Hewlett Packard Enterprise

INTRODUCTION TO CHAPEL PARALLEL PROGRAMMING LANGUAGE

Michelle Strout and Jeremiah Corrado

CUF23: Sponsored by OLCF, NERSC, and ECP
July 26-27, 2023

INTRODUCTION TO CHAPEL

- What Chapel is and how programmers are using Chapel in their applications
- Chapel execution model with a parallel and distributed "Hello World"
- 2D Heat Diffusion example: variants and how to compile and run them
- Learning objectives for today's 90-minute Chapel tutorial

CHAPEL PROGRAMMING LANGUAGE

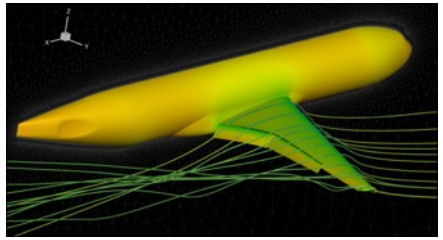
Chapel is a general-purpose programming language that provides

ease of parallel programming,
high performance, and
portability.

And is being used in applications in various ways:

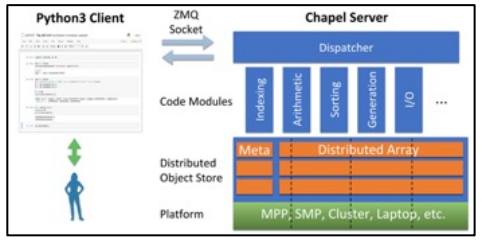
refactoring existing codes,
developing new codes,
serving high performance to Python codes **(Chapel server with Python client),** and
providing distributed and shared memory parallelism for existing codes.

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



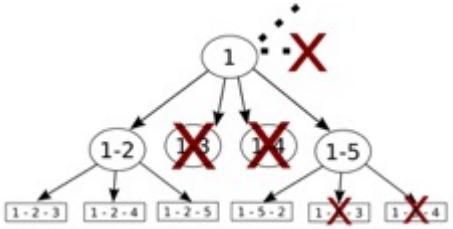
CHAMPS: 3D Unstructured CFD

CHIUW 2021 CHIUW 2022



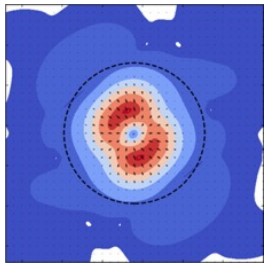
Arkouda: Interactive Data Science at Massive Scale

CHIUW 2020 CHIUW 2023



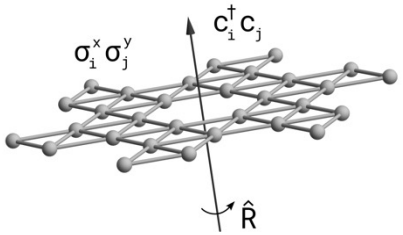
ChOp: Chapel-based Optimization

CHIUW 2021 CHIUW 2023



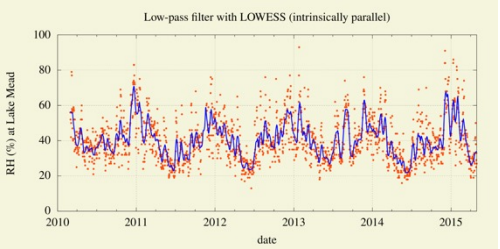
ChplUltra: Simulating Ultralight Dark Matter

CHIUW 2020 CHIUW 2022



Lattice-Symmetries: a Quantum Many-Body Toolbox

CHIUW 2022

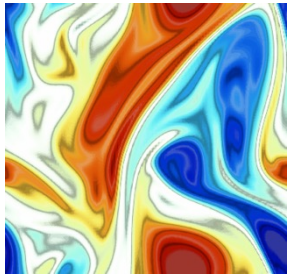


Desk dot chpl: Utilities for Environmental Eng.

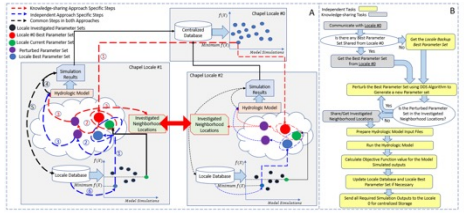
CHIUW 2022

RapidQ: Mapping Coral Biodiversity

CHIUW 2023

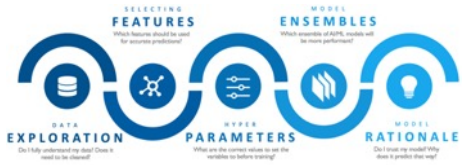


ChapQG: Layered Quasigeostrophic CFD



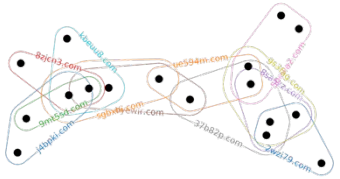
Chapel-based Hydrological Model Calibration

CHIUW 2023



CrayAI HyperParameter Optimization (HPO)

CHIUW 2021



CHGL: Chapel Hypergraph Library

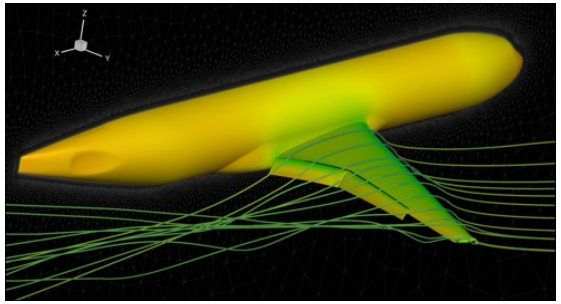
CHIUW 2020

?

Your Application Here?

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

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

Recent Journal Paper on using Chapel for calibrating hydrologic models

- Marjan Asgari et al, "Development of a knowledge-sharing parallel computing approach for calibrating distributed watershed hydrologic models", Environmental Modeling and Software.
- They report super-linear speedup

ARKOUDA ARGSORT PERFORMANCE

HPE Apollo (May 2021)

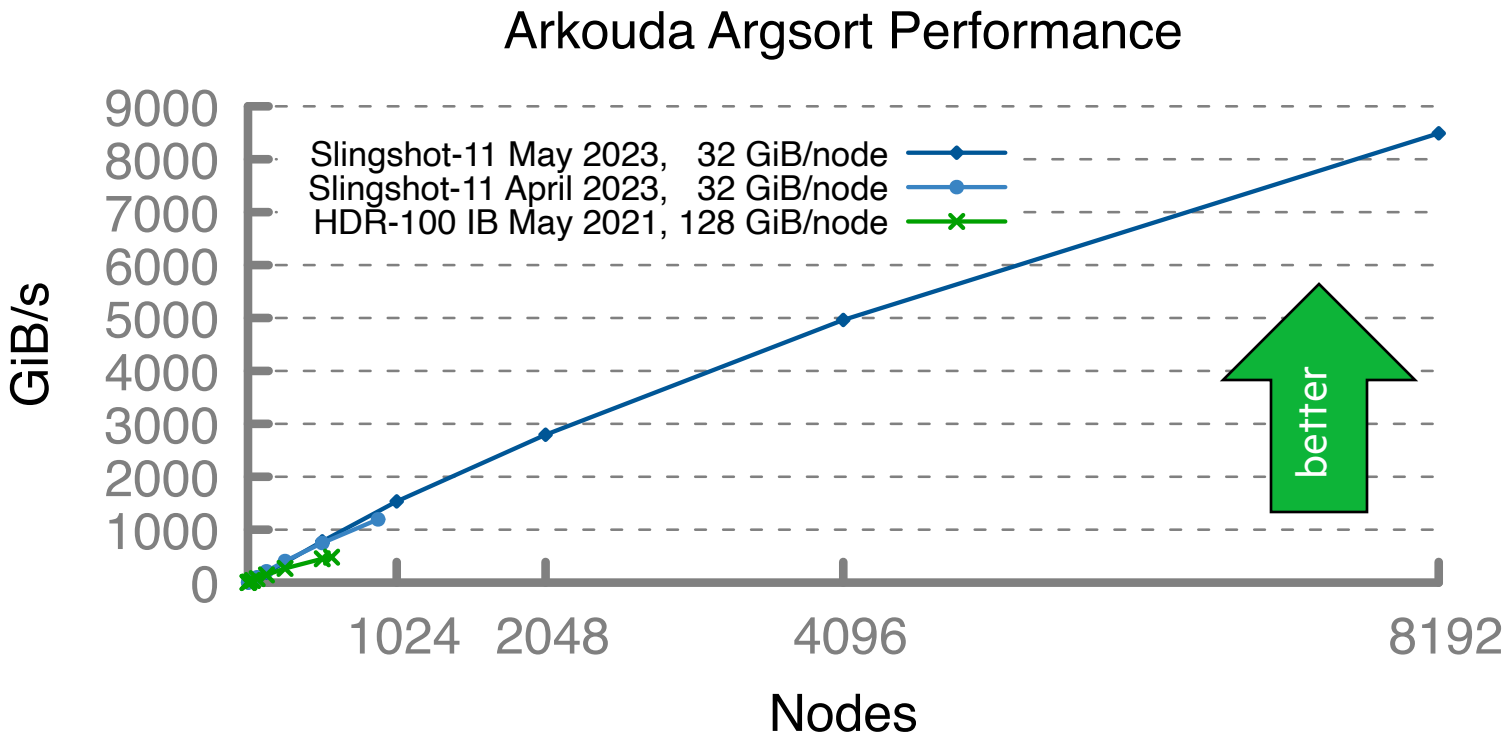
- HDR-100 Infiniband network (100 Gb/s)
- 576 compute nodes
- 72 TiB of 8-byte values
- ~480 GiB/s (~150 seconds)

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)



A notable performance achievement in ~100 lines of Chapel

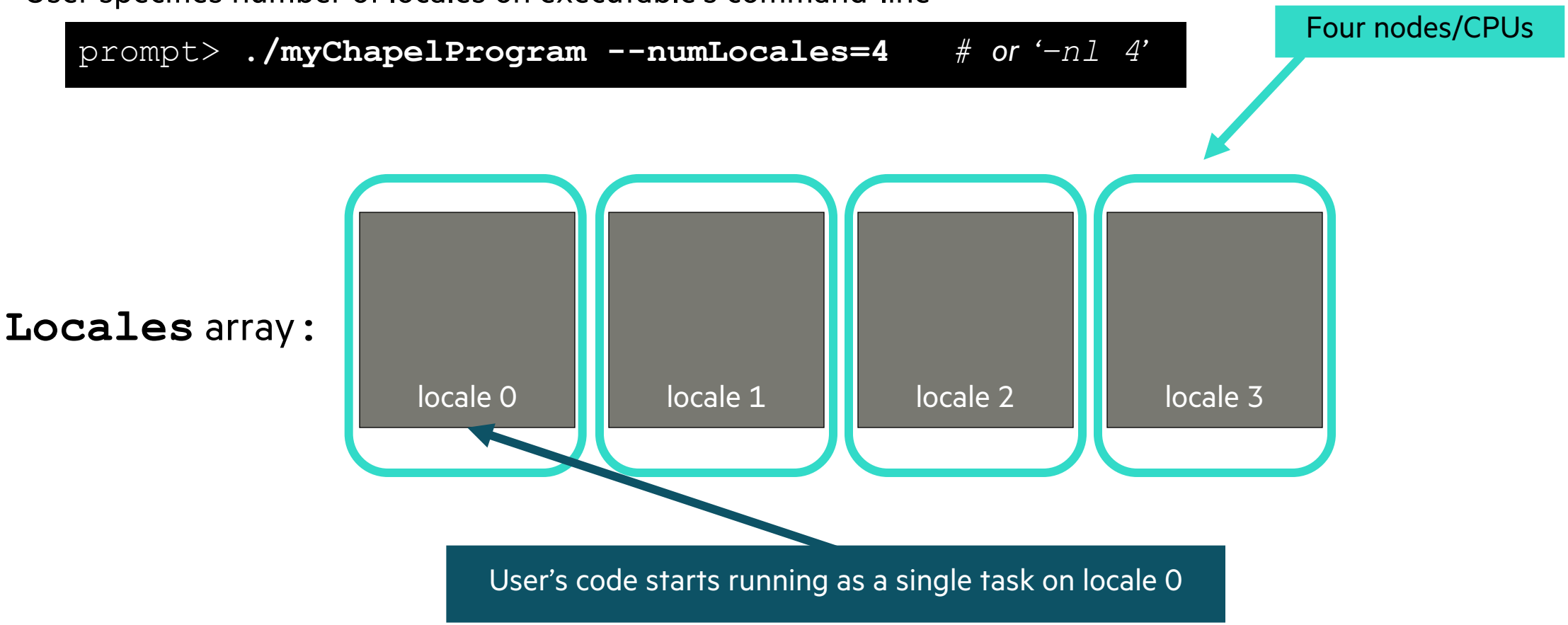
INTRODUCTION TO CHAPEL

- What Chapel is and how programmers are using Chapel in their applications
- Chapel execution model with a parallel and distributed "Hello World"
- 2D Heat Diffusion example: variants and how to compile and run them
- Learning objectives for today's 90-minute Chapel tutorial

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'
```



TASK-PARALLEL "HELLO WORLD"

hello-dist-node-names.chpl

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

TASK-PARALLEL "HELLO WORLD"

hello-dist-node-names.chpl

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

'here' refers to the locale on which we're currently running

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

what's my locale's name?

TASK-PARALLEL "HELLO WORLD"

hello-dist-node-names.chpl

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

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

```
> chpl hello-dist-node-names.chpl
> ./hello-dist-node-names
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"

hello-dist-node-names.chpl

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

```
> chpl hello-dist-node-names.chpl
> ./hello-dist-node-names
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

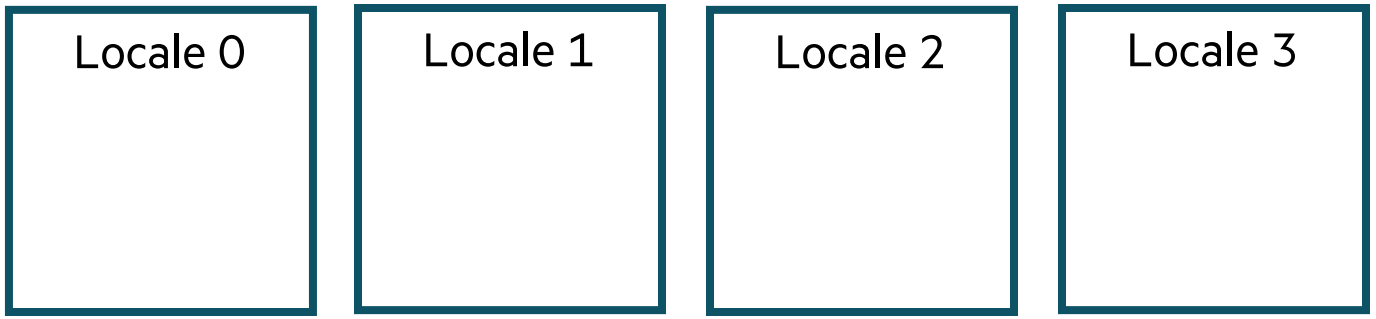
TASK-PARALLEL "HELLO WORLD" (DISTRIBUTED VERSION)

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 (introduced a few slides back)

Locales array:



TASK-PARALLEL "HELLO WORLD" (DISTRIBUTED VERSION)

hello-dist-node-names.chpl

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

create a task per locale on which the program is running

have each task run 'on' its locale

then print a message per core, as before

```
> chpl hello-dist-node-names.chpl
> ./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
…
```

INTRODUCTION TO CHAPEL

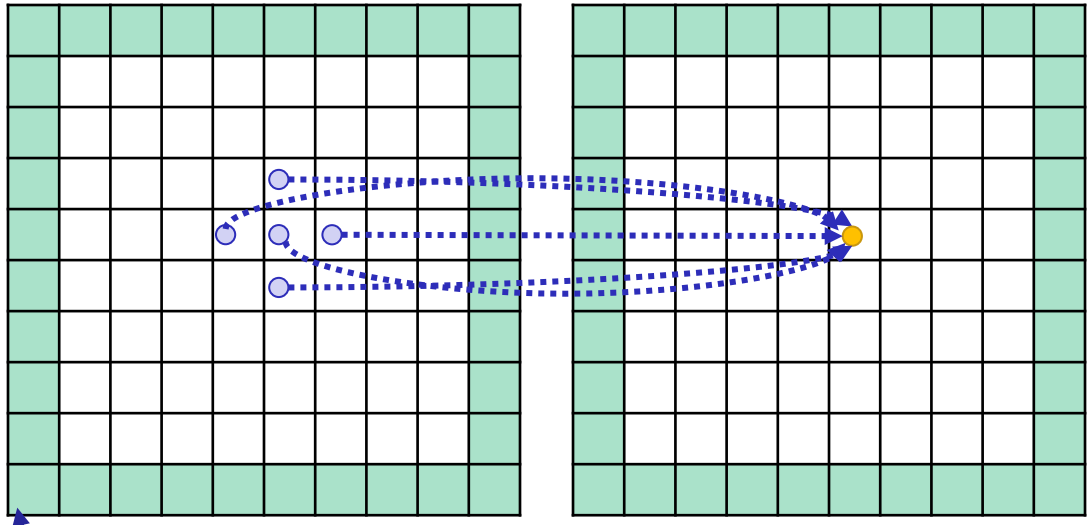
- What Chapel is and how programmers are using Chapel in their applications
- Chapel execution model with a parallel and distributed "Hello World"
- 2D Heat Diffusion example: variants and how to compile and run them
- Learning objectives for today's 90-minute Chapel tutorial

2D HEAT DIFFUSION EXAMPLE

See https://go.lbl.gov/cuf23-repo for more info and for example code.

- **See 'heat_2D.*.chpl' in the Chapel examples**
  - 'heat_2D.chpl' - shared memory parallel version that runs in locale 0
  - 'heat_2D_dist.chpl' - parallel and distributed version that is the same as 'heat_2D.chpl' but with distributed arrays
  - 'heat_2D_dist_buffers.chpl' - parallel and distributed version that copies to neighbors landing pad and then into local halos

PARALLEL HEAT DIFFUSION IN HEAT_2D.CHPL



u^n

Stored in un

u^{n+1}

Stored in u

Fixed boundary values

- 2D heat diffusion PDE

$$\frac{\partial u}{\partial t} = \nu \frac{\partial^2 u}{\partial x^2} + \nu \frac{\partial^2 u}{\partial y^2}$$

Simplified form for below

assume **Δx=Δy**, and let **α=νΔt/Δx²**

- Solving for next temperatures at each time step using finite difference method

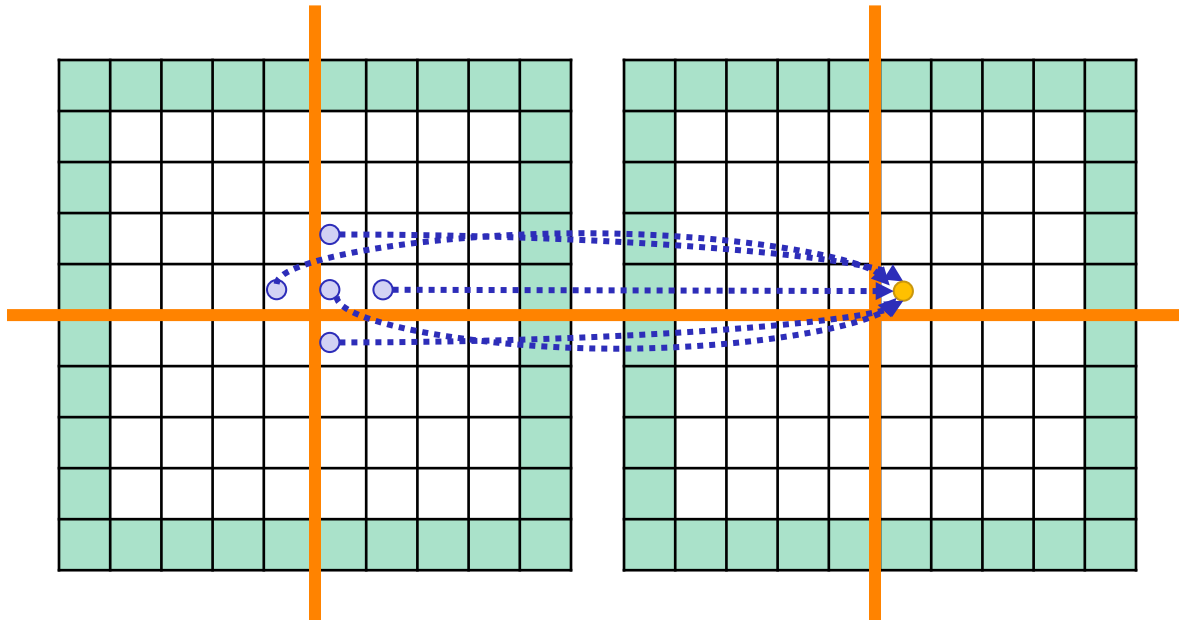
$$u_{i,j}^{n+1} = u_{i,j}^{n} + \alpha\left(u_{i+1,j}^{n} + u_{i-1,j}^{n} - 4u_{i,j}^{n} + u_{i,j+1}^{n} + u_{i,j-1}^{n}\right)$$

- All updates in a timestep can be done in parallel

```
forall (i, j) in indicesInner do
    u[i, j] = un[i, j] + alpha *
      (un[i, j-1] + un[i-1, j] + un[i+1, j] +
       un[i, j+1] - 4 * un[i, j]);
```

- Output is the mean and standard deviation of all the values and time to solution

DISTRIBUTED AND PARALLEL HEAT DIFFUSION IN HEAT_2D_DIST.CHPL



Stored in un Stored in u

- Declaring 'u' and 'un' arrays

```
const indices = {0..<nx, 0..<ny}
var u: [indices] real;
```

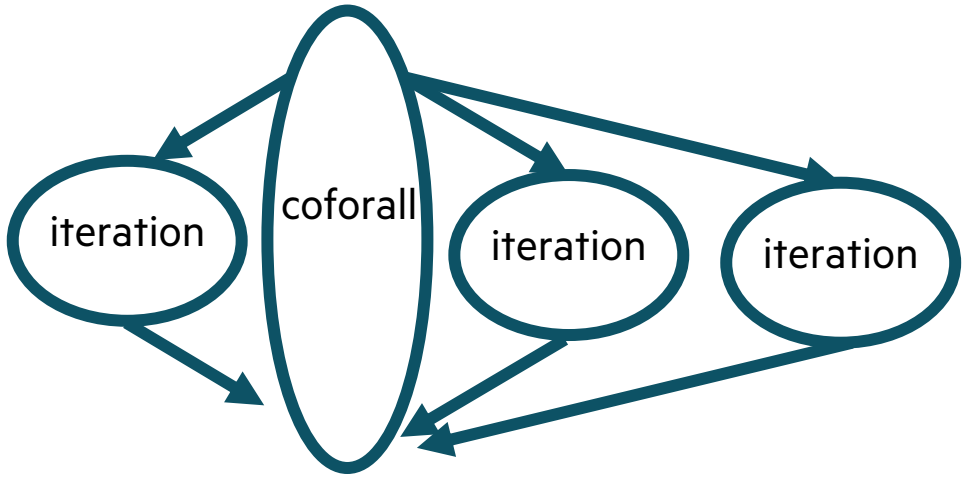
- Declaring 'u' and 'un' arrays as distributed (e.g., 2x2 distribution is shown)

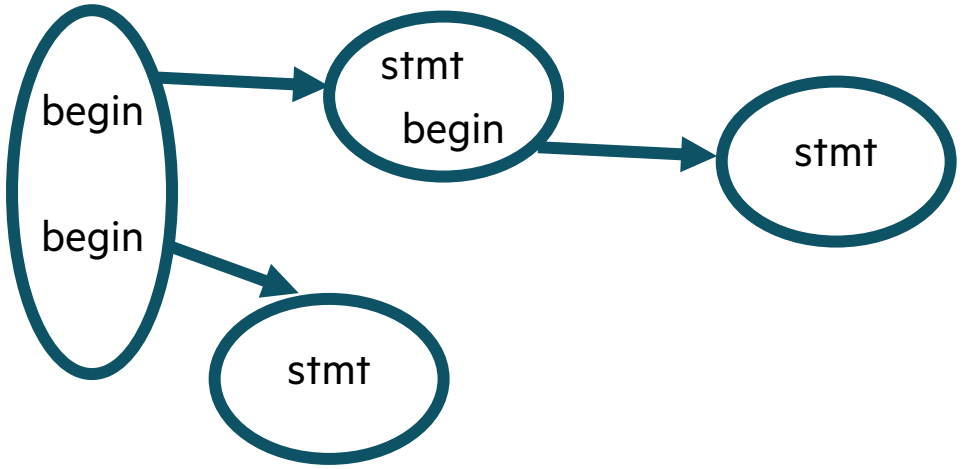
```
const indices = {0..<nx, 0..<ny},
      INDICES = Block.createDomain(indices);
var u: [INDICES] real;
```

- Reads that cross the distribution boundary will result in a remote get

PARALLELISM SUPPORTED BY CHAPEL

- **Synchronous parallellism**
  - 'coforall', distributed memory parallelism across processes/locales with 'on' syntax
  - 'coforall', shared-memory parallelism over threads
  - 'cobegin', executes all statements in block in parallel
- **Asynchronous parallelism**
  - 'begin', creates an asynchronous task
  - 'sync' and 'atomic' vars for task coordination
  - spawning subprocesses
- **Higher-level parallelism abstractions**
  - 'forall', data parallelism and iterator abstraction
  - 'foreach', SIMD parallelism
  - 'scan', operations such as cumulative sums
  - 'reduce', operations such as summation





LEARNING OBJECTIVES FOR TODAY'S CHAPEL TUTORIAL

- Compile and run Chapel programs
- 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)
  - Distributed parallelism and 2D arrays, (heat diffusion problem will see in UPC++ and CAF)
  - Distributed parallel image processing, (coral reef diversity example)
  - GPU parallelism (stream example)

- Where to get help and how you can participate in the Chapel community

Hewlett Packard Enterprise

PROGRAMING IN CHAPEL

Michelle Strout and Jeremiah Corrado

CUF23: Sponsored by OLCF, NERSC, and ECP

July 26-27, 2023

LEARNING OBJECTIVES FOR TODAY'S CHAPEL TUTORIAL

- Compile and run Chapel programs
- 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)
  - Distributed parallelism and 2D arrays, (heat diffusion problem will see in UPC++ and CAF)
  - Distributed parallel image processing, (coral reef diversity example)
  - GPU parallelism (stream example)

- Where to get help and how you can participate in the Chapel community

HOW TO PARTICIPATE IN THIS TUTORIAL AND AFTERWARDS

- **During the tutorial today and tomorrow (July 26-27, 2023)**
  - Download the tarball of examples and follow the instructions in the README

```
curl -LO https://go.lbl.gov/cuf23.tar.gz
tar xzf cuf23.tar.gz
cd cuf23/
```

Check out the chapel-quickReference.pdf in the cuf23/chapel/ subdirectory

- **After the tutorial**
  - **The cuf23 tarball will still be available or clone from** *https://go.lbl.gov/cuf23-repo for Chapel code*
  - **Attempt this Online website for running Chapel code**
    - Go to main Chapel webpage at https://chapel-lang.org/ and click on the ATO icon on the lower left
  - **Using a container on your laptop**
    - First, install docker for your machine and then start it up
    - Then, the below commands work with docker

```
docker pull docker.io/chapel/chapel-gasnet    # takes about 5 minutes
docker run --rm -v "$PWD":/myapp -w /myapp chapel/chapel-gasnet chpl hello.chpl
docker run --rm -v "$PWD":/myapp -w /myapp chapel/chapel-gasnet ./hello -nl 1
```

SERIAL CODE USING MAP/DICTIONARY: K-MER COUNTING

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

EXPERIMENTING WITH THE K-MER EXAMPLE

```
make run-kmer
```

- **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

LEARNING OBJECTIVES FOR TODAY'S CHAPEL TUTORIAL

- Compile and run Chapel programs
- 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)
  - Distributed parallelism and 2D arrays, (heat diffusion problem will see in UPC++ and CAF)
  - Distributed parallel image processing, (coral reef diversity example)
  - GPU parallelism (stream example)

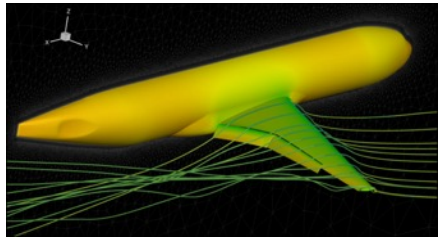
- Where to get help and how you can participate in the Chapel community

PARALLELISM SUPPORTED BY CHAPEL

- **Synchronous parallellism**
  - 'coforall', distributed memory parallelism across processes/locales with 'on' syntax
  - 'coforall', shared-memory parallelism over threads
  - 'cobegin', executes all statements in block in parallel
- **Asynchronous parallelism**
  - 'begin', creates an asynchronous task
  - 'sync' and 'atomic' vars for task coordination
  - spawning subprocesses
- **Higher-level parallelism abstractions**
  - 'forall', data parallelism and iterator abstraction
  - 'foreach', SIMD parallelism
  - 'scan', operations such as cumulative sums
  - 'reduce', operations such as summation

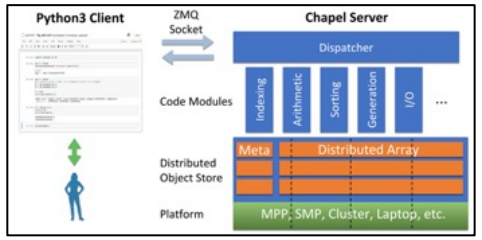
```
coforall loc in Locales do on loc { /* ... */ }
coforall tid in 0..<numTasks { /* ... */ }

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

var x : atomic int = 0, y : sync int = 0;
sync {
  begin x.add(1);
  begin y.writeEF(1);
  begin x.sub(1);
  begin y.writeFF(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;
```

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



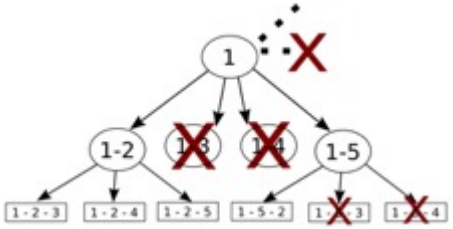
CHAMPS: 3D Unstructured CFD

CHIUW 2021 CHIUW 2022



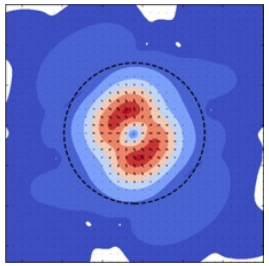
Arkouda: Interactive Data Science at Massive Scale

CHIUW 2020 CHIUW 2023



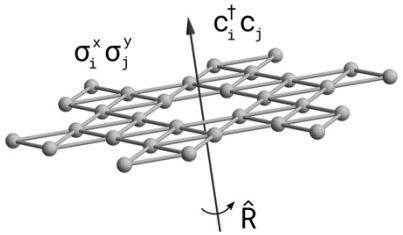
ChOp: Chapel-based Optimization

CHIUW 2021 CHIUW 2023



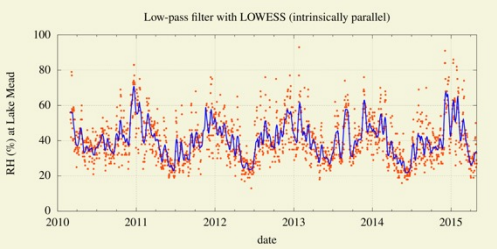
ChplUltra: Simulating Ultralight Dark Matter

CHIUW 2020 CHIUW 2022



Lattice-Symmetries: a Quantum Many-Body Toolbox

CHIUW 2022

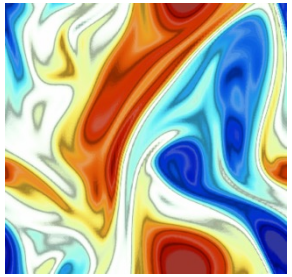


Desk dot chpl: Utilities for Environmental Eng.

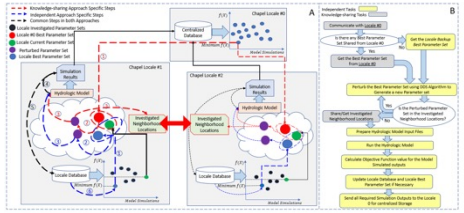
CHIUW 2022

RapidQ: Mapping Coral Biodiversity

CHIUW 2023

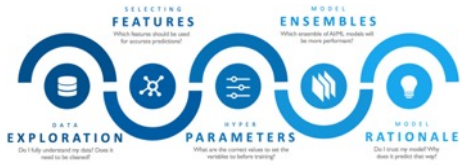


ChapQG: Layered Quasigeostrophic CFD



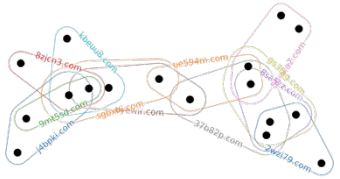
Chapel-based Hydrological Model Calibration

CHIUW 2023



CrayAI HyperParameter Optimization (HPO)

CHIUW 2021



CHGL: Chapel Hypergraph Library

CHIUW 2020

?

Your Application Here?

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

USE OF PARALLELISM IN SOME APPLICATIONS AND BENCHMARKS

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

In this tutorial will be working with examples of parallelism from the yellow highlighted columns.

PARALLELISM ACROSS LOCALES AND WITHIN LOCALES

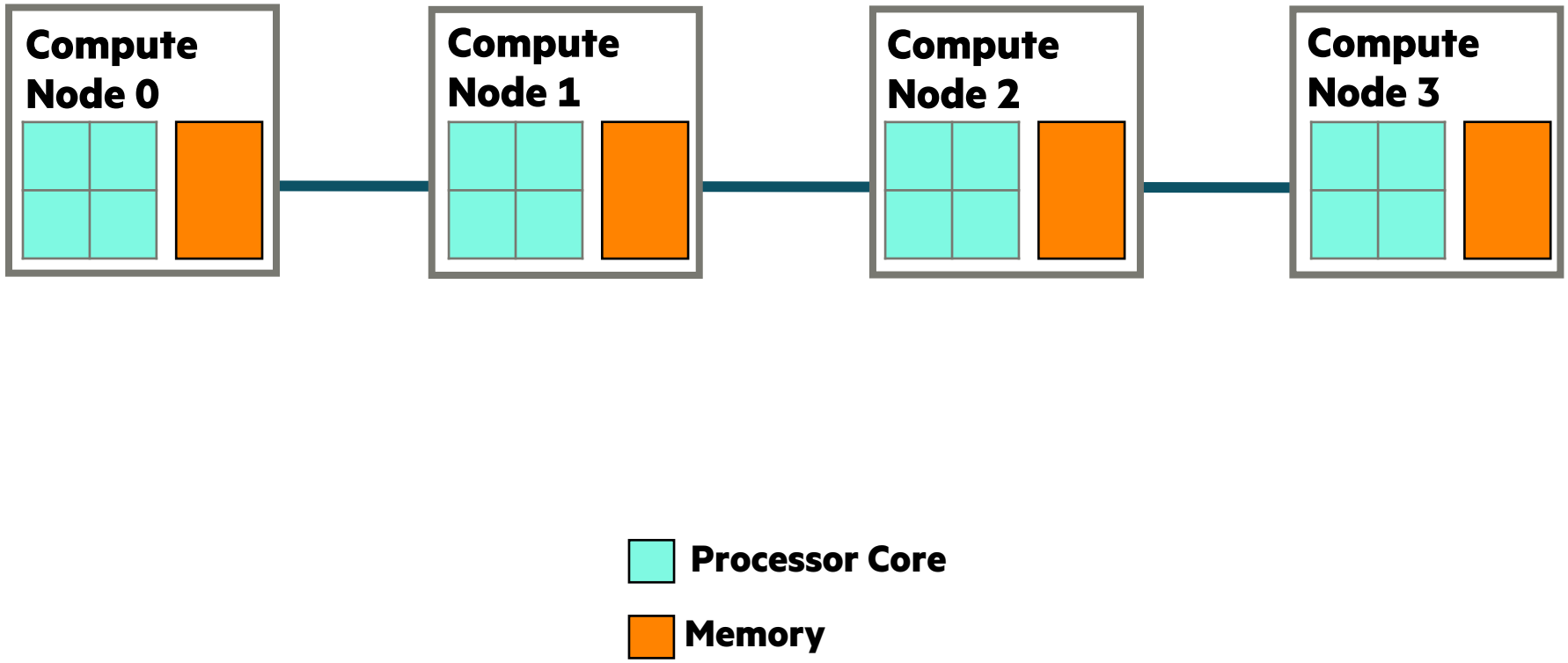
```
make run-hellopar
```

- **Parallel hello world**
  - hellopar.chpl
- **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, ")" );
  }
}
```

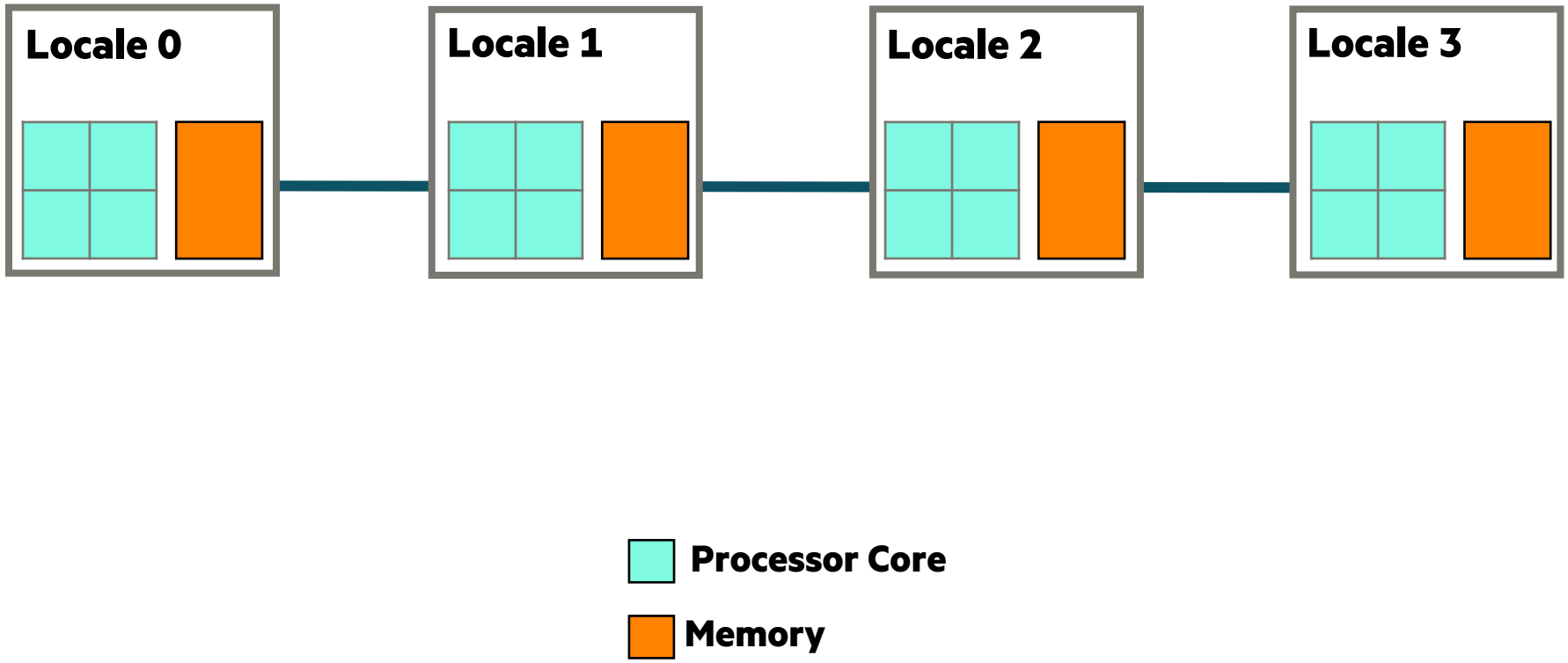
LOCALES AND EXECUTION MODEL IN CHAPEL

- 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



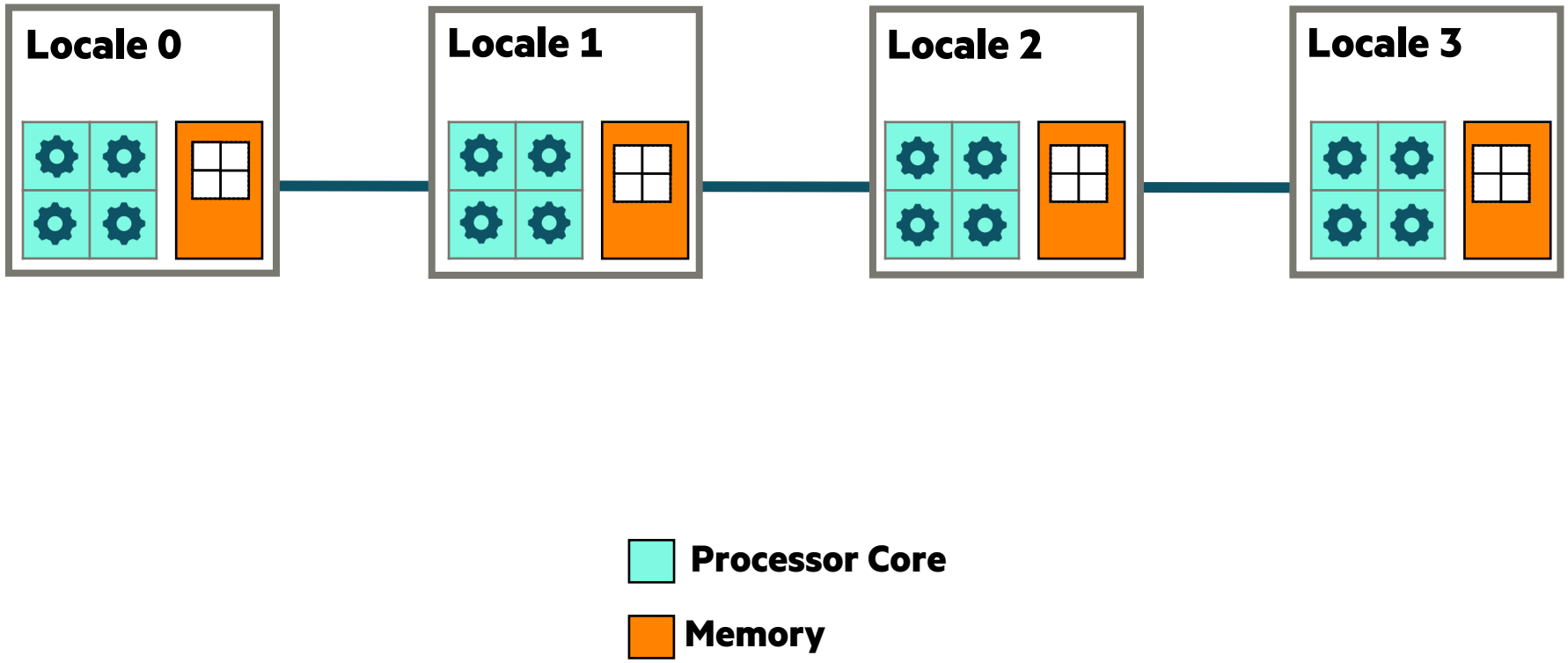
LOCALES AND EXECUTION MODEL IN CHAPEL

- 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



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

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;
}
```

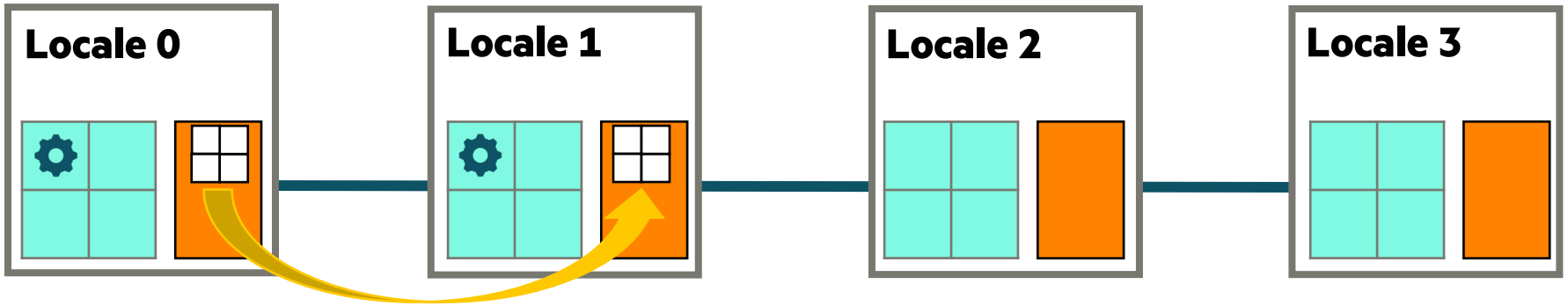
All Chapel programs begin running as a single task on locale 0

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



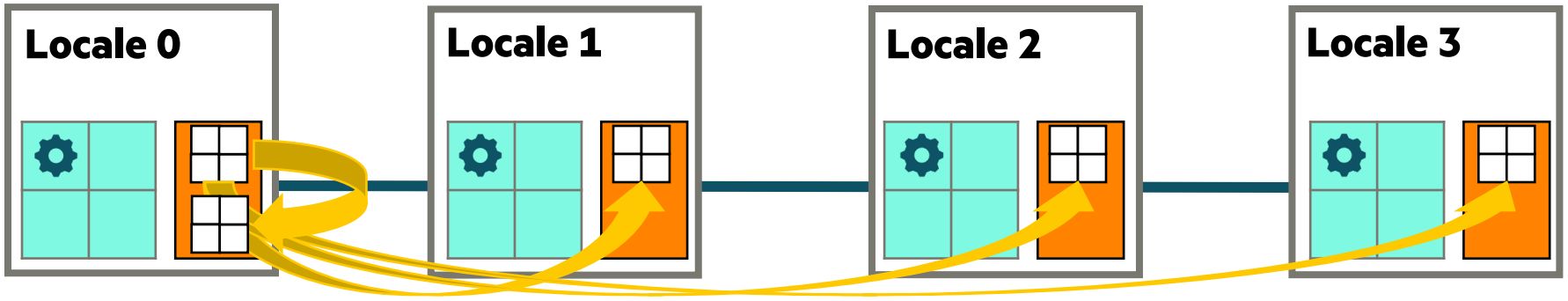
BASIC FEATURES FOR LOCALITY

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



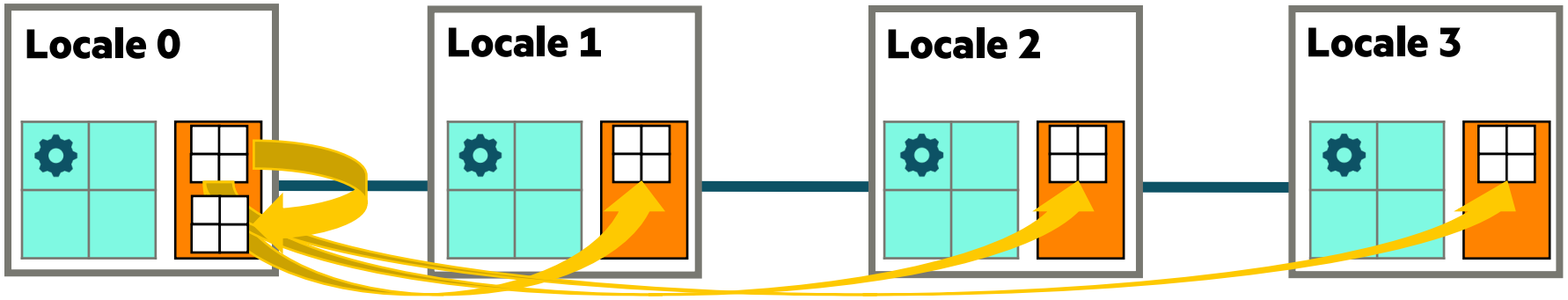
MIXING LOCALITY WITH TASK PARALLELISM

basics-coforall.chpl

```
writeln("Hello from locale ", here.id);

var A: [1..2, 1..2] real;

coforall loc in Locales {
  on loc {
    var B = A;
  }
}
```

The coforall loop creates a parallel task per iteration

This results in a parallel distributed computation



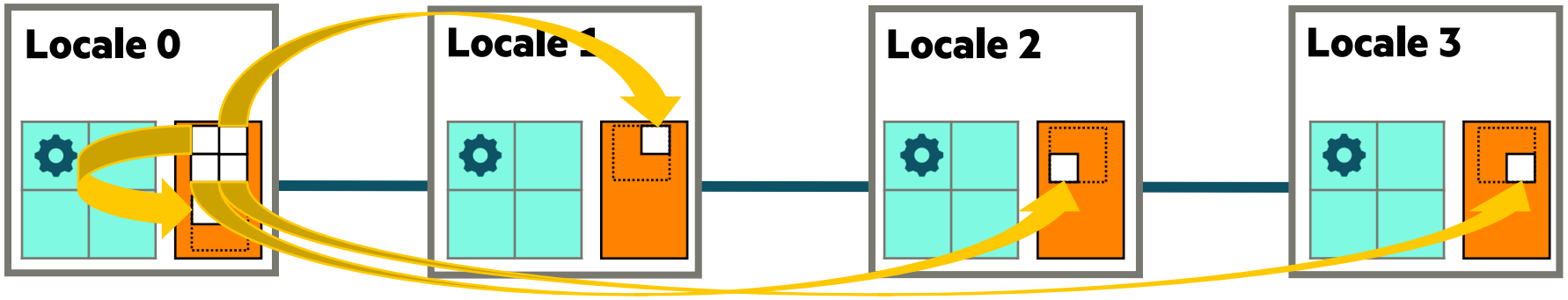
ARRAY-BASED PARALLELISM AND LOCALITY

basics-distarr.chpl

```
writeln("Hello from locale ", here.id);

var A: [1..2, 1..2] real;

use BlockDist;

var D = Block.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



PARALLELISM ACROSS LOCALES AND WITHIN LOCALES

```
make run-hellopar
```

- **Parallel hello world**
  - hellopar.chpl
- **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 '//'
- **Things to try**

```
./run-hellopar -nl 1 --tasksPerLocale=3
./run-hellopar -nl 2 --tasksPerLocale=3
```

```
// 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

- 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);
```

LEARNING OBJECTIVES FOR TODAY'S CHAPEL TUTORIAL

- Compile and run Chapel programs
- 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)
  - Distributed parallelism and 2D arrays, (heat diffusion problem will see in UPC++ and CAF)
  - Distributed parallel image processing, (coral reef diversity example)
  - GPU parallelism (stream example)

- Where to get help and how you can participate in the Chapel community

PROCESSING FILES IN PARALLEL

```
make run-parfilekmer
```

- **See 'parfilekmer.chpl' in the repository**

- **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
```

ANALYZING MULTIPLE FILES USING PARALLELISM

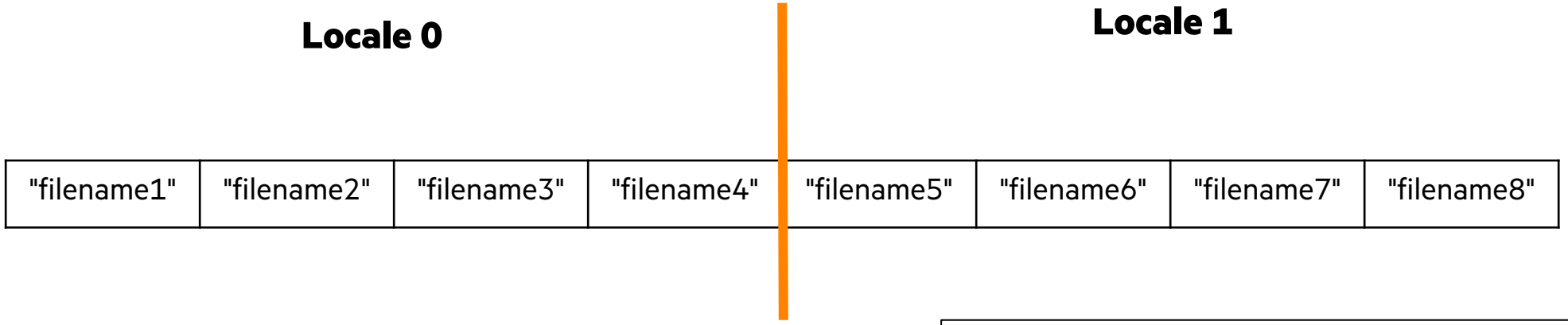
parfilekmer.chpl

```
use FileSystem;
config const dir = "DataDir";
var fList = findFiles(dir);
var filenames =
  Block.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




```
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

PROCESSING FILES IN PARALLEL

```
make run-parfilekmer
```

- **See 'parfilekmer.chpl' in the repository**

- **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 puts and gets happening yet

LEARNING OBJECTIVES FOR TODAY'S CHAPEL TUTORIAL

- Compile and run Chapel programs
- 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)
  - Distributed parallelism and 2D arrays, (heat diffusion problem will see in UPC++ and CAF)
  - Distributed parallel image processing, (coral reef diversity example)
  - GPU parallelism (stream example)

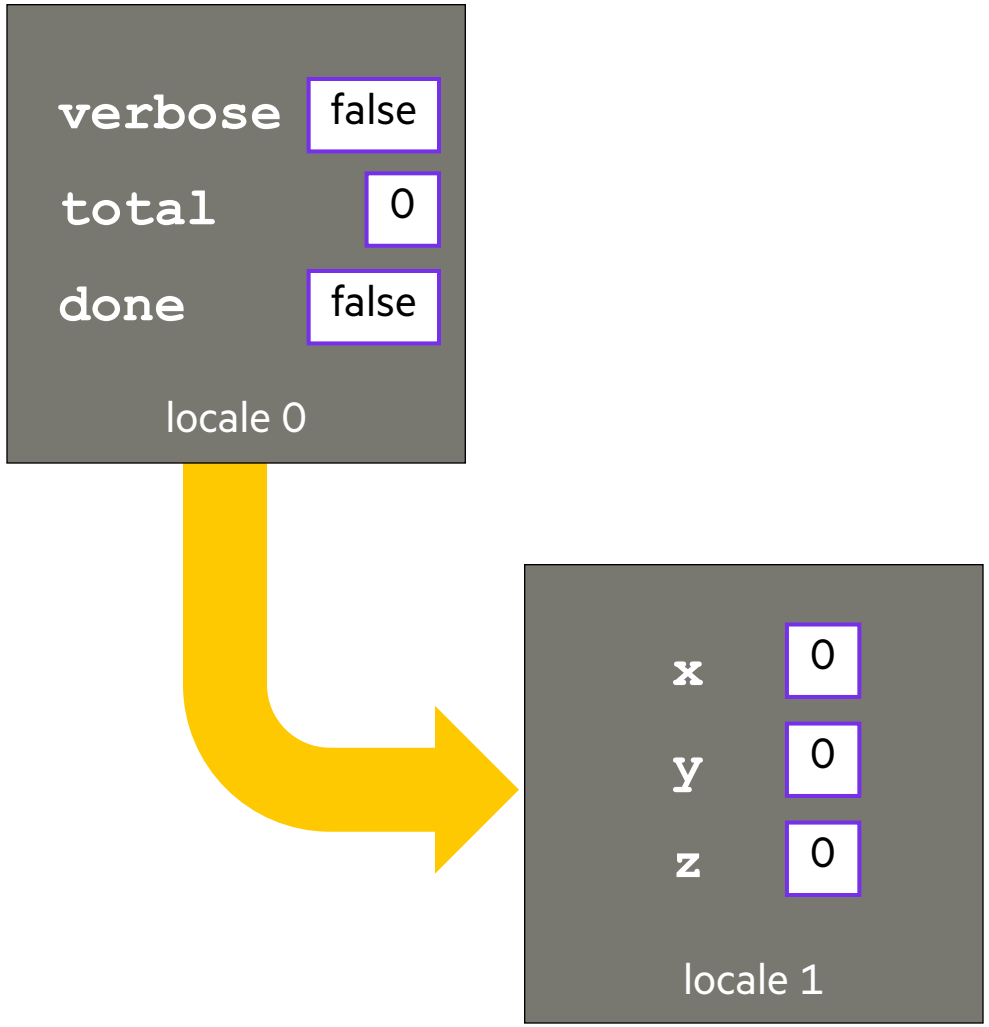
- Where to get help and how you can participate in the Chapel community

CHAPEL SUPPORTS A GLOBAL NAMESPACE WITH PUTS AND GETS

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

onClause.chpl

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

…

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



}
```

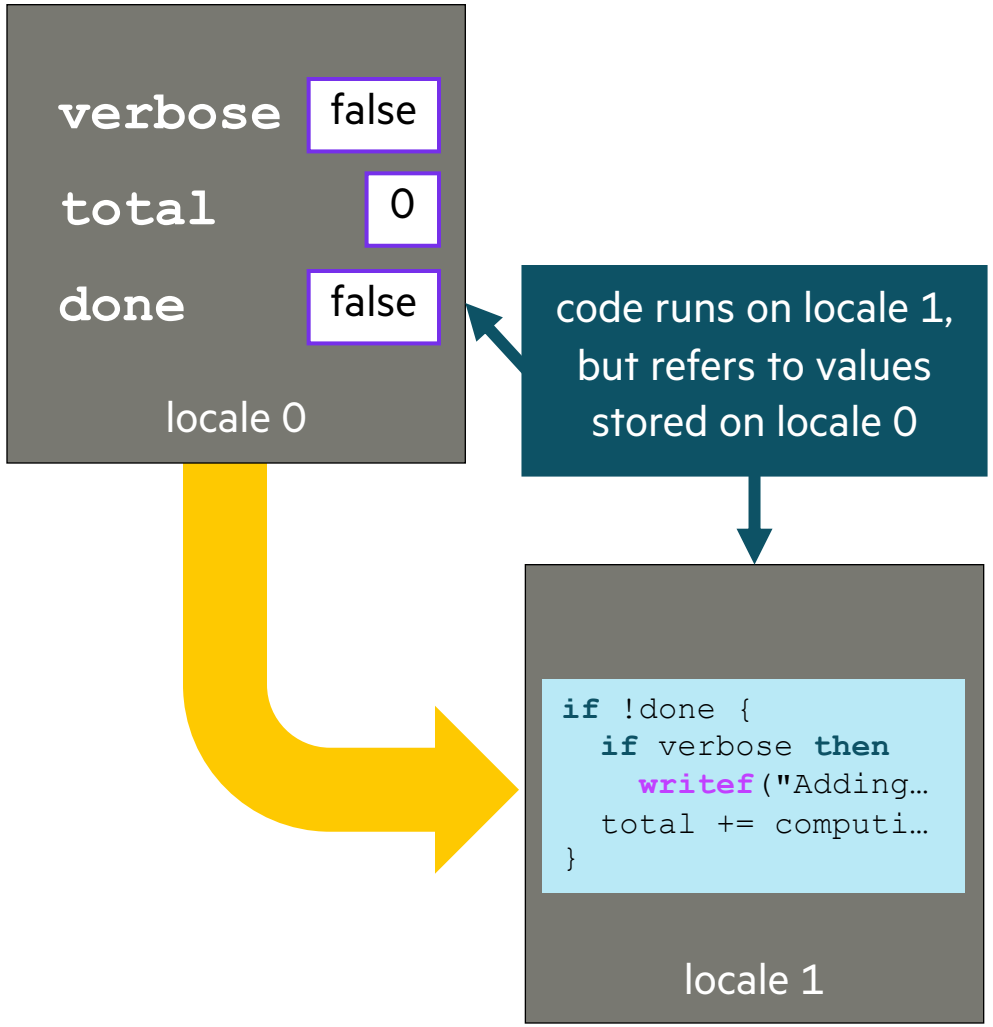


CHAPEL SUPPORTS A GLOBAL NAMESPACE

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

onClause.chpl

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

…

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



2D HEAT DIFFUSION EXAMPLE

```
make run-heat_2D
make run-heat_2D_dist
make run-heat_2D_buffers
```

- **See 'heat_2D.*.chpl' in the Chapel examples**
  - 'heat_2D.chpl' - shared memory parallel version that runs in locale 0
  - 'heat_2D_dist.chpl' - parallel and distributed version that is the same as 'heat_2D.chpl' but with distributed arrays
  - 'heat_2D_dist_buffers.chpl' - parallel and distributed version that copies to neighbors landing pad and then into local halos

- **Some things to try out with these variants**

```
chpl heat_2D.chpl
./heat_2D -nl 1

--nt 10 --nx=2048 --ny=2048    # decreases the number of time steps
                               # and reduces the size of the domain
                               # along each dimension from default 4096
```

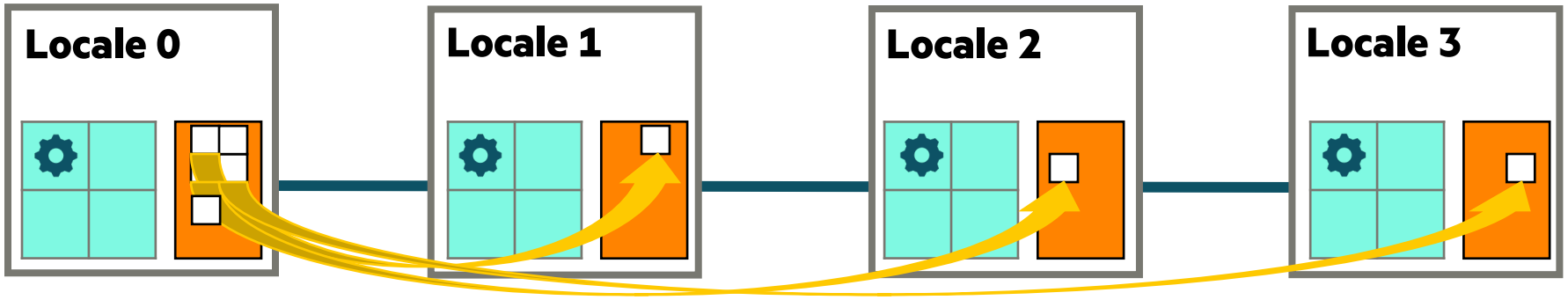
ARRAY-BASED PARALLELISM AND LOCALITY

basics-distarr.chpl

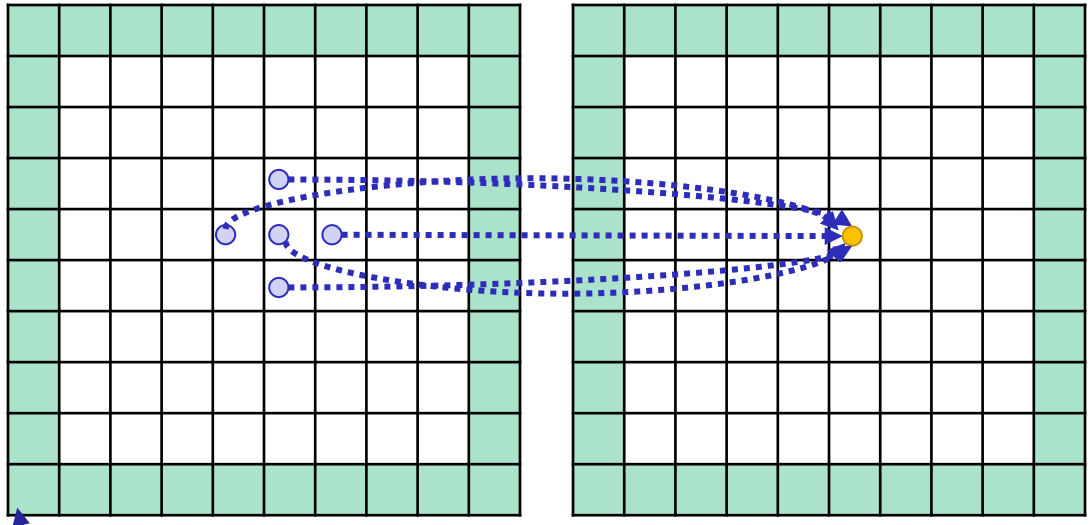
```
writeln("Hello from locale ", here.id);

var A: [1..2, 1..2] real;

use BlockDist;

var D = Block.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



PARALLEL HEAT DIFFUSION IN HEAT_2D.CHPL



u^n

Stored in un

u^{n+1}

Stored in u

Fixed boundary values

- 2D heat diffusion PDE

$$\frac{\partial u}{\partial t} = \nu \frac{\partial^2 u}{\partial x^2} + \nu \frac{\partial^2 u}{\partial y^2}$$

Simplified form for below

assume **Δx=Δy**, and let **α=νΔt/Δx²**

- Solving for next temperatures at each time step using finite difference method

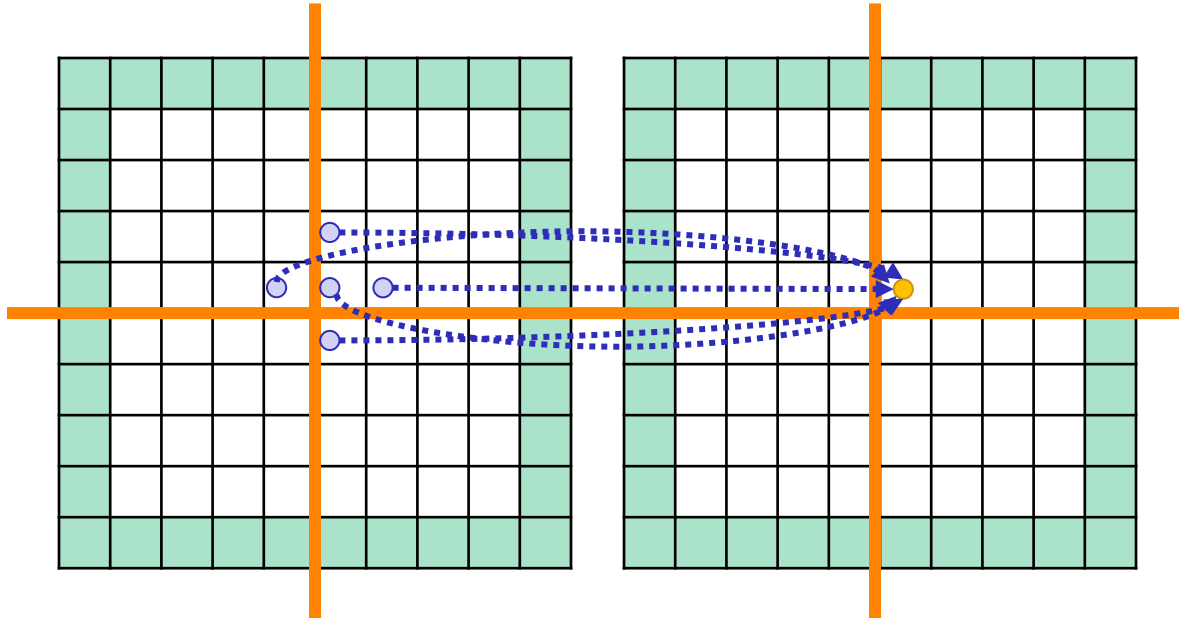
$$\boxed{u_{i,j}^{n+1} = u_{i,j}^{n} + \alpha\left(u_{i+1,j}^{n} + u_{i-1,j}^{n} - 4u_{i,j}^{n} + u_{i,j+1}^{n} + u_{i,j-1}^{n}\right)}$$

- All updates in a timestep can be done in parallel

```
forall (i, j) in indicesInner do
    u[i, j] = un[i, j] + alpha *
      (un[i, j-1] + un[i-1, j] + un[i+1, j] +
       un[i, j+1] - 4 * un[i, j]);
```

- Output is the mean and standard deviation of all the values and time to solution

DISTRIBUTED AND PARALLEL HEAT DIFFUSION IN HEAT_2D_DIST.CHPL



u^n

Stored in un

u^{n+1}

Stored in u

- Declaring 'u' array

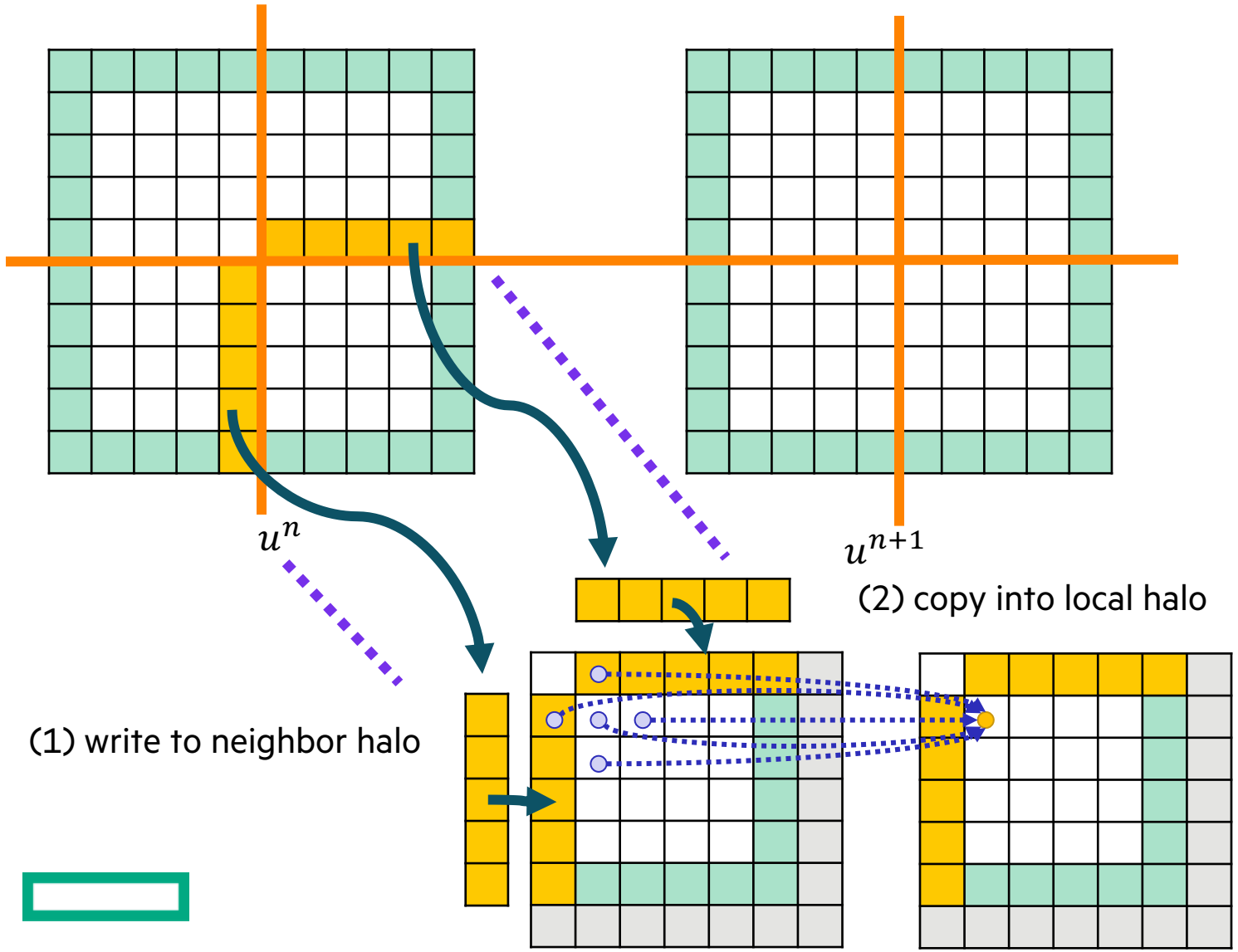
```
const indices = {0..<nx, 0..<ny}
var u: [indices] real;
```

- Declaring 'u' array as distributed

```
const indices = {0..<nx, 0..<ny},
      INDICES = Block.createDomain(indices);
var u: [INDICES] real;
```

- Reads that cross the distribution boundary will result in a remote get

HALO BUFFER OPTIMIZATION IN HEAT_2D_DIST_BUFFERS.CHPL



- Each locale has own copies of 'u' and 'un' subdomains with a one-cell halo
- (1) Array assignment writes edge values into neighbors' halo landing pads
- (2) copy into local halo
- (3) compute next u in parallel locally

HALO BUFFER OPTIMIZATION CODE

```
const indices = {0..<nx, 0..<ny},
      indicesInner = indices.expand(-1),
      INDICES = Block.createDomain(indices);
const u: [INDICES] real;
...
var   LOCALE_DOM = Block.createDomain(u.targetLocales().domain);
var   haloArrays: [LOCALE_DOM][0..<4] haloArray;
param N = 0, S = 1, E = 2, W = 3;
...
for 1..nt {
  haloArrays[tidX, tidY-1][E].v = uLocal2[.., WW+1];
  ...
  b.barrier();
  uLocal1 <=> uLocal2;

  uLocal1[.., WW] = haloArrays[tidX, tidY][W].v;
  ...
  forall (i,j) in localIndicesInner do
    uLocal2[i,j] = uLocal1[i,j] + alpha*(uLocal1[i-1,j] + uLocal1[i+1,j]
      + uLocal1[i,j-1] + uLocal1[i,j+1] - 4*uLocal1[i,j]);
  b.barrier();
}
```

Declare and distribute 'u' array.

Declare North, South, East, and West halo arrays per locale

Copy local edge results into neighbor's halo array. 'tidX' and 'tidY' are the locale's task id X and Y coordinates. Using array slicing in 'uLocal2[..,WW+1]'.

Copy halo array into local halo.

Compute u[I,j] in local subdomain.

Barrier over all locales

2D HEAT DIFFUSION EXAMPLE

```
make run-heat_2D
make run-heat_2D_dist
make run-heat_2D_dist_buffers
```

- **See 'diffusion/heat_2D.*.chpl' in the Chapel examples**
  - 'heat_2D.chpl' - shared memory parallel version that runs in locale 0
  - 'heat_2D_dist.chpl' - parallel and distributed version that is the same as 'heat_2D.chpl' but with distributed arrays
  - 'heat_2D_dist_buffers.chpl' - parallel and distributed version that copies to neighbors landing pad and then into local halos

- **Concepts illustrated**
  - 'forall' provides distributed and shared memory parallelism when do a 'forall' over the 2D Block distributed array
  - 'heat_2D_dist.chpl' version doesn't do any special handling of the halo exchange
  - 'heat_2D_dist_buffers.chpl' shows an optimization that explicitly copies subarrays into buffers

IMAGE PROCESSING EXAMPLE

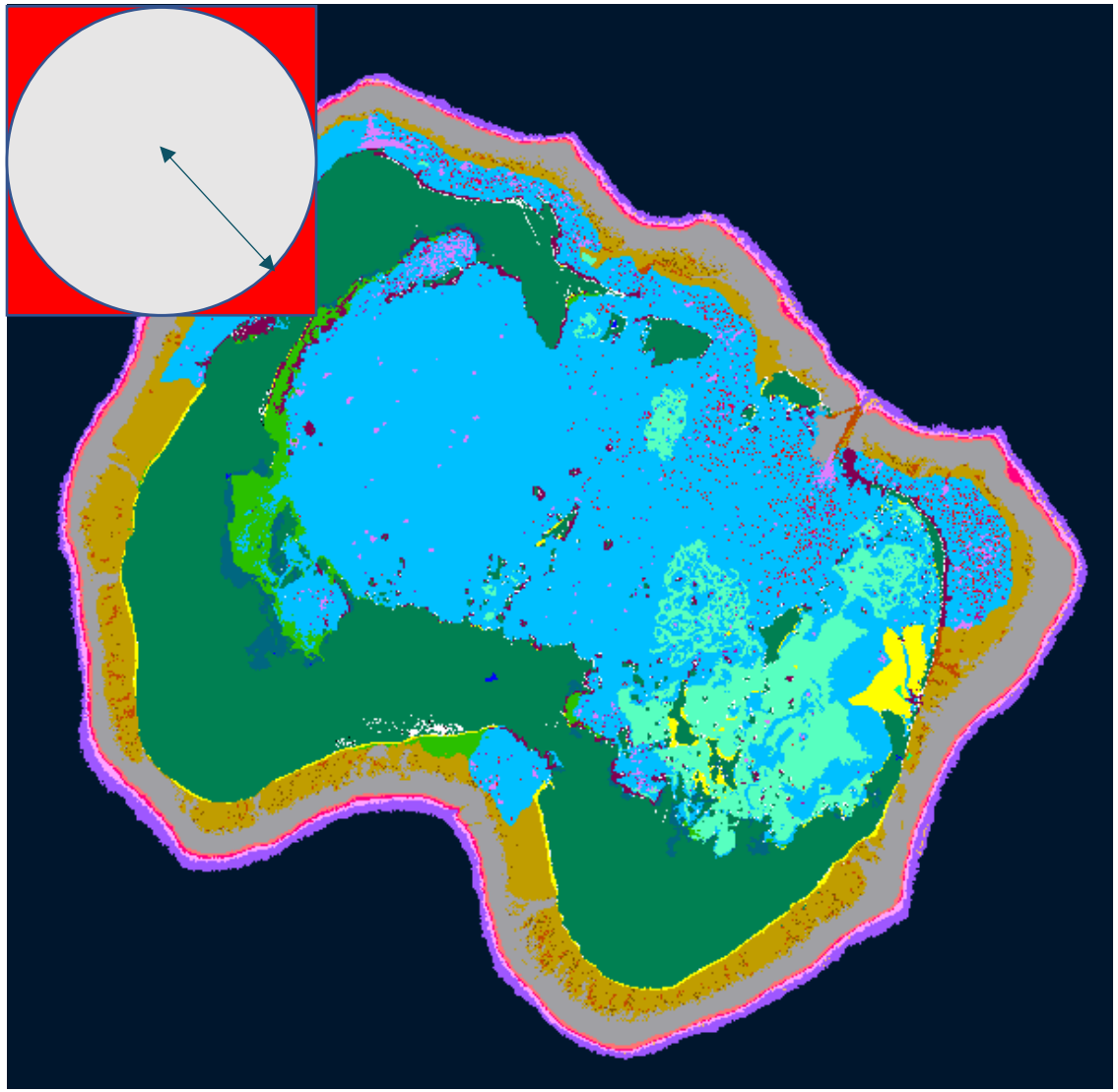
- **See 'image_analysis/' subdirectory in the Chapel examples**
  - Coral reef diversity analysis written by Scott Bachman
  - Reads a single file in parallel
  - Uses distributed and shared memory parallelism
  - Is being used and modified by Scott and collaborators for climate research

- **'image_analysis/README' explains how to compile and run it**

```
cd image_analysis
chpl main.chpl --fast
./main -nl 2 --in_name=banda_ai --map_type=benthic --window_size=100000
```

IMAGE PROCESSING FOR CORAL REEF DISSIMILARITY

- **Analyzing images for coral reef diversity**
  - Important for prioritizing interventions
- **Algorithm implemented productively**
  - Add up weighted values of all points in a neighborhood, i.e., convolution over image
  - Developed by Scott Bachman, NCAR scientist who is a visiting scholar on the Chapel team
  - Scott started learning Chapel in Sept 2022, started Coral Reef app in Dec 2022, already had collaborators presenting results in Feb 2023
  - Last week with ~5 lines changed, ran on a GPU
- **Performance**
  - Less than 300 lines of Chapel code scales out to 100s of processors on Cheyenne (NCAR)
  - Full maps calculated in *seconds*, rather than days



Distributed Parallelism: Divide the domain into “strips” and allocate a task per strip

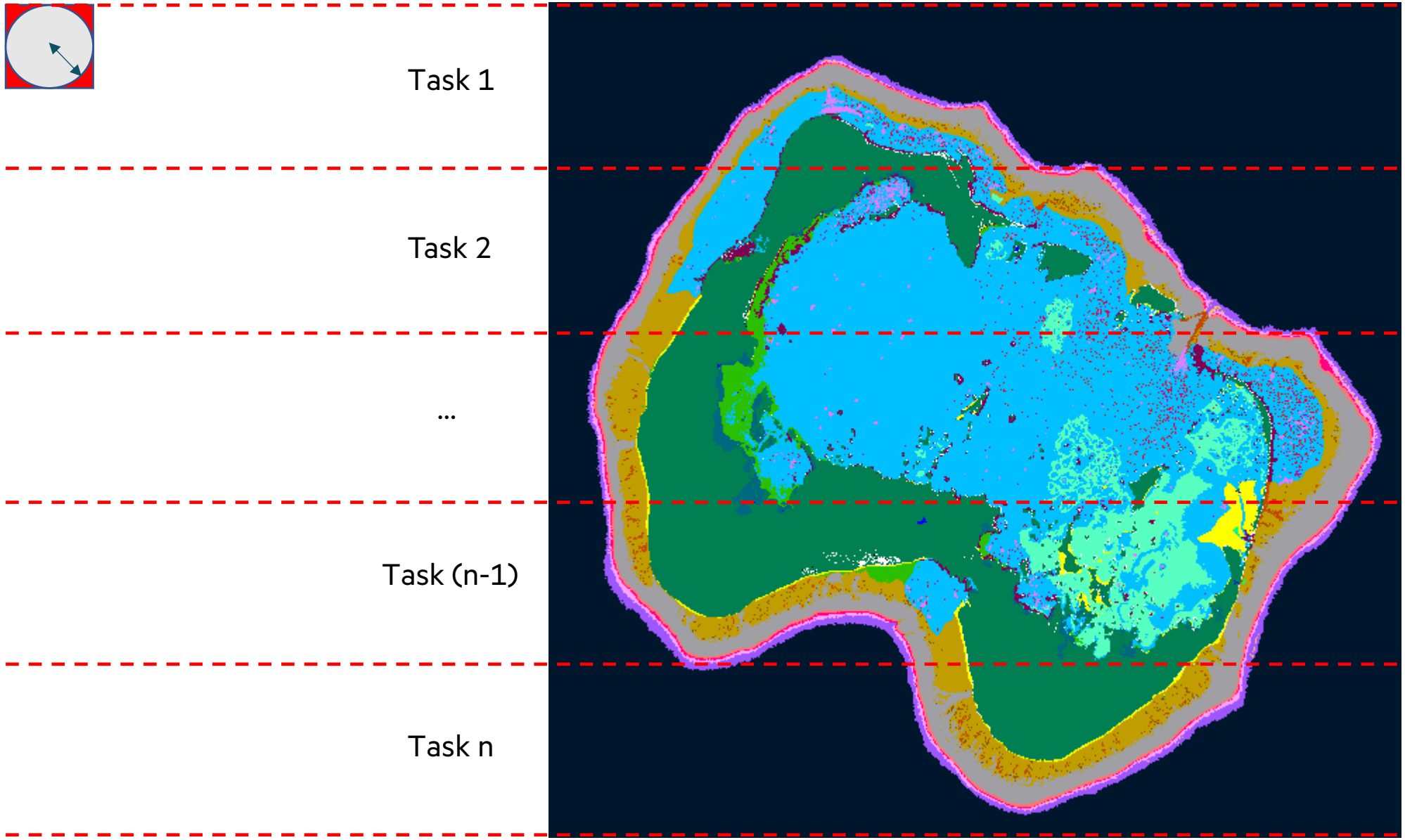


IMAGE PROCESSING EXAMPLE

- **See 'image_analysis/' subdirectory in the Chapel examples**
  - Coral reef diversity analysis written by Scott Bachman
  - Reads a single file in parallel
  - Uses distributed and shared memory parallelism
  - Is being used and modified by Scott and collaborators for climate research

- **'image_analysis/README' explains how to compile and run it**

- **Concepts illustrated**
  - User-defined modules
  - Reading a single file in parallel
  - Sparse domains used to create masks in 'distance_mask.chpl'
  - Creating a 1D block distribution by reshaping the 'Locales' array
  - Gets to locale 0 will occur for some smaller arrays that live on locale 0

GPU SUPPORT IN CHAPEL

- **Generate code for GPUs**
  - Support for NVIDIA and AMD GPUs
  - Exploring Intel support
- **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
- **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
- **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());
```

STREAM TRIAD: DISTRIBUTED MEMORY, GPUS AND CPUS

stream-ep.chpl

```
config var n = 1_000_000,
           alpha = 0.01;

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

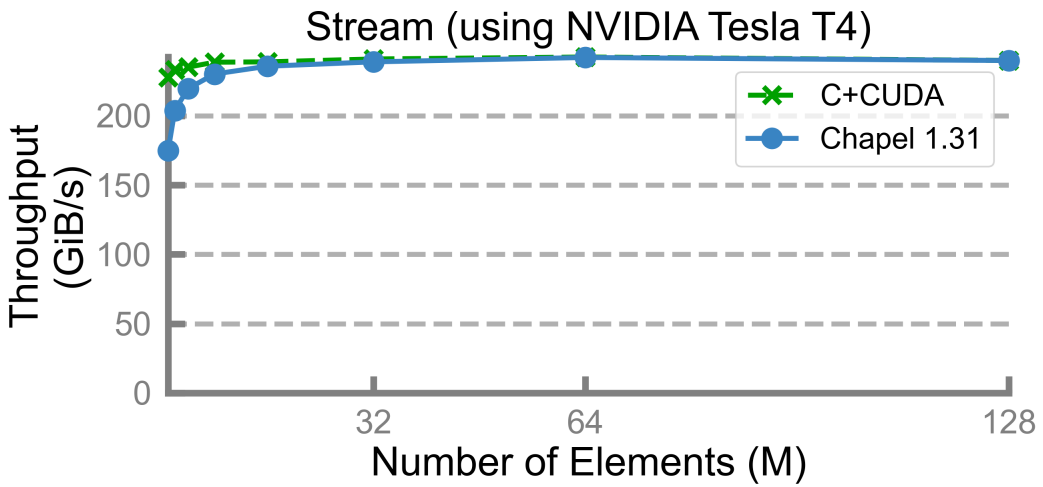
'cobegin { … }' creates a task per child statement

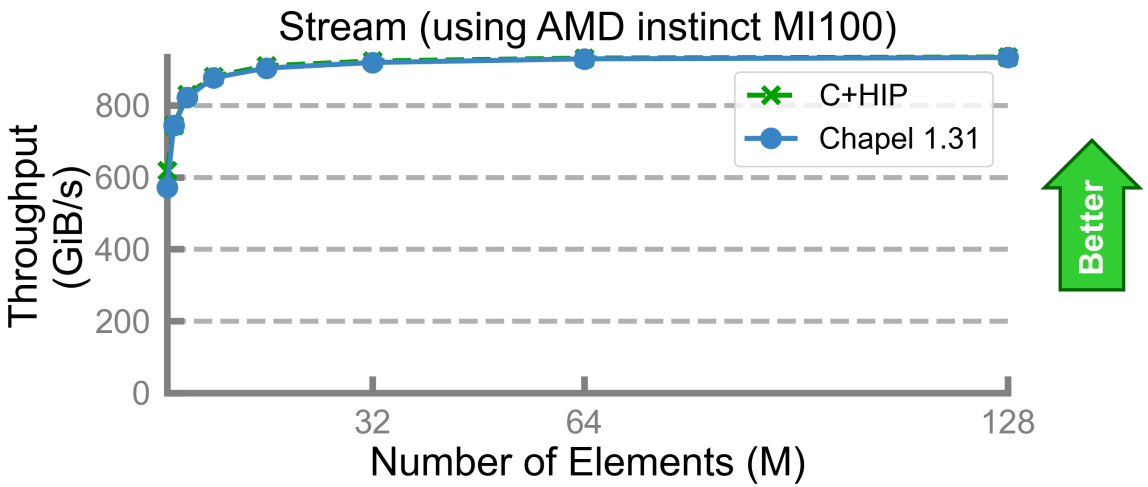
one task runs our multi-GPU triad

the other runs the multi-CPU triad

This program uses all CPUs and GPUs across all of your compute nodes

STREAM TRIAD: PERFORMANCE VS. REFERENCE VERSIONS

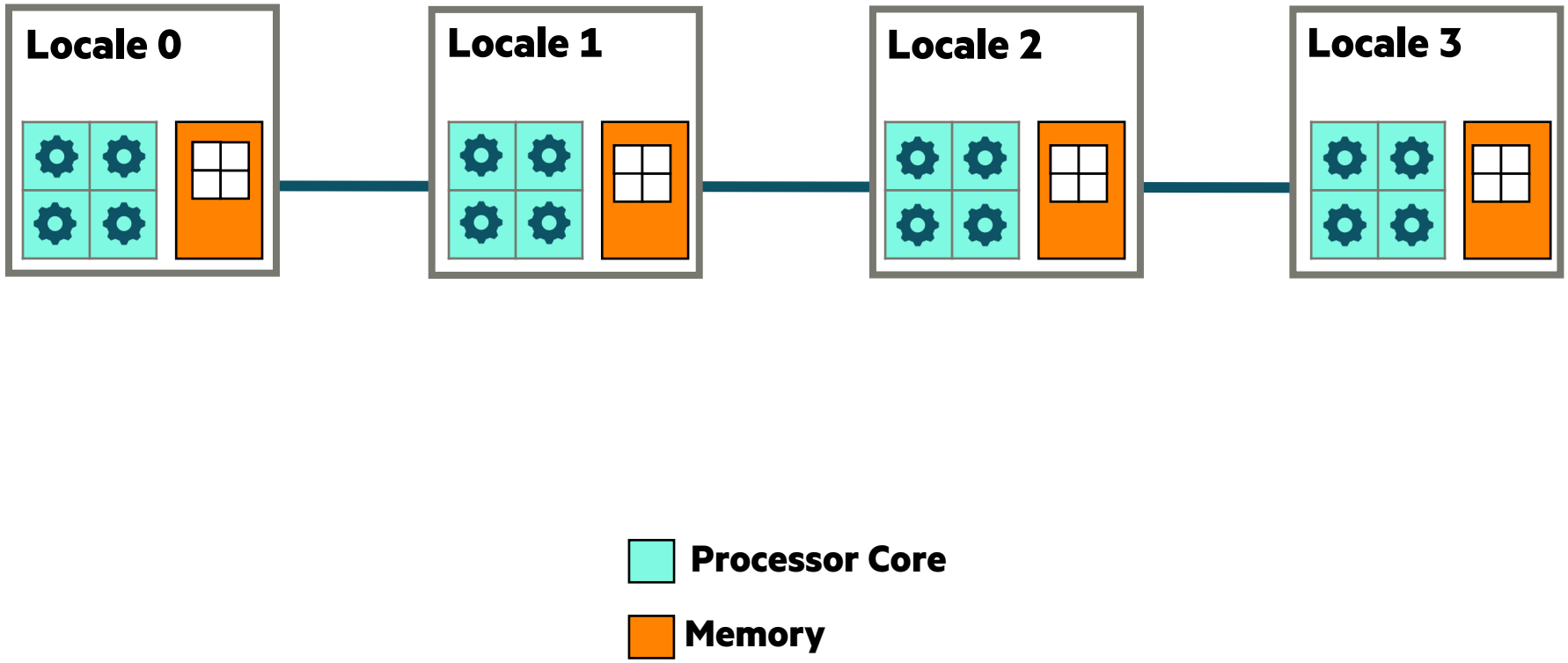




Performance vs. reference versions has become competitive as of the last release

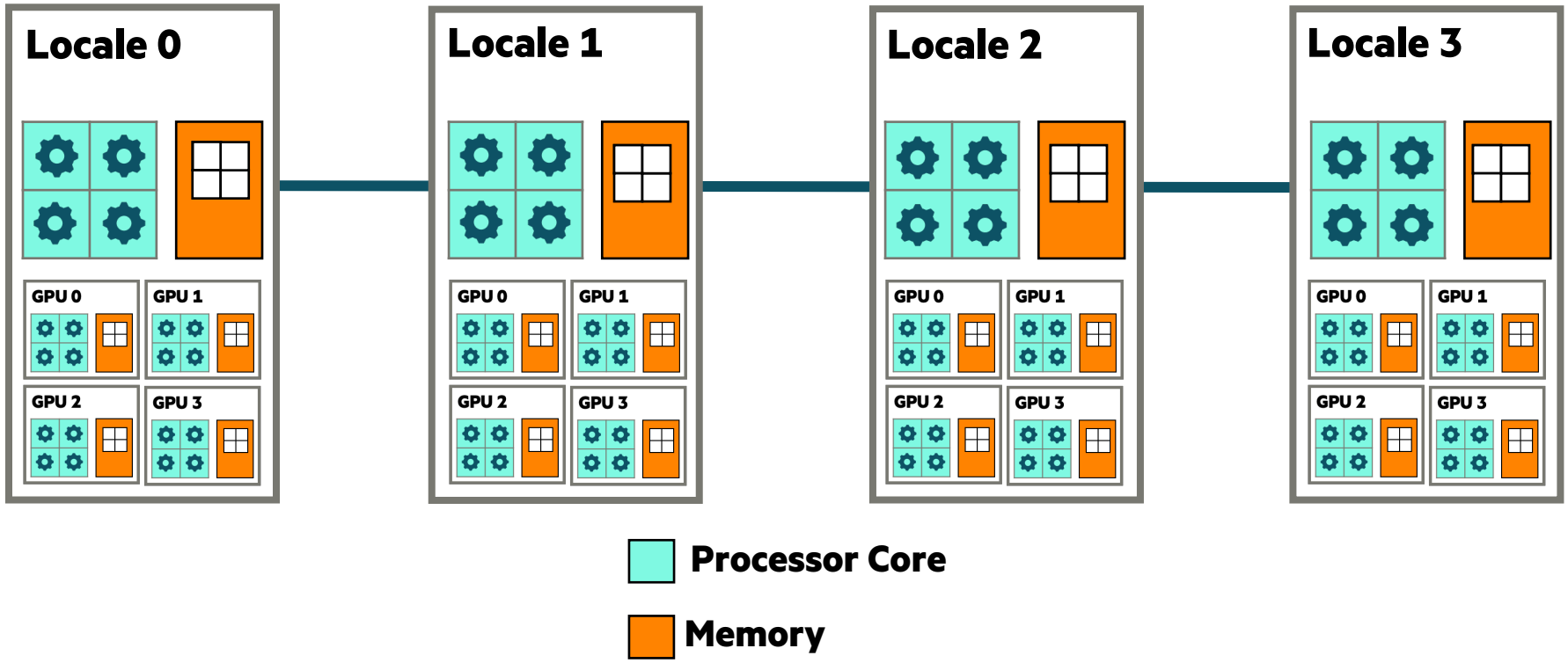
KEY CONCERNS FOR SCALABLE PARALLEL COMPUTING

1. **parallelism:** What 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:** What 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



STREAM TRIAD: DISTRIBUTED MEMORY, CPUS ONLY

stream-glbl.chpl

```
config const n = 1_000_000,
             alpha = 0.01;

use BlockDist;

const Dom = Block.createDomain({1..n});
var A, B, C: [Dom] real;

A = B + alpha * C;
```

These programs are both CPU-only

Nothing refers to GPUs, explicitly or implicitly

stream-ep.chpl

```
config const n = 1_000_000,
             alpha = 0.01;

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

STREAM TRIAD: DISTRIBUTED MEMORY, GPUS ONLY

stream-ep.chpl

```
config const n = 1_000_000,
             alpha = 0.01;

coforall loc in Locales do on loc {

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



}
```

Use a similar ‘coforall’ + ‘on’ idiom to run a Triad concurrently on each of this locale’s GPUs

This is a GPU-only program

Nothing other than coordination code runs on the CPUs

STREAM TRIAD: DISTRIBUTED MEMORY, GPUS AND CPUS

stream-ep.chpl

```
config const n = 1_000_000,
             alpha = 0.01;

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

‘cobegin { … }’ creates a task per child statement

one task runs our multi-GPU triad

the other runs the multi-CPU triad

This program uses all CPUs and GPUs across all of our compute nodes

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

TUTORIAL SUMMARY

- **Takeaways**
  - Chapel is a PGAS 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**
  - Ask the Chapel team and users questions on discourse, gitter, or stack overflow
  - Also feel free to email me at michelle.strout@hpe.com
- **Engaging with the community**
  - Share your sample codes with us and your research community!
  - Join us at our free, virtual workshop in June, https://chapel-lang.org/CHIUW.html

CHAPEL RESOURCES

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

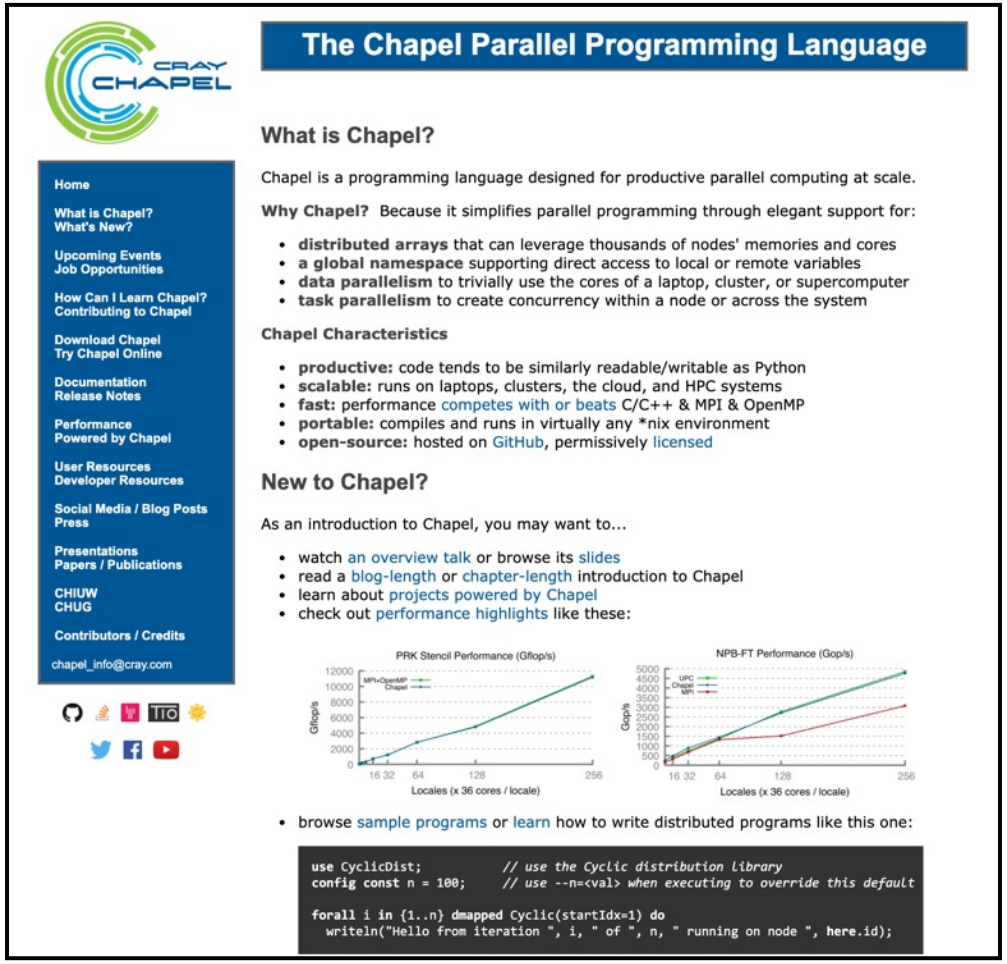
- (points to all other resources)

Social Media:

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

Community Discussion / Support:

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



CURRENT CHAPEL TEAM AT HPE

BACKUP SLIDES AND ADDITIONAL CONTENT

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

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

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

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

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

OTHER TASK PARALLEL FEATURES

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

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

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

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

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

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

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

SPECTRUM OF CHAPEL FOR-LOOP STYLES

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

- predictable execution order, similar to conventional languages

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

- a candidate for vectorization, SIMD execution on GPUs

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

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

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

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

- explicit parallelism; supports synchronization between iterations (tasks)

SIDEBAR: PROMOTION OF SCALAR SUBROUTINES

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

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

- So, in the Jacobi computation,

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

==

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