelism including for asynchronous parallelism (task parallelism)
- 4:30-5:00: Wrap-up including gathering further questions from attendees



OUTLINE: PARALLELISM IN CHAPEL

- Recall processing files in parallel
- Data parallelism concepts and examples including multi-locale parallelism with distributions
- Domains
- Forall Loops
- Domain Distributions
- Using a Different Domain Distribution
- Implicit Communication: Remote writes/Puts and Reads/Gets
- Parallelizing a 1D heat diffusion solver (Hands On)
- Heat 2D example with CommDiagnostics (Hands On)



RECALL PROCESSING FILES IN PARALLEL

RECALL: ANALYZING MULTIPLE FILES USING PARALLELISM

 parfilekmer.chpl

parfilekmer.chpl

```
use FileSystem;
config const dir = "DataDir";
var fList = findFiles(dir);
var filenames =
    blockDist.createArray(0..<fList.size, string);
filenames = fList;

// per file word count
forall f in filenames {
    ...
    // code from kmer.chpl
    ...
}
```

```
prompt> chpl --fast parfilekmer.chpl
prompt> ./parfilekmer -nl 1
prompt> ./parfilekmer -nl 4
```

- shared and distributed-memory parallelism using 'forall'
 - in other words, parallelism within the locale/node and across locales/nodes
- a distributed array
- command line options to indicate number of locales



RECALL: BLOCK DISTRIBUTION OF ARRAY OF STRINGS

Locale 0

Locale 1

"filename1"	"filename2"	"filename3"	"filename4"	"filename5"	"filename6"	"filename7"	"filename8"
-------------	-------------	-------------	-------------	-------------	-------------	-------------	-------------

```
prompt> chpl --fast parfilekmer.chpl  
prompt> ./parfilekmer -nl 2
```

- Array of strings for filenames is distributed across locales
- 'forall' will do parallelism across locales and then within each locale to take advantage of multicore



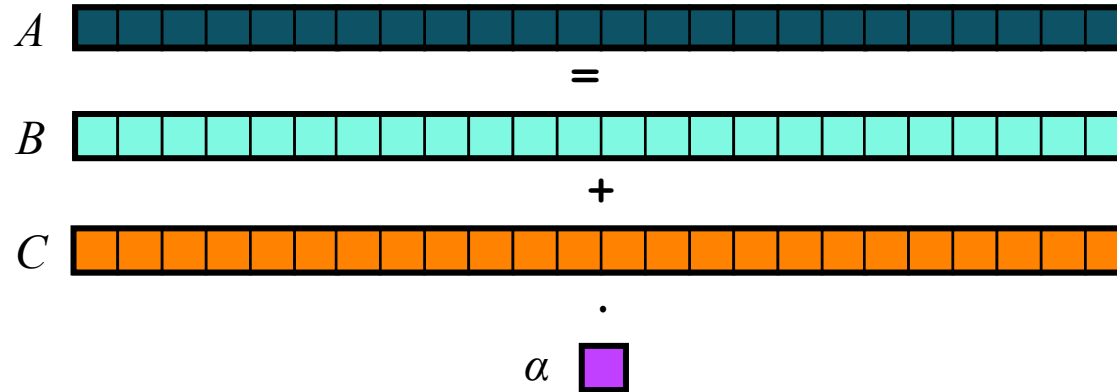
DATA PARALLELISM CONCEPTS AND EXAMPLES INCLUDING MULTI-LOCALE PARALLELISM WITH DISTRIBUTIONS

STREAM TRIAD: A PARALLEL COMPUTATION

Given: m -element vectors A, B, C

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

In pictures:

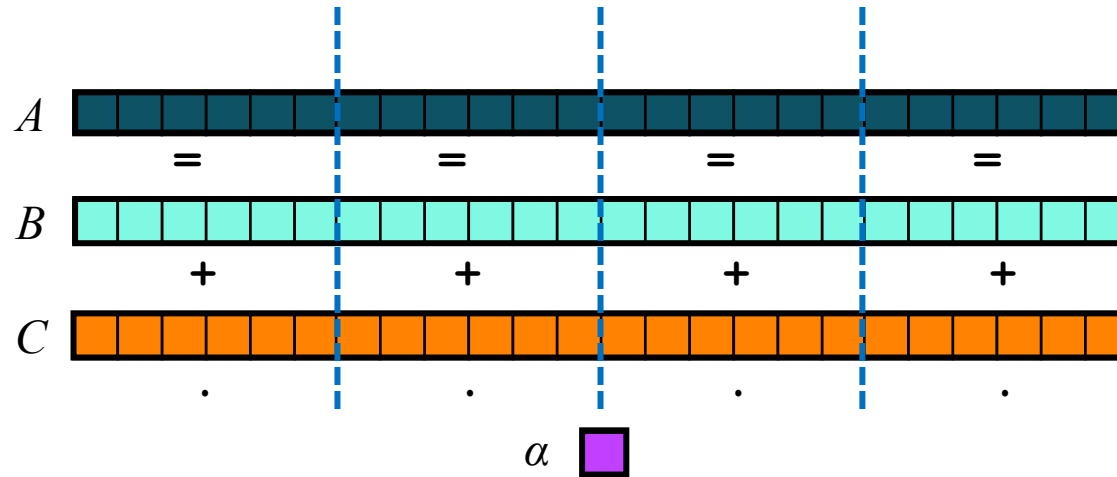


STREAM TRIAD: A PARALLEL COMPUTATION

Given: m -element vectors A, B, C

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

In pictures, in parallel (shared memory / multicore):

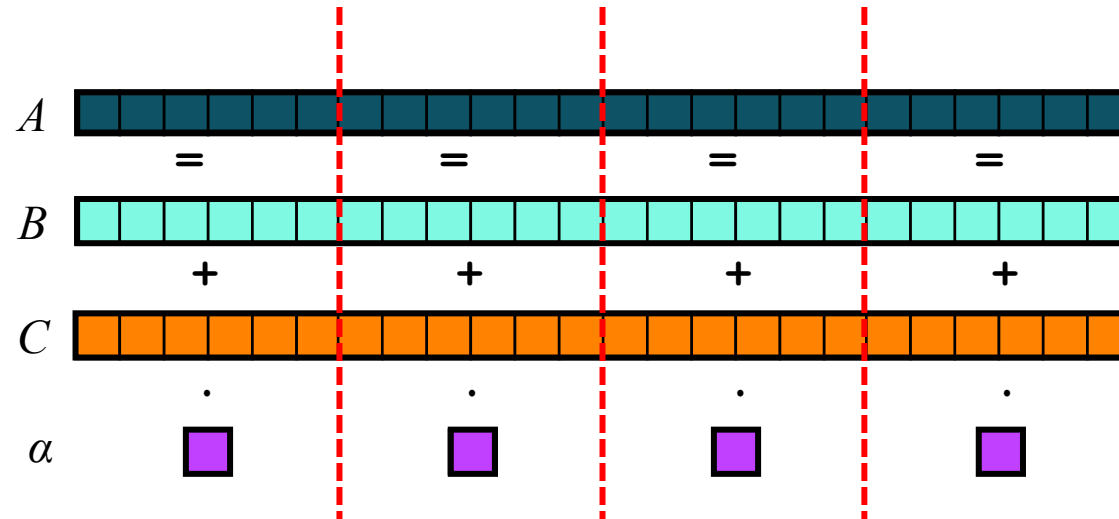


STREAM TRIAD: A PARALLEL COMPUTATION

Given: m -element vectors A, B, C

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

In pictures, in parallel (distributed memory):

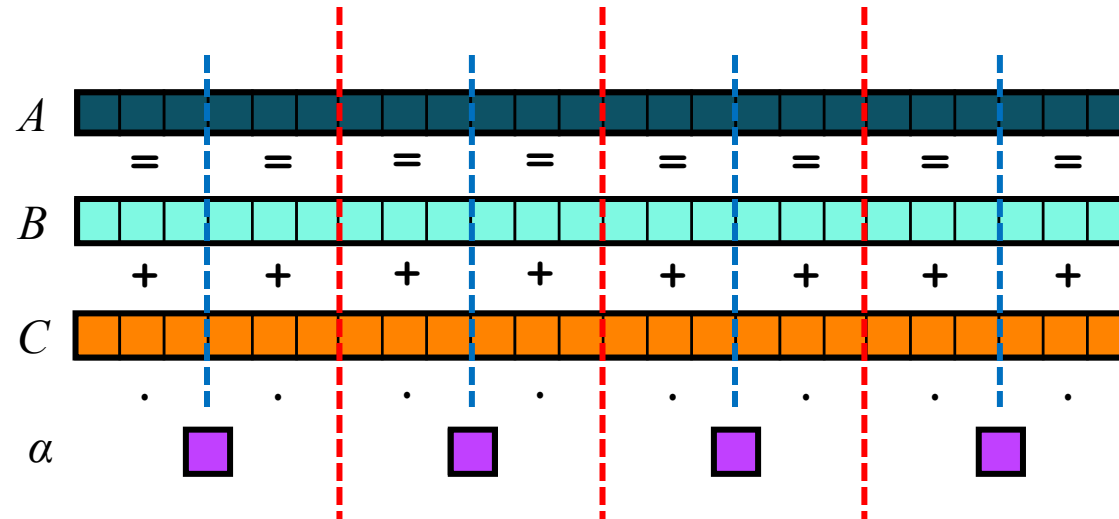


STREAM TRIAD: A PARALLEL COMPUTATION

Given: m -element vectors A, B, C

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

In pictures, in parallel (distributed memory multicore):

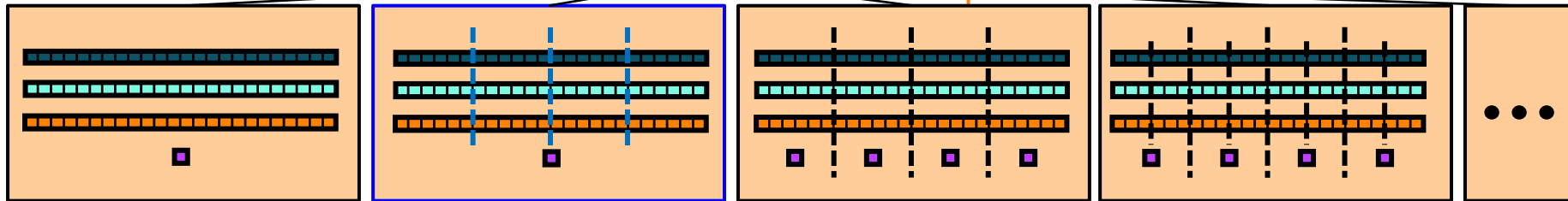


STREAM TRIAD: CHAPEL

```
use BlockDist;  
  
config const m = 1000,  
           alpha = 3.0;  
  
const ProblemSpace = blockDist.createDomain({1..m});  
  
var A, B, C: [ProblemSpace] real;  
  
B = 2.0;  
C = 1.0;  
  
A = B + alpha * C;
```

The special sauce:

How should this index set—and any arrays and computations over it—be mapped to the system?



Philosophy: Good, *top-down* language design can tease system-specific implementation details away from an algorithm, permitting the compiler, runtime, applied scientist, and HPC expert to each focus on their strengths.

DATA PARALLELISM, BY EXAMPLE



03-domain-distributions.chpl

```
config const n = 1000;  
var D = {1..n, 1..n};  
  
var A: [D] real;  
forall (i,j) in D with (ref A) do  
    A[i,j] = i + (j - 0.5)/n;  
writeln(A);
```


```
prompt> chpl dataParallel.chpl  
prompt> ./dataParallel -nl 1 --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
```

DOMAINS

DATA PARALLELISM, BY EXAMPLE

 03-domain-distributions.chpl

Domains (Index Sets)



```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D with (ref A) do
    A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

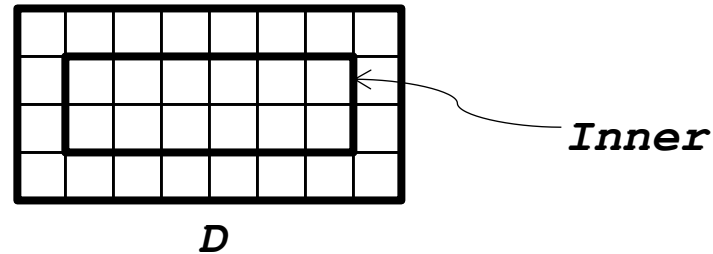
```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel -nl 1 --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

DOMAINS

Domain:

- A first-class index set
- The fundamental Chapel concept for data parallelism

```
config const m = 4, n = 8;  
  
const D = {1..m, 1..n};  
const Inner = {2..m-1, 2..n-1};
```

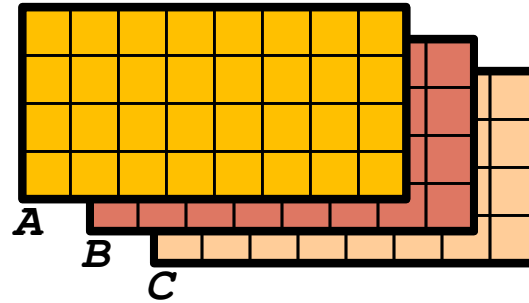


DOMAINS

Domain:

- A first-class index set
- The fundamental Chapel concept for data parallelism
- Useful for declaring arrays and computing with them

```
config const m = 4, n = 8;  
  
const D = {1..m, 1..n};  
const Inner = {2..m-1, 2..n-1};  
  
var A, B, C: [D] real;
```



DATA PARALLELISM, BY EXAMPLE

 03-domain-distributions.chpl

Arrays



```
config const n = 1000;  
var D = {1..n, 1..n};  
  
var A: [D] real;  
forall (i,j) in D with (ref A) do  
    A[i,j] = i + (j - 0.5)/n;  
writeln(A);
```

```
prompt> chpl dataParallel.chpl  
prompt> ./dataParallel -nl 1 --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
```

FORALL LOOPS

DATA PARALLELISM, BY EXAMPLE



03-domain-distributions.chpl

Data-Parallel Forall Loops

```
config const n = 1000;  
var D = {1..n, 1..n};  
  
var A: [D] real;  
forall (i,j) in D with (ref A) do  
    A[i,j] = i + (j - 0.5)/n;  
writeln(A);
```

```
prompt> chpl dataParallel.chpl  
prompt> ./dataParallel -nl 1 --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
```


FORALL LOOPS

Forall loops: Central concept for data parallel computation

- Like for-loops, but parallel
- Implementation details determined by iterand (e.g., D below)
 - specifies number of tasks, which tasks run which iterations, ...
 - in practice, typically uses a number of tasks appropriate for target HW

```
forall (i,j) in D with (ref A) do  
    A[i,j] = i + j/10.0;
```

Forall loops assert...

- ...parallel safety:** OK to execute iterations simultaneously
- ...order independence:** iterations could occur in any order
- ...serializability:** all iterations could be executed by one task
 - e.g., can't have synchronization dependences between iterations

1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8
2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8
3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8
4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.8

COMPARISON OF LOOPS: FOR, FORALL, AND COFORALL

For loops: executed using one task

- use when a loop must be executed serially
- or when one task is sufficient for performance

Forall loops: typically executed using $1 < \#tasks << \#iters$

- use when a loop *should* be executed in parallel...
- ...but *can* legally be executed serially
- use when desired $\# \text{ tasks } << \# \text{ of iterations}$

Coforall loops: executed using a task per iteration

- use when the loop iterations *must* be executed in parallel
- use when you want $\# \text{ tasks } == \# \text{ of iterations}$
- use when each iteration has substantial work



DATA PARALLELISM, BY EXAMPLE



03-domain-distributions.chpl

This is a shared memory program

Nothing has referred to remote
locales, explicitly or implicitly

```
config const n = 1000;  
var D = {1..n, 1..n};  
  
var A: [D] real;  
forall (i,j) in D with (ref A) do  
    A[i,j] = i + (j - 0.5)/n;  
writeln(A);
```

```
prompt> chpl dataParallel.chpl  
prompt> ./dataParallel -nl 1 --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
```

DOMAIN DISTRIBUTIONS

DISTRIBUTED DATA PARALLELISM, BY EXAMPLE

 03-domain-distributions.chpl

Domain Distribution
(Map Data Parallelism to the System)

```
use CyclicDist;
config const n = 1000;
var D = cyclicDist.createDomain({1..n, 1..n});


var A: [D] real;
forall (i,j) in D with (ref A) do
    A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5 -nl 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
```

DISTRIBUTED DATA PARALLELISM, BY EXAMPLE

 03-domain-distributions.chpl

High-level distributed and shared
memory parallelism



```
use CyclicDist;
config const n = 1000;
var D = cyclicDist.createDomain({1..n, 1..n});

var A: [D] real;
forall (i,j) in D with (ref A) do
    A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

Provides programmability and control

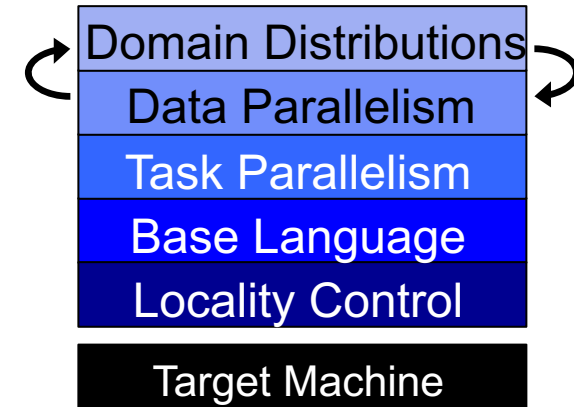
- Lowering of code is well-defined
- User can control details
- Part of Chapel's *multiresolution philosophy*...

```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5 --nl 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'S MULTIREOLUTION PHILOSOPHY

Multiresolution Design: Support multiple tiers of features

- higher levels for programmability, productivity
- lower levels for greater degrees of control
- build the higher-level concepts in terms of the lower
- permit users to intermix layers arbitrarily



DISTRIBUTED DATA PARALLELISM, BY EXAMPLE

 03-domain-distributions.chpl

Chapel's prescriptive approach:

```
forall (i,j) in D do...
```

- ⇒ invoke and inline D's
default parallel iterator
- defined by D's type /
domain distribution

default domain distribution

- create a task per local core
- block indices across tasks

```
config const n = 1000;  
var D = {1..n, 1..n};  
  
var A: [D] real;  
forall (i,j) in D with (ref A) do  
    A[i,j] = i + (j - 0.5)/n;  
writeln(A);
```

```
prompt> chpl dataParallel.chpl  
prompt> ./dataParallel --n=5 -nl 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
```


DISTRIBUTED DATA PARALLELISM, BY EXAMPLE



03-domain-distributions.chpl

Chapel's prescriptive approach:

```
forall (i,j) in D do...
```

⇒ invoke and inline D's
default parallel iterator

- defined by D's type /
domain distribution

cyclic domain distribution

on each target locale...

- create a task per core
- block local indices across tasks

```
use CyclicDist;
config const n = 1000;
var D = cyclicDist.createDomain({1..n, 1..n});

var A: [D] real;
forall (i,j) in D with (ref A) do
    A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5 -nl=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
```

DISTRIBUTED DATA PARALLELISM, BY EXAMPLE

 03-domain-distributions.chpl

Chapel's prescriptive approach:

```
forall (i,j) in D do...
```

What if I don't like D's iteration strategy?

```
use CyclicDist;
config const n = 1000;
var D = cyclicDist.createDomain({1..n, 1..n});
var A: [D] real;
forall (i,j) in D with (ref A) do
    A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

Write and call your own parallel iterator:

```
forall (i,j) in myParIter(D) do...
```



DISTRIBUTED DATA PARALLELISM, BY EXAMPLE

 03-domain-distributions.chpl

Chapel's prescriptive approach:

```
forall (i,j) in D do...
```

What if I don't like D's iteration strategy?

```
use CyclicDist;
config const n = 1000;
var D = cyclicDist.createDomain({1..n, 1..n});
var A: [D] real;
forall (i,j) in D with (ref A) do
    A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

Write and call your own parallel iterator:

```
forall (i,j) in myParIter(D) do...
```

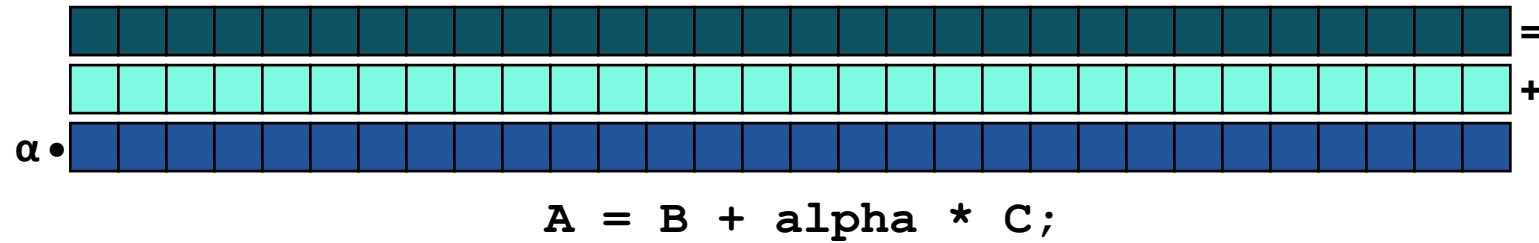
Or use a different domain distribution:

```
var D = blockDist.createDomain({1..n, 1..n});
```

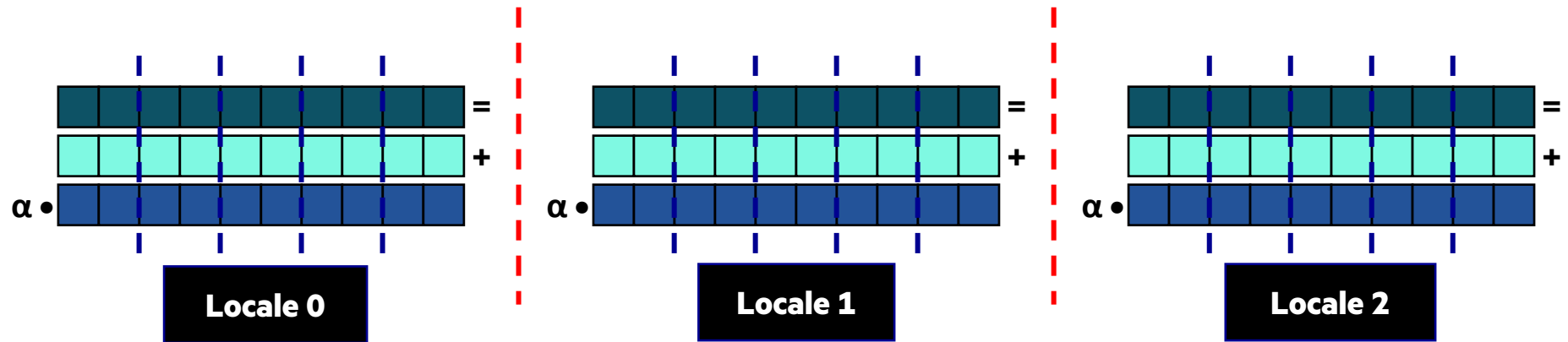
USING A DIFFERENT DOMAIN DISTRIBUTION

DOMAIN DISTRIBUTIONS: A MULTIREOLUTION FEATURE

Domain distributions are “recipes” that instruct the compiler how to map the global view of a computation...

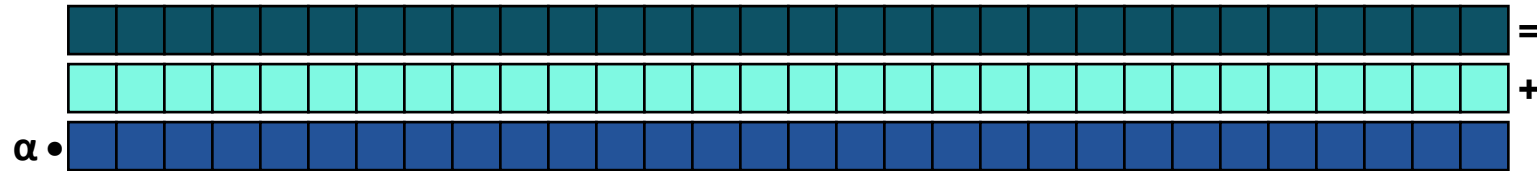


...to the target locales' memory and processors:



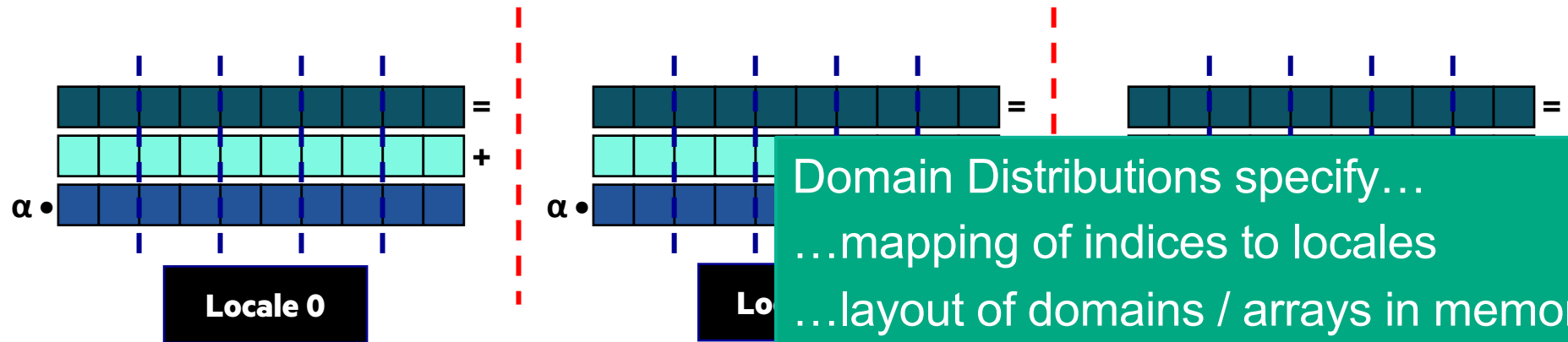
DOMAIN DISTRIBUTIONS: A MULTIREOLUTION FEATURE

Domain distributions are “recipes” that instruct the compiler how to map the global view of a computation...



$$A = B + \text{alpha} * C;$$

...to the target locales' memory and processors:



Domain Distributions specify...

...mapping of indices to locales

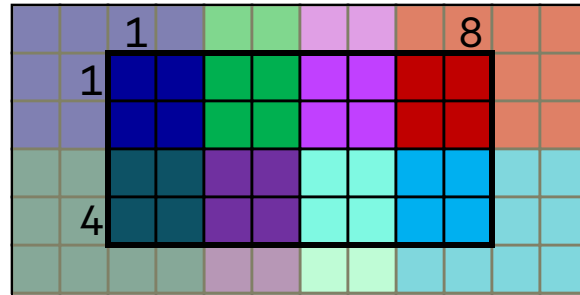
...layout of domains / arrays in memory

...parallel iteration strategies

...core operations on arrays / domains

SAMPLE DOMAIN DISTRIBUTIONS: BLOCK AND CYCLIC

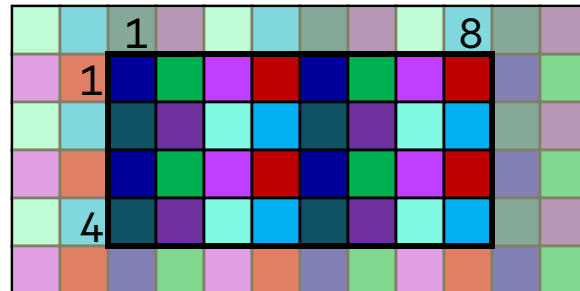
```
var Dom = blockDist.createDomain({1..4, 1..8});
```



distributed to

L0	L1	L2	L3
L4	L5	L6	L7

```
var Dom = cyclicDist.createDomain({1..4, 1..8});
```



distributed to

L0	L1	L2	L3
L4	L5	L6	L7

**IMPLICIT COMMUNICATION:
REMOTE WRITES/PUTS AND READS/GETS**

CHAPEL SUPPORTS A GLOBAL NAMESPACE WITH PUTS AND GETS

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

 03-onClause.chpl

03-onClause.chpl

```
config const verbose = false;  
var total = 0,  
    done = false;  
  
...  
  
on Locales[1] {  
    var x, y, z: int;  
    ...  
}
```

verbose false
total 0
done false

locale 0

x 0
y 0
z 0

locale 1

CHAPEL SUPPORTS A GLOBAL NAMESPACE WITH PUTS AND GETS

Note 2: Tasks can refer to lexically visible variables, whether local or remote

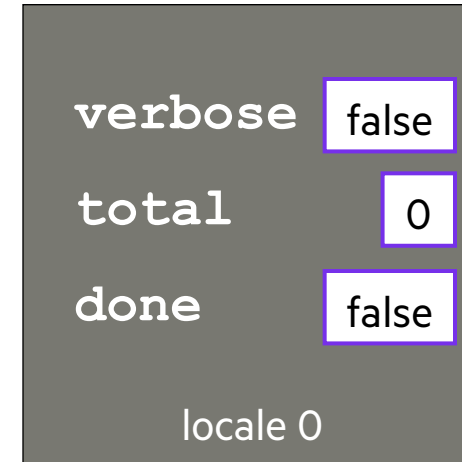
 03-onClause.chpl

03-onClause.chpl

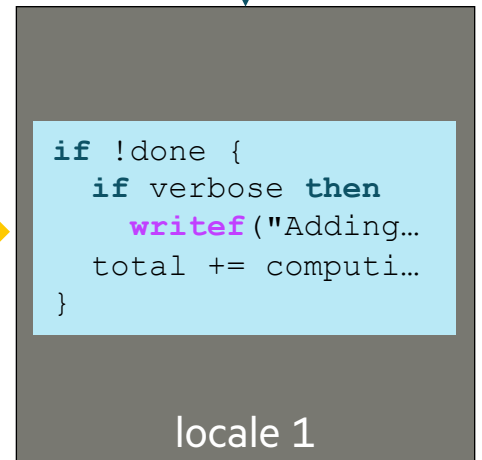
```
config const verbose = false;
var total = 0,
    done = false;

...

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



code runs on locale 1,
but refers to values
stored on locale 0



ARRAY-BASED PARALLELISM AND LOCALITY



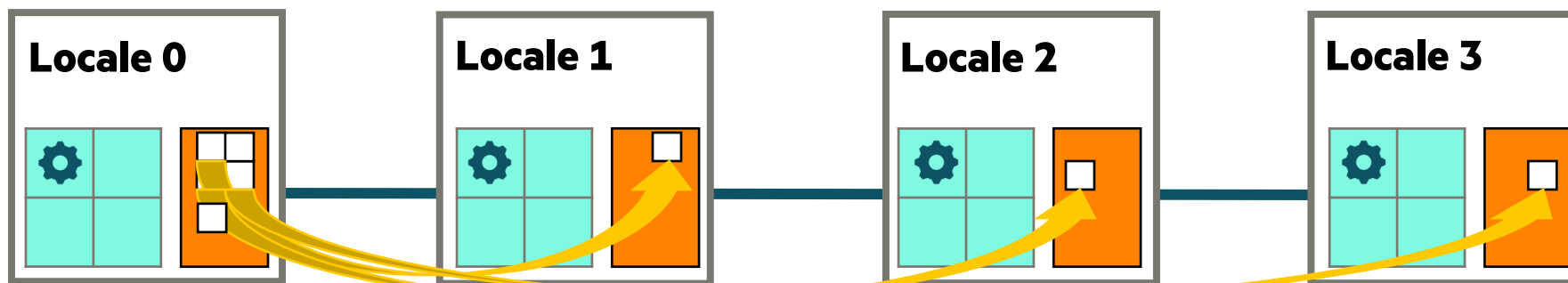
03-basics-distarr.chpl

03-basics-distarr.chpl

```
writeln("Hello from locale ", here.id);  
  
var A: [1..2, 1..2] real;  
  
use BlockDist;  
  
var D = blockDist.createDomain({1..2, 1..2});  
var B: [D] real;  
B = A;
```

Chapel also supports distributed domains (index sets) and arrays

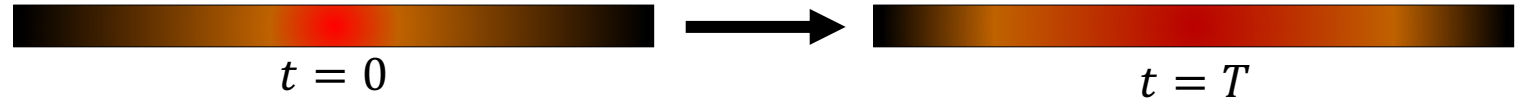
They also result in parallel distributed computation



PARALLELIZING A 1D HEAT DIFFUSION SOLVER (HANDS ON)

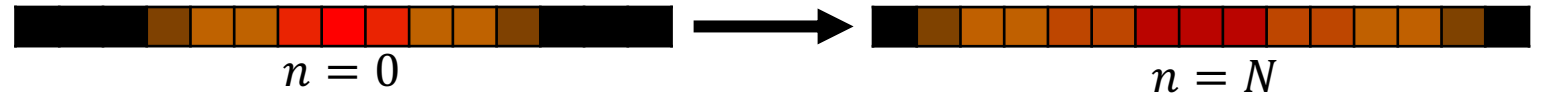
1D HEAT EQUATION EXAMPLE

Differential equation: $\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$



Discretized (finite difference) equation: $u_i^{n+1} = u_i^n + \alpha (u_{i-1}^n - 2u_i^n + u_{i+1}^n)$

- where $i \in \Omega \subset \mathbb{R}^1$ are discrete points in space, and $(n, n+1, \dots)$ are discrete instances in time



Finite difference algorithm:

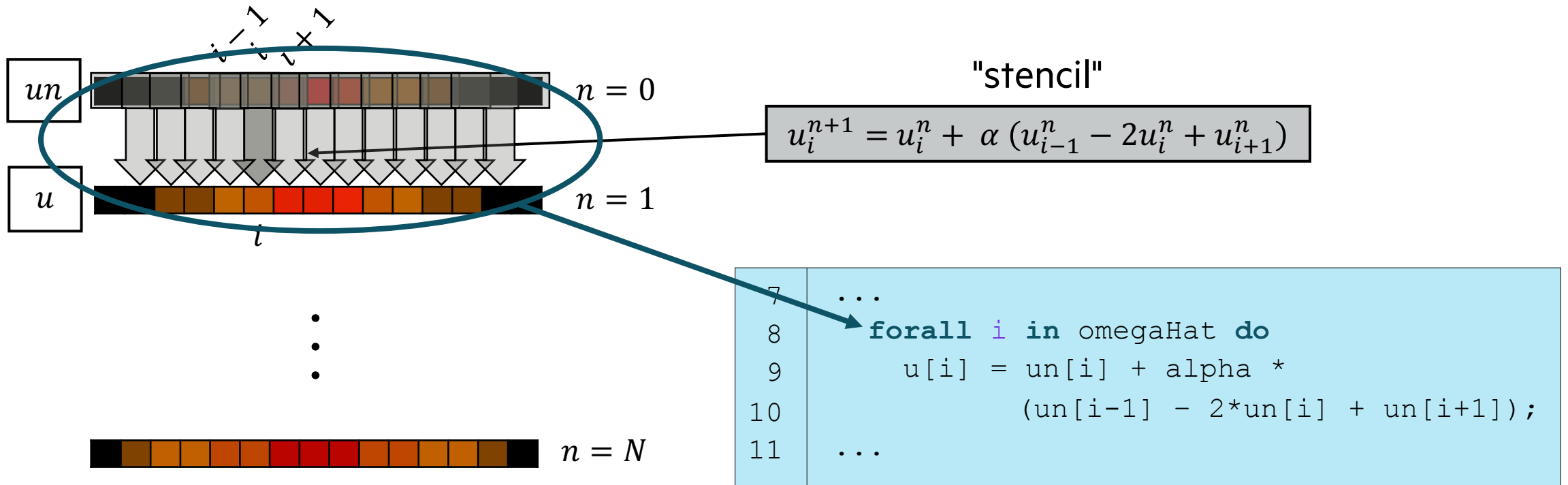
- define Ω to be a set of discrete points along the x-axis
- define $\hat{\Omega}$ over the same points, excluding the boundaries
- define an array u to over Ω
- set some initial conditions
- create a temporary copy of u , named un
- for N timesteps:
 - (1) swap u and un
 - (2) compute u in terms of un over $\hat{\Omega}$

```
1  const omega = {0..<nx},
2      omegaHat = omega.expand(-1);
3  var u: [omega] real = 1.0;
4  u[nx/4..3*nx/4] = 2.0;
5  var un = u;
6  for 1..N {
7      un <=> u;
8      forall i in omegaHat do
9          u[i] = un[i] + alpha *
10              (un[i-1] - 2*un[i] + un[i+1]);
11  }
```

1D HEAT EQUATION EXAMPLE

This pattern is often referred to as a **Stencil Computation**

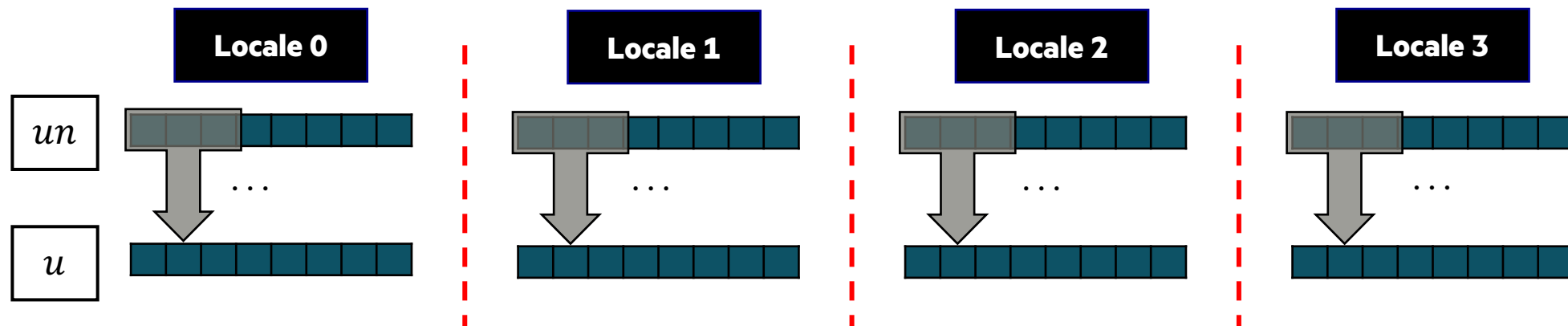
- The values in the array can be computed by applying a "stencil" to its previous state
- Note that in this case, the stencil can be applied to the entire array in parallel
 - each value in un depends strictly on values in u



HANDS ON: DISTRIBUTING THE 1D HEAT EQUATION

Imagine we want to simulate a very large domain

- We could use the Block distribution to distribute u and un across multiple locales
 - taking advantage of their memory and compute resources



Look at **heat-1D-block.chpl** and fill in the blanks to make the arrays block-distributed

Hint | Define a block-distributed domain:

```
use BlockDist;  
...  
const myBlockDom = blockDist.createDomain({1..10});
```

HANDS ON: DISTRIBUTING THE 1D HEAT EQUATION

 heat-1D-block-solution.chpl

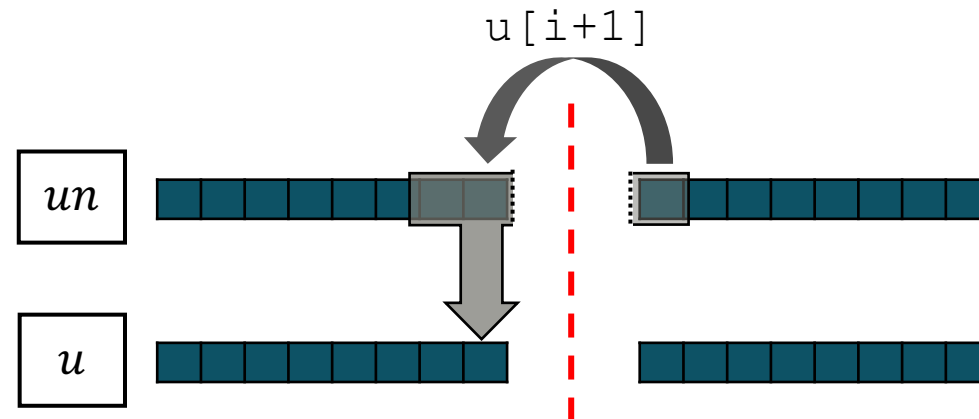
Solution: make 'omega' block-distributed:

```
omega = blockDist.createDomain({0.. $\text{nx}$ });
```

Why does this work?

- 'omegaHat' inherits 'omega's distribution
- 'u' is block-distributed
- 'un' inherits 'u's domain (and distribution)
- 'omegaHat' invokes 'blockDist's parallel/distr. iterator
 - the body of the loop is automatically split across multiple tasks on each locale
- Communication occurs automatically when a loop references a value stored on a remote locale

```
1  const omega =  
2      blockDist.createDomain({0.. $\text{nx}$ }),  
3      omegaHat = omega.expand(-1);  
4  var u: [omega] real = 1.0;  
5  u[nx/4.. $3 \times \text{nx}/4$ ] = 2.0;  
6  var un = u;  
7  for 1.. $N$  {  
8      un <=> u;  
9      forall i in omegaHat do  
10         u[i] = un[i] + alpha *  
11             (un[i-1] - 2*un[i] + un[i+1]);  
12 }
```



HEAT 2D EXAMPLE WITH COMMDIAGNOSTICS (HANDS ON)

2D HEAT EQUATION EXAMPLE

2D and 3D stencil codes are more common and practical

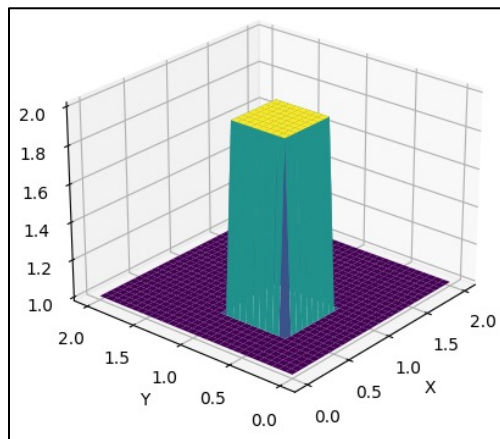
- They also present more interesting considerations for parallelization and distribution

2D heat / diffusion PDE:

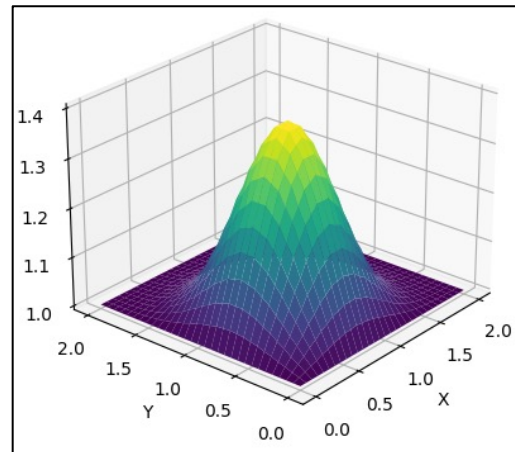
$$\frac{\partial u}{\partial t} = \alpha \Delta u = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$

Discretized (finite-difference) form:

$$u_{i,j}^{n+1} = u_{i,j}^n + \alpha (u_{i+1,j}^n + u_{i-1,j}^n - 4u_{i,j}^n + u_{i,j+1}^n + u_{i,j-1}^n)$$



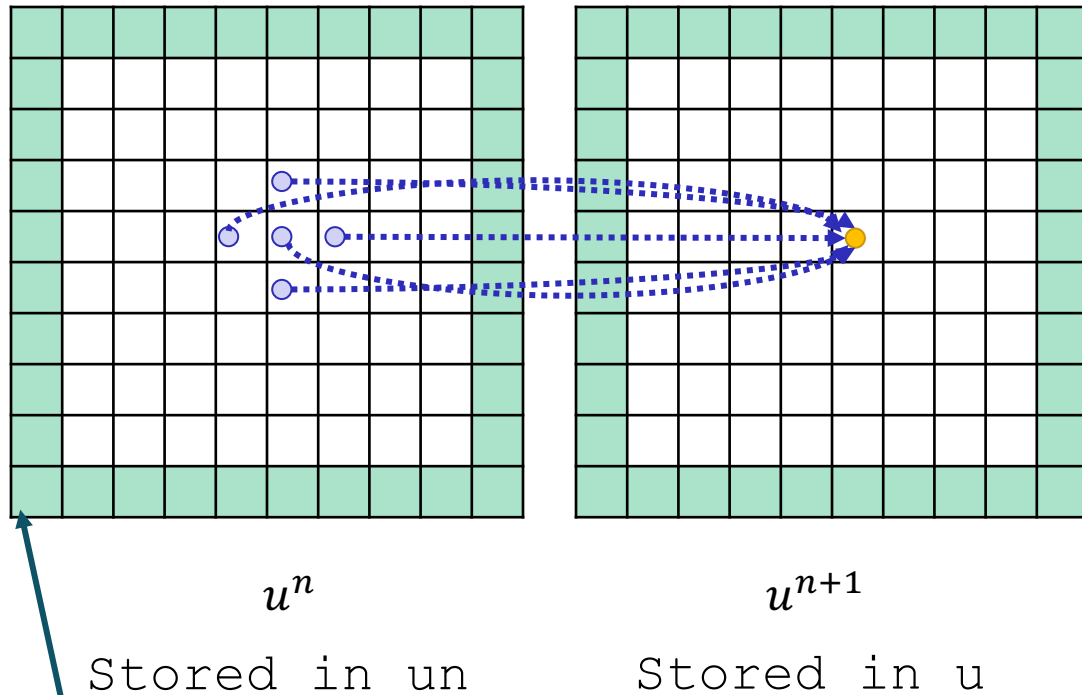
$n = 0$



$n = N$

```
1  const omega = {0.. $\text{nx}$ , 0.. $\text{ny}$ },  
2      omegaHat = omega.expand(-1);  
3  var u: [omega] real = 1.0;  
4  u[nx/4.. $3 \times \text{nx}/4$ ] = 2.0;  
5  var un = u;  
6  for 1.. $N$  {  
7      un <=> u  
8      forall (i, j) in omegaHat do  
9          u[i, j] = un[i, j] + alpha * (  
10              un[i-1, j] + un[i, j-1] +  
11              un[i+1, j] + un[i, j+1] -  
12              4 * un[i, j]);  
13  }
```

PARALLEL 2D HEAT EQUATION



- This computation uses a "5 point stencil"
- Each point in 'u' can be computed in parallel
 - this is accomplished using a 'forall' loop

```
7 ...  
8   forall (i, j) in omegaHat do  
9       u[i, j] = un[i, j] + alpha * (  
10           un[i-1, j] + un[i, j-1] +  
11           un[i+1, j] + un[i, j+1] -  
12           4 * un[i, j]);  
13 ...
```

$$u_{i,j}^{n+1} = u_{i,j}^n + \alpha(u_{i-1,j}^n + u_{i,j-1}^n + u_{i+1,j}^n + u_{i,j+1}^n - 4u_{i,j}^n)$$

BLOCK DISTRIBUTED & PARALLEL 2D HEAT EQUATION

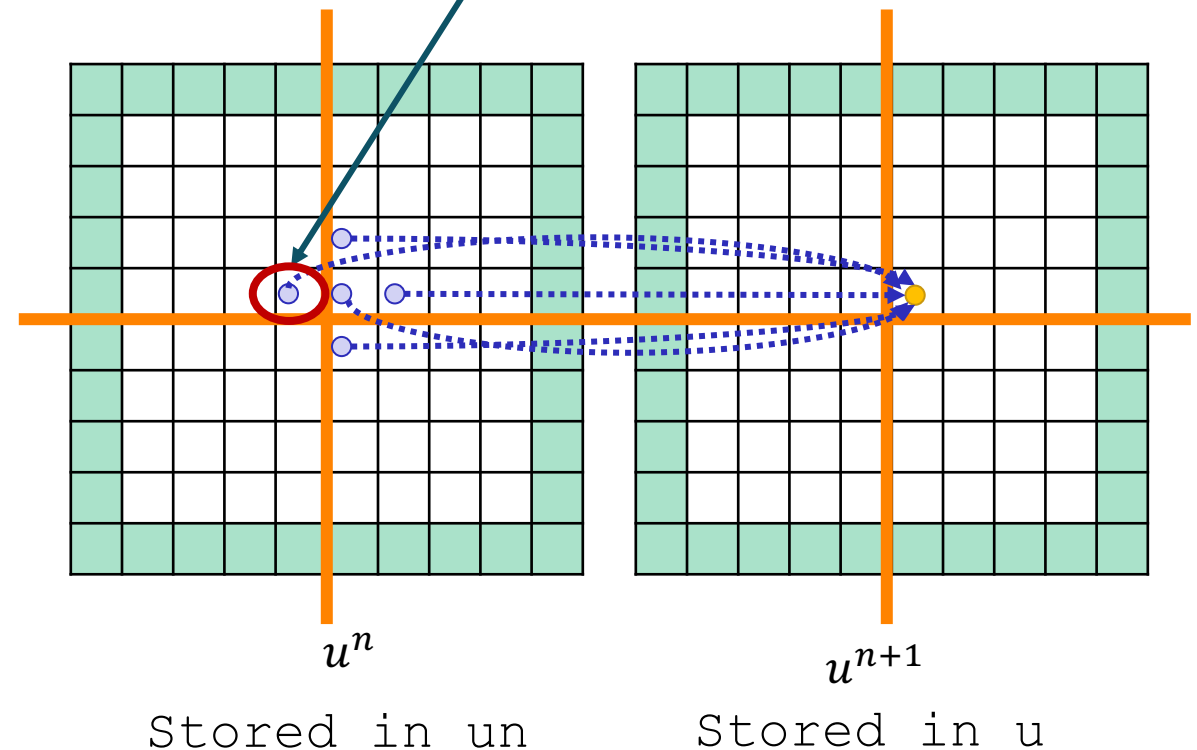
- Declaring distributed domains with the block distribution

```
const Omega = blockDist.createDomain(0.. $nx$ , 0.. $ny$ ),  
      OmegaHat = Omega.expand(-1);
```

- Distributed & Parallel loop over 'OmegaHat'

```
for 1.. $nt$  {  
  u <=> un;  
  
  forall (i, j) in OmegaHat do  
    u[i, j] = un[i, j] + alpha * (  
      un[i-1, j] + un[i, j-1] +  
      un[i+1, j] + un[i, j+1] -  
      4 * un[i, j]);  
}
```

Array access across locale boundaries automatically invokes communication



STENCIL DISTRIBUTED & PARALLEL 2D HEAT EQUATION

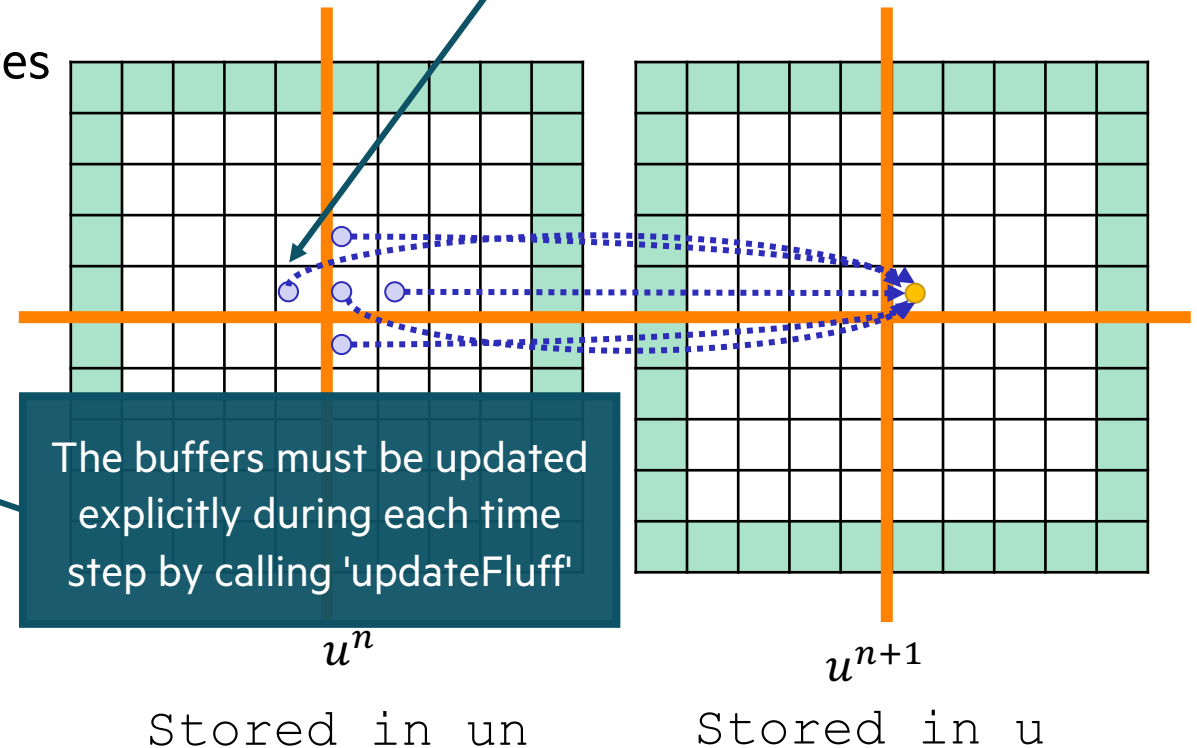
- Declaring distributed domains with the stencil distribution

```
const Omega = stencilDist.createDomain(  
    {0..  
nx, 0..  
ny}, fluff=(1,1)),  
OmegaHat = Omega.expand(-1);
```

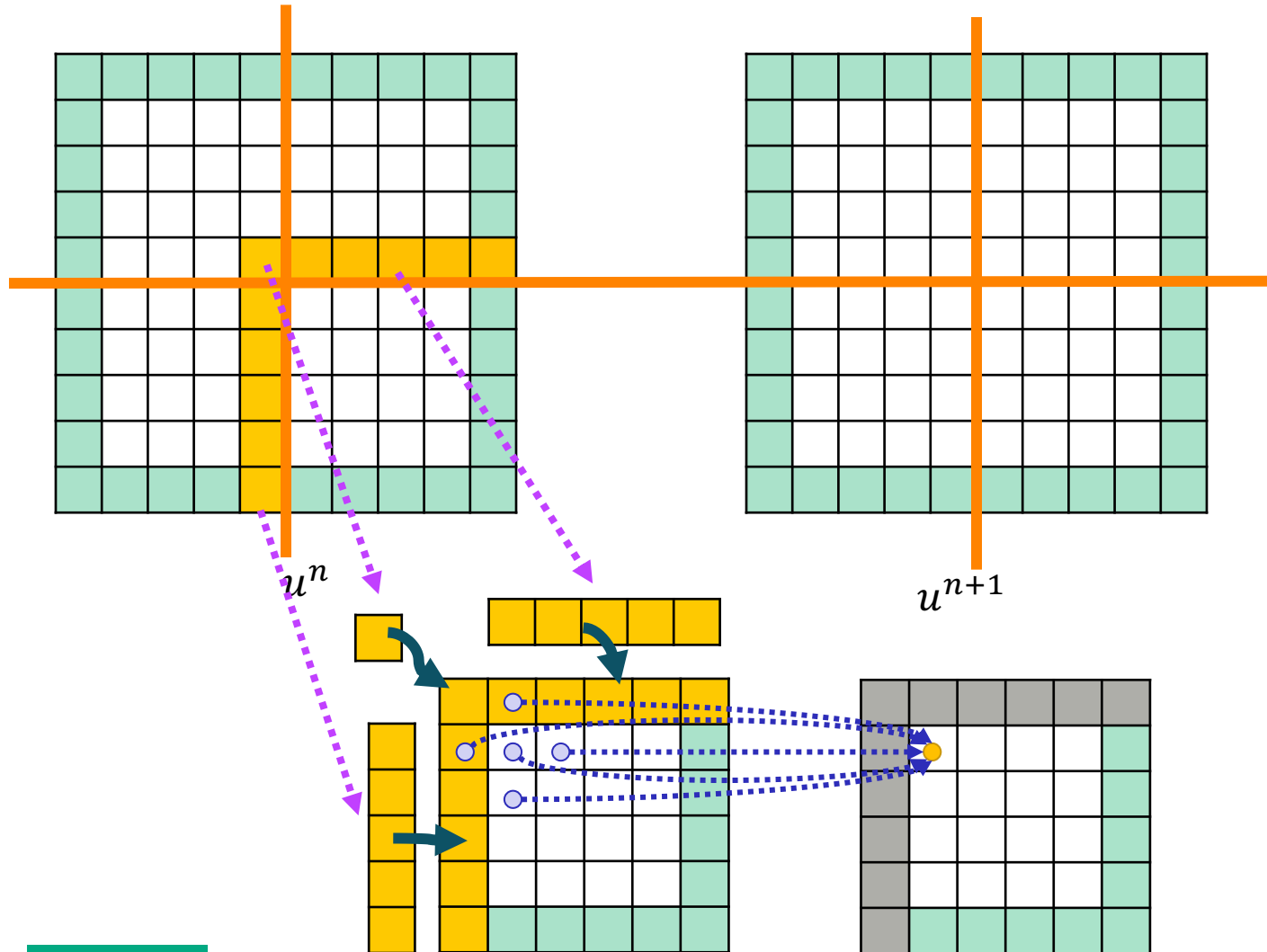
- Distributed & Parallel loop including buffer updates

```
for 1..  
nt {  
    u <=> un;  
    un.updateFluff();  
    forall (i, j) in OmegaHat do  
        u[i, j] = un[i, j] + alpha * (  
            un[i-1, j] + un[i, j-1] +  
            un[i+1, j] + un[i, j+1] -  
            4 * un[i, j]);  
}
```

Array access across locale boundaries (within the fluff region) results in a local buffer access — no communication is required



STENCIL DISTRIBUTED & PARALLEL 2D HEAT EQUATION



- Each locale owns a region of the array surrounded by a "fluff" (buffer) region
- Calling 'updateFluff' copies values from neighboring regions of the array into the local buffered region
- Subsequent accesses of those values result in a local memory access, rather than a remote communication

COMM DIAGNOSTICS

The 'CommDiagnostics' module provides functions for tracking comm between locales

- the following is a common pattern:

```
use CommDiagnostics;
...
startCommDiagnostics();
potentiallyCommHeavyOperation();
stopCommDiagnostics();
...
printCommDiagnosticsTable();
```

- which results in a table summarizing comm counts between the **start** and **stop** calls, e.g.,

locale	get	put	execute_on	execute_on_nb
-----:	--:	--:	-----:	-----:
0	10	0	6	12
1	105	5	0	0
2	105	4	0	0
3	105	7	0	0

- Compiling with '--no-cache-remote' before collecting comm diagnostics is recommended



HANDS ON: HEAT 2D COMM DIAGNOSTICS RESULTS

heat-2D-block.chpl, heat-2D-stencil.chpl

- Comparing comm diagnostics for:

- heat-2D-block.chpl
- heat-2D-stencil.chpl

- *Compilation:*

```
chpl heat-2D-block.chpl --fast  
      --no-cache-remote -sRunCommDiag=true
```

```
chpl heat-2D-stencil.chpl -fast  
      --no-cache-remote -sRunCommDiag=true
```

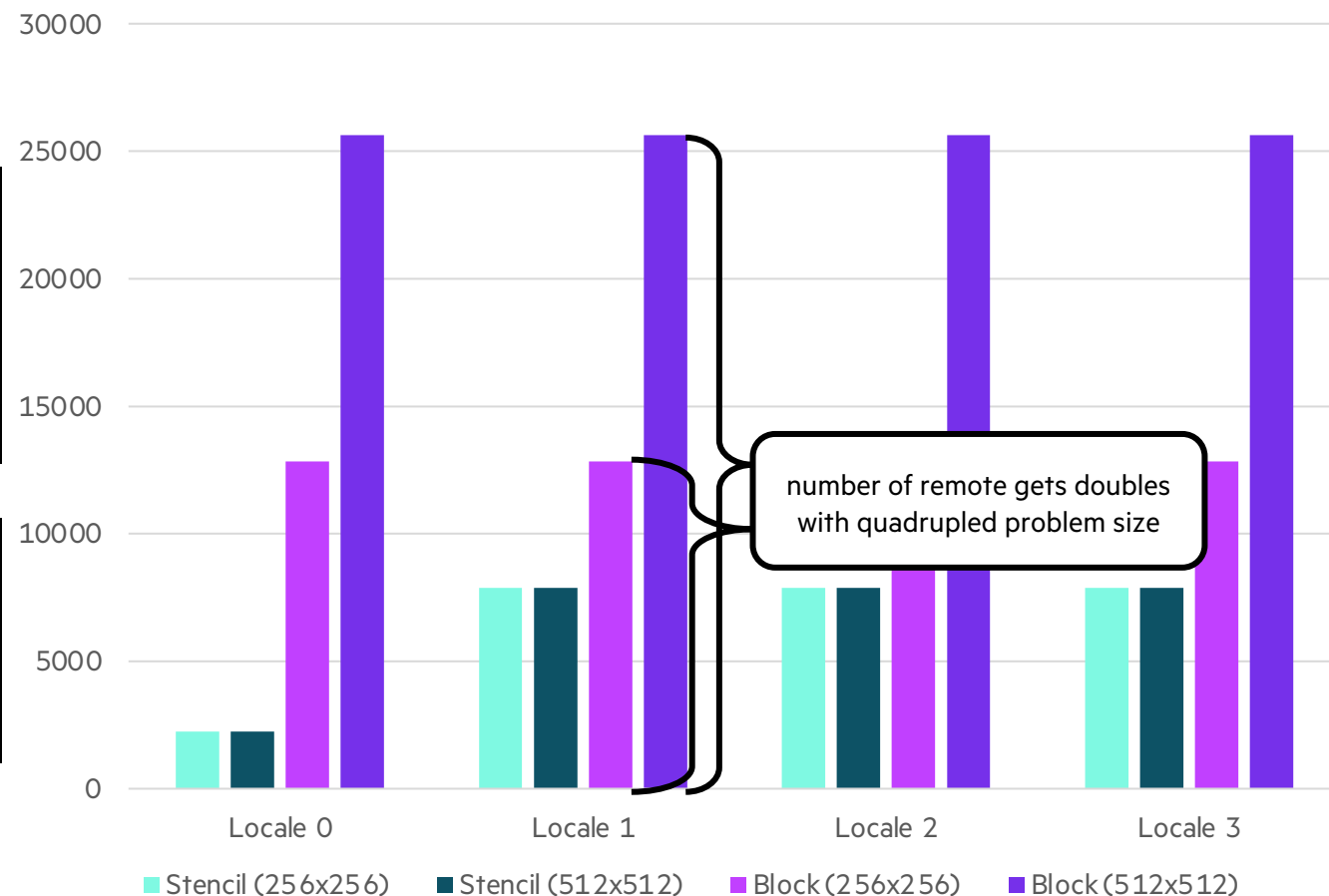
- *Execution:*

```
./heat-2D-block -nl4 --nx=256 --ny=256
```

```
./heat-2D-stencil -nl4 --nx=512 --ny=512
```

- **Block:** number of gets scales with size
- **Stencil:** static number of gets per iteration

Number of Gets on 4 Locales – Block vs. Stencil



OUTLINE: PARALLELISM IN CHAPEL

- Recall processing files in parallel
- Data parallelism concepts and examples including multi-locale parallelism with distributions
- Domains
- Forall Loops
- Domain Distributions
- Using a Different Domain Distribution
- Implicit Communication: Remote writes/Puts and Reads/Gets
- Parallelizing a 1D heat diffusion solver (Hands On)
- Heat 2D example with CommDiagnostics (Hands On)



SUMMARIZING WHAT WE LEARNED IN SESSION 3

- Data parallelism session
 - Provides shared memory and distributed memory parallelism
 - Distributions like block and cyclic can be applied to arrays of any dimension
 - Main control abstraction is the 'forall' loop
 - 'forall' loop uses default iterator over provided array or domain, but can use own iterator
 - This is an example of multi-resolution design in Chapel, i.e., the 'forall' loop is mapped down to lower-level abstractions like 'coforall'
 - CommDiagnostics module can be used to observe the number of remote puts/writes and gets/reads at runtime



LEARNING OBJECTIVES FOR TODAY'S CHAPEL TUTORIAL

- Familiarity with the Chapel execution model including how to run codes in parallel on a single node, across nodes, and both
- Learn Chapel concepts by compiling and running provided code examples
 - ✓ Serial code using map/dictionary, (k-mer counting from bioinformatics)
 - ✓ Parallelism and locality in Chapel
 - ✓ Distributed parallelism and 1D arrays, (processing files in parallel)
 - ✓ Chapel basics in the context of an n-body code
 - ✓ Distributed parallelism and 2D arrays, (heat diffusion problem)
- How to parallelize histogram
- Using CommDiagnostics for counting remote reads and writes
- Chapel and Arkouda best practices including avoiding races and performance gotchas
- Where to get help and how you can participate in the Chapel community



ONE DAY CHAPEL TUTORIAL

- 9-10:30: Getting started using Chapel for parallel programming
- 10:30-10:45: break
- 10:45-12:15: Chapel basics in the context of the n-body example code
- 12:15-1:15: lunch
- 1:15-2:45: Distributed and shared-memory parallelism especially w/arrays (data parallelism)
- 2:45-3:00: break
- 3:00-4:30: More parallelism including for asynchronous parallelism (task parallelism)
- 4:30-5:00: Wrap-up including gathering further questions from attendees



CHAPEL RESOURCES

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


- (points to all other resources)

Social Media:

- Twitter: [@ChapelLanguage](https://twitter.com/ChapelLanguage)
- Facebook: [@ChapelLanguage](https://facebook.com/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>



The Chapel Parallel Programming Language

What is Chapel?

Chapel is a programming language designed for productive parallel computing at scale.

Why Chapel? Because it simplifies parallel programming through elegant support for:

- **distributed arrays** that can leverage thousands of nodes' memories and cores
- **a global namespace** supporting direct access to local or remote variables
- **data parallelism** to trivially use the cores of a laptop, cluster, or supercomputer
- **task parallelism** to create concurrency within a node or across the system

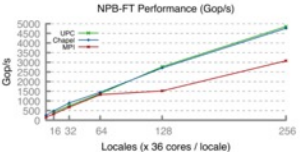
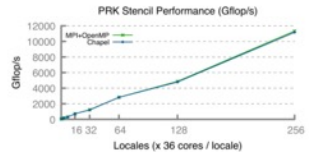
Chapel Characteristics

- **productive:** code tends to be similarly readable/writable as Python
- **scalable:** runs on laptops, clusters, the cloud, and HPC systems
- **fast:** performance competes with or beats C/C++ & MPI & OpenMP
- **portable:** compiles and runs in virtually any *nix environment
- **open-source:** hosted on GitHub, permissively licensed

New to Chapel?

As an introduction to Chapel, you may want to...

- watch an [overview talk](#) or browse its [slides](#)
- read a [blog-length](#) or [chapter-length](#) introduction to Chapel
- learn about [projects powered by Chapel](#)
- check out [performance highlights](#) like these:



PRK Stencil Performance (Gflop/s)

NPB-FT Performance (Gop/s)

- browse [sample programs](#) or [learn](#) how to write distributed programs like this one:

```
use CyclicDist;           // use the Cyclic distribution library
config const n = 100;     // use --n=<val> when executing to override this default

forall i in {1..n} dmapped Cyclic(startIdx=1) do
  writeln("Hello from iteration ", i, " of ", n, " running on node ", here.id);
```



Hewlett Packard
Enterprise

ONE-DAY CHAPEL TUTORIAL

SESSION 4: MORE PARALLELISM

Chapel Team

October 16, 2023

ONE DAY CHAPEL TUTORIAL

- 9-10:30: Getting started using Chapel for parallel programming
- 10:30-10:45: break
- 10:45-12:15: Chapel basics in the context of the n-body example code
- 12:15-1:15: lunch
- 1:15-2:45: Distributed and shared-memory parallelism especially w/arrays (data parallelism)
- 2:45-3:00: break
- 3:00-4:30: More parallelism including for asynchronous parallelism (task parallelism)
- 4:30-5:00: Wrap-up including gathering further questions from attendees



OUTLINE: MORE PARALLELISM AND SOME BEST PRACTICES

- Spectrum of Chapel loops
- Task intents including reduce intents, and atomics
- Parallelizing histogram (Hands On)
- Story of index gather parallelization
- Other parallel constructs: 'cobegin', 'begin', 'sync',
- Avoiding races with task intents and task-private variables
- Performance gotchas
- Memory in Chapel and Arkouda
- Using CommDiagnostics



SPECTRUM OF CHAPEL LOOPS

SPECTRUM OF CHAPEL FOR-LOOP STYLES

See <https://chapel-lang.org/docs/primers/loops.html> for more details on loops.

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)

IMPLICIT LOOPS: PROMOTION OF SCALAR SUBROUTINES & ARRAY OPS

- 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*z;  
}
```

- 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*B;
```



TASK INTENTS INCLUDING REDUCE INTENTS

USING TASK INTENTS IN LOOPS

Procedure argument intents (<https://chapel-lang.org/docs/primers/procedures.html?highlight=intents#argument-intents>)

- Tell how to pass a symbol actual argument into a formal parameter
- Default intent is 'const', which means formal can't be modified in procedure body
- 'ref' means formal can be changed AND that change will be visible elsewhere, e.g., at the callsite
- Others: 'in', 'out', and 'inout' refer to copying the actual argument in, the formal out, or both

Task intents in loops

- Similar to argument intents in syntax and philosophy
- Also have a 'reduce' intent similar to OpenMP
- 'reduce' intent means each task has its own copy and specified operation like '+' will combine at end of loop

Design principles

- Avoid common race conditions
- Avoid copies of (potentially) large data structures



TASK INTENTS IN FORALL LOOPS: SCALARS



04-task-intents-forall.chpl

```
var sum: real;  
forall i in 1..n do  
    sum += computeMyResult(i);
```

Default intent of scalars is 'const in' so this is illegal (and avoids a race)

```
var sum: real;  
forall i in 1..n with (ref sum) do  
    sum += computeMyResult(i);
```

With 'ref' intent, we are requesting a race

```
var sum: real;  
forall i in 1..n with (+ reduce sum) do  
    sum += computeMyResult(i);
```

Override default intent so that each task accumulates its own copy. On loop exit, all tasks combine their results into original 'sum'

FORALL INTENT EXAMPLES: ARRAYS



04-task-intents-forall.chpl

```
var bucketCount: [0.. $m$ ] real;  
forall i in 1.. $n$  with (ref bucketCount) do  
    bucketCount[i %  $m$ ] += 1;
```

'ref' intent avoids array copies,
but can result in data races

```
var bucketCount: [0.. $m$ ] real;  
forall i in 1.. $n$  with (in bucketCount) do  
    bucketCount[i %  $m$ ] += 1;
```

*'in' intent will result in
each task having its own
copy*

```
var bucketCount : [0.. $m$ ] real;  
forall i in 1.. $n$  with (+ reduce bucketCount) do  
    bucketCount[i %  $m$ ] += 1;
```

*'reduce' intent will result in
each task having own copy,
but then on loop exit tasks
combine their results into the
original 'bucketCount' variable*

ATOMIC VARIABLES

ATOMIC VARIABLES

Meaning

- Atomic means 'indivisible'
- An atomic operation is indivisible.
- A thread of computation cannot interfere with another thread that is doing an atomic operation.

Atomic Type Semantics in Chapel

- Supports operations on variable atomically w.r.t. other tasks
- Based on C/C++ atomic operations

Example: Counting barrier

```
var count: atomic int, done: atomic bool;  
  
proc barrier(numTasks) {  
    const myCount = count.fetchAdd(1);  
    if (myCount < numTasks - 1) then  
        done.waitFor(true);  
    else  
        done.testAndSet();  
}
```

ARRAY OF ATOMIC

```
var bucketCount: [0..<m] atomic real;  
forall i in 1..n with (ref bucketCount) do  
    bucketCount[i % m].add(1);
```

Make the 'bucketCount' array
contain 'atomic real's

Use the atomic 'add' operation

```
var bucketCount: [0..<m] atomic real;  
forall i in 1..n do  
    bucketCount[i % m].add(1);
```

Can leave off 'ref' intent, since that
is the default for 'atomic' types

PARALLELIZING HISTOGRAM (HANDS ON)

HANDS ON: PARALLELIZING HISTOGRAM

Goals

- Parallelize a program that computes a histogram using reductions
- Parallelize it using an array of atomic integers
- Compare the performance of both versions versus each other and the serial version

Parallelize 'histogram-serial.chpl' using a 'forall' loop and a 'reduction' intent

1. Copy 'histogram-serial.chpl' into 'histogram-reduce.chpl'
2. Parallelize the serial 'for' loop using concepts from '04-task-intents.chpl'

Parallelize 'histogram-serial.chpl' using an array of atomic integers

1. Copy 'histogram-serial.chpl' into 'histogram-atomic.chpl'
2. Parallelize the serial 'for' loop using concepts from '04-atomic-type.chpl'

Compare the performance of all three

```
./histogram-serial --numNumbers=100000000 --printRandomNumbers=false --useRandomSeed=false  
./histogram-reduce --numNumbers=100000000 --printRandomNumbers=false --useRandomSeed=false  
./histogram-atomic --numNumbers=100000000 --printRandomNumbers=false --useRandomSeed=false
```

HANDS ON EXTRA CREDIT: PARALLELIZE N-BODY



nbody.chpl

Goals and Questions to Answer

- Parallelize as many loops in n-body as possible
- Determine when a 'reduce' intent or 'atomic' variable type is needed
- How can you check if you got the same answer?
- Is it possible for floating-point roundoff differences to change what the answers are slightly? For which loops?
- Did you get a performance improvement by doing the parallelization?



ATOMIC METHODS

- `read() : t` return current value
- `write(v : t)` store *v* as current value
- `exchange(v : t) : t` store *v*, returning previous value
- `compareExchange(old : t, new : t) : bool`
store *new* iff previous value was *old*; returns true on success
- `waitFor(v : t)` wait until the stored value is *v*
- `add(v : t)` add *v* to the value atomically
- `fetchAdd(v : t)` same, returning pre-sum value
(*sub, or, and, xor* also supported similarly)
- `testAndSet()` like *exchange(true)* for atomic bool
- `clear()` like *write(false)* for atomic bool



REDUCTIONS IN CHAPEL

- Recall the following snippet of code from the histogram exercise

```
// verify number of items in histogram is equal to number of random
// numbers and output timing results
if + reduce histogram != numNumbers then
  halt("Number of items in histogram does not match number of random numbers");
writeln("Histogram computed in ", timer.elapsed( ), " seconds\n");
```

- Standard reductions supported by default:

```
+, *, min, max, &, |, &&, ||, minloc, maxloc, ...
```

- Reductions can reduce arbitrary iterable expressions:

```
const total = + reduce Arr,
  factN = * reduce 1..n,
  biggest = max reduce (forall i in myIter() do foo(i));
```

STORY OF INDEX GATHER PARALLELIZATION

STORY ABOUT PARALLELIZING INDEX GATHER

- Computation in Bale that gathers spread out data into a packed array

```
for i in D do
    Dest[i] = Src[Inds[i]];
```

- Parallelize over threads using a 'forall'

```
forall i in D with (ref Dest) do
    Dest[i] = Src[Inds[i]];
```

- Parallelize by distributing the D2 domain and using a 'forall'

```
const D = blockDist.createDomain({0..numUpdates-1});
var Inds: [D] int;

forall i in D with (ref Dest) do
    Dest[i] = Src[Inds[i]];
```



CHAPEL TENDS TO BE COMPACT, CLEAN, AND FAST (BALE INDEX-GATHER)

Exstack version

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

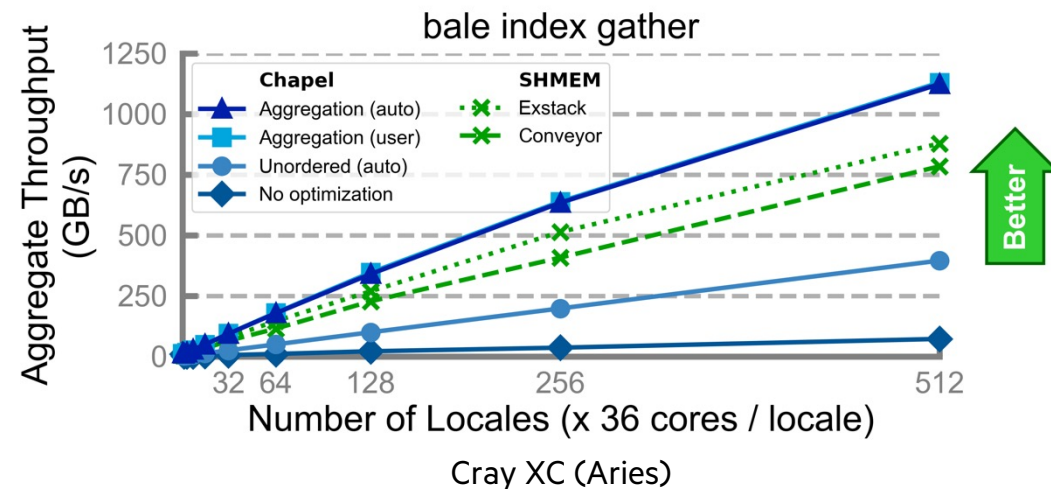
Conveyors version

```
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;
}
```



Manually Tuned Chapel version (using explicit aggregator type)

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

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

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

OTHER PARALLEL CONSTRUCTS

DEFINING OUR TERMS

Task: a unit of computation that can/should execute in parallel with other tasks

Thread: a system resource that executes tasks

- not exposed in the language
- occasionally exposed in the implementation

Task Parallelism: a style of parallel programming in which parallelism is driven by programmer-specified tasks

(in contrast with):

Data Parallelism: a style of parallel programming in which parallelism is driven by computations over collections of data elements or their indices



PARALLELISM SUPPORTED BY CHAPEL

Synchronous task parallelism

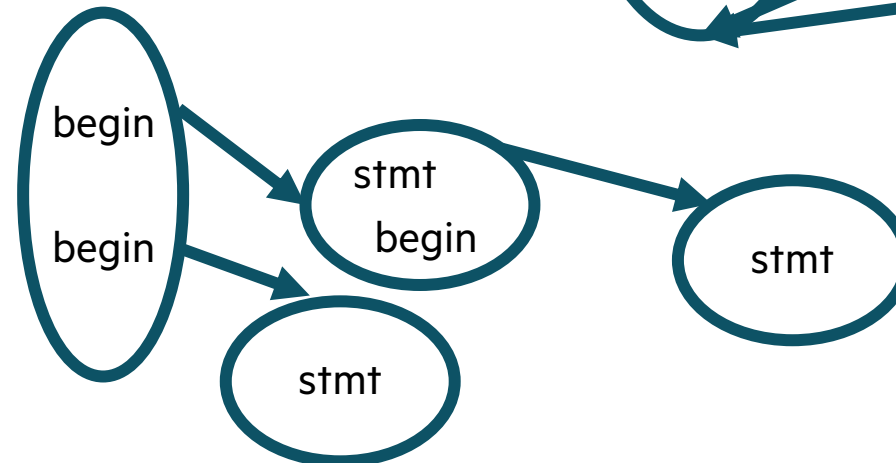
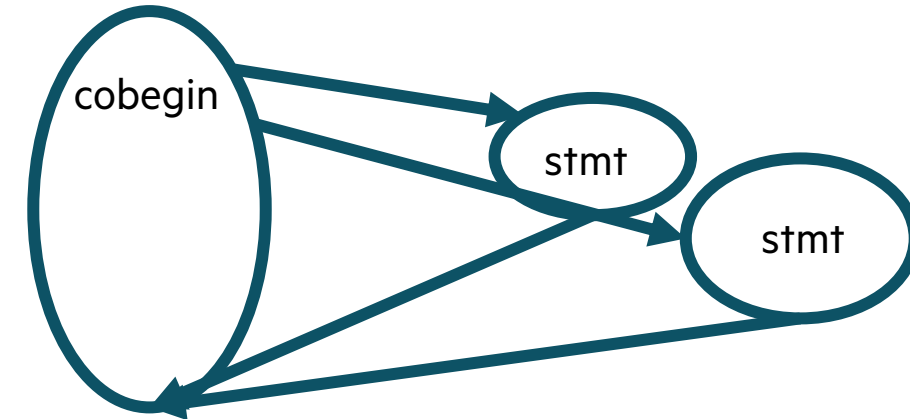
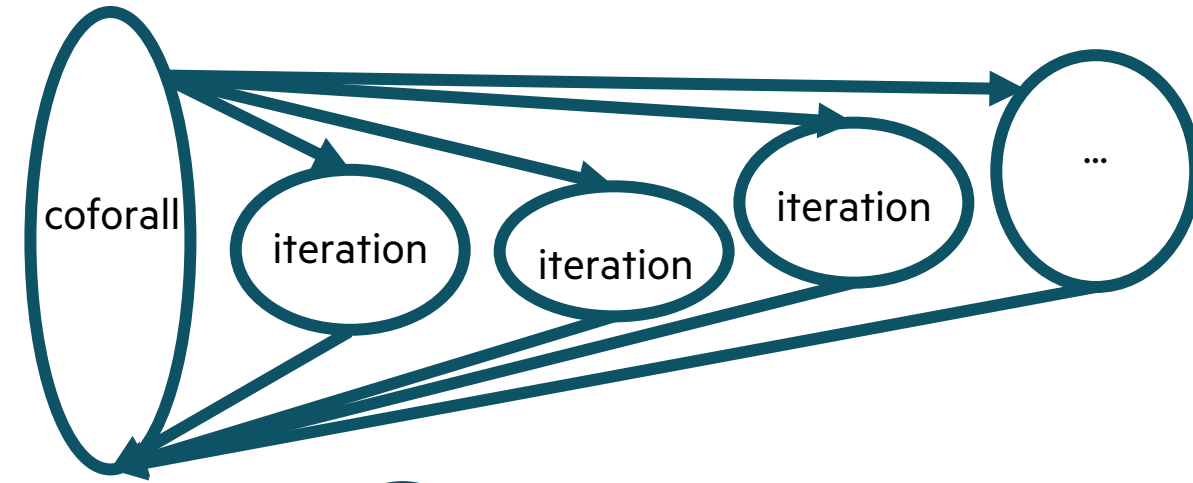
- 'coforall', parallel task per iteration
- 'cobegin', executes all statements in block in parallel

Asynchronous task parallelism

- 'begin', creates an asynchronous task
- 'sync' and 'atomic' vars for task coordination

Higher-level parallelism abstractions

- 'forall', data parallelism and iterator abstraction
- 'foreach', SIMD parallelism
- 'scan', operations such as cumulative sums
- 'reduce', operations such as summation



PARALLELISM SUPPORTED BY CHAPEL



04-parallelism-in-chapel.chpl

Synchronous task parallelism

- 'coforall', parallel task per iteration
- 'cobegin', executes all statements in block in parallel

Asynchronous task parallelism

- 'begin', creates an asynchronous task
- 'sync' and 'atomic' vars for task coordination

Higher-level parallelism abstractions

- 'forall', data parallelism and iterator abstraction
- 'foreach', SIMD parallelism
- 'scan', operations such as cumulative sums
- 'reduce', operations such as summation

```
coforall loc in Locales do on loc { /* ... */ }
coforall tid in 0..<numTasks { /* ... */ }

cobegin { doTask0(); doTask1(); ... doTaskN(); }

var x : atomic int = 0, y : sync int;
sync {
    begin x.add(1);
    begin y.writeEF(1);
    begin x.sub(1);
    begin { y.readFE(); y.writeEF(0); }
}
assert(x.read() == 0);
assert(y.readFE() == 0);

var n = [i in 1..10] i*i;
forall x in n do x += 1;

var nPartialSums = + scan n;
var nSum = + reduce n;
```

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 .readFE(), .writeEF()
```

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

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



USE OF PARALLELISM IN SOME APPLICATIONS AND BENCHMARKS

Application	Distributed 'coforall'	Threaded 'coforall'	Asynchronous 'begin'	'cobegin'	sync or atomic	forall	scan
Arkouda	✓	✓			✓	✓	✓
CHAMPS	✓	✓			✓		
ChOp	✓		✓		✓	✓	
ParFlow						✓	
Coral Reef	✓	✓		✓		✓	



TASK PARALLELISM: BEGIN STATEMENTS

```
// create a fire-and-forget task for a statement  
begin writeln("hello world");  
writeln("goodbye");
```

Possible outputs:

```
hello world  
goodbye
```

```
goodbye  
hello world
```



JOINING SUB-TASKS: SYNC-STATEMENTS

Syntax

```
sync-statement:  
  sync stmt
```

Definition

- Executes *stmt*
- Waits for all *dynamically-scoped* begins to complete

Examples

```
sync {  
  for i in 1..numConsumers {  
    begin consumer(i);  
  }  
  producer();  
}
```

```
proc search(node: TreeNode) {  
  if (node != nil) {  
    begin search(node.left);  
    begin search(node.right);  
  }  
}  
sync { search(root); }
```

TASK PARALLELISM: COBEGIN STATEMENTS

```
// create a task per child statement  
cobegin {  
    producer(1);  
    producer(2);  
    consumer(1);  
} // implicit join of the three tasks here
```



COBEGINS/SERIAL BY EXAMPLE: QUICKSORT



'cobegin' will start both
'quickSort' calls in parallel
unless the number of running
tasks would exceed the
available HW parallelism

```
proc quickSort(arr: [?D],
               low: int = D.low,
               high: int = D.high) {
  if high - low < 8 {
    bubbleSort(arr, low, high);
  } else {
    const pivotLoc = partition(arr, low, high);
    serial (here.runningTasks() > here.maxTaskPar) do
      cobegin {
        quickSort(arr, low, pivotLoc-1);
        quickSort(arr, pivotLoc+1, high);
      }
  }
}
```

TASK PARALLELISM: COFORALL LOOPS

```
// create a task per iteration  
coforall t in 0..#numTasks {  
    writeln("Hello from task ", t, " of ", numTasks);  
} // implicit join of the numTasks tasks here  
  
writeln("All tasks done");
```

Sample output:

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



COMPARISON OF BEGIN, COBEGIN, AND COFORALL

begin:

- Use to create a dynamic task with an unstructured lifetime
- “fire and forget” (or at least “leave running for awhile”)

cobegin:

- Use to create a related set of heterogeneous tasks
...or a small, fixed set of homogenous tasks
- The parent task depends on the completion of the tasks

coforall:

- Use to create a fixed or dynamic # of homogenous tasks
- The parent task depends on the completion of the tasks

Note: All these concepts can be composed arbitrarily



SYNCHRONIZATION VARIABLES

TASK PARALLELISM: DATA-DRIVEN SYNCHRONIZATION

- **sync variables:** store full-empty state along with value
- **atomic variables:** support atomic operations
 - e.g., compare-and-swap; atomic sum, multiply, etc.
 - similar to C/C++



BOUNDED BUFFER PRODUCER/CONSUMER EXAMPLE



04-bounded-buffer-with-sync.chpl

```
// 'sync' types store full/empty state along with value
var buff: [0..#buffersize] sync real;

begin producer();
      consumer();

proc producer() {
  var i = 0;
  for ... {
    i = (i+1) % buffersize;
    buff[i].writeEF( ... ); // wait for empty, write, leave full
  } }

proc consumer() {
  var i = 0;
  while ... {
    i = (i+1) % buffersize;
    ...buff[i].readFE()...; // wait for full, read, leave empty
  } }
```

Syntax

```
sync-type:  
  sync type
```

Semantics

- Stores *full/empty* state along with normal value
- Initially *full* if initialized, *empty* otherwise

Examples: Critical sections and futures

```
var lock: sync bool;  
  
lock.writeEF(true);  
critical();  
lock.readFE();
```

```
var future: sync real;  
  
begin future.writeEF(compute());  
res = computeSomethingElse();  
useComputedResults(future.readFE(), res);
```

SYNCHRONIZATION VARIABLE METHODS

- **readFE** () : t block until *full*, leave *empty*, return value
- **readFF** () : t block until *full*, leave *full*, return value
- **writeEF** (v : t) block until *empty*, set value to v, leave *full*



COMPARISON OF SYNCHRONIZATION TYPES

sync:

- Best for producer/consumer style synchronization
 - “this task should block until something happens”
 - use single for write-once values

atomic:

- Best for uncoordinated accesses to shared state
 - “these tasks are unlikely to interfere with each other, at least for very long...”



AVOIDING RACES WITH TASK INTENTS AND TASK PRIVATE VARIABLES

TASK INTENTS

- Tells how to “pass” variables from outer scopes to tasks
 - Similar to argument intents in syntax and philosophy
 - also adds a “reduce intent”, similar to OpenMP
 - Design principles:
 - “principle of least surprise”
 - avoid simple race conditions
 - avoid copies of (potentially) expensive data structures
 - support coordination via sync/atomic variables



TASK INTENT EXAMPLES



04-task-intents-coforall.chpl

```
var sum: real;  
coforall i in 1..n do  
    sum += computeMyResult(i);
```

Default task intent of scalars is 'const in' so this is illegal (and avoids a race)

```
var sum: real;  
coforall i in 1..n with (ref sum) do  
    sum += computeMyResult(i);
```

Use a 'ref' task intent for 'sum' variable. We've now requested a race.

```
var sum: real;  
coforall i in 1..n with (+ reduce sum) do  
    sum += computeMyResult(i);
```

Use a 'reduce' task intent. Per-task sums will be reduced on task exit.

```
var sum: atomic real;  
coforall i in 1..n do  
    sum.add(computeMyResult(i));
```

Default task intent of atomics is 'ref' so this is legal, meaningful, and safe

TASK-PRIVATE VARIABLES



- Task-parallel features support task-private variables easily

```
coforall i in 1..numTasks {  
    var mySum: real; // each task gets its own copy of mySum  
    for j in 1..n do  
        mySum += A[i][j];  
}
```

- Forall loops need special support for task-private variables

```
var oneSingleVariable: real;  
forall i in 1..n {  
    var onePerIteration: real;  
}
```



TASK-PRIVATE VARIABLES



- Task-parallel features support task-private variables easily

```
coforall i in 1..numTasks {  
    var mySum: real; // each task gets its own copy of mySum  
    for j in 1..n do  
        mySum += A[i][j];  
}
```

- Forall loops need special support for task-private variables

```
var oneSingleVariable: real;  
forall i in 1..n with (var onePerTask: real) {  
    var onePerIteration: real;  
}
```



TASK-PRIVATE VARIABLES



- Task-parallel features support task-private variables easily

```
coforall i in 1..numTasks {  
    var mySum: real; // each task gets its own copy of mySum  
    for j in 1..n do  
        mySum += A[i][j];  
}
```

- Forall loops need special support for task-private variables

```
var oneSingleVariable: real;  
forall i in 1..n with (var onePerTask = 3.14) {  
    var onePerIteration: real;  
}
```



PERFORMANCE / ARKOUDA ROADMAP

- Chapel best practices: General and for performance
 - Tips for compiling Arkouda faster
- Performance gotchas
- Memory in Chapel and Arkouda
- Stopwatches and benchmarks
- Using CommDiagnostics



CHAPEL BEST PRACTICES

CHAPEL BEST PRACTICES: GENERAL AND FOR PERFORMANCE

The three most common ways to build Chapel

- **‘quickstart’** configuration
 - Low performance, quickest build time, minimal dependency requirements
 - Not recommended for testing performance, not a fully-featured version of Chapel
- **‘CHPL_COMM=none’** local configuration
 - Fully featured and best performance when running on a non-distributed system (e.g., your laptop)
 - Can potentially hit scaling issues when extending to multi-locale, as communication does not factor in
 - When comparing performance against non-distributed code from other languages, typically preferred configuration
- **‘CHPL_COMM=gasnet’** multi-locale configuration
 - Enables multi-locale features, inserts code for remote accesses, works everywhere, but not always most optimized
 - Can be run on laptop for debugging purposes, but distribution is only simulated, so performance doesn’t mean much
- See <https://chapel-lang.org/docs/usingchapel/QUICKSTART.html> for more info

Make sure you are in the correct configuration for your system when testing performance

- Ask for help in the Chapel Discourse if you need help determining correct configuration!



TIPS FOR COMPILING ARKOUDA FASTER

Quick, 1-step compilation time improvements that all developers should be using

- 'export ARKOUDA_QUICK_COMPILE=1'
 - Disable optimizations, but performance will be worse (does not compile with '--fast')
 - Recommended when developing new features or running correctness tests
- 'export ARKOUDA_SKIP_CHECK_DEPS=1'
 - Skip compiling and running each of the Arkouda dependency tests when building Arkouda
 - Typically, the dependency tests only need to be run once per-machine, once you know they pass, they can be disabled
- Make sure that 'CHPL_DEVELOPER' is unset
 - If the 'chpl' compiler was built when 'CHPL_DEVELOPER' was set, this can have an adverse effect on compilation times
 - If this was set when 'chpl' was built, 'chpl' should be rebuilt without it (doesn't apply to brew installs)

Effectively use the modular build system, only compiling features necessary for what is being tested

- See <https://bears-r-us.github.io/arkouda/setup/MODULAR.html> for more info

See <https://github.com/Bears-R-Us/arkouda/issues/2073> for more tips on speeding up compilation



PERFORMANCE GOTCHAS

PERFORMANCE GOTCHAS

- Compile with ‘--fast’
- Locate bottlenecks using stopwatches to identify which portion of code is running slowly
- Check the slow portion of the code for...
 - ...algorithmic overheads or tight-loop complexity
 - ...excessive remote accesses (use ‘CommDiagnostics’ to assess)
 - Are arrays distributed that should be? Are all locales accessing a variable declared on locale 0? Can you use aggregators?
 - ...extraneous dynamic (i.e., class object) allocations (use ‘MemDiagnostics’ to assess)
 - Could class allocations be reused in loops? Could a record be used instead of a class?
- Can be informative to compare standalone Chapel program to equivalent C/C++ code for local codes
 - Is Chapel slower than the same code in other languages?



MEMORY IN CHAPEL AND ARKOUDA

MEMORY IN CHAPEL AND ARKOUDA

Arkouda's Memory Tracking

- Arkouda uses MemDiagnostics to estimate the total memory available and the total memory used
 - The maximum memory usage is set as 90% of Chapel's estimated available memory
- Only allocations larger than 1 MB are tracked
 - Tracking all allocations would have too large a performance impact

Most Common Memory Allocation Modes in Chapel

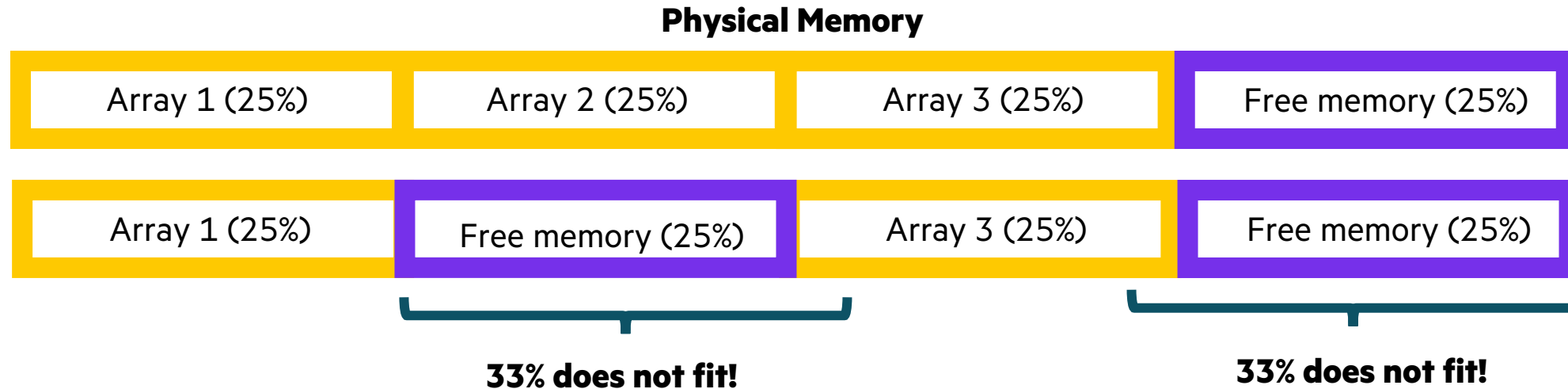
- **simple:** comm none, gasnet-everything
 - Fragmentation is handled completely by allocator, can use virtual memory and mmap for large arrays
- **fixed-heap:** gasnet-ibv, ofi-cxi
 - At program startup, a fixed segment of memory is allocated from comm layer
 - Allocator can only use provided memory region and not entire virtual address range



MEMORY IN CHAPEL AND ARKOUDA

Fragmentation

- Fragmentation can occur when allocating and freeing large blocks of memory
 - Common pattern in Arkouda



1. Allocate 3 arrays that are 25% of maximum memory each (25% memory available)
2. Free the second array (50% memory available)
3. Attempt to allocate an array that is 33% of maximum memory
 - Oh no! Our memory is fragmented, so we can't satisfy allocation, even though 50% of memory is available



STOPWATCHES AND BENCHMARKS

BENCHMARKS AND STOPWATCHES

Benchmarks

- The Arkouda repository runs a number of performance tests that time Arkouda operations nightly
 - See <https://chapel-lang.org/perf/arkouda/>
- Benchmarks are useful for tracking historical performance data and gauging overall performance

Stopwatches

- Stopwatches can be used to segment out portions of Chapel code and identify bottlenecks
- Rather than trying to guess what part of a function is slow, can isolate the code to optimize



USING COMMDIAGNOSTICS

USING COMM DIAGNOSTICS

Chapel provides a comm table as well as a more advanced verbose comm mode

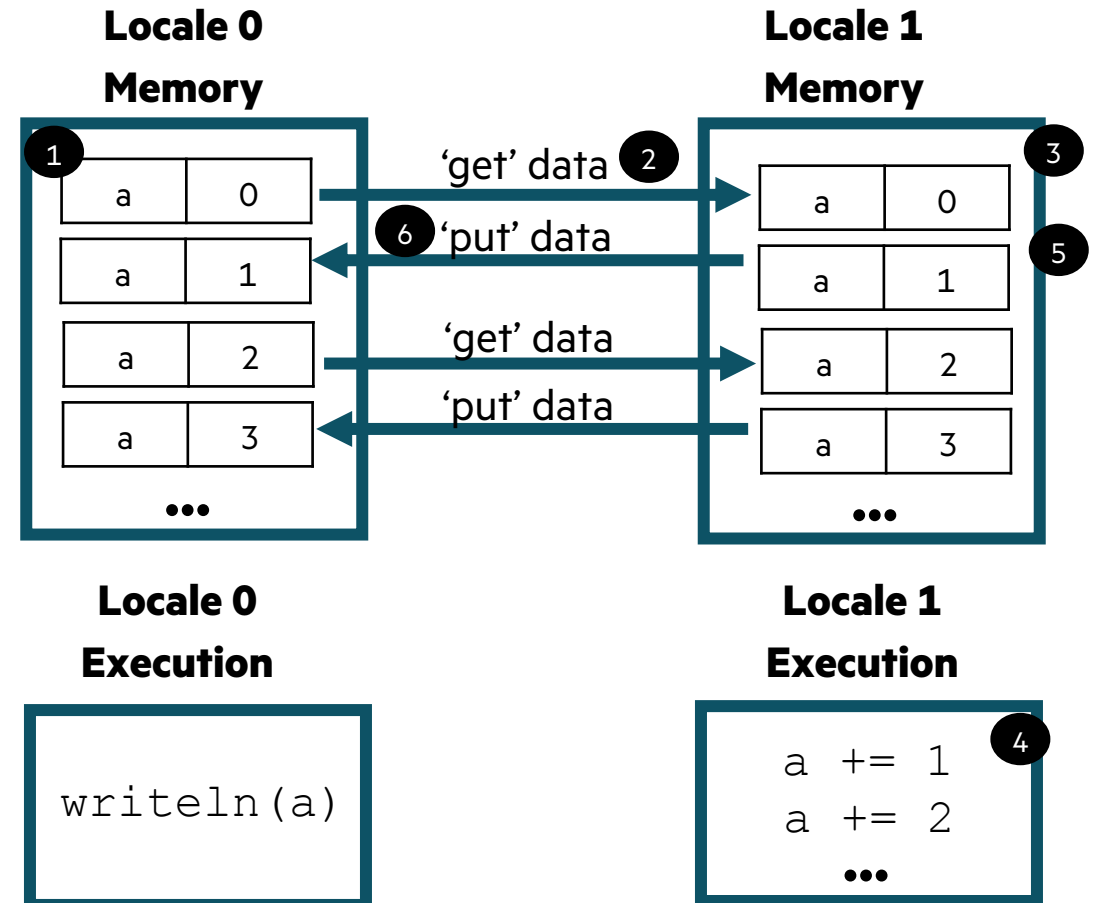
- Let's look at some code...

commTable.chpl

```
var a = 0; // allocated on locale 0

on Locales[1] do
  for i in 0..#5 do
    a += 1;

writeln(a); // print on locale 0
```



USING COMMDIAGNOSTICS

commTable.chpl

```
use CommDiagnostics;
var a = 0; // allocated on locale 0

startCommDiagnostics();
on Locales[1] do
  for i in 0..#5 do
    a += 1;
stopCommDiagnostics();
printCommDiagnosticsTable();

writeln(a); // print on locale 0
```

```
prompt> chpl commTabl.chpl --no-cache-remote
prompt> ./commTable -nl 2
```

	locale		get		put		execute_on	
	-----:		--:		--:		-----:	
	0		0		0		1	
	1		5		5		0	

5 'get's = 5 remote reads

5 'put's = 5 remote writes

```
prompt> chpl commTable.chpl
prompt> ./commTable -nl 2
```

	locale		get_nb		put_nb		execute_on		cache_get_hits		cache_get_misses		cache_put_hits		cache_put_misses	
	-----:		-----:		-----:		-----:		-----:		-----:		-----:		-----:	
	0		0		0		1		0		0		0		0	
	1		1		1		0		4		1		4		1	

USING COMMDIAGNOSTICS

verboseComm.chpl

```
var a = 0; // allocated on locale 0

startVerboseComm();
on Locales[1] do
  for i in 0..#5 do
    a += 1;
stopVerboseComm();

writeln(a); // print on locale 0
```

```
prompt> chpl verboseComm.chpl --no-cache-remote
prompt> ./verboseComm -nl 2
0: verboseComm.chpl:6: remote executeOn, node 1
1: verboseComm.chpl:8: remote get, node 0, 8 bytes, commid 5
1: verboseComm.chpl:8: remote put, node 0, 8 bytes, commid 6
1: verboseComm.chpl:8: remote get, node 0, 8 bytes, commid 5
1: verboseComm.chpl:8: remote put, node 0, 8 bytes, commid 6
1: verboseComm.chpl:8: remote get, node 0, 8 bytes, commid 5
1: verboseComm.chpl:8: remote put, node 0, 8 bytes, commid 6
1: verboseComm.chpl:8: remote get, node 0, 8 bytes, commid 5
1: verboseComm.chpl:8: remote put, node 0, 8 bytes, commid 6
1: verboseComm.chpl:8: remote get, node 0, 8 bytes, commid 5
1: verboseComm.chpl:8: remote put, node 0, 8 bytes, commid 6
```

USING COMMDIAGNOSTICS

- A more interesting example...

commOverDom.chpl

```
use CommDiagnostics;
use BlockDist;

config const size = 5;
var D = blockDist.createDomain(0..#size);
var a = 0;
startCommDiagnostics();
forall i in D with (ref a) do
  a += 1; //race condition!
stopCommDiagnostics();
printCommDiagnosticsTable();

writeln(a); // print on locale 0
```

```
prompt> chpl commOverDom.chpl --no-cache-remote
prompt> ./commOverDom -nl 2
```

	locale	get	put	execute_on_nb
	-----:	--:	--:	-----:
	0	0	0	1
	1	2	2	0

```
prompt> chpl commOverDom.chpl --no-cache-remote
prompt> ./commOverDom -nl 4 --size=100
```

	locale	get	put	execute_on_nb
	-----:	--:	--:	-----:
	0	0	0	3
	1	25	25	0
	2	25	25	0
	3	25	25	0

GENERAL TIPS WHEN GETTING STARTED WITH CHAPEL

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)



OUTLINE: MORE PARALLELISM AND SOME BEST PRACTICES

- Spectrum of Chapel loops
- Task intents including reduce intents, and atomics
- Parallelizing histogram (Hands On)
- Story of index gather parallelization
- Other parallel constructs: 'cobegin', 'begin', 'sync',
- Avoiding races with task intents and task-private variables
- Performance gotchas
- Memory in Chapel and Arkouda
- Using CommDiagnostics



LEARNING OBJECTIVES FOR TODAY'S CHAPEL TUTORIAL

- Familiarity with the Chapel execution model including how to run codes in parallel on a single node, across nodes, and both
- Learn Chapel concepts by compiling and running provided code examples
 - ✓ Serial code using map/dictionary, (k-mer counting from bioinformatics)
 - ✓ Parallelism and locality in Chapel
 - ✓ Distributed parallelism and 1D arrays, (processing files in parallel)
 - ✓ Chapel basics in the context of an n-body code
 - ✓ Distributed parallelism and 2D arrays, (heat diffusion problem)
 - ✓ How to parallelize histogram
 - ✓ Using CommDiagnostics for counting remote reads and writes
 - ✓ Chapel and Arkouda best practices including avoiding races and performance gotchas
- Where to get help and how you can participate in the Chapel community



TUTORIAL SUMMARY

- **Takeaways**

- Chapel is a general-purpose programming language designed to leverage parallelism
- It is being used in some large production codes
- Our team is responsive to user questions and would enjoy having you participate in our community

- **How to get more help and engage with the community**

- Ask us questions on discourse, gitter, or stack overflow
- Share your sample codes with us and your research community!
- Join us at our free, virtual workshop in June, <https://chapel-lang.org/CHI UW.html>

- **Potential follow-on topics**

- Using classes in Chapel including memory management
- Generics in Chapel: enabling the same code to work for multiple types
- Chapel interoperability with C
- Your suggestions?



ONE DAY CHAPEL TUTORIAL

- 9-10:30: Getting started using Chapel for parallel programming
- 10:30-10:45: break
- 10:45-12:15: Chapel basics in the context of the n-body example code
- 12:15-1:15: lunch
- 1:15-2:45: Distributed and shared-memory parallelism especially w/arrays (data parallelism)
- 2:45-3:00: break
- 3:00-4:30: More parallelism including for asynchronous parallelism (task parallelism)
- 4:30-5:00: Wrap-up including gathering further questions from attendees



CHAPEL RESOURCES

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


- (points to all other resources)

Social Media:

- Twitter: [@ChapelLanguage](https://twitter.com/ChapelLanguage)
- Facebook: [@ChapelLanguage](https://facebook.com/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>



The Chapel Parallel Programming Language

What is Chapel?

Chapel is a programming language designed for productive parallel computing at scale.

Why Chapel? Because it simplifies parallel programming through elegant support for:

- **distributed arrays** that can leverage thousands of nodes' memories and cores
- **a global namespace** supporting direct access to local or remote variables
- **data parallelism** to trivially use the cores of a laptop, cluster, or supercomputer
- **task parallelism** to create concurrency within a node or across the system

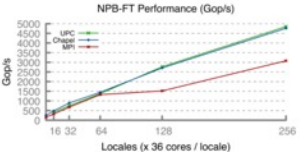
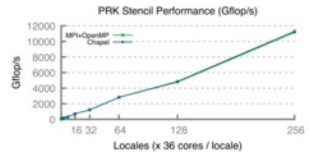
Chapel Characteristics

- **productive:** code tends to be similarly readable/writable as Python
- **scalable:** runs on laptops, clusters, the cloud, and HPC systems
- **fast:** performance competes with or beats C/C++ & MPI & OpenMP
- **portable:** compiles and runs in virtually any *nix environment
- **open-source:** hosted on GitHub, permissively licensed

New to Chapel?

As an introduction to Chapel, you may want to...

- watch an [overview talk](#) or browse its [slides](#)
- read a [blog-length](#) or [chapter-length](#) introduction to Chapel
- learn about [projects powered by Chapel](#)
- check out [performance highlights](#) like these:



PRK Stencil Performance (Gflop/s)

NPB-FT Performance (Gop/s)

- browse [sample programs](#) or learn how to write distributed programs like this one:

```
use CyclicDist;           // use the Cyclic distribution library
config const n = 100;     // use --n=<val> when executing to override this default

forall i in {1..n} dmapped Cyclic(startIdx=1) do
  writeln("Hello from iteration ", i, " of ", n, " running on node ", here.id);
```