

HPC Programming Language Chapel: Parallel Approaches

Jemmy Hu

SHARCNET/Compute Canada

Oct. 07, 2020

Previous Base Language Overview:

<https://www.youtube.com/watch?v=FG0L4Xmkml4>

Example codes are available on [Graham](#) under
</home/jemmyhu/chapel/>

What is Chapel?

Chapel: A productive parallel programming language

- developed by *Cray Inc.*
- portable & scalable
- open-source & collaborative



Designed around a high-level abstraction and multi-resolution philosophy. Users can incrementally add more detail to their original code prototype.

Goals:

- Support general parallel programming
 - “any parallel algorithm on any parallel hardware”
- Make parallel programming at scale far more productive

Partitioned Global Address Space (PGAS) Languages

A programming model suited for shared and distributed memory parallel machines

- support a shared namespace on distributed memory
 - permit parallel tasks to access remote variables by naming them
- establish a strong sense of ownership
 - every variable has a well-defined location
 - local variables are cheaper to access than remote ones

Traditional PGAS languages have been SPMD in nature

- best-known examples: Fortran Co-Arrays, UPC

Chapel is PGAS, but unlike most, it's not inherently SPMD

- not “the other copies of the program”
- “global name/address space” comes from lexical scoping
 - as in traditional languages, each declaration yields one variable
 - variables are stored on the locale where the task declaring it is executing
- Communication though implicit, users can reason about communication
 - semantic model is explicit about where data is placed / tasks execute
 - execution-time queries support reasoning about locality ● e.g., here, x.locale

Chapel compiler

Chapel is a compiled language, Chapel source code must be compiled to generate a binary or executable to be run on the computer.

Chapel source code must be written in text files with the extension `.chpl`.

Chapel compiler command is `chpl`.

```
chpl -o hello hello.chpl
```

```
chpl --fast -o hello hello.chpl
```

`--fast` indicates the compiler to optimise the binary to run as fast as possible in the given architecture.

```
//Chapel hellow.chpl
```

```
writeln('Hello World from Chapel!');
```

```
writeln('If we can see this, everything works!');
```

```
[jemmyhu@gra-login1 chapel]$ chpl -o hello hello.chpl
```

```
[jemmyhu@gra-login1 chapel]$ ./hello
```

```
Hello World from Chapel!
```

```
If we can see this, everything works!
```

Chapel on CC clusters

On Compute Canada clusters [Cedar](#) and [Graham](#) we have two versions of Chapel:

one is a single-locale (single-node) Chapel,

[chapel-single/1.15.0](#)

the other is a multi-locale (multi-node) Chapel,

[chapel-slurm-gasnetrun_ibv/1.15.0](#)

```
module spider chapel
```

Interactive job for testing, e.g., serial

```
salloc --time=01:00:00 --ntasks=1 --mem-per-cpu=3G  
--account=def-username  
./hello
```

4 cpus on one node/locale

```
salloc --time=01:00:00 --cpus-per-task=4 --mem-per-  
cpu=3G --account=def-username
```

2 nodes/locales, 2 cpus per locale

```
salloc --time=01:00:00 --nodes=2 --cpus-per-task=2  
--mem-per-cpu=3G --account=def-username
```

```
[jemmyhu@gra-login1 chapel]$ module load  
nixpkgs/16.09 gcc/5.4.0
```

```
[jemmyhu@gra-login1 chapel]$ module load chapel-  
single/1.15.0
```

```
[jemmyhu@gra-login1 chapel]$ which chpl  
/cvmfs/soft.computecanada.ca/easybuild/software/2017/av  
x2/Compiler/gcc5.4/chapel-single/1.15.0/bin/linux64/chpl
```

```
[jemmyhu@gra-login1 chapel]$ chpl -o hello hello.chpl
```

```
[jemmyhu@gra-login1 chapel]$ ./hello
```

Hello World from Chapel!

If we can see this, everything works!

For production jobs, submit a batch script to the queue
[sbatch hello.sh](#)

A few terms:

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

Thread: a system resource that executes tasks

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

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

(in contrast with):

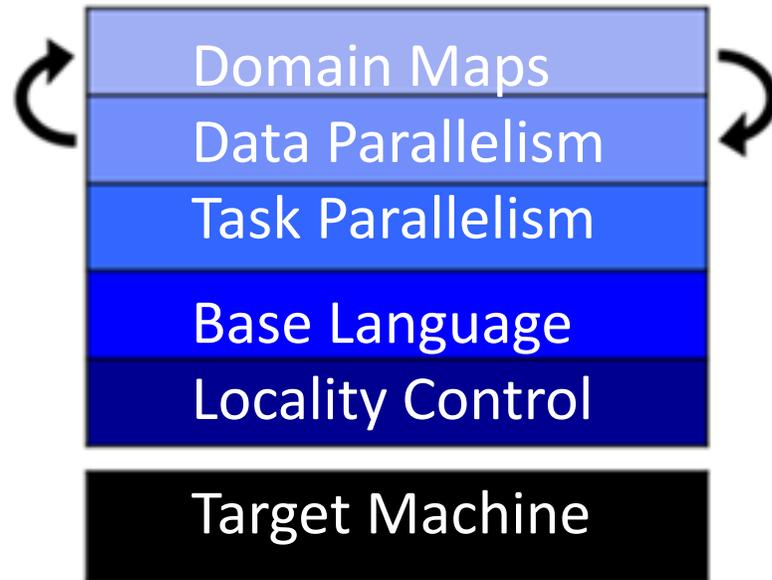
Data Parallelism: a style of parallel programming in which parallelism is driven by computations over collections of data elements or their indices

Chapel's Multiresolution Philosophy

Multiresolution Design: Support multiple tiers of features

- higher levels for programmability, productivity
- lower levels for greater degrees of control

Chapel language concepts



- build the higher-level concepts in terms of the lower
- permit the user to intermix layers arbitrarily Domain Maps Data Parallelism

Locales in Chapel:

Definition:

- Abstract unit of target architecture
- Supports reasoning about locality
defines “here vs. there” / “local vs. remote”
- Capable of running tasks and storing variables
i.e., has processors and memory

Typically: A compute node (multicore processor or SMP)

Using locales:

- Users specify # of locales when running Chapel programs

```
a.out --numLocales=4
```

```
a.out -nl 4
```

- Chapel provides built-in locale variables

```
config const numLocales: int = ...;
```

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

```
Locales: L0 L1 L2 L3
```

- User's main() begins executing on locale #0

Parallel loops: forall, coforall

forall loops: Central concept for data parallelism

- use when a loop should be executed in parallel...
...but can legally be executed serially
- use when desired # tasks \ll # of iterations,
typically executed using $1 < \#tasks \ll \#iters$

coforall loops: create a new task per iteration,
executed using a task per iteration

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

```
var counter = 0;
forall i in 1..100 with (+ reduce counter) {
  counter += i;
}
writeln("sum of the index = ", counter);
```

```
[jemmyhu@gra796 single_locale]$ chpl -o forall forall.chpl
[jemmyhu@gra796 single_locale]$ ./forall
sum of the index = 5050
```

```
config const numTasks = here.numPUs();

coforall tid in 0..#numTasks do
  writeln("Hello, world! from task ", tid, " of ",
numTasks);
```

```
[jemmyhu@gra768 Parallel]$ ./coforall_2
Hello, world! from task 3 of 4
Hello, world! from task 2 of 4
Hello, world! from task 1 of 4
Hello, world! from task 0 of 4
```

Serial for loop code: pi.chpl

- **Numerical integration**

$$\int_0^1 \frac{4}{1+x^2} dx = \pi$$

- **Discretization:**

$$\Delta = 1/N: \text{step} = 1/\text{NBIN}$$
$$x_i = (i+0.5)\Delta \quad (i = 0, \dots, N-1)$$

$$\sum_{i=0}^{N-1} \frac{4}{1+x_i^2} \Delta \cong \pi$$

```
[jemmyhu@gra-login3 Base]$ chpl -o pi pi.chpl
```

```
[jemmyhu@gra-login3 Base]$ ./pi
3.14159273692 8.33333e-08
```

```
[jemmyhu@gra-login3 Base]$ ./pi --n=100000
3.1415926536 8.36842e-12
```

```
[jemmyhu@gra-login3 Base]$ ./pi --n=10000000
3.14159265359 6.21725e-14
```

```
//Sample serial Chapel code to calculate Pi
```

```
const pi = 3.14159265358979323846;
```

```
config const n = 1000;
```

```
var h, sum = 0.0, i: int;
```

```
h = 1.0 / n;
```

```
for i in 1..n {
```

```
    var x = h * ( i - 0.5 );
```

```
    sum += 4.0 / ( 1.0 + x**2);
```

```
}
```

```
sum *= h;
```

```
writeln("%.12n %.6n\n", sum, abs(sum-pi));
```

Simple loop parallel: for to forall

```
[jemmyhu@gra768 single_locale]$ chpl -o pi_forall
pi_forall.chpl
[jemmyhu@gra768 single_locale]$ ./pi_forall
3.14159273692 8.33333e-08
[jemmyhu@gra768 single_locale]$ ./pi_forall --n=100000
3.1415926536 8.33378e-12
[jemmyhu@gra768 single_locale]$ ./pi_forall --n=1000000
3.14159265359 8.26006e-14
```

```
// forall parallel to calculate Pi

const pi = 3.14159265358979323846;

config const n = 1000;

var h, sum = 0.0, i: int;

h = 1.0 / n;

forall i in 1..n with (+ reduce sum) {

    var x = h * ( i - 0.5 );

    sum += 4.0 / ( 1.0 + x**2);

}

sum *= h;

writef("%.12n  %.6n\n", sum, abs(sum-pi));
```

Parallel tasks statement: begin

begin:

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

A Chapel program always start as a single main thread. You can then start concurrent tasks with the **begin** statement. A task spawned by the **begin** statement will run in a different thread while the main thread continues its normal execution.

begin statements run **asynchronously**

```
var x=0;
writeln("This is the main thread starting first task");
begin{
  var c=0;
  while c<5 {
    c+=1;
    writeln('thread 1: ',x+c);
  }
}
writeln("This is the main thread starting second task");
begin{
  var c=0;
  while c<5 {
    c+=1;
    writeln('thread 2: ',x+c);
  }
}
writeln('this is main thread, I am done...');
```

```
This is the main thread starting first task
This is the main thread starting second task
This is the main thread, I am done!
thread 2: 101
...
thread 1: 6
...
```

Parallel tasks: cobegin

cobegin:

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

cobegin statements are different in that the calling code waits for the cobegin's block of parallelized code to finish before continuing.

Essentially, the **cobegin** example above is almost equivalent to the **begin** example because all the **writeln** statements run asynchronously, yet the main difference is that no code can run until the **cobegin** block has finished. Thus, the **writeln** statement outside the **cobegin** block will always run last.

```
var x=0;
writeln("This is the main thread, my value of x is ",x);

cobegin{
  {
    var x=5;
    writeln("this is task 1, my value of x is ",x);
  }
  writeln("this is task 2, my value of x is ",x);
}

writeln("this message won't appear until all tasks are done...");
```

Output:

```
This is the main thread, my value of x is 0
this is task 2, my value of x is 0
this is task 1, my value of x is 5
this message won't appear until all tasks are done...
```

Synchronization: sync

The keyword **sync** provides all sorts of mechanisms to synchronise tasks in Chapel.

sync can be applied to a variable, statement or block of code. We can simply use sync to force the parent task to stop and wait until its spawned-child-task ends.

Output:

```
This is the main thread starting a synchronous task
thread 1: 1
thread 1: 2
thread 1: 3
The first task is done...
This is the main thread starting an asynchronous task
this is main thread, I am done...
thread 2: 1
thread 2: 2
thread 2: 3
```

```
var x=0;
writeln("This is the main thread starting a synchronous task");
sync {
  begin{
    var c=0;
    while c<3{
      c+=1;
      writeln('thread 1: ',x+c);
    }
  }
}
writeln("The first task is done...");
writeln("This is the main thread starting an asynchronous
task");
begin{
  var c=0;
  while c<3{
    c+=1;
    writeln('thread 2: ',x+c);
  }
}
writeln('this is main thread, I am done...');
```

Sync statement with loops

Example: computes the value of pi by adding up the area of many rectangles under half of a circle and doubling that value.

Task begin implementation, race condition; the code might print the value of `globalSum*2` before the tasks launched with `begin` all add their partial sum to the `globalSum`.

```
[jemmyhu@gra768 single_locale]$ ./pi_1  
This code estimates pi as 0.0  
Thread: 4 globalSum: 0.392699  
Thread: 3 globalSum: 0.785398  
Thread: 1 globalSum: 1.1781  
Thread: 2 globalSum: 1.5708
```

```
const numRect = 10000000;  
const width = 2.0 / numRect; // rectangle width  
const numThreads = here.numPUs(); // no of cores the  
computers processor has  
var globalSum: real = 0.0;  
  
proc calculateArea(init) {  
  var partialSum: real = 0.0;  
  var x: real;  
  var i: int = init;  
  do {  
    x = -1 + ( i + 0.5) * width;  
    partialSum += sqrt(1.0 - x*x) * width;  
    i += numThreads;  
  } while (i < numRect-1);  
  globalSum += partialSum;  
  writeln("Thread: ", init, " globalSum: ", globalSum);  
}  
  
for i in 1..numThreads {  
  begin calculateArea(i);  
}  
  
writeln("This code estimates pi as ", globalSum*2);
```

Sync statement with loops

Fix this race condition by adding a sync statement to the for loop that creates the tasks:

```
sync for i in 1..numThreads { ... }
```

```
[jemmyhu@gra768 single_locale]$ ./pi_1_sync  
Thread: 3 globalSum: 0.392699  
Thread: 4 globalSum: 0.785398  
Thread: 2 globalSum: 1.1781  
Thread: 1 globalSum: 1.5708  
This code estimates pi as 3.14159
```

```
const numRect = 10000000;  
const width = 2.0 / numRect; // rectangle width  
const numThreads = here.numCores; // number of cores the  
computers processor has  
var globalSum: real = 0.0;
```

```
proc calculateArea(init) {  
  var partialSum: real = 0.0;  
  var x: real;  
  var i: int = init;  
  do {  
    x = -1 + ( i + 0.5) * width;  
    partialSum += sqrt(1.0 - x*x) * width;  
    i += numThreads;  
  } while (i < numRect-1);  
  globalSum += partialSum;  
  writeln("Thread: ", init, " globalSum: ", globalSum);  
}
```

```
sync for i in 1..numThreads {  
  begin calculateArea(i);  
}
```

```
writeln("This code estimates pi as ", globalSum*2);
```

Parallel for loops: forall

```
[jemmyhu@gra-login3 single_locale]$ chpl -o pi_1_forall
pi_1_forall.chpl
[jemmyhu@gra-login3 single_locale]$ ./pi_1_forall
Thread: 16 globalSum: 0.0981748
Thread: 9 globalSum: 0.19635
Thread: 8 globalSum: 0.294524
Thread: 11 globalSum: 0.392699
Thread: 10 globalSum: 0.490874
Thread: 12 globalSum: 0.589049
Thread: 14 globalSum: 0.687223
Thread: 6 globalSum: 0.785398
Thread: 5 globalSum: 0.883573
Thread: 15 globalSum: 0.981748
Thread: 2 globalSum: 1.07992
Thread: 1 globalSum: 1.1781
Thread: 3 globalSum: 1.27627
Thread: 4 globalSum: 1.37445
Thread: 13 globalSum: 1.47262
Thread: 7 globalSum: 1.5708
This code estimates pi as 3.14159
```

* Chapel utilises all shared memory cores on a node

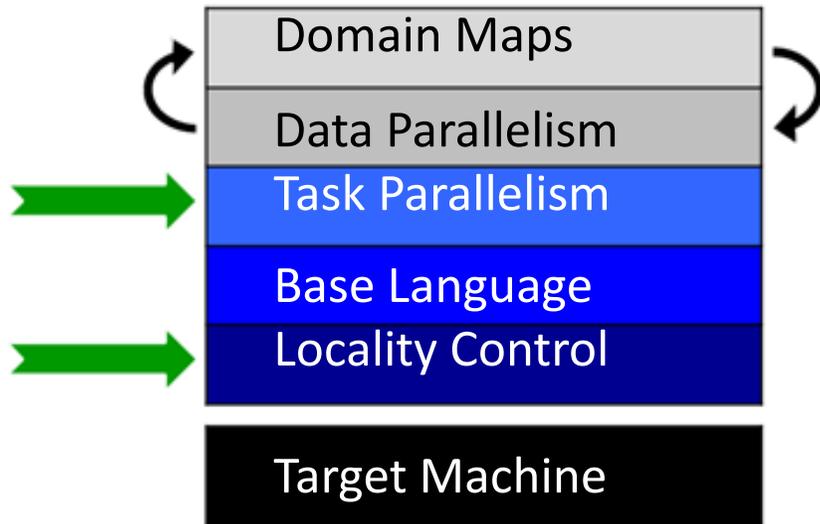
```
const numRect = 10000000;
const width = 2.0 / numRect; // rectangle width
const numThreads = here.numPUs(); // number of cores
the computers processor has
var globalSum: real = 0.0;
```

```
proc calculateArea(init) {
  var partialSum: real = 0.0;
  var x: real;
  var i: int = init;
  do {
    x = -1 + (i + 0.5) * width;
    partialSum += sqrt(1.0 - x*x) * width;
    i += numThreads;
  } while (i < numRect-1);
  globalSum += partialSum;
  writeln("Thread: ", init, " globalSum: ", globalSum);
}
```

```
forall i in 1..numThreads {
  calculateArea(i);
}
```

```
writeln("This code estimates pi as ", globalSum*2);
```

Task Parallelism, Locality Control, by example



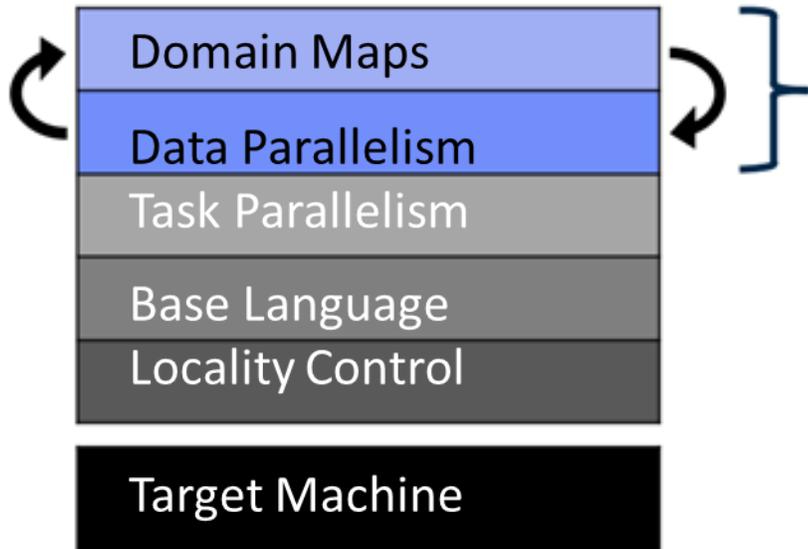
```
//taskParallel.chpl

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

```
[jemmyhu@gra800 single_locale]$ ./taskParallel
Hello from task 1 of 2 running on gra800
Hello from task 2 of 2 running on gra800
```

```
[jemmyhu@gra796 multi_locale]$ srun ./taskParallel_real -nl 2
Hello from task 2 of 2 running on gra796
Hello from task 1 of 2 running on gra796
Hello from task 2 of 2 running on gra800
Hello from task 1 of 2 running on gra800
```

Data Parallelism, by example



dataParallel.chpl

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

```
[jemmyhu@gra-login1 single_locale]$ ./dataParallel --n=5  
1.1 1.3 1.5 1.7 1.9  
2.1 2.3 2.5 2.7 2.9  
3.1 3.3 3.5 3.7 3.9  
4.1 4.3 4.5 4.7 4.9  
5.1 5.3 5.5 5.7 5.9
```

Distributed Data Parallelism, by example

```
dataParallel.chpl
use CyclicDist;

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

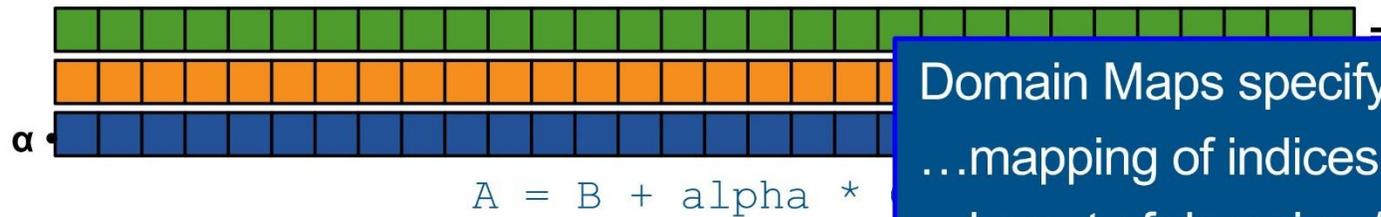
```
[jemmyhu@gra796 multi_locale]$ srun ./dis_dataParallel_real --n=5 -nl 3
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

```
[jemmyhu@gra796 multi_locale]$ srun ./dis_dataParallel_real --n=5 -nl 2
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

Domain Maps: A Multiresolution Feature

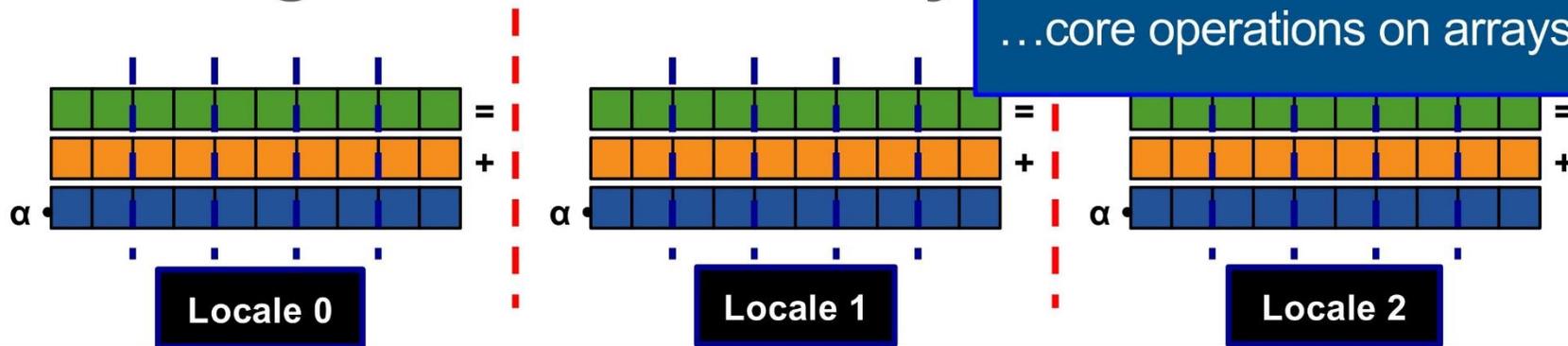


Domain maps are “recipes” that instruct the compiler how to map the global view of a computation...



Domain Maps specify...
...mapping of indices to locales
...layout of domains / arrays in memory
...parallel iteration strategies
...core operations on arrays / domains

...to the target locales' memory



COMPUTE

STORE

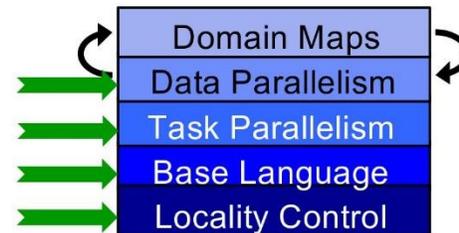
ANALYZE



Chapel's Domain Map Philosophy



- 1. Chapel provides a library of standard domain maps**
 - to support common array implementations effortlessly
- 2. Expert users can write their own domain maps in Chapel**
 - to cope with any shortcomings in our standard library



- 3. Chapel's standard domain maps are written using the end-user framework**
 - to avoid a performance cliff between "built-in" and user-defined cases
 - in fact every Chapel array is implemented using this framework



References

<https://chapel-lang.org/docs/>

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

<http://faculty.knox.edu/dbunde/teaching/chapel/tutorial-1.9.html>

<https://hpc-carpentry.github.io/hpc-chapel/01-intro/>

<https://learnxinyminutes.com/docs/chapel/>