Hewlett Packard
Enterprise

OCT 2023 CHAPEL TUTORIAL
UPDATED FOR USE IN SPRING 2025 CS372 CLASS

Chapel Team, edited by Michelle Strout

March 27 and April 1, 2025

PLAN

- **Announcements**
  - MT2 and LA2 grades have been posted
  - SA6 is due Monday, survey for final project preferences
  - Some reading assignment for Chapel have been posted
- **Last time**
  - Midterm 2
- **Today**
  - Chapel introduction
  - Please start the docker pull for Chapel (see next slide)

HANDS ON: HOW TO DO THE HANDS ON

01-hello.chpl

Example codes for Chapel tutorial slides

- https://github.com/UofA-CSc-372-Spring-2025/CSc372Spring2025-CourseMaterials/tree/main/Sandboxes/ChapelTutorialExamples

Using a container on your laptop

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

```
docker pull docker.io/chapel/chapel-gasnet    # takes about 5 minutes
cd CSc372Spring2025-CourseMaterials/Sandboxes/ChapelTutorial/
docker run --rm -it -v "$PWD":/workspace chapel/chapel-gasnet
root@589405d07f6a:/opt/chapel# cd /workspace
root@xxxxxxxxx:/myapp# chpl 01-hello.chpl
root@xxxxxxxxx:/myapp# ./01-hello -nl 1
```

OUTLINE: OVERVIEW OF PROGRAMMING IN CHAPEL

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

CHAPEL GOALS, USAGE, AND COMPARISON WITH OTHER TOOLS

CHAPEL PROGRAMMING LANGUAGE

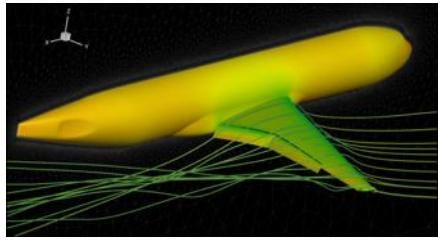
Chapel is a general-purpose programming language that provides

ease of parallel programming,
high performance, and
portability.

And is being used in applications in various ways:

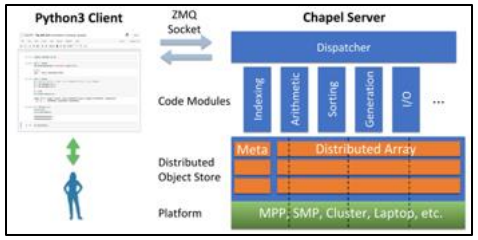
refactoring existing codes,
developing new codes,
serving high performance to Python codes **(Chapel server with Python client),** and
providing distributed and shared memory parallelism for existing codes.

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



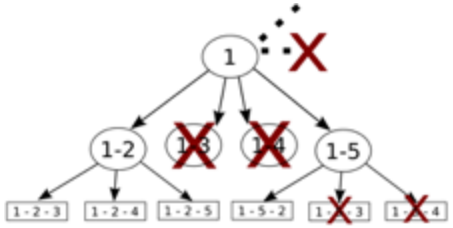
CHAMPS: 3D Unstructured CFD

CHIUW 2021 CHIUW 2022



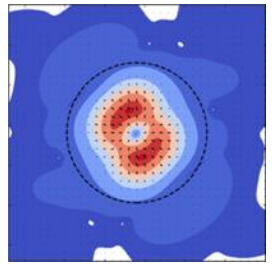
Arkouda: Interactive Data Science at Massive Scale

CHIUW 2020 CHIUW 2023



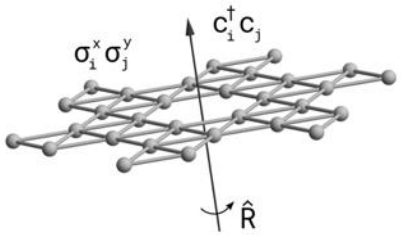
ChOp: Chapel-based Optimization

CHIUW 2021 CHIUW 2023



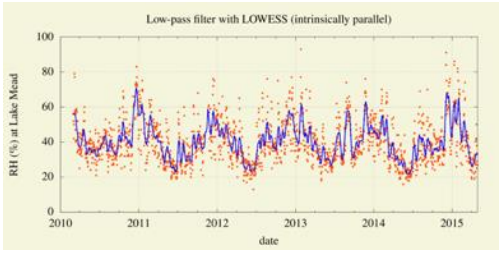
ChplUltra: Simulating Ultralight Dark Matter

CHIUW 2020 CHIUW 2022



Lattice-Symmetries: a Quantum Many-Body Toolbox

CHIUW 2022

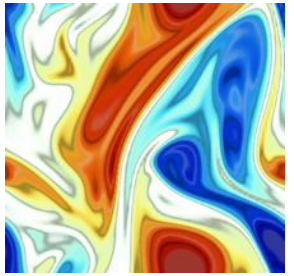


Desk dot chpl: Utilities for Environmental Eng.

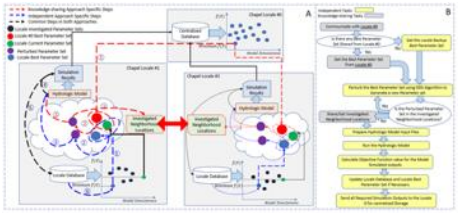
CHIUW 2022

RapidQ: Mapping Coral Biodiversity

CHIUW 2023

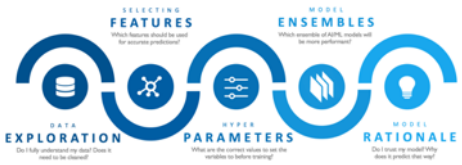


ChapQG: Layered Quasigeostrophic CFD



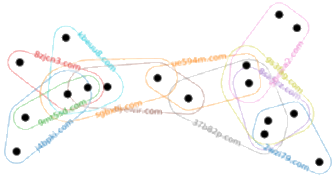
Chapel-based Hydrological Model Calibration

CHIUW 2023



CrayAI HyperParameter Optimization (HPO)

CHIUW 2021



CHGL: Chapel Hypergraph Library

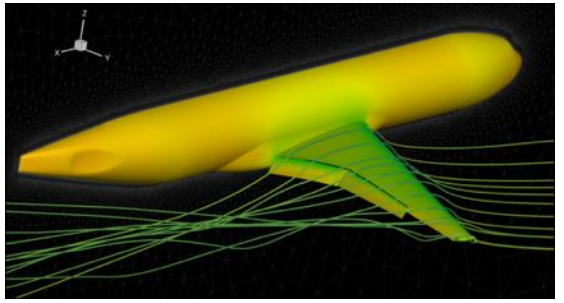
CHIUW 2020

?

Your Application Here?

(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

Other recent users

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

CHAPEL IS HIGHLY PERFORMANT AND SCALABLE

HPE Apollo (May 2021)

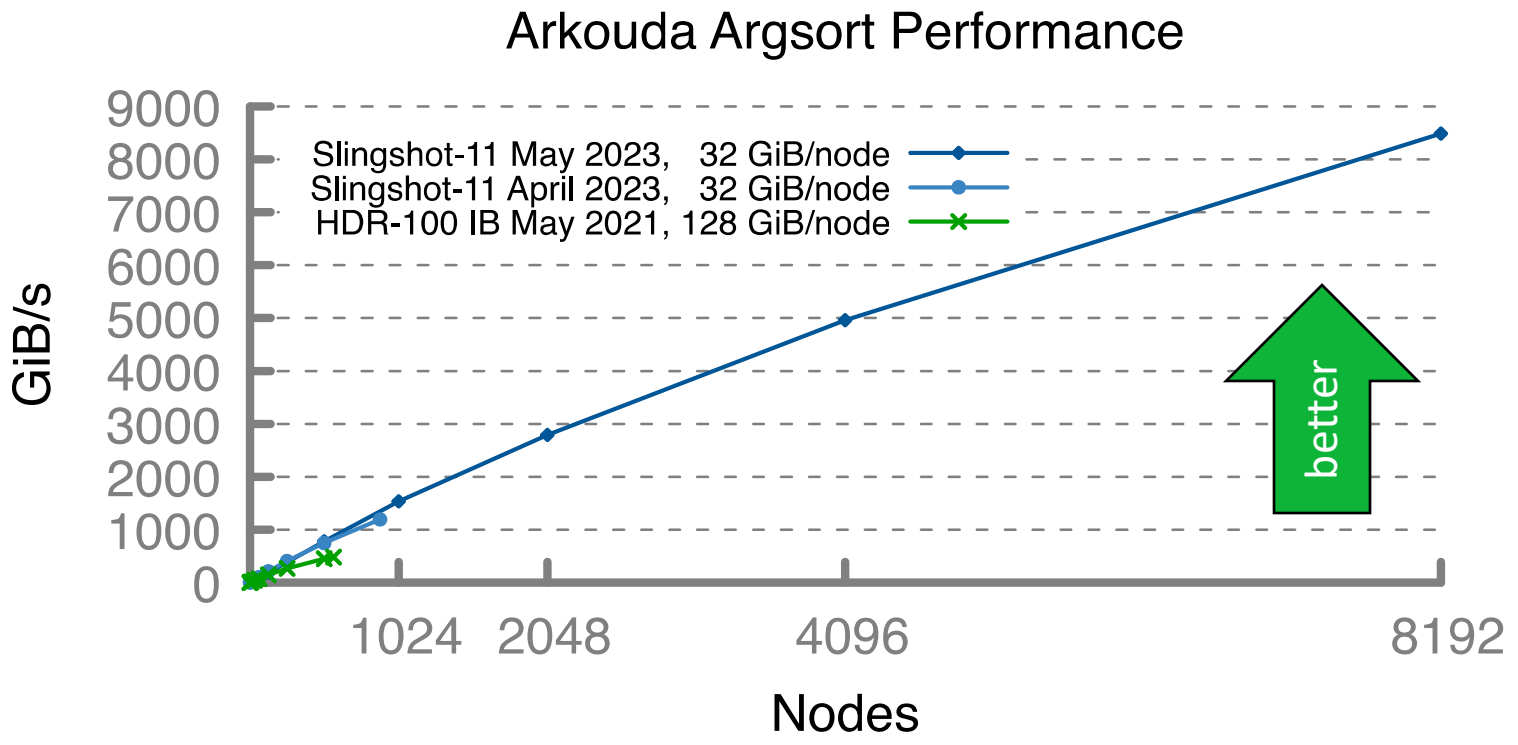
- HDR-100 Infiniband network (100 Gb/s)
- 576 compute nodes
- 72 TiB of 8-byte values
- ~480 GiB/s (~150 seconds)

HPE Cray EX (April 2023)

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

HPE Cray EX (May 2023)

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



A notable performance achievement in ~100 lines of Chapel

COMPARE WITH OTHER PARALLEL PROGRAMMING MODELS

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

Chapel:

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

All can be expressed in the same programming language.

COMPARE WITH OTHER PARALLEL PROGRAMMING MODELS (W/OUT CHAPEL BOX)

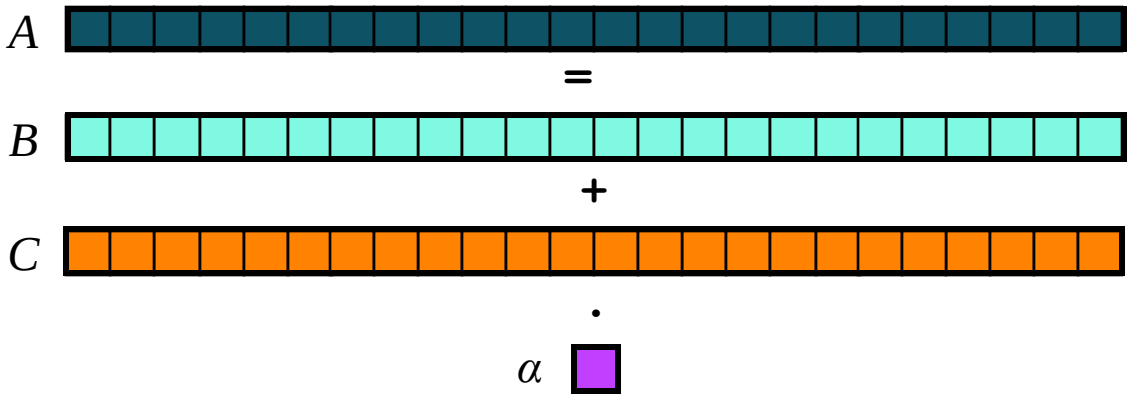
- Shared-memory parallelism
  - Pthreads: low-level library for creating and managing threads of execution that share memory
  - OpenMP: pragmas added before loops and other statements in C/C++/Fortran programs
  - Rust, Julia: programming languages with some threaded parallelism constructs
  - RAJA,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

LET'S COMPARE WITH STREAM TRIAD: A PARALLEL COMPUTATION

Given: m-element vectors A, B, C

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

In pictures:

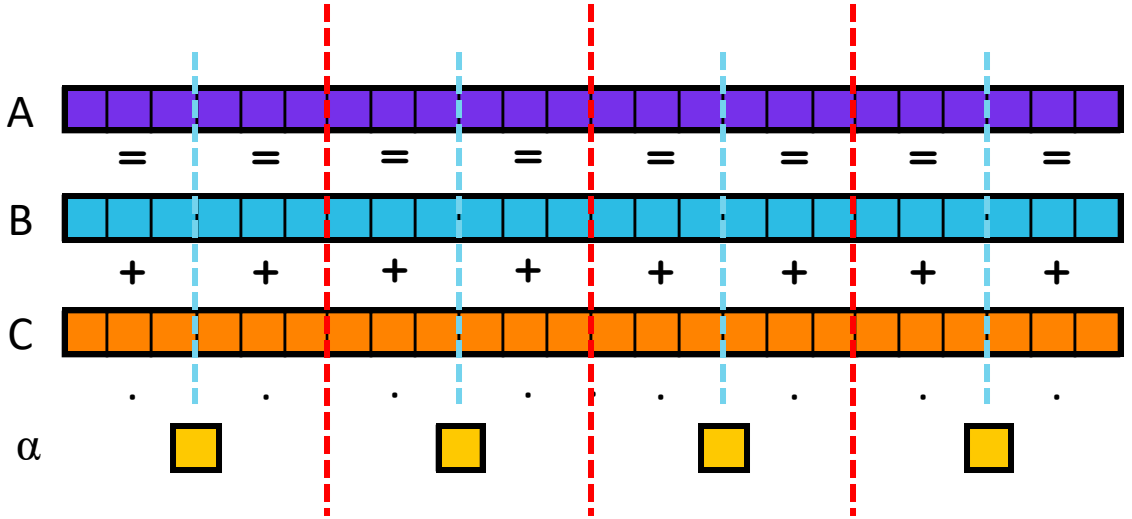


LET'S COMPARE WITH STREAM TRIAD: A PARALLEL COMPUTATION

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

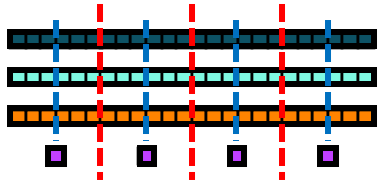
Compute: $\forall i \in 1..n, A_i = B_i + \alpha \cdot C_i$

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



STREAM TRIAD: IN MPI+OPENMP

MPI + OpenMP

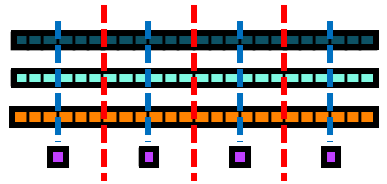


```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

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 to
      allocate memory (%d).\n",
      VectorSize );
    fclose( outFile );
  }
```

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

#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.

WHY SO MANY PROGRAMMING MODELS?

HPC tends to approach programming models bottom-up:

Given a system and its core capabilities…

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

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

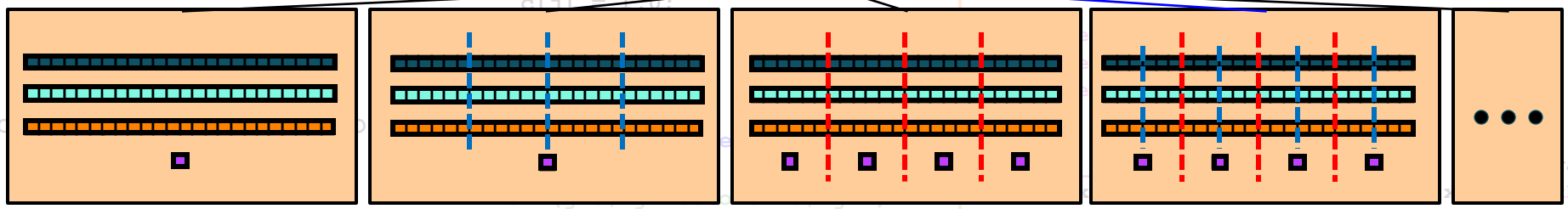
benefits: lots of control; decent generality; easy to implement

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

STREAM TRIAD: IN CHAPEL

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

```
use BlockDist;

config const m = 1000,
             alpha = 3.0;

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

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

B = 2.0;
C = 1.0;

A = B + alpha * C;
```



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

HELLO WORLD (HANDS ON)

HANDS ON: HOW TO DO THE HANDS ON

Example codes for Chapel tutorial slides

- https://github.com/UofA-CSc-372-Spring-2025/CSc372Spring2025-CourseMaterials/tree/main/Sandboxes/ChapelTutorialExamples

Using a container on your laptop

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

```
docker pull docker.io/chapel/chapel-gasnet    # takes about 5 minutes
cd CSc372Spring2025-CourseMaterials/Sandboxes/ChapelTutorial/
docker run --rm -it -v "$PWD":/workspace chapel/chapel-gasnet
root@589405d07f6a:/opt/chapel# cd /workspace
root@xxxxxxxxx:/myapp# chpl 01-hello.chpl
root@xxxxxxxxx:/myapp# ./01-hello -nl 1
```

"HELLO WORLD" IN CHAPEL: TWO VERSIONS

- Fast prototyping

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

01-hello.chpl

- “Production-grade”

```
module Hello {

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

}
```

01-hello-production.chpl

"HELLO WORLD" IN CHAPEL: TWO VERSIONS

- Fast prototyping (configurable)

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

01-hello-configurable.chpl

- "Production-grade" (configurable)

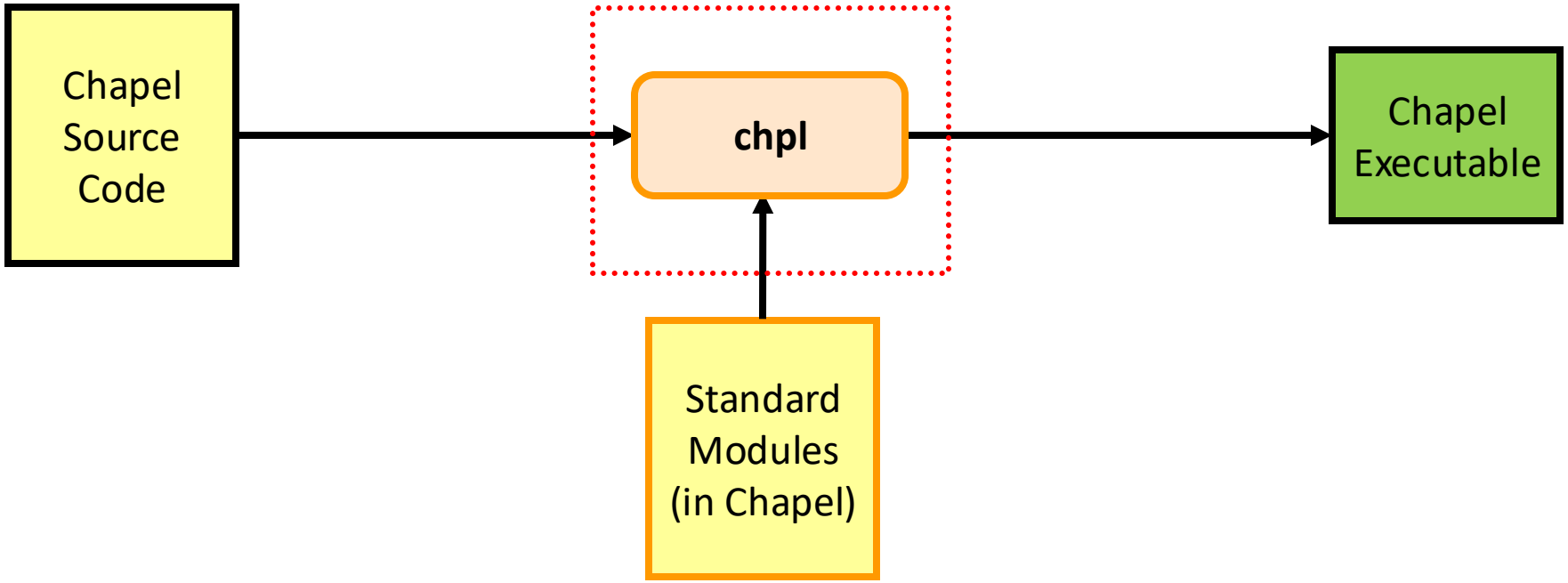
```
module Hello {
  config const audience = "world";

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

01-hello-production-configurable.chpl

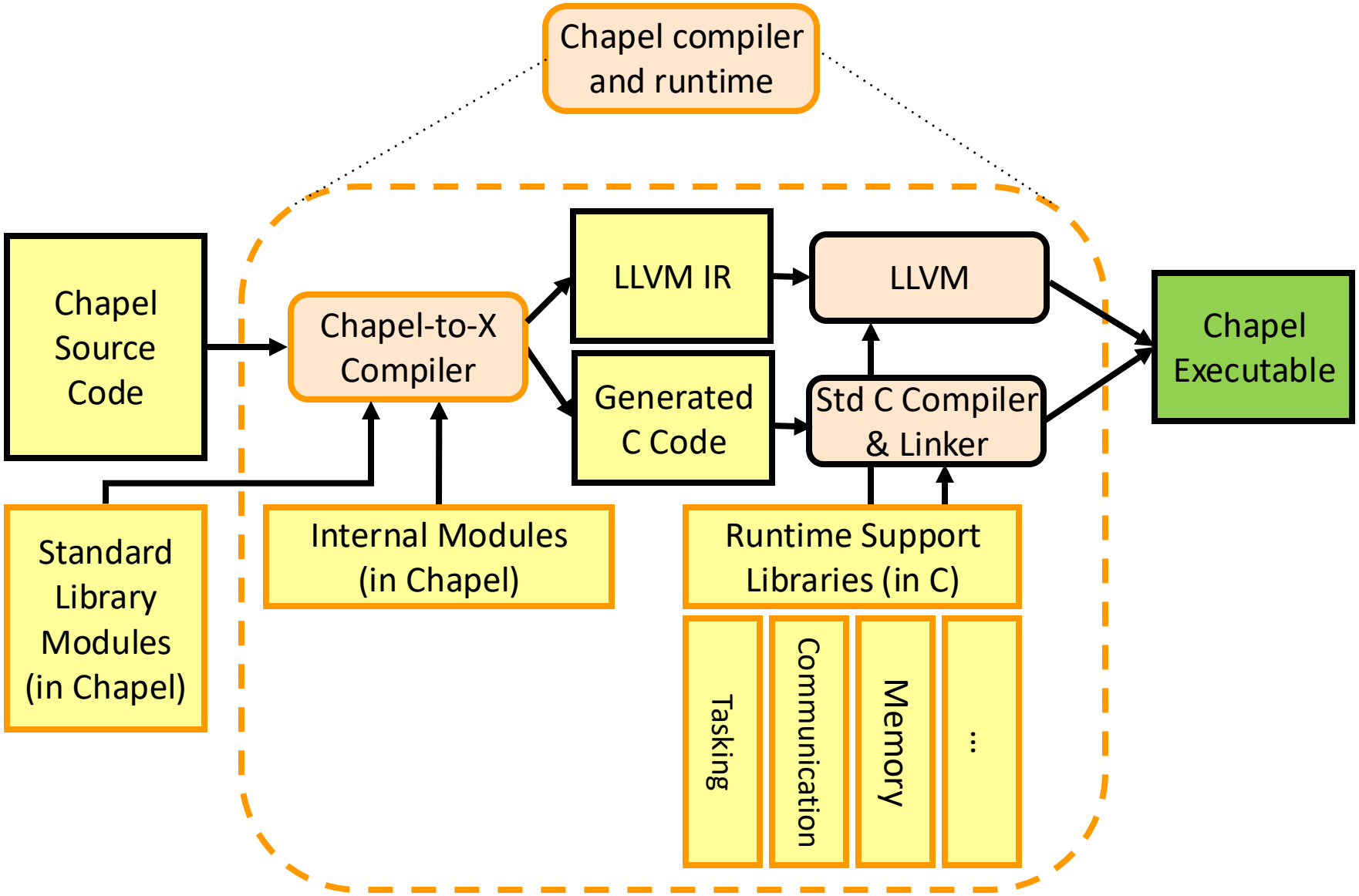
- To change 'audience' for a given run:

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

COMPILING CHAPEL



CHAPEL COMPILER ARCHITECTURE



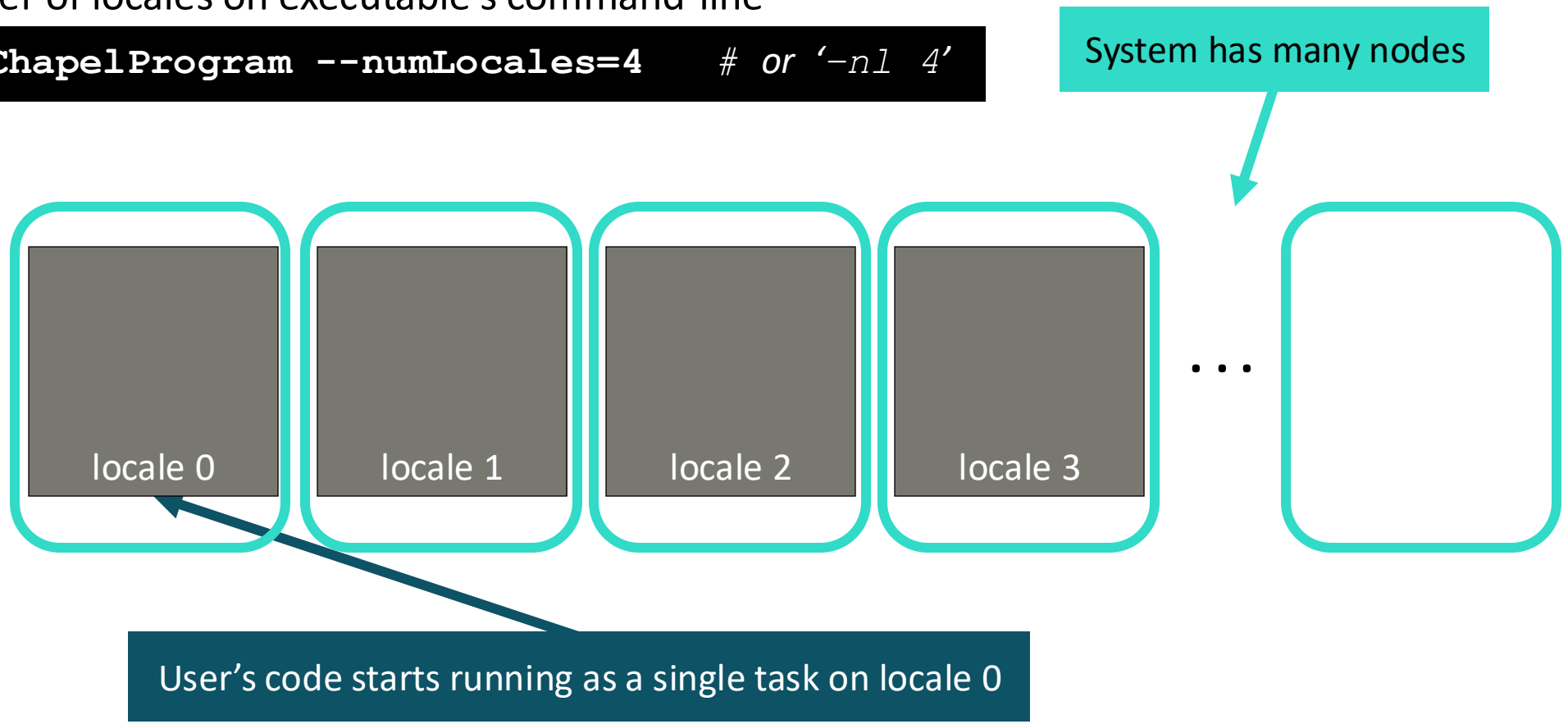
CHAPEL EXECUTION MODEL AND PARALLEL HELLO WORLD (HANDS ON)

CHAPEL EXECUTION MODEL AND TERMINOLOGY: LOCALES

Locales can run tasks and store variables

- Each locale executes on a "compute node" on a parallel system
- User specifies number of locales on executable's command-line

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



TASK-PARALLEL “HELLO WORLD”

01-hello-dist-node-names.chpl

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

TASK-PARALLEL “HELLO WORLD”

01-hello-dist-node-names.chpl

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

‘here’ refers to the locale on which we’re currently running

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

what’s my locale’s name?

TASK-PARALLEL “HELLO WORLD”

01-hello-dist-node-names.chpl

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

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

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

TASK-PARALLEL “HELLO WORLD”

01-hello-dist-node-names.chpl

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

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

So far, this is a shared-memory program

Nothing refers to remote locales, explicitly or implicitly

TASK-PARALLEL “HELLO WORLD” (DISTRIBUTED VERSION)

01-hello-dist-node-names.chpl

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

the array of locales we’re running on

Locales array:

TASK-PARALLEL “HELLO WORLD” (DISTRIBUTED VERSION)

01-hello-dist-node-names.chpl

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

create a task per locale on which the program is running

have each task run ‘on’ its locale

then print a message per core, as before

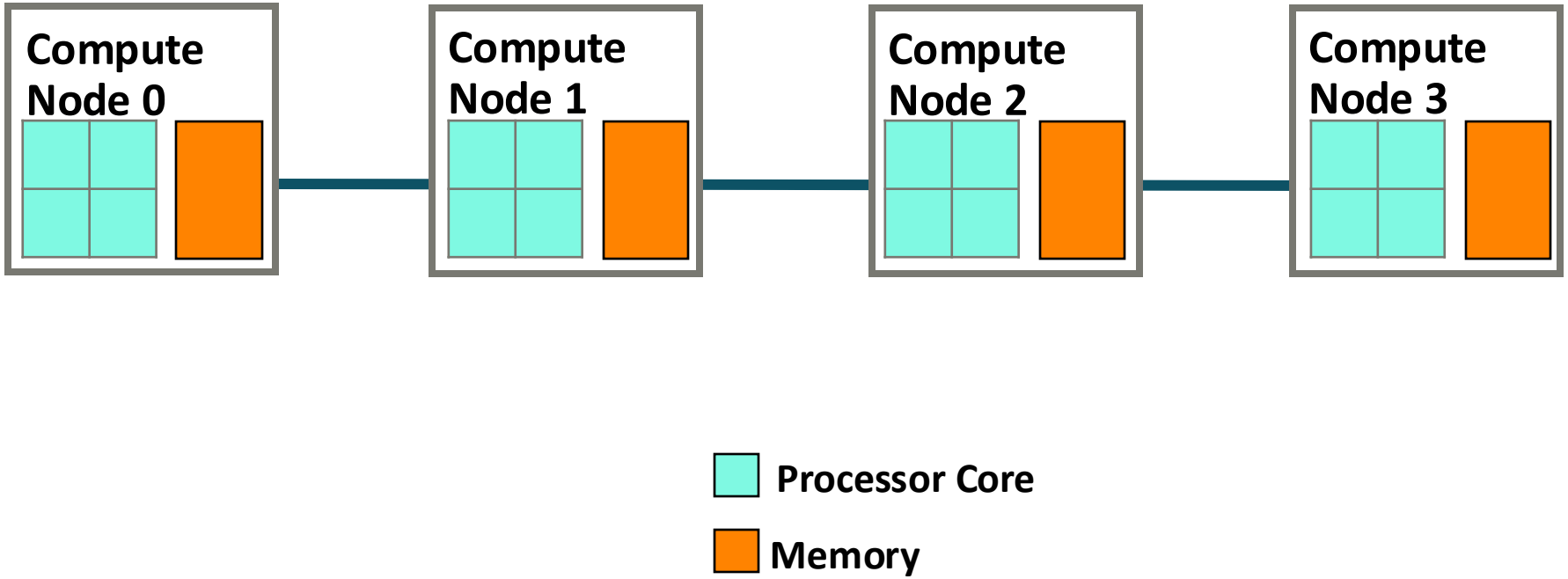
```
> chpl 01-hello-dist-node-names.chpl
> ./01-hello-dist-node-names -nl=4
Hello from task 1 of 4 on n1032
Hello from task 4 of 4 on n1032
Hello from task 1 of 4 on n1034
Hello from task 2 of 4 on n1032
Hello from task 1 of 4 on n1033
Hello from task 3 of 4 on n1034
Hello from task 1 of 4 on n1035
…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

LOCALES AND EXECUTION MODEL IN CHAPEL

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

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

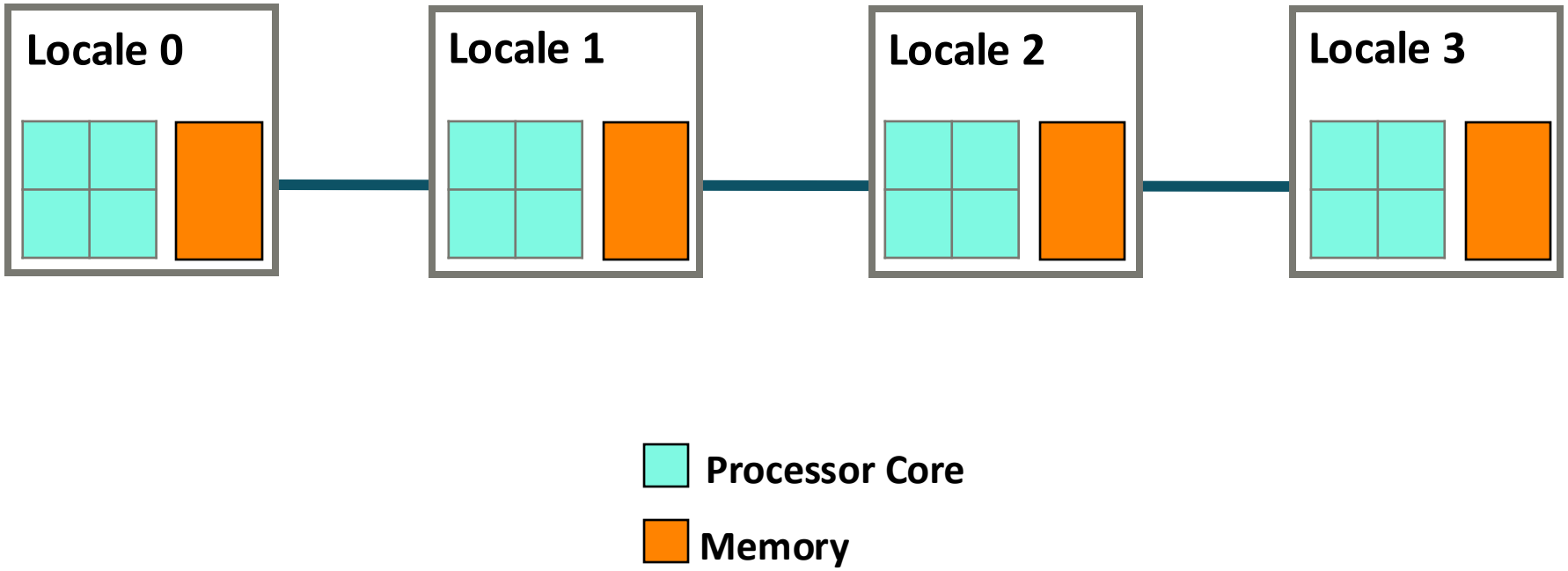
For now, think of each compute node as having one locale run on it



LOCALES AND EXECUTION MODEL IN CHAPEL

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

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



GETTING STARTED WITH LOCALES

- Users specify # of locales when running Chapel programs

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

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

- Chapel provides built-in locale variables

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

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

LOCALE OPERATIONS

- Locale methods support queries about the target system:

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

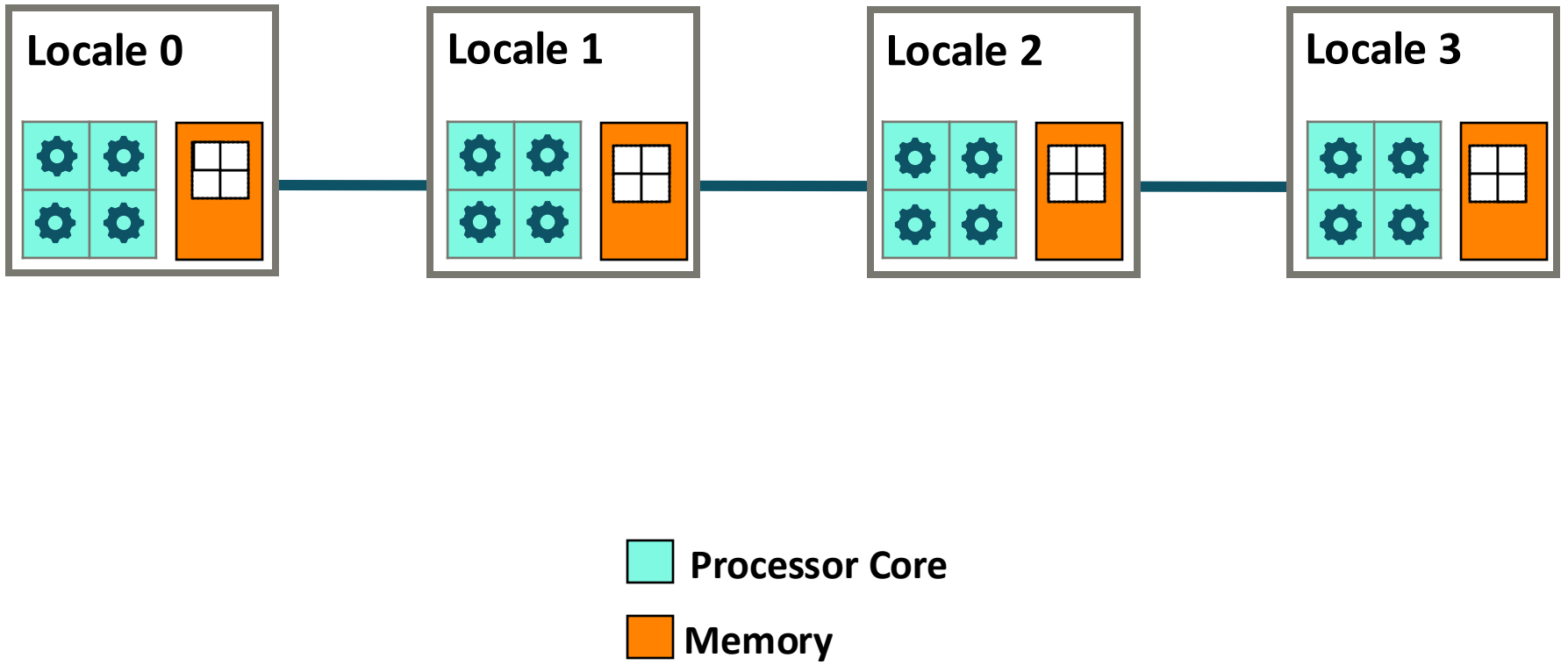
- *On-clauses* support placement of computations:

```
writeln("on locale 0");

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

writeln("on locale 0 again");
```

```
on A[i,j] do
  bigComputation(A);

on node.left do
  search(node.left);
```

KEY CONCERNS FOR SCALABLE PARALLEL COMPUTING

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



BASIC FEATURES FOR LOCALITY

01-basics-on.chpl

01-basics-on.chpl

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

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

on Locales[1] {
  var B: [1..2, 1..2] real;

  B = 2 * A;
}
```

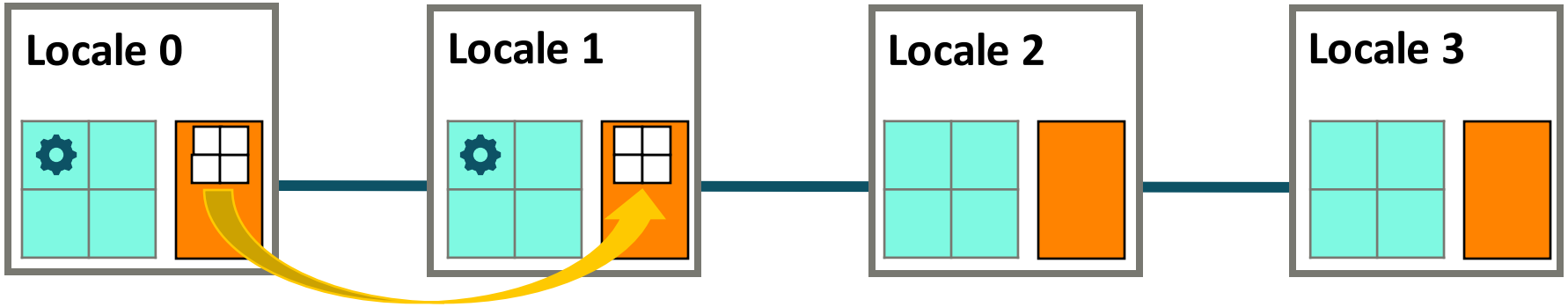
All Chapel programs begin running as a single task on locale 0

Variables are stored using the memory local to the current task

on-clauses move tasks to other locales

remote variables can be accessed directly

This is a serial, but distributed computation



BASIC FEATURES FOR LOCALITY

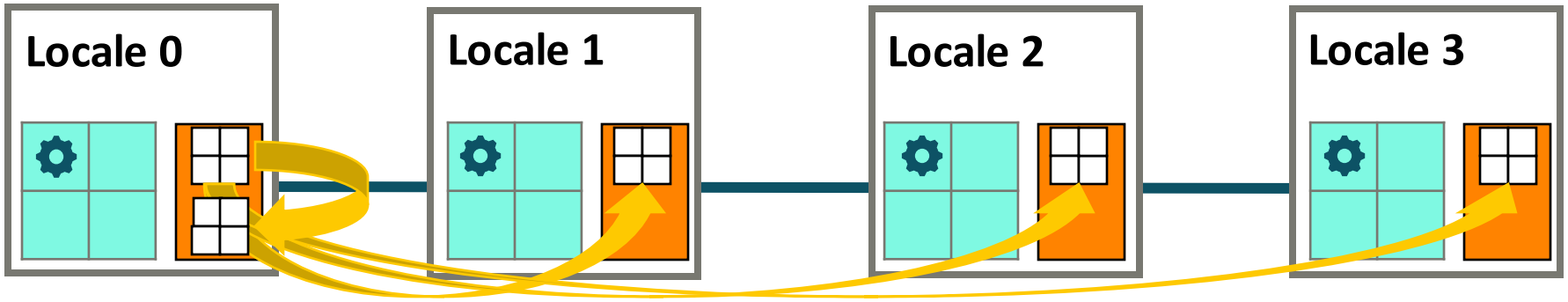
01-basics-for.chpl

01-basics-for.chpl

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

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

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

This loop will serially iterate over the program's locales

This is also a serial, but distributed computation



MIXING LOCALITY WITH TASK PARALLELISM

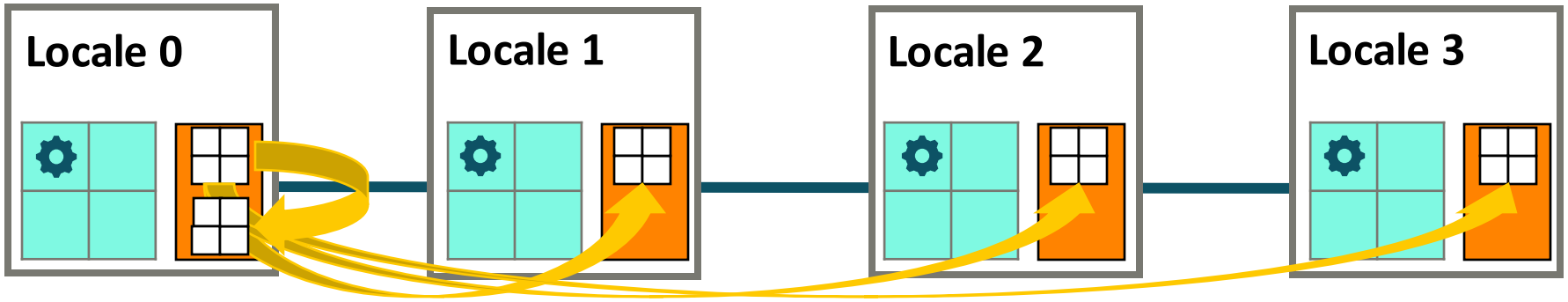
01-basics-coforall.chpl

01-basics-coforall.chpl

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

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

coforall loc in 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

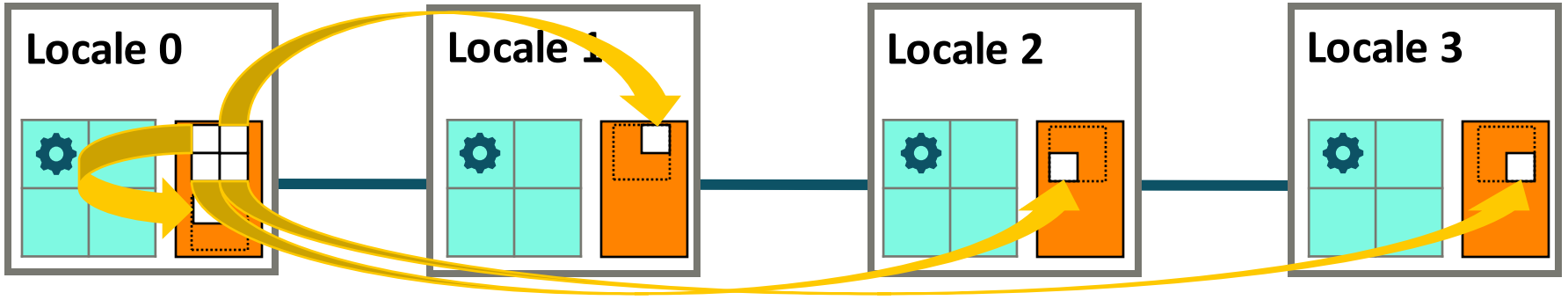
01-basics-distarr.chpl

01-basics-distarr.chpl

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

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

use BlockDist;

var D = blockDist.createDomain({1..2, 1..2});
var B: [D] real;
B = A;
```

Chapel also supports distributed domains (index sets) and arrays

They also result in parallel distributed computation



HANDS ON: PARALLELISM ACROSS AND WITHIN LOCALES

01-hellopar.chpl

Parallel hello world

- 01-hellopar.chpl

Things to try

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

Key concepts

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

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

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

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

PARALLELISM AND LOCALITY ARE ORTHOGONAL IN CHAPEL

- This is a parallel, but local program:

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

- This is a distributed, but serial program:

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

- This is a distributed parallel program:

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

HANDS ON: PARALLELISM AND LOCALITY IN CHAPEL

Goals

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

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

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

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

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