SC23
Denver, CO | i am hpc.

Getting Started Using Chapel for Parallel Programming

SC23 Tutorial - Denver

Michelle Strout and the Chapel Team

November 12, 2023

Hewlett Packard
Enterprise

GETTING STARTED USING CHAPEL FOR PARALLEL PROGRAMMING

SC23 Tutorial - Denver
Michelle Strout and the Chapel Team
November 12, 2023

OUTLINE: OVERVIEW OF PROGRAMMING IN CHAPEL

- Chapel Goals, Usage, and Comparison with other Tools
- Hello World (Demo with Example Codes)
- Chapel Execution Model and Parallel Hello World
- Serial programming in Chapel: k-mer counting using file IO, config consts, strings, maps
- Parallelizing a program that processes files
- Distributed parallelism for Heat 2D problem
- GPU programming support

CHAPEL GOALS, USAGE, AND COMPARISON WITH OTHER TOOLS

CHAPEL PROGRAMMING LANGUAGE

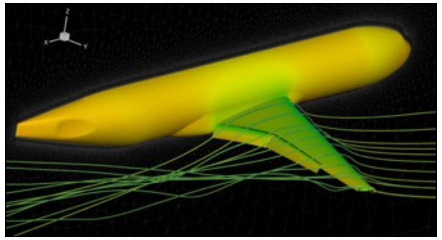
Chapel is a general-purpose programming language that provides

ease of parallel programming,
high performance, and
portability.

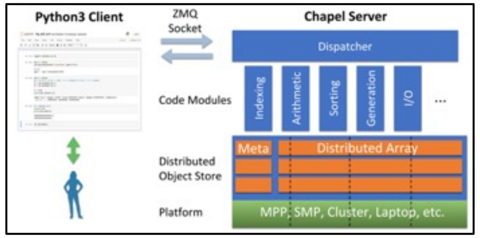
And is being used in applications in various ways:

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.

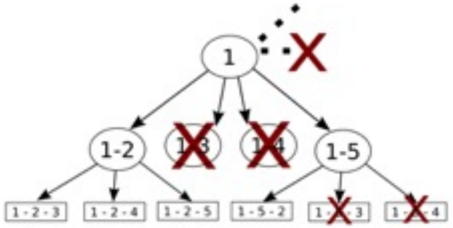
APPLICATIONS OF CHAPEL



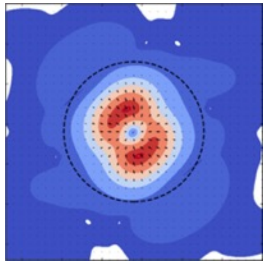
CHAMPS: 3D Unstructured CFD
Laurendeau, Bourgault-Côté, Parenteau, Plante, et al.
École Polytechnique Montréal



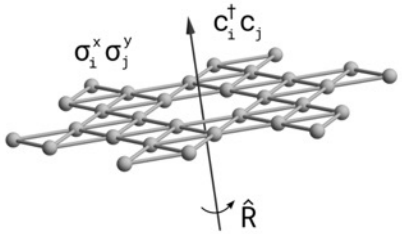
Arkouda: Interactive Data Science at Massive Scale
Mike Merrill, Bill Reus, et al.
U.S. DoD



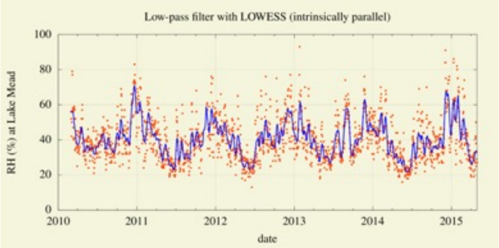
ChOp: Chapel-based Optimization
T. Carneiro, G. Helbecque, N. Melab, et al.
INRIA, IMEC, et al.



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

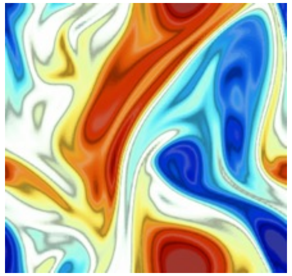


Lattice-Symmetries: a Quantum Many-Body Toolbox
Tom Westerhout
Radboud University

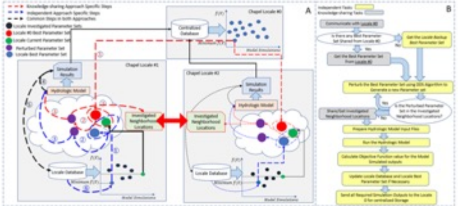


Desk dot chpl: Utilities for Environmental Eng.
Nelson Luis Dias
The Federal University of Paraná, Brazil

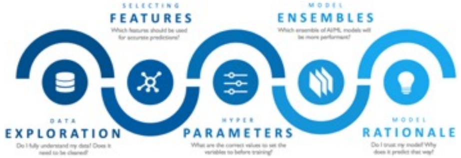
RapidQ: Mapping Coral Biodiversity
Rebecca Green, Helen Fox, Scott Bachman, et al.
The Coral Reef Alliance



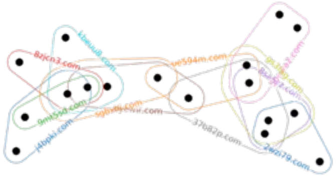
ChapQG: Layered Quasigeostrophic CFD
Ian Grooms and Scott Bachman
University of Colorado, Boulder et al.



Chapel-based Hydrological Model Calibration
Marjan Asgari et al.
University of Guelph



CrayAI HyperParameter Optimization (HPO)
Ben Albrecht et al.
Cray Inc. / HPE



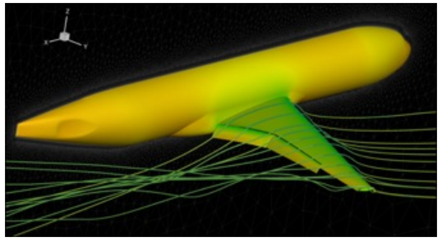
CHGL: Chapel Hypergraph Library
Louis Jenkins, Cliff Joslyn, Jesun Firoz, et al.
PNNL

?

Your Application Here?

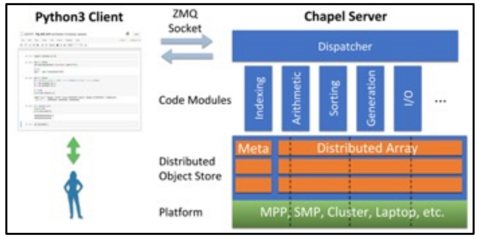
(images provided by their respective teams and used with permission)

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



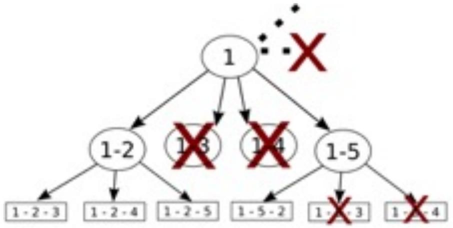
CHAMPS: 3D Unstructured CFD

CHIUW 2021 CHIUW 2022



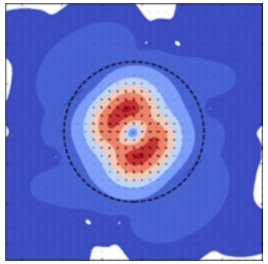
Arkouda: Interactive Data Science at Massive Scale

CHIUW 2020 CHIUW 2023



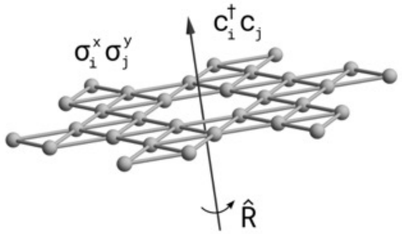
ChOp: Chapel-based Optimization

CHIUW 2021 CHIUW 2023



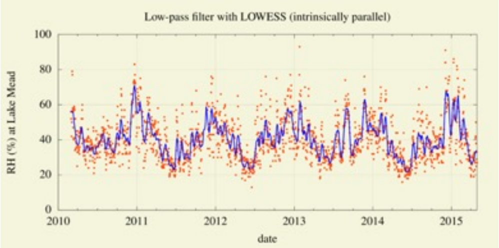
ChplUltra: Simulating Ultralight Dark Matter

CHIUW 2020 CHIUW 2022



Lattice-Symmetries: a Quantum Many-Body Toolbox

CHIUW 2022

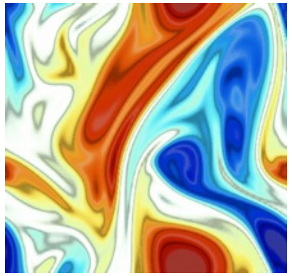


Desk dot chpl: Utilities for Environmental Eng.

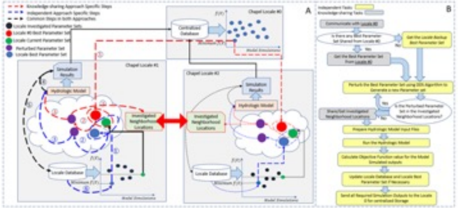
CHIUW 2022

RapidQ: Mapping Coral Biodiversity

CHIUW 2023

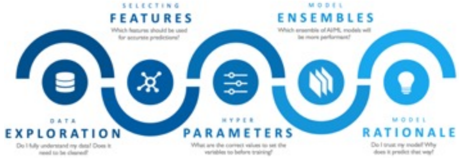


ChapQG: Layered Quasigeostrophic CFD



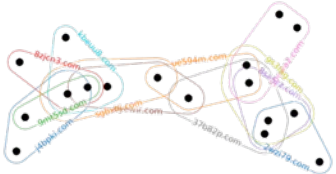
Chapel-based Hydrological Model Calibration

CHIUW 2023



CrayAI HyperParameter Optimization (HPO)

CHIUW 2021



CHGL: Chapel Hypergraph Library

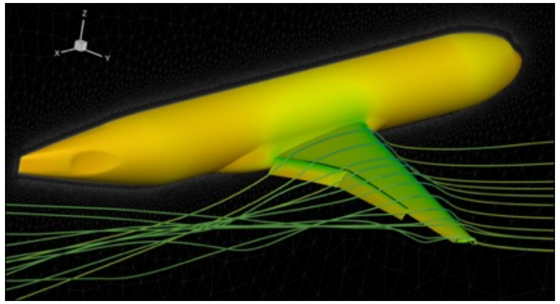
CHIUW 2020

?

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
- 9 TB/s for argsort on an HPE EX system

CHAPEL IS HIGHLY PERFORMANT AND SCALABLE

HPE Apollo (May 2021) ✕━✕

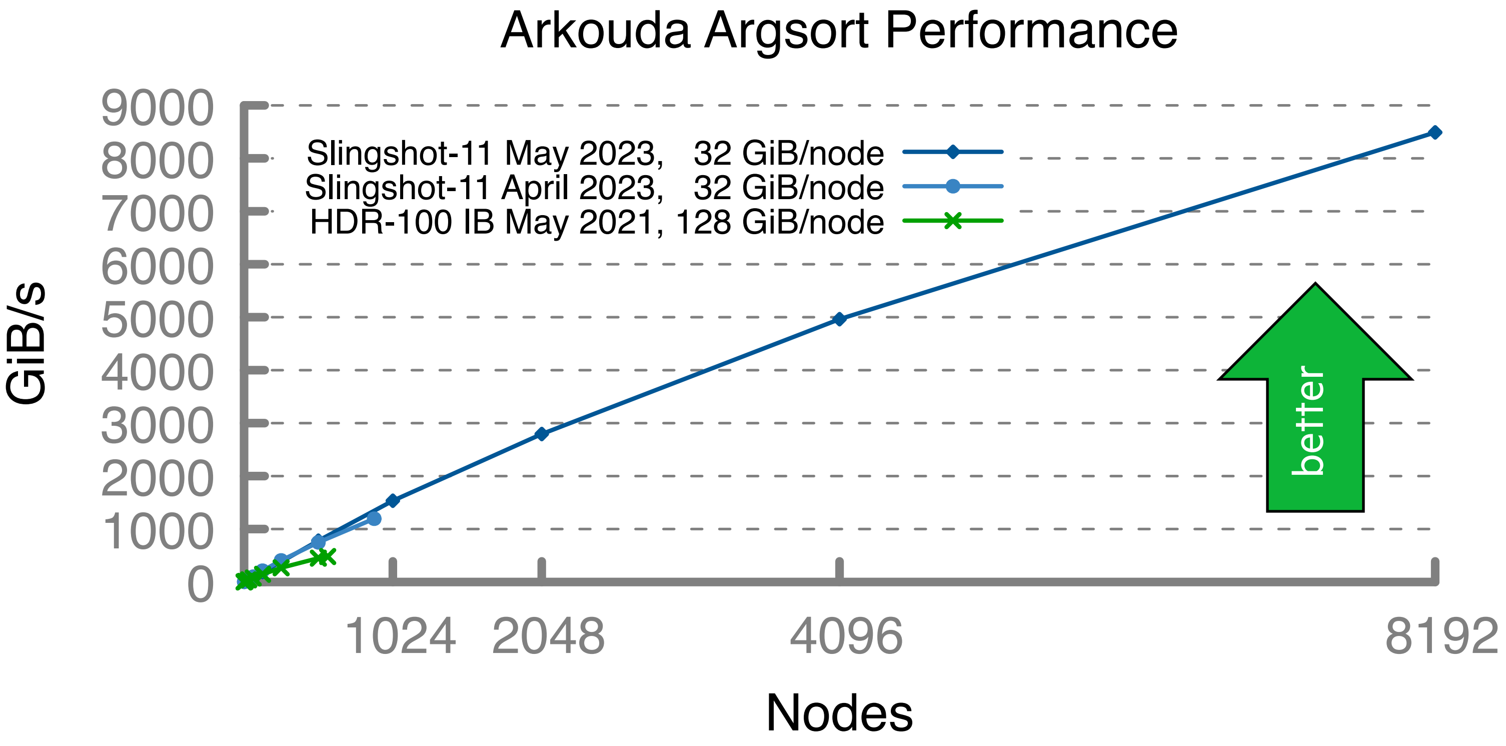
- HDR-100 Infiniband network (100 Gb/s)
- 576 compute nodes
- 72 TiB of 8-byte values
- ~480 GiB/s (~150 seconds)

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)



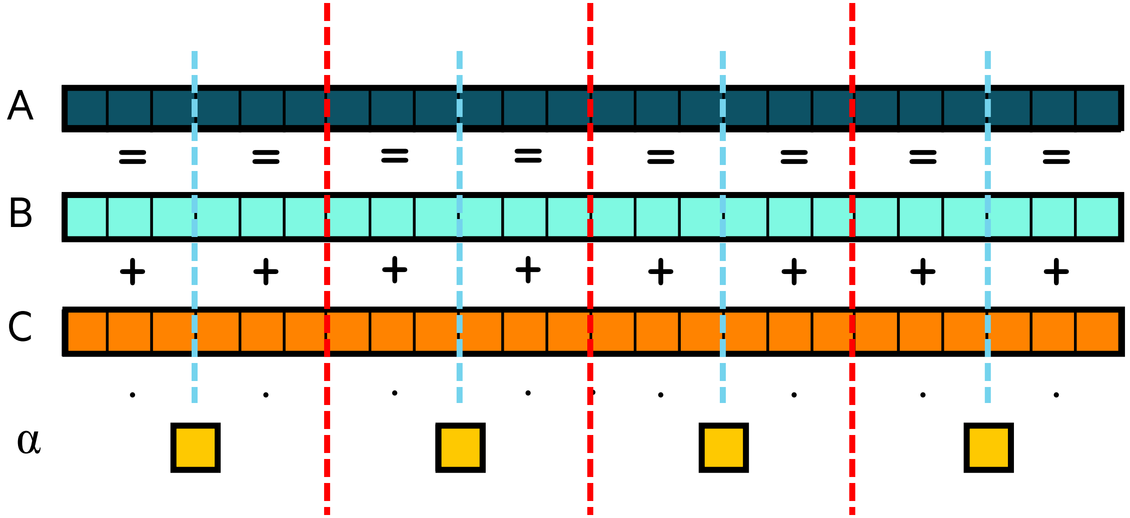
A notable performance achievement in ~100 lines of Chapel

LET'S COMPARE WITH MPI+OPENMP+CUDA USING STREAM TRIAD

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

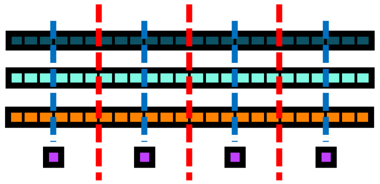
Compute: $\forall i \in 1..n,\ A_i = B_i + \alpha \cdot C_i$

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



STREAM TRIAD: IN MPI+OPENMP+CUDA

MPI + OpenMP



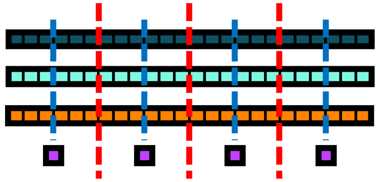
```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

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

int HPCC_StarStream(HPCC_Params *params) {
  int myRank, commSize;
```

```
  MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM,
    0, comm );

  return errCount;
}

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

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

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

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

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

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

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

  return 0;  }
```

CUDA



```
#define N 2000000

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

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

  dim3 dimBlock(128);
```

```
  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];  }
```

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

BOTTOM UP LANGUAGE DESIGN

Given a system and its core capabilities…

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

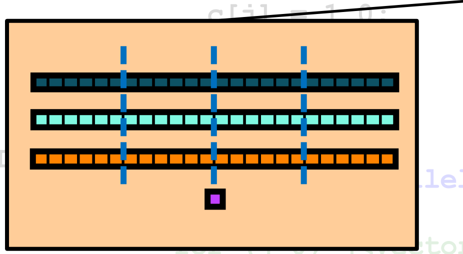
Type of HW Parallelism	Programming Model	Unit of Parallelism
Inter-node	MPI	executable
Intra-node/multicore	OpenMP / pthreads	iteration/task
Instruction-level vectors/threads	pragmas	iteration
GPU/accelerator	CUDA, ROCm, OpenMP, …	SIMD function/task

benefits: lots of control; decent generality

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

STREAM TRIAD: IN CHAPEL

```
use BlockDist;

config const m = 1000,
             alpha = 3.0;

const ProblemSpace =  {1..m};

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

B = 2.0;
C = 1.0;

A = B + alpha * C;
```



MPI

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

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

int HPCC_StarStream(HPCC_Params *pa
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;

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

  rv = HPCC_Stream( params, 0 == my
  MPI_Reduce( &rv, &errCount, 1, MP
    0, comm );

  return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doI
  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 );
```

```
lel for
torSize; j++)
  a[j] = b[j]+scalar*c[j];

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

return 0;  }
```

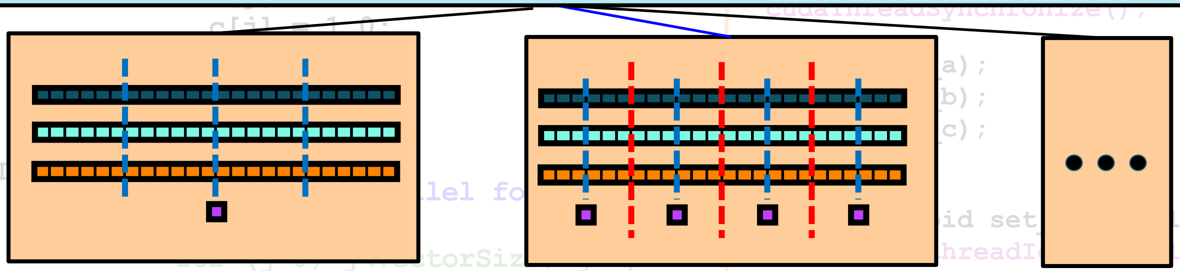
```
(float)*N);
(float)*N);
(float)*N);

id

>(d_b, .5f, N);
>(d_c, .5f, N);

>>>(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];  }
```

STREAM TRIAD: IN CHAPEL

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

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

int HPCC_StarStream(HPCC_Params *pa
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;

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

  rv = HPCC_Stream( params, 0 == my
  MPI_Reduce( &rv, &errCount, 1, MP
    0, comm );

  return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doI
  register int j;
  double  scalar;

  VectorSize = HPCC_
    sizeof(double),

  a = HPCC_XMALLOC(
  b = HPCC_XMALLOC(
  c = HPCC_XMALLOC(
```

```
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:
Indicate how this index set—and any arrays and computations over it—should be mapped to the system.



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.

LEARNING OBJECTIVES FOR TODAY'S CHAPEL TUTORIAL

- Familiarity with the Chapel execution model including how to run codes in parallel on a single node and across nodes

- Learn some Chapel programming concepts
  - Parallelism and locality in Chapel
  - Serial code using map/dictionary, (k-mer counting from bioinformatics)
  - Distributed parallelism and 1D arrays, (processing files in parallel)
  - Distributed parallelism and 2D arrays, (heat diffusion problem)
  - GPU support in Chapel

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

HELLO WORLD (DEMO WITH EXAMPLE CODES)

DEMO OF HOW TO USE EXAMPLE CODES IN DOCKER

Tarball with example codes and slides

```
curl -LO https://go.lbl.gov/sc23.tar.gz
tar xzf sc23.tar.gz
cd sc23/
```

Check out the chapel-quickReference.pdf in the sc23/chapel/ subdirectory

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 connected to the 'sc23/chapel/' directory

```
docker pull docker.io/chapel/chapel-gasnet    # takes about 5+ minutes
cd chapel/
docker run --rm -it -v "$PWD":/myapp -w /myapp chapel/chapel-gasnet /bin/bash
root@xxxxxxxxx:/myapp# chpl hello.chpl
root@xxxxxxxxx:/myapp# ./hello -nl 1
```

```
make run-hello
```

[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

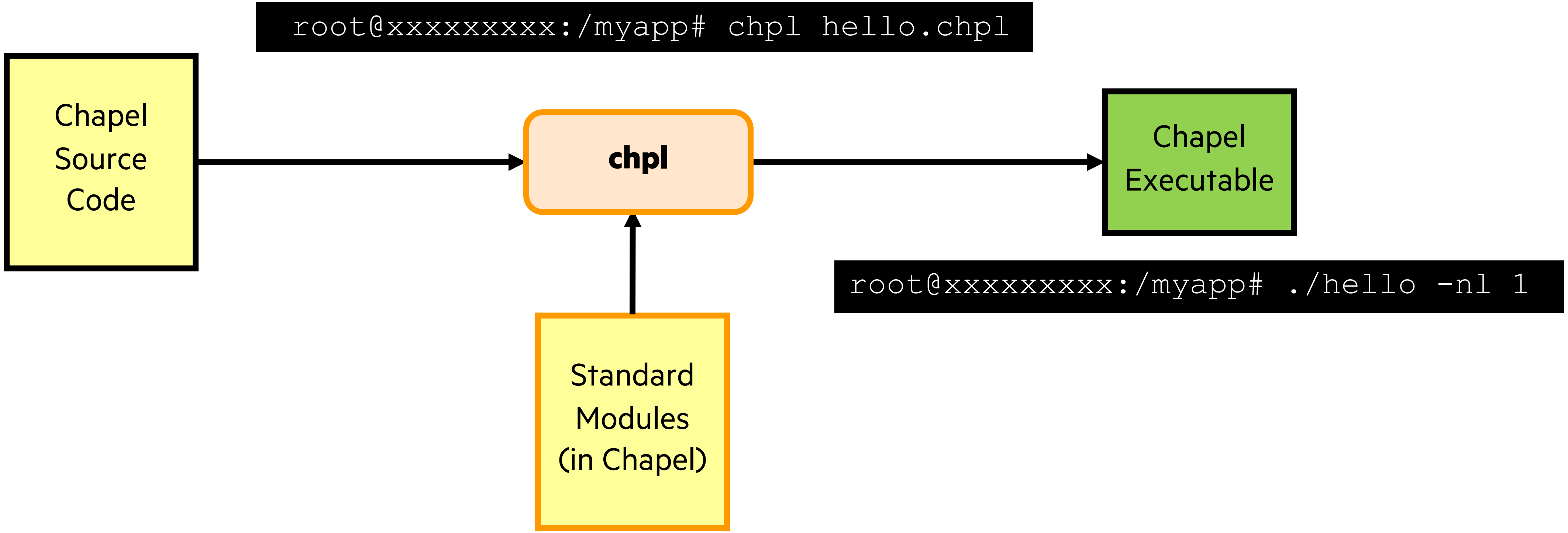
```
writeln("Hello, world!");
```

- “Production-grade”

```
module Hello {

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

}
```

COMPILING CHAPEL

```
make run-hello
```



```
root@xxxxxxxxx:/myapp# chpl hello.chpl
```

```
root@xxxxxxxxx:/myapp# ./hello -nl 1
```

CHAPEL EXECUTION MODEL AND PARALLEL HELLO WORLD

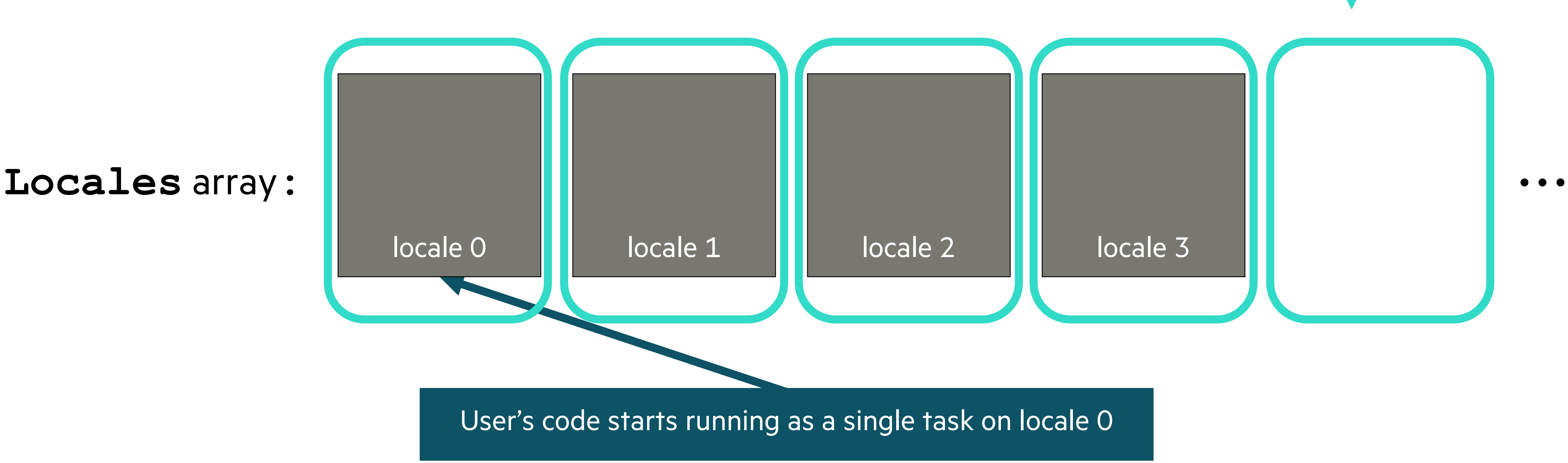
CHAPEL EXECUTION MODEL AND TERMINOLOGY: LOCALES

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:

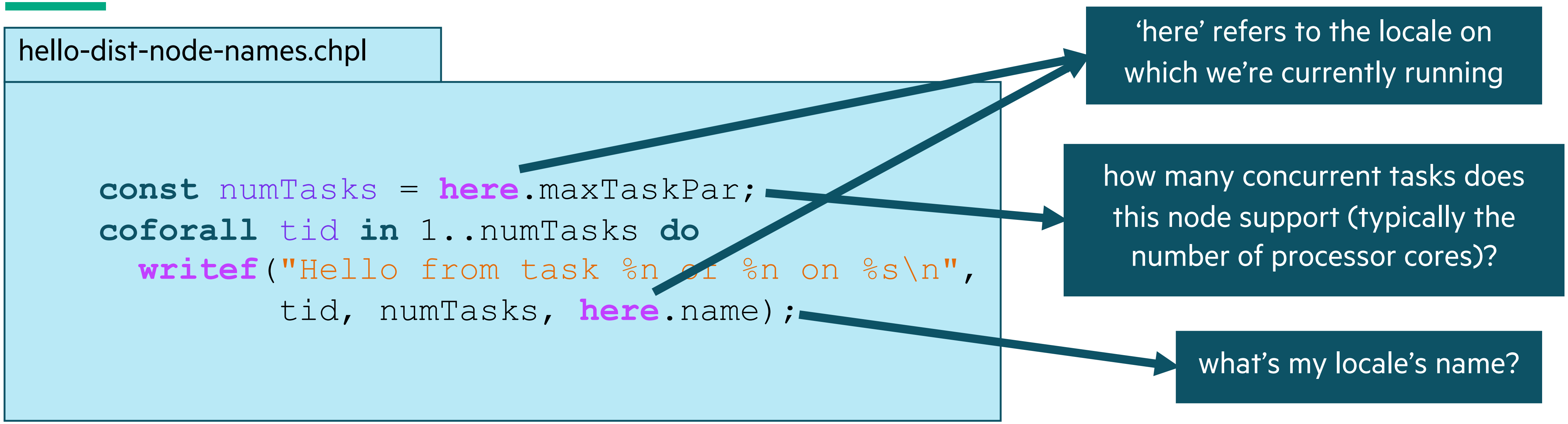
User’s code starts running as a single task on locale 0

TASK-PARALLEL “HELLO WORLD”

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”



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”

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 hello-dist-node-names.chpl
> ./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”

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

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:

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);
  }
}
```

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

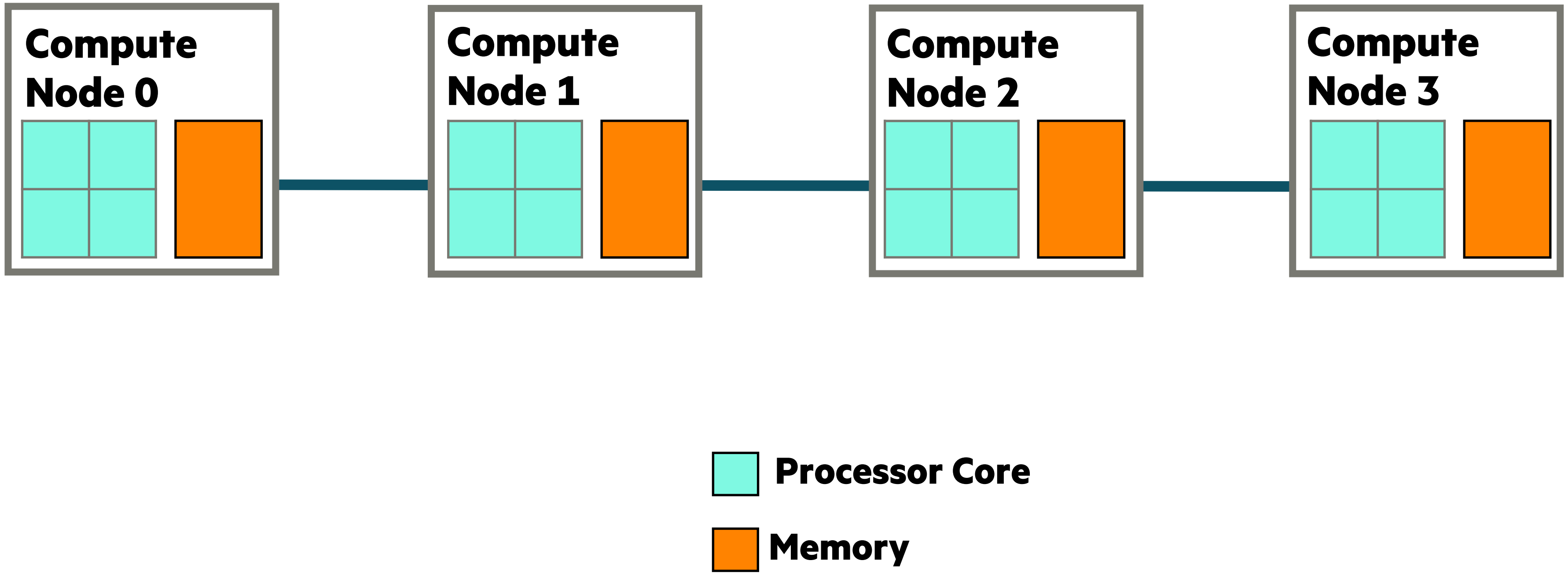
```
> chpl hello-dist-node-names.chpl
> ./hello-dist-node-names -nl=4
Hello from task 1 of 4 on n1032
Hello from task 4 of 4 on n1032
Hello from task 1 of 4 on n1034
Hello from task 2 of 4 on n1032
Hello from task 1 of 4 on n1033
Hello from task 3 of 4 on n1034
Hello from task 1 of 4 on n1035
…
```

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

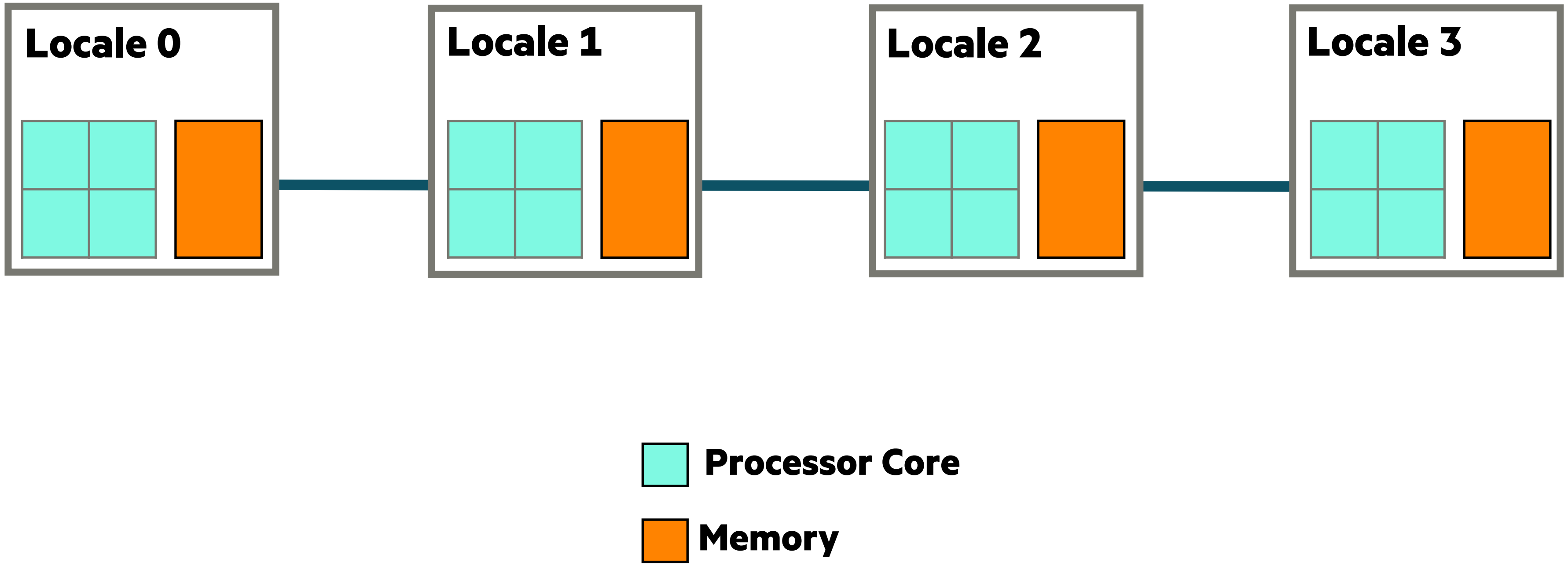
For now, think of each compute node as having one locale run on it



LOCALES AND EXECUTION MODEL IN CHAPEL

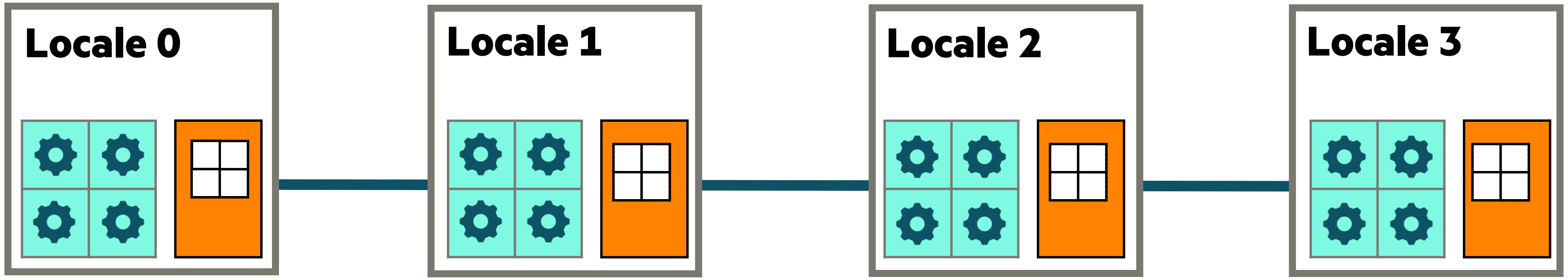
Two key built-in variables for referring to locales in Chapel programs:

- **`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?



🟦 **Processor Core**

🟧 **Memory**

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;
}
```

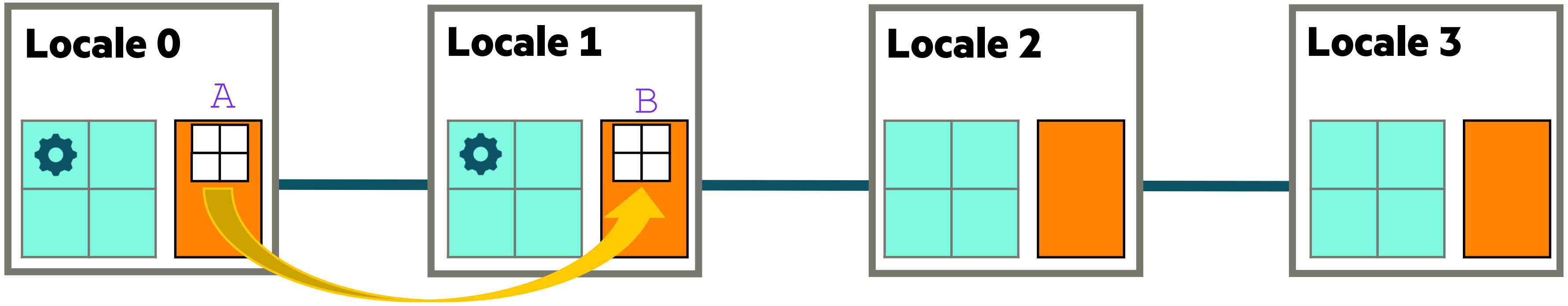
All Chapel programs begin running as a single task on locale 0

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



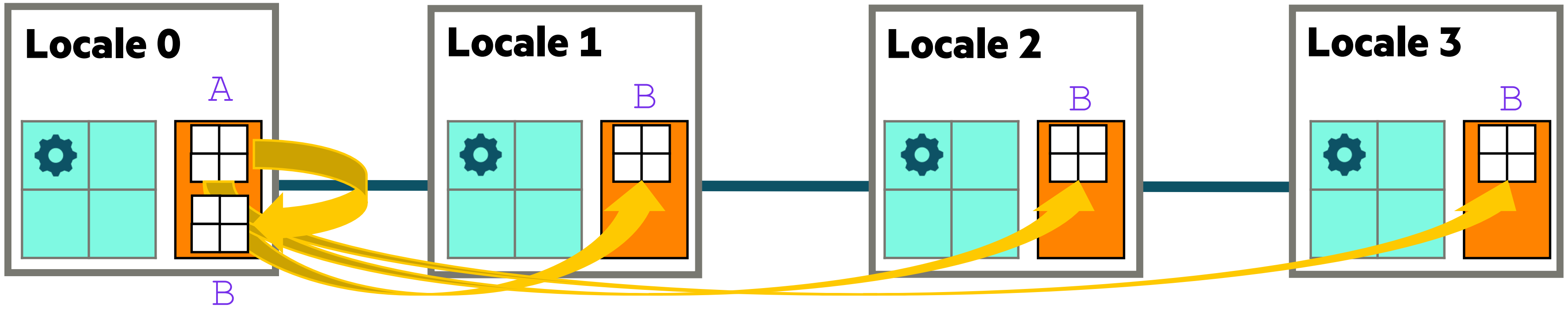
BASIC FEATURES FOR LOCALITY

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



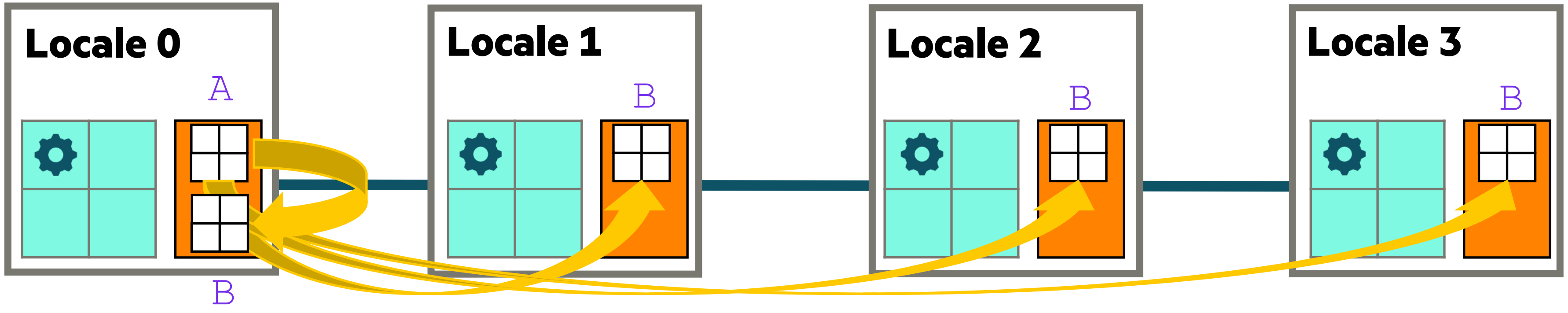
MIXING LOCALITY WITH TASK PARALLELISM

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



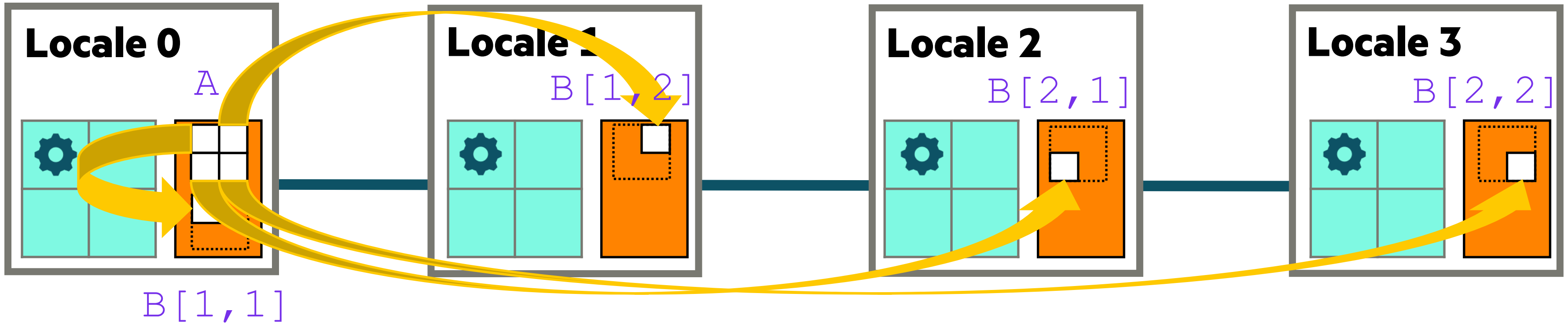
ARRAY-BASED PARALLELISM AND LOCALITY

basics-distarr.chpl

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

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

use BlockDist;

var D = 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



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);
```

OUTLINE: OVERVIEW OF PROGRAMMING IN CHAPEL

- Chapel Goals, Usage, and Comparison with other Tools
- Hello World (Demo with Example Codes)
- Chapel Execution Model and Parallel Hello World
- Serial programming in Chapel: k-mer counting using file IO, config consts, strings, maps
- Parallelizing a program that processes files
- Distributed parallelism for Heat 2D problem
- GPU programming support

K-MER COUNTING USING FILE IO, CONFIG CONSTS, AND STRINGS

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

EXERCISES: EXPERIMENTING WITH THE K-MER EXAMPLE

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

ANALYZING MULTIPLE FILES USING PARALLELISM

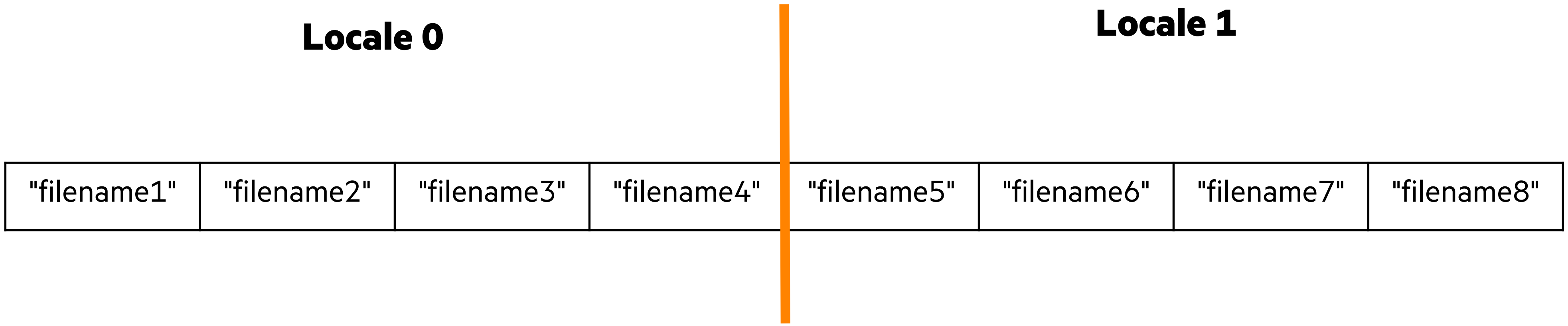
parfilekmer.chpl

```
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



Locale 0

Locale 1

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

- 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

EXERCISES: PROCESSING FILES IN PARALLEL

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' over a block distributed array provides distributed and shared memory parallelism
- No remote writes/puts and reads/gets

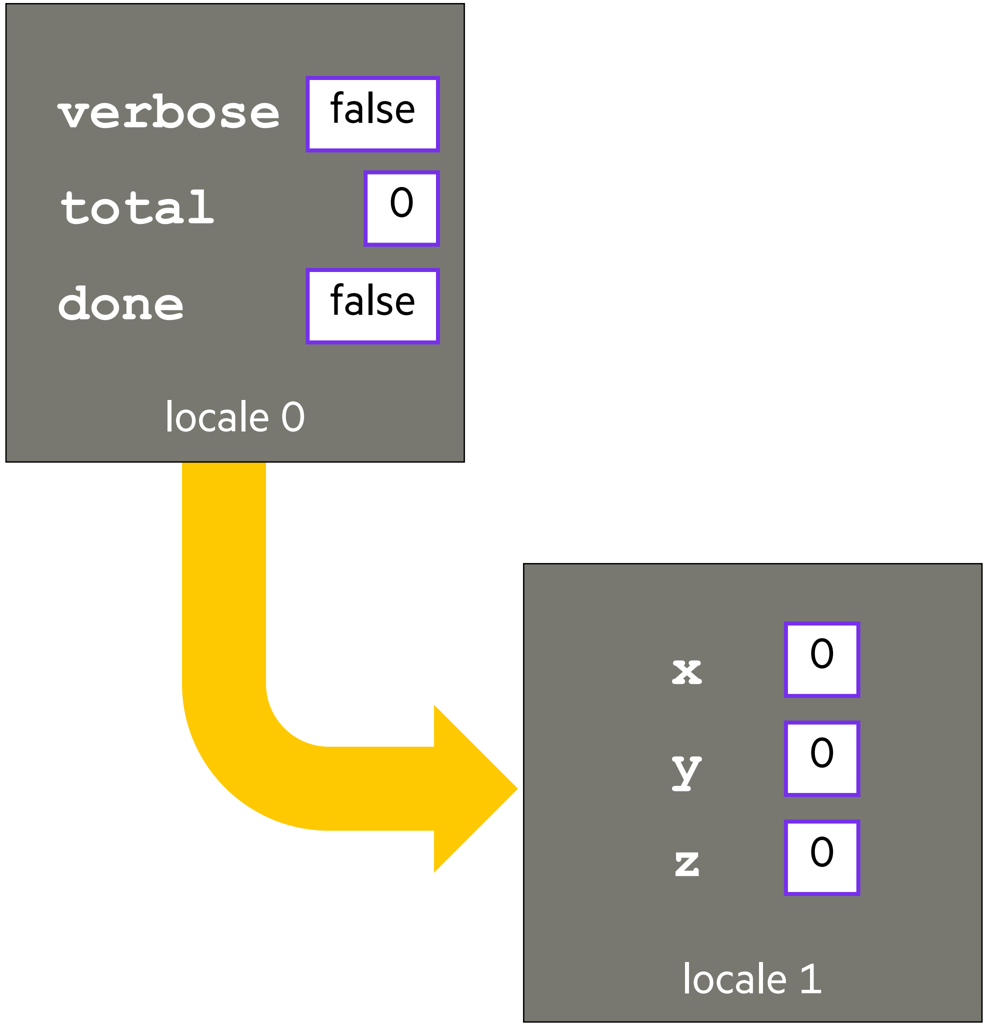
IMPLICIT COMMUNICATION:
REMOTE WRITES/PUTS AND READS/GETS

CHAPEL SUPPORTS A GLOBAL NAMESPACE WITH PUTS AND GETS

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

onClause.chpl

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

…

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



}
```

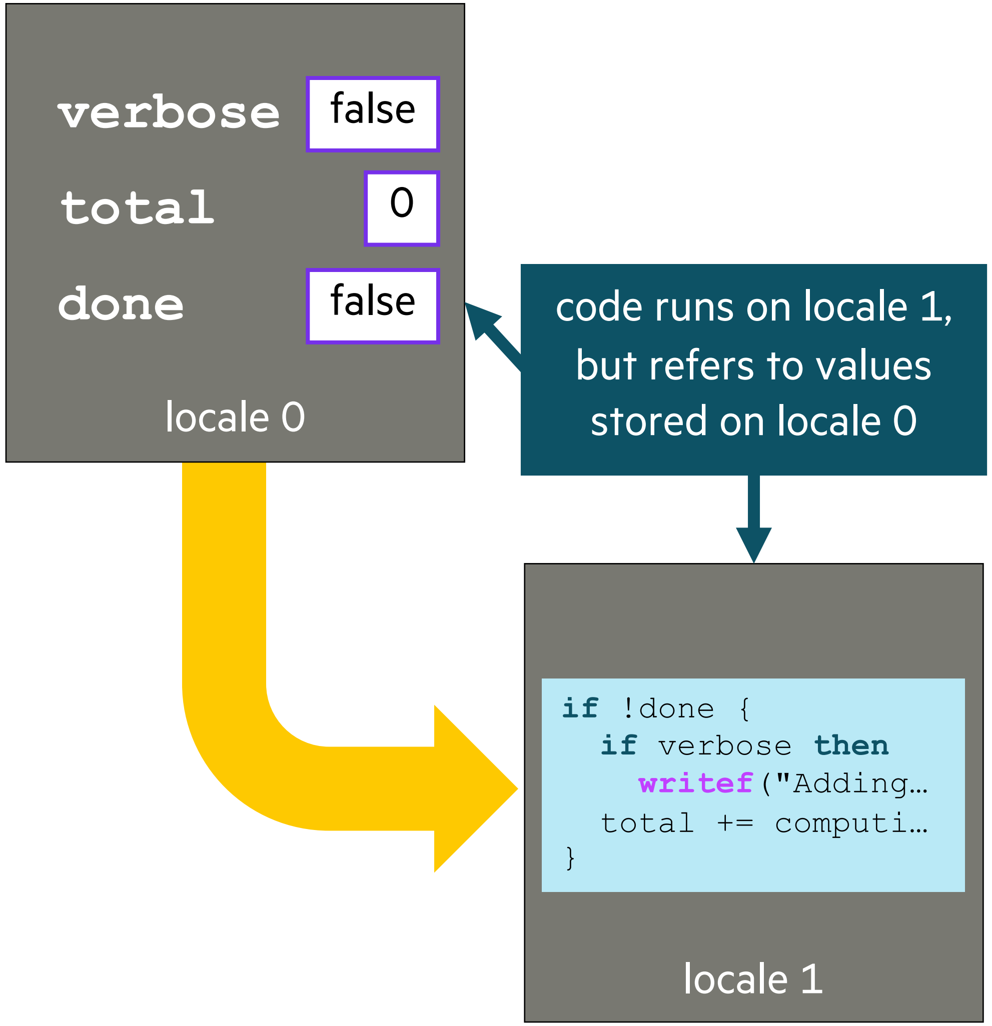


CHAPEL SUPPORTS A GLOBAL NAMESPACE WITH PUTS AND GETS

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();
  }
}
```



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

```
if !done {
  if verbose then
    writef("Adding…
  total += computi…
}
```

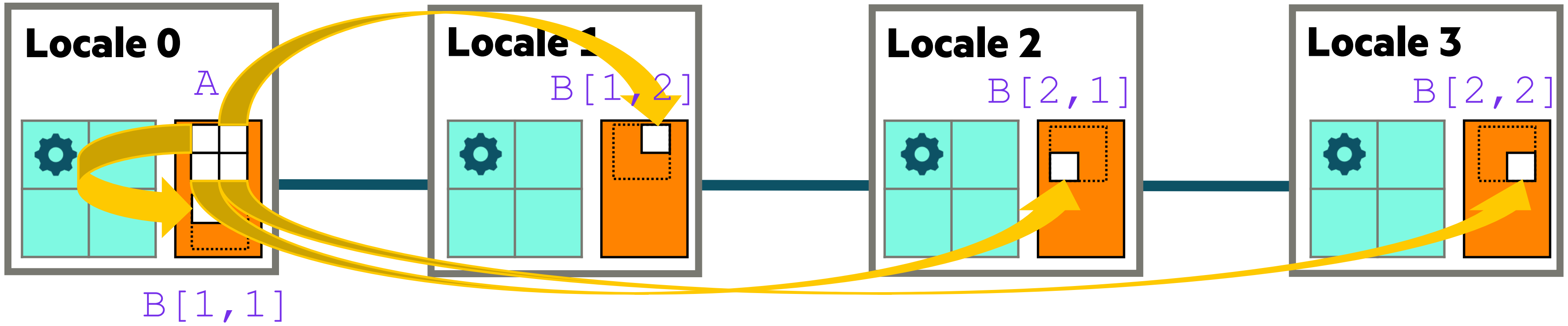
ARRAY-BASED PARALLELISM AND LOCALITY

basics-distarr.chpl

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

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

use BlockDist;

var D = 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



HEAT 2D EXAMPLE

2D HEAT EQUATION EXAMPLE

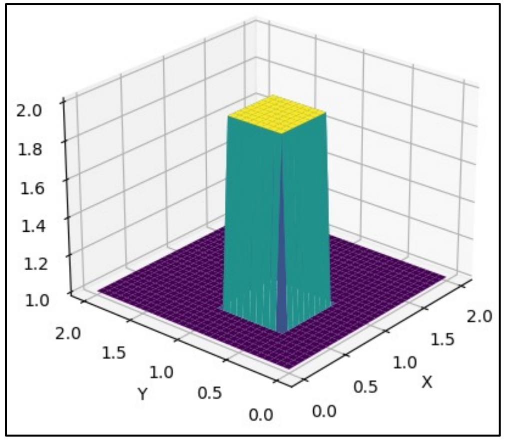
heat-2D.chpl

2D heat / diffusion PDE:

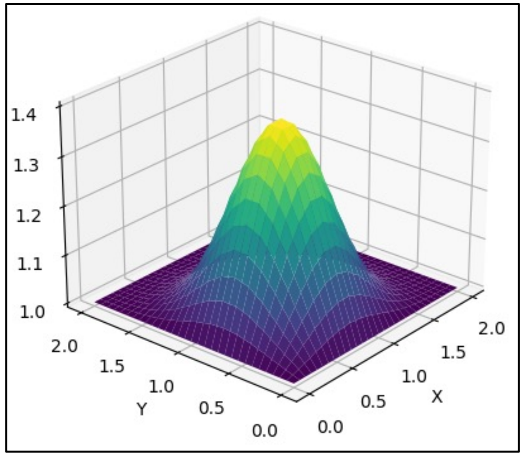
$$\frac{\partial u}{\partial t} = \alpha \Delta u = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$

Discretized (finite-difference) form:

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



$n = 0$



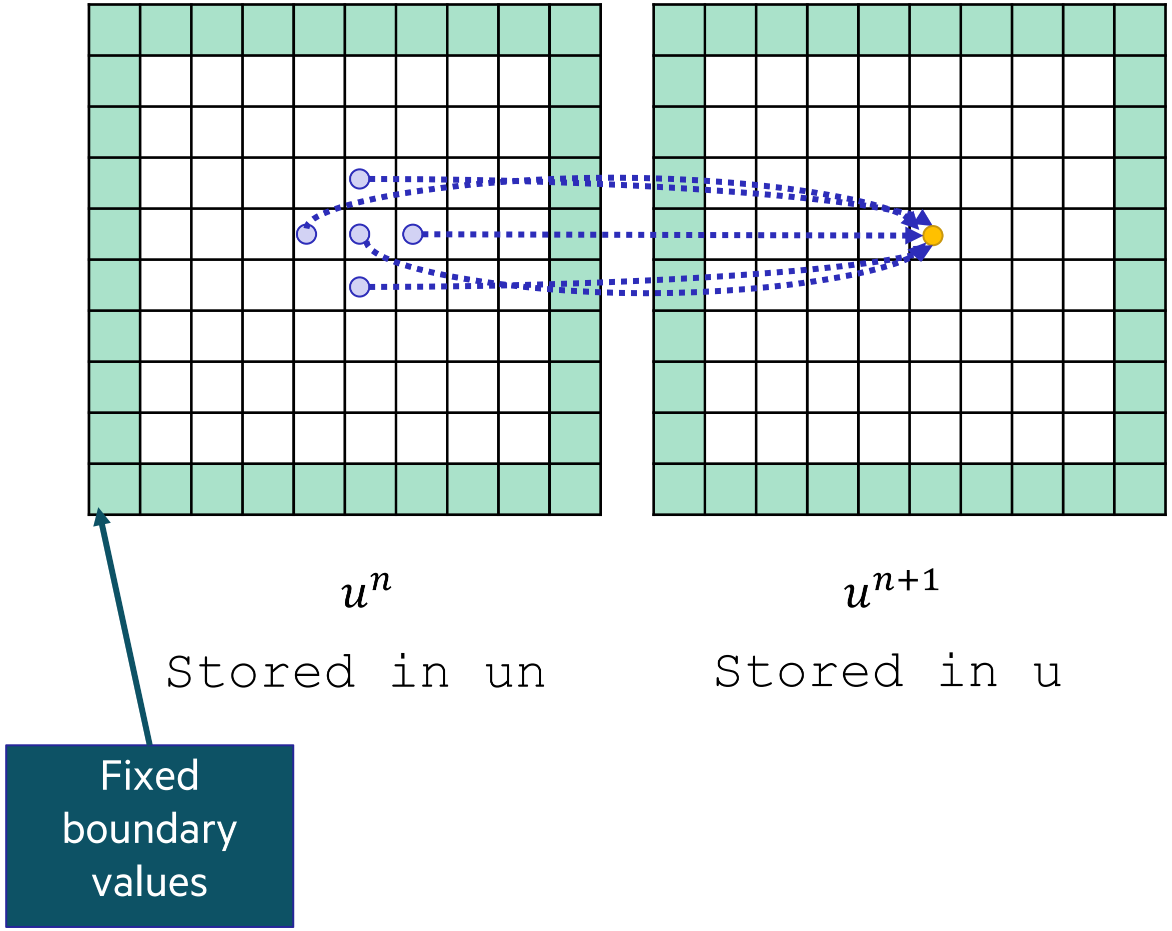
$n = N$

2D heat / diffusion PDE in Chapel:

```
const omega = {0..<nx, 0..<ny},
      omegaHat = omega.expand(-1);
var u: [omega] real = 1.0;
u[nx/4..3*nx/4] = 2.0;
var un = u;
for 1..N {
  un <=> u
  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]);
}
```

PARALLEL 2D HEAT EQUATION

heat-2D.chpl



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

```
...
  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]);
...
```

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

BLOCK DISTRIBUTED & PARALLEL 2D HEAT EQUATION

heat-2D-block.chpl

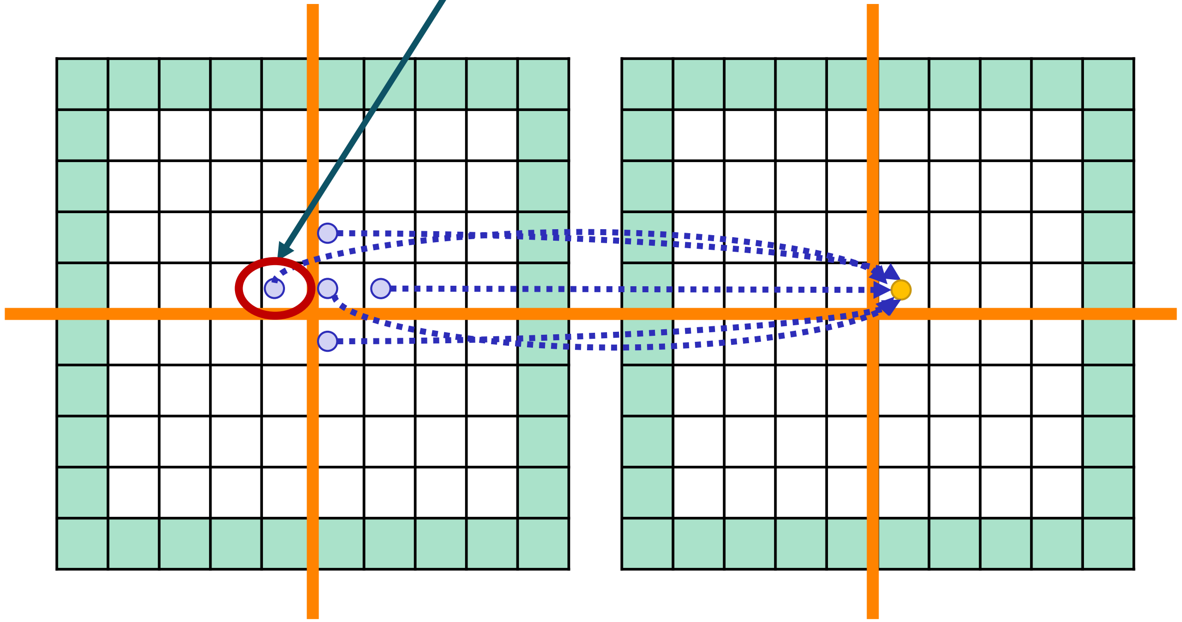
- Declaring distributed domains with the block distribution

```
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



u^n

Stored in un

u^{n+1}

Stored in u

STENCIL DISTRIBUTED & PARALLEL 2D HEAT EQUATION

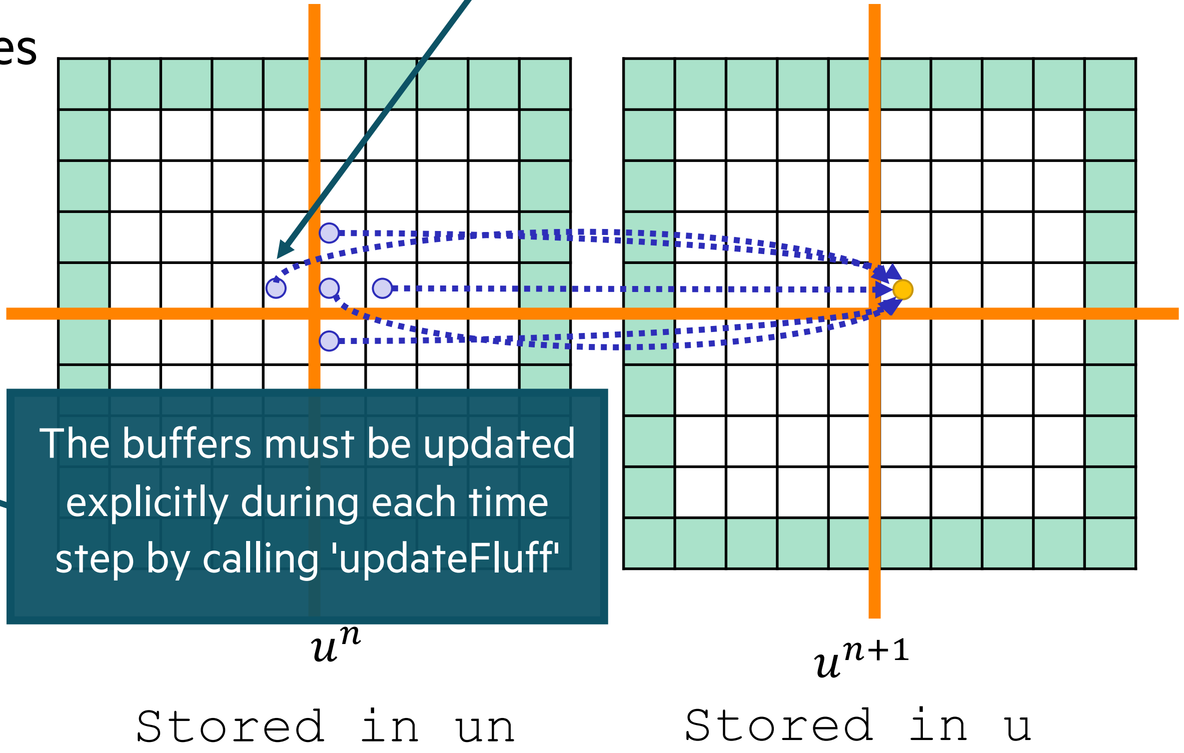
heat-2D-stencil.chpl

- 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]);
}
```

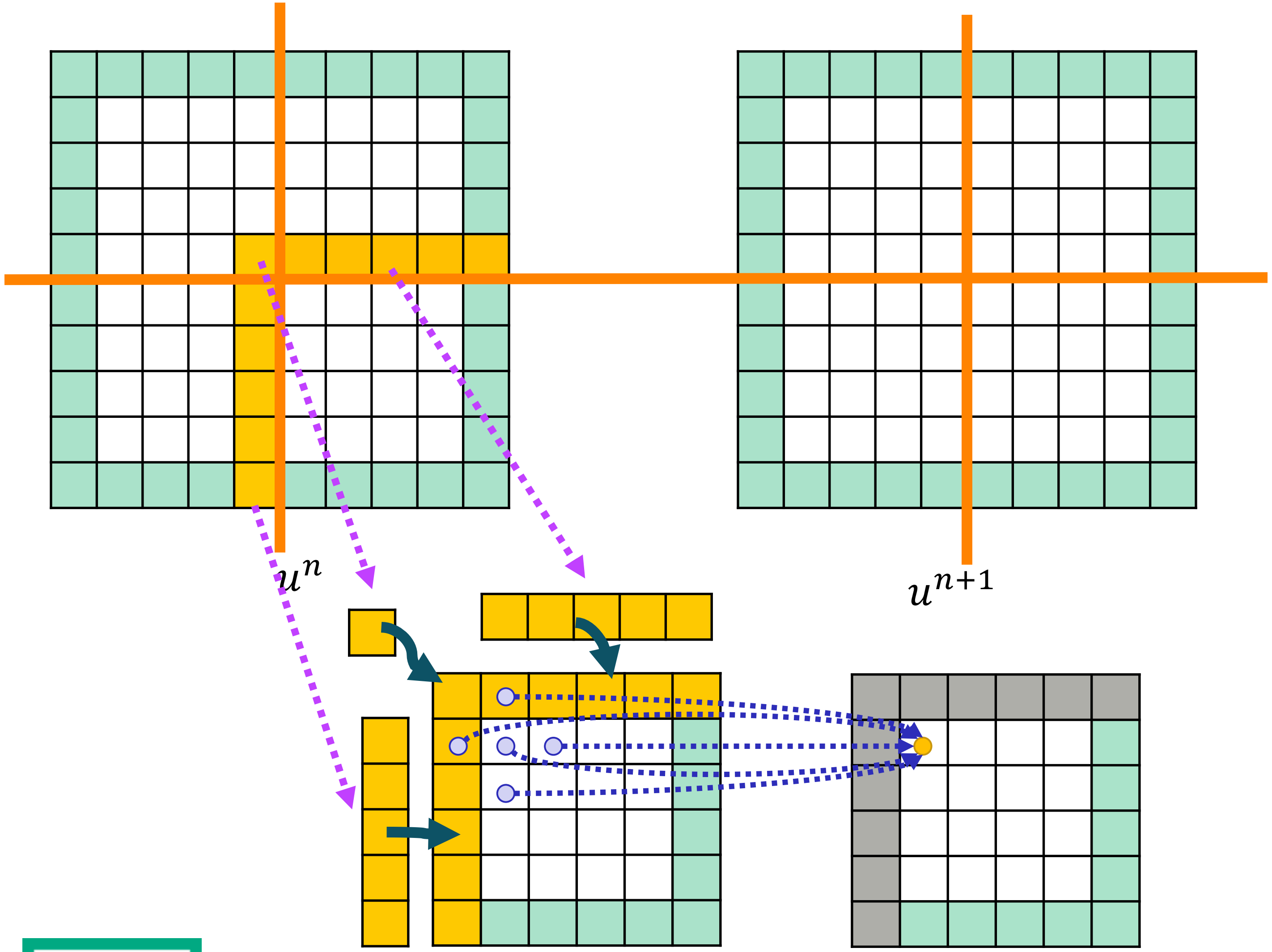


Array access across locale boundaries (within the fluff region) results in a local buffer access — no communication is required

The buffers must be updated explicitly during each time step by calling 'updateFluff'

STENCIL DISTRIBUTED & PARALLEL 2D HEAT EQUATION

heat-2D-stencil.chpl



- 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

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;
 forall a in A do a+=1;
 writeln(A);
}

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

TUTORIAL SUMMARY

OUTLINE: OVERVIEW OF PROGRAMMING IN CHAPEL

- Chapel Goals, Usage, and Comparison with other Tools
- Hello World (Demo with Example Codes)
- Chapel Execution Model and Parallel Hello World
- Serial programming in Chapel: k-mer counting using file IO, config consts, strings, maps
- Parallelizing a program that processes files
- Distributed parallelism for Heat 2D problem
- GPU programming support

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

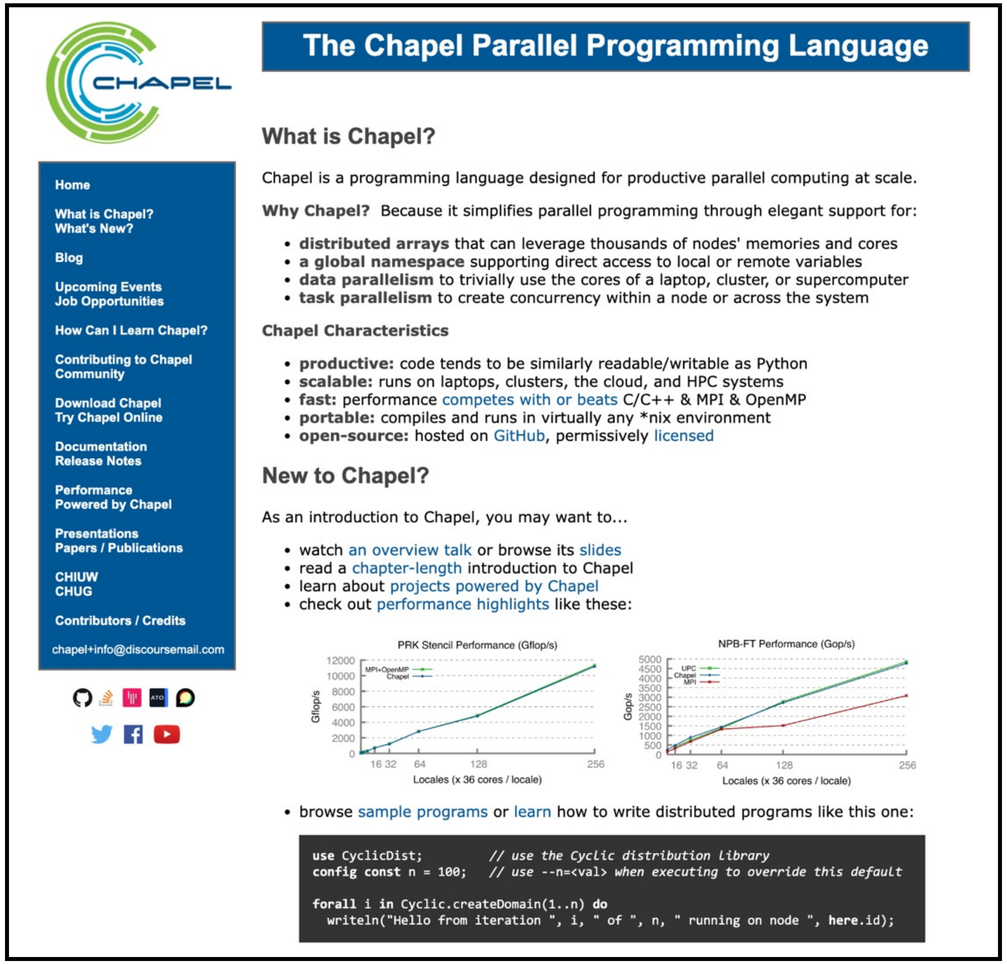
- (points to all other resources)

Social Media:

- Blog: https://chapel-lang.org/blog/
- Twitter: @ChapelLanguage
- Facebook: @ChapelLanguage
- YouTube: @ChapelLanguage

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 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 Cyclic.createDomain(1..n) do
  writeln("Hello from iteration ", i, " of ", n, " running on node ", here.id);
```

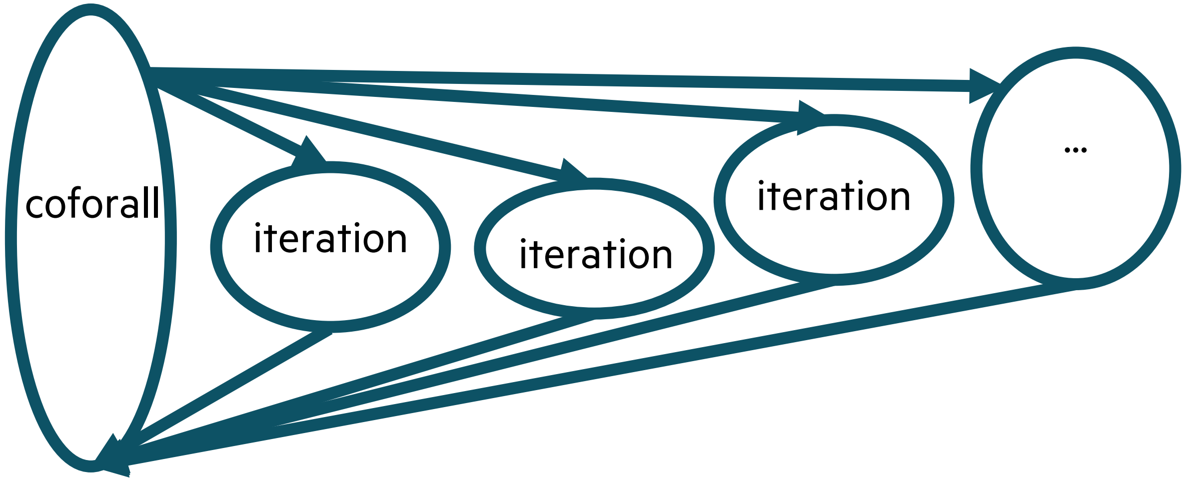
ADDITIONAL CONTENT

ADDITIONAL CONTENT

- Parallelism supported in Chapel
- Parallelism and locality in the context of GPUs

PARALLELISM SUPPORTED BY CHAPEL

PARALLELISM SUPPORTED BY CHAPEL



Synchronous task parallellism

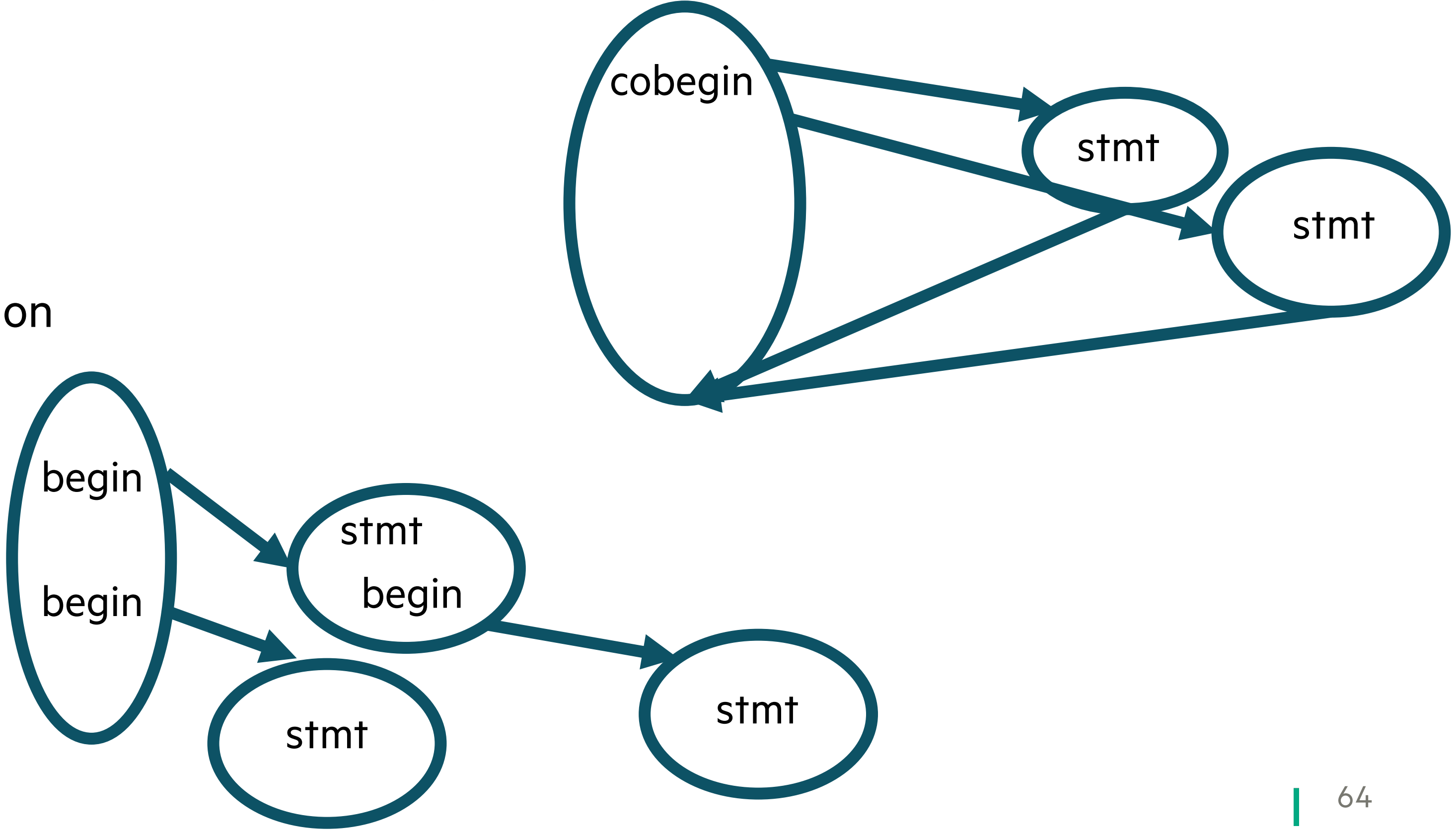
- 'coforall', parallel task per iteration
- 'cobegin', executes all statements in block in parallel

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



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;
```

REDUCE INTENT AND REDUCTIONS IN CHAPEL

- Variables can have 'reduce' intent within tasks:

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

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

- Reductions can reduce arbitrary iterable expressions:

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

- Standard reductions supported by default:

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

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

```
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

- **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()
```

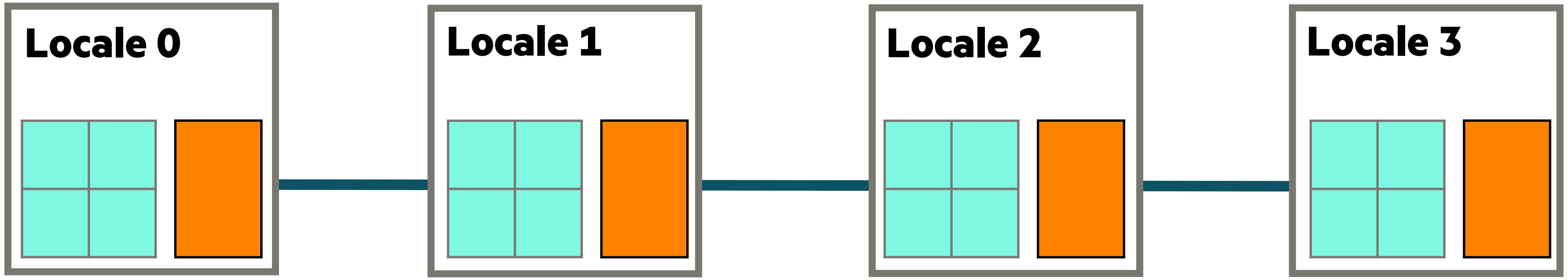
- **task intents / task-private variables:** control how variables and tasks relate

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

PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS

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

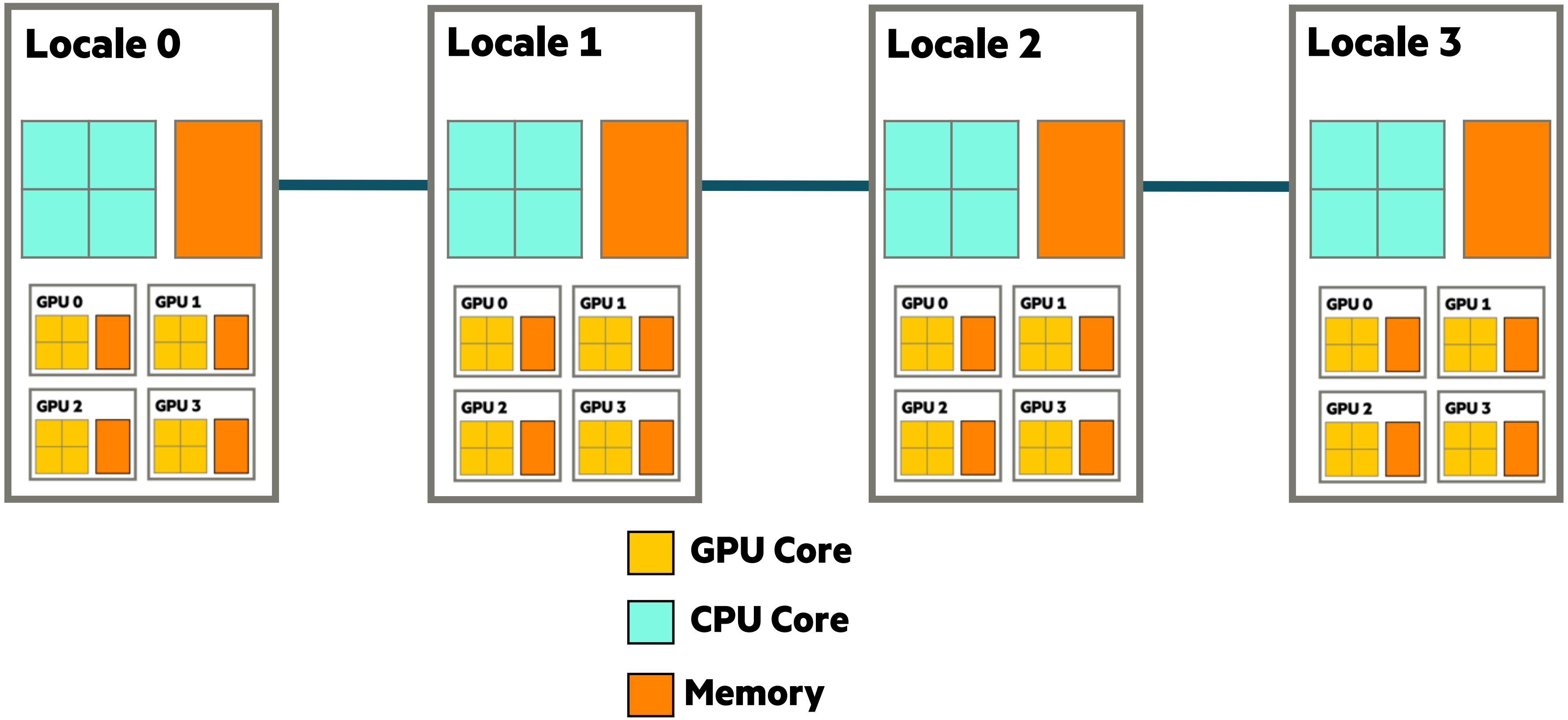


🟦 **CPU Core**

🟧 **Memory**

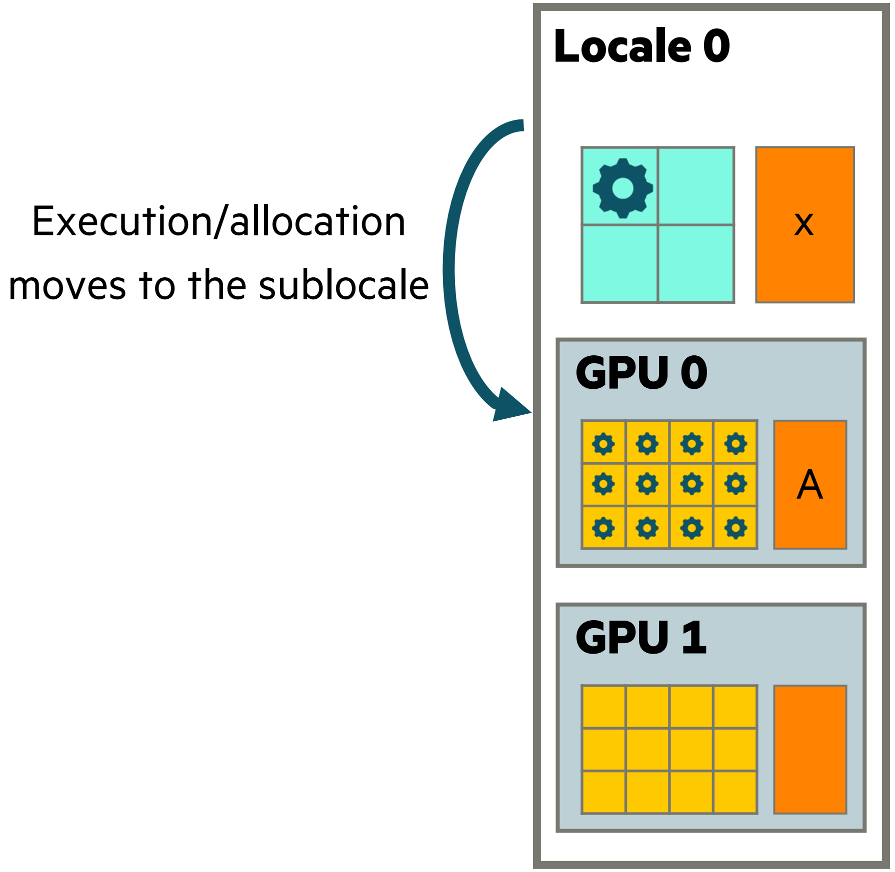
KEY CONCERNS FOR SCALABLE PARALLEL COMPUTING

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

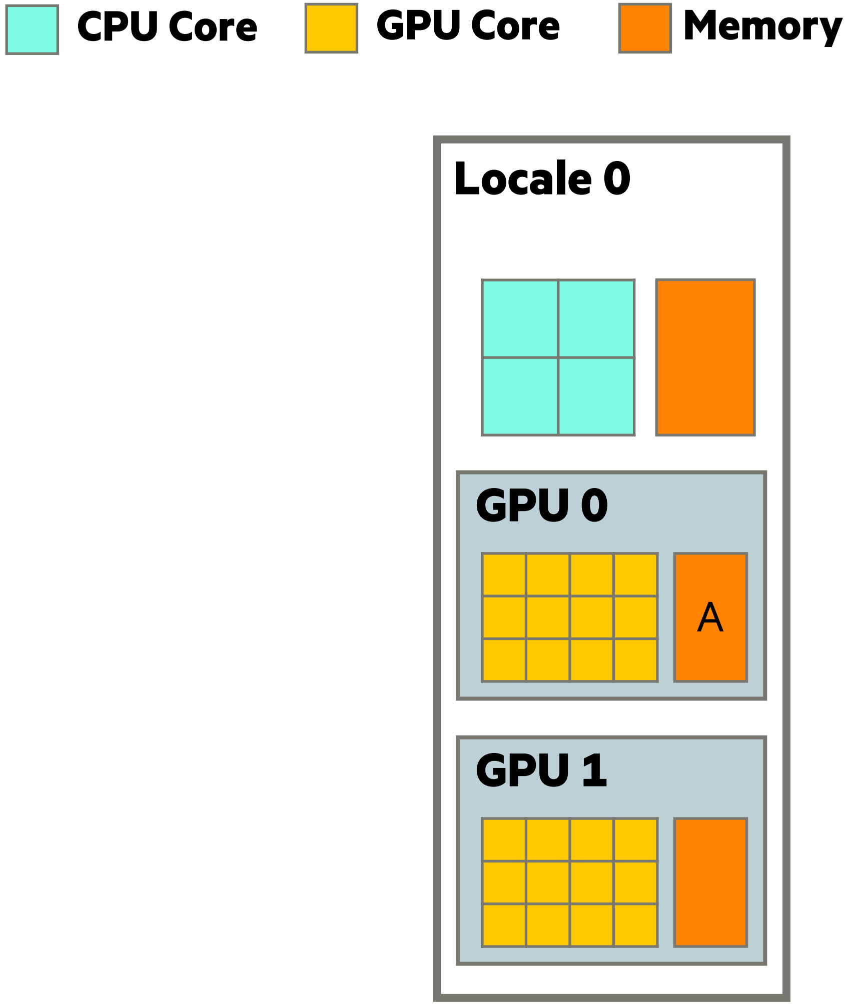
🟦 CPU Core 🟨 GPU Core 🟧 Memory



Execution/allocation moves to the sublocale

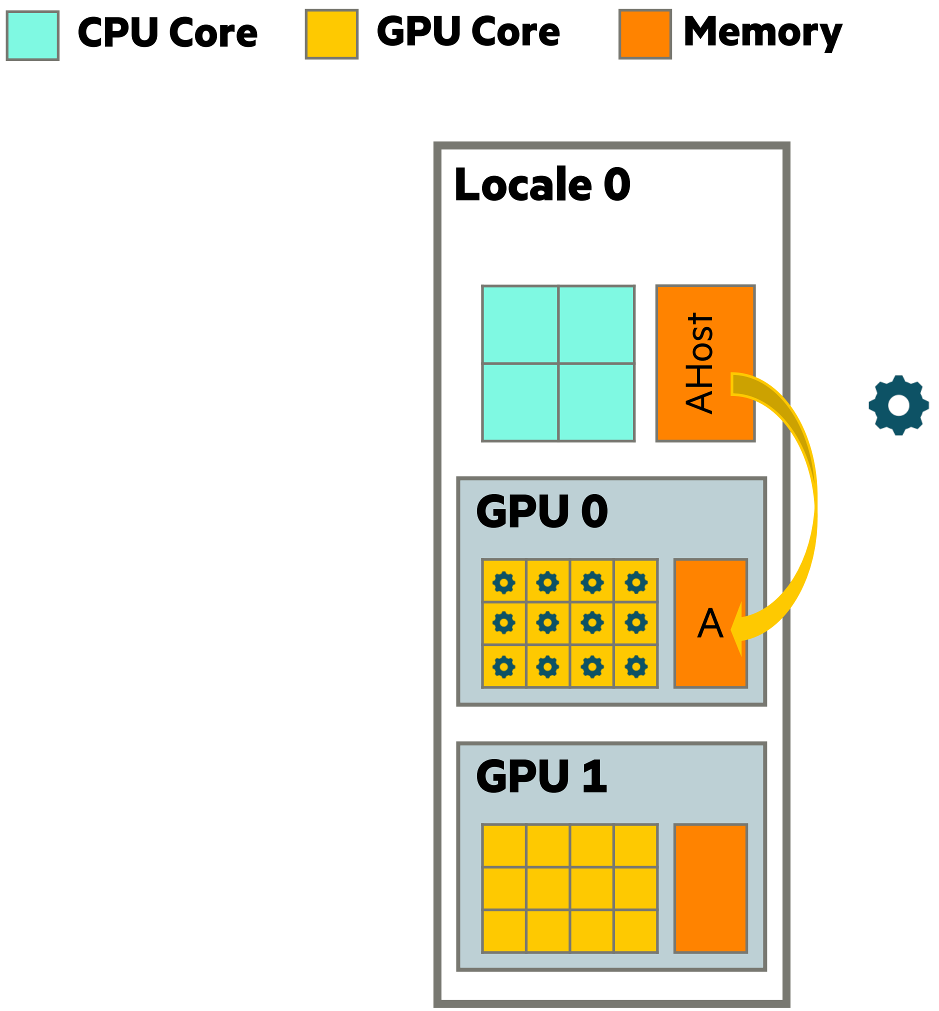
```
var x = 10;


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


writeln(x);
```

PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS



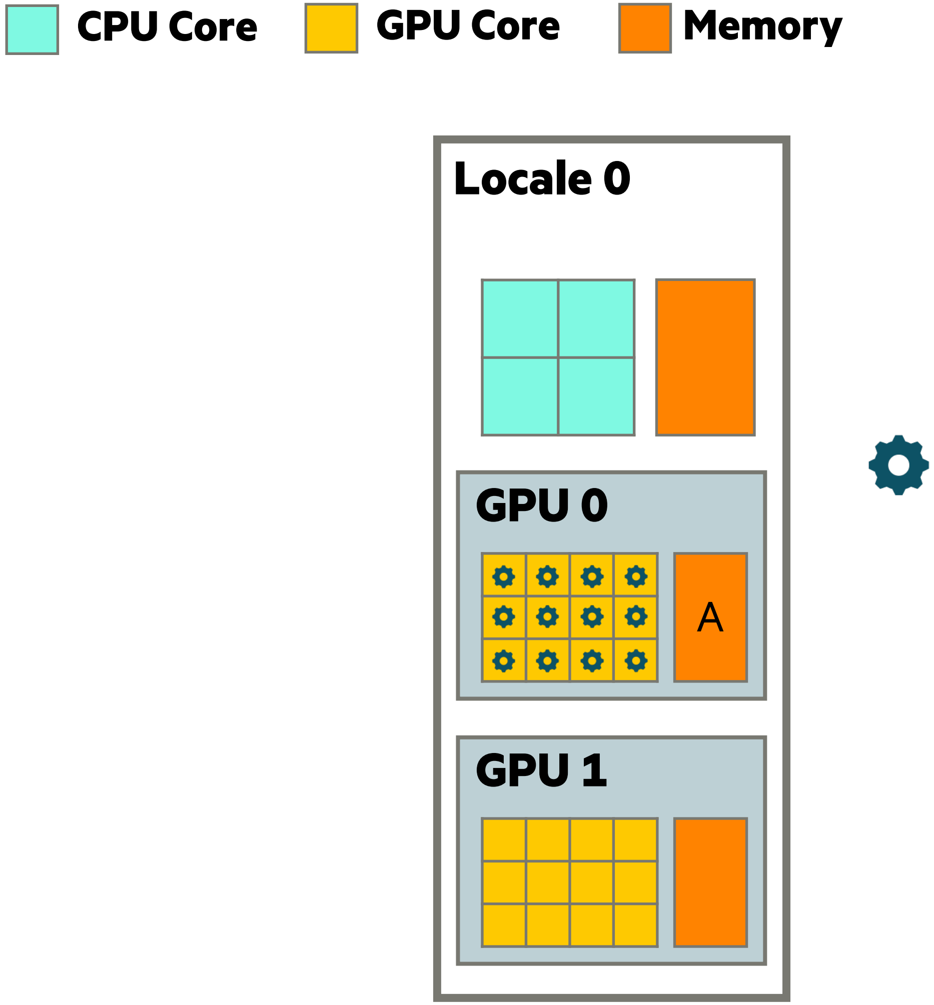
```
var x = 10;


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






writeln(x);
```

PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS



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

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



var writeln(x);
```

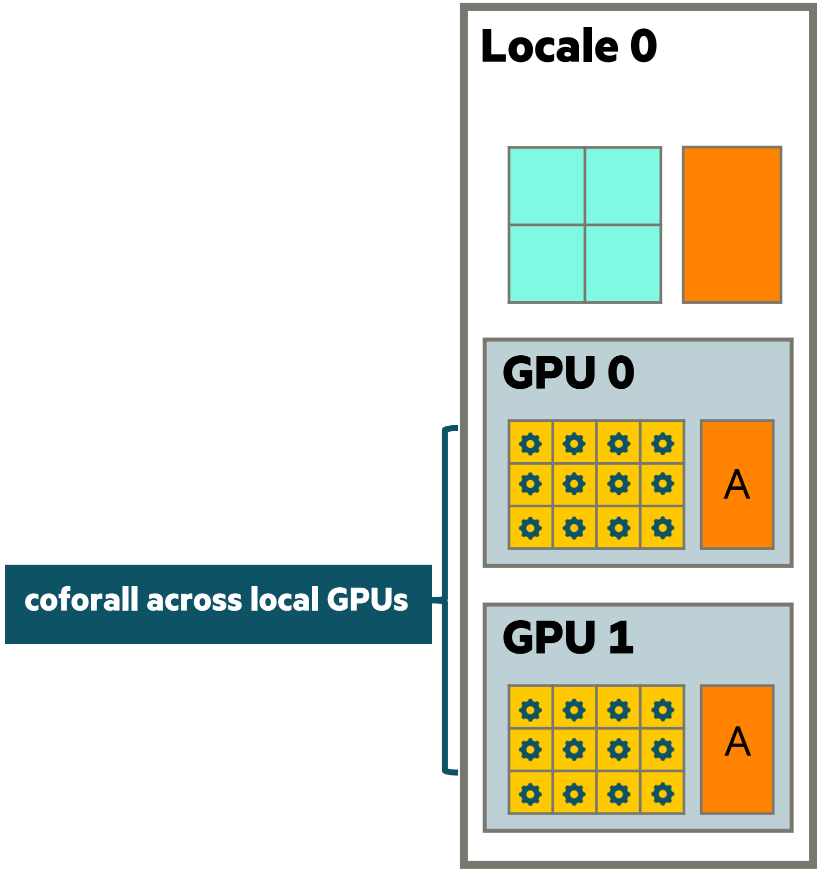
PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS



```
var x = 10;


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






writeln(x);
```

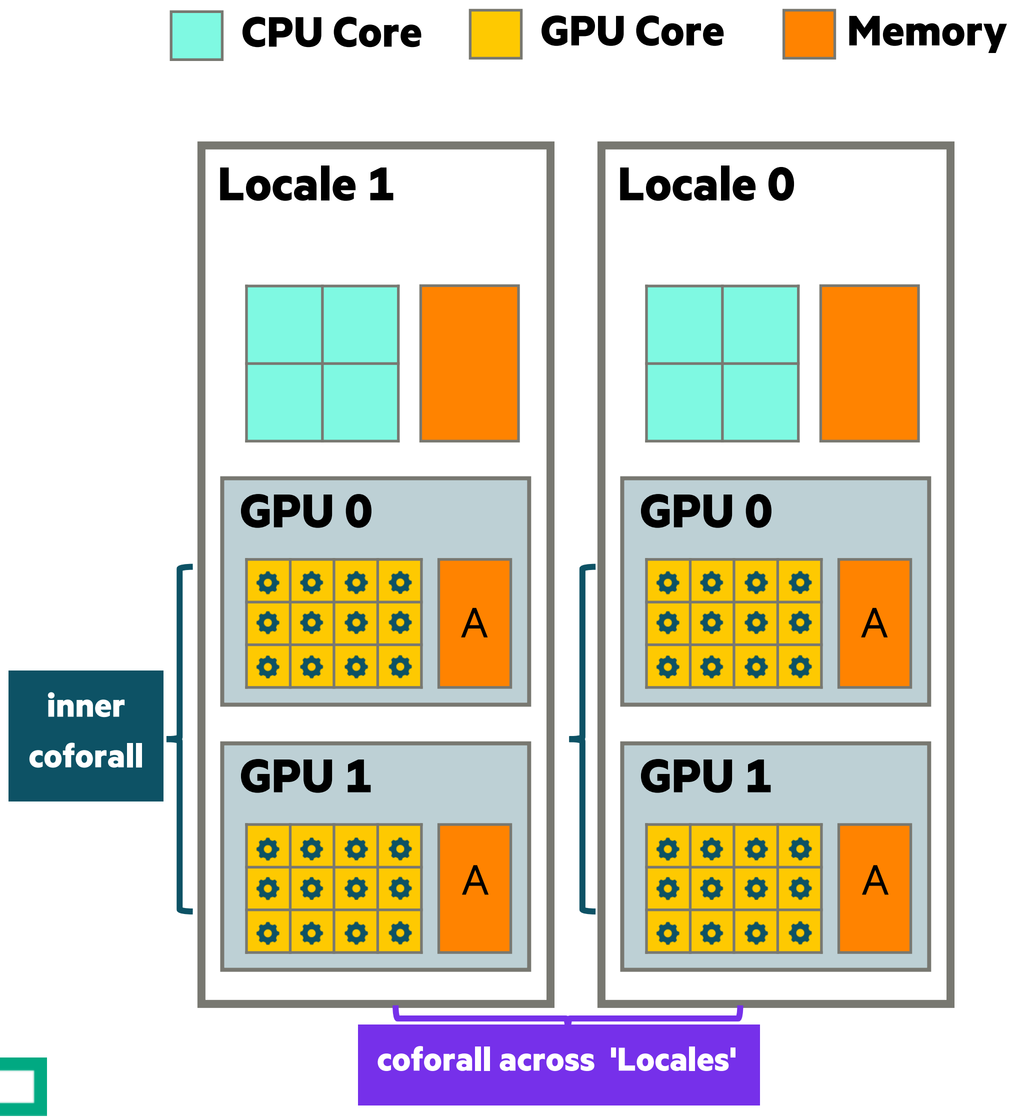
PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS

🟦 **CPU Core** 🟨 **GPU Core** 🟧 **Memory**



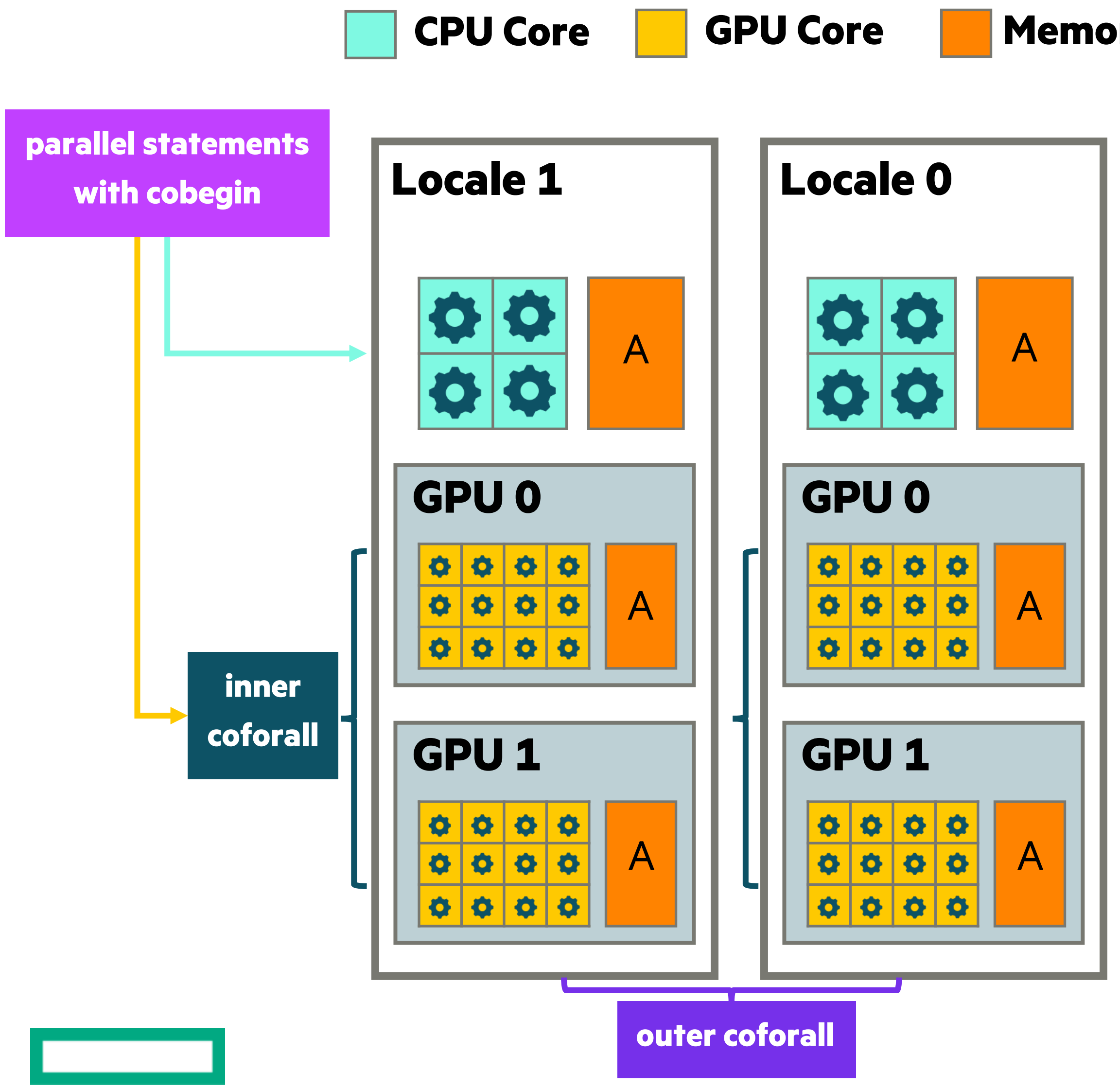
```
var x = 10;

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

writeln(x);
```

PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS



```
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, ...];
   forall a in A do a += 1;
  }






}
writeln(x);
```

PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS



```
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, ...];
   forall a in A do a += 1;
  }
  {
   var A = [1, 2, 3, 4, 5, ...];
   forall a in A do a += 1;
  }
 }
}
writeln(x);
```