An Introduction to Chapel

Daniel Fedorin

Productive parallel computing at every scale.

Productive parallel computing at every scale.

How?

Why?

Presentation Outline

1. Why is parallelism important?

- It can speed up your work
- It can make intractable problems, tractable
- It is everywhere

2. How do I use Chapel to make parallel computing productive?

- Parallel computation is baked into the language, not an add-on
- Multi-resolution philosophy lets you work at the level of abstraction you need
- Many powerful features fall out from these guiding principles

Parallelism is Important

Parallel Computing For Performance

— Parallel computing allows programs to run much, much faster

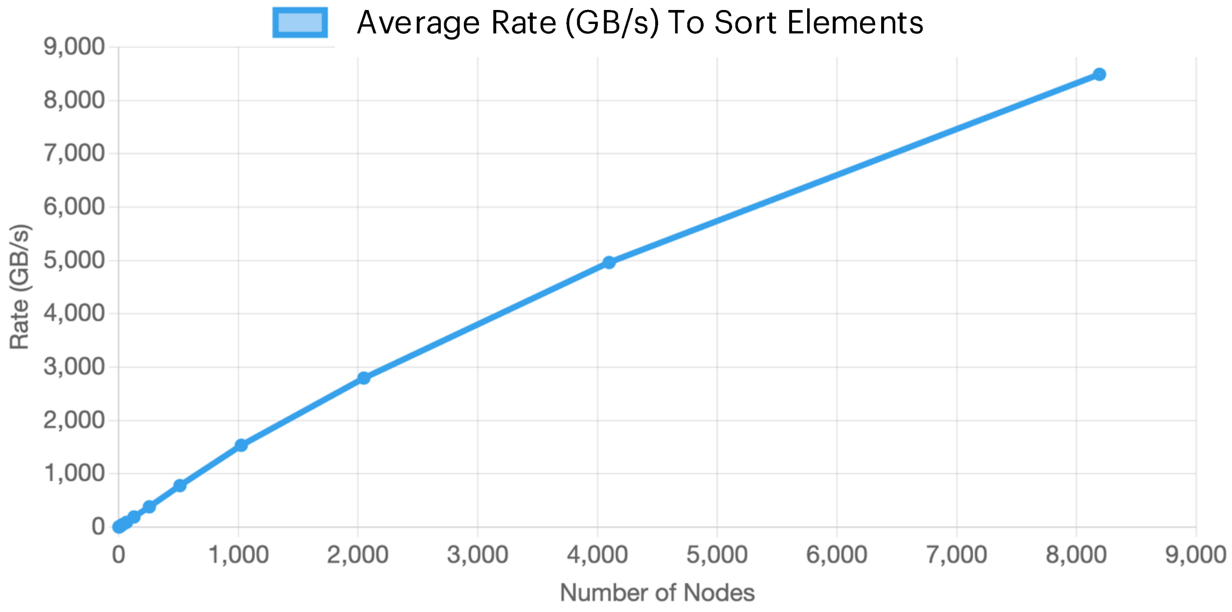
- Consider an analogy to wheat. It would take 3-4 months for a single seed to mature
- If grown one-by-one, a single-acre wheat field would take 225,000 years
- Fortunately, wheat can be grown in parallel

Sequential
Grow a single seed before planting anything else

Parallel
Grow plants simultaneously

Parallel Computing For Tractability

— Some problems are too big to solve on a single machine
- E.g., Large, detailed physics simulations, massive computations
- As part of one of our benchmarks, Arkouda sorted 256 TiB of data in 30 seconds
  - This far exceeds the memory capacity of a single machine (Linux kernel can handle 64TB)



Source: Arkouda `argsort` Benchmark

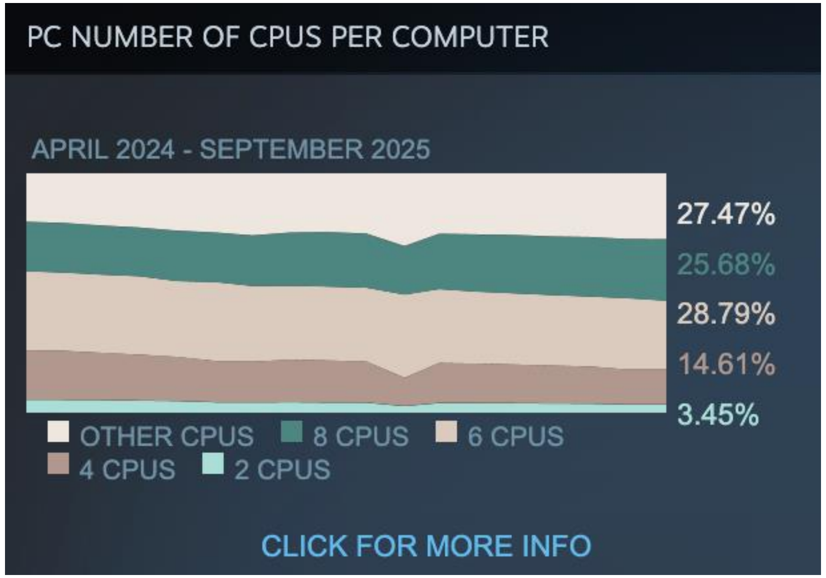
Hardware: HPE Cray EX with a Slingshot-11 network (200 Gb/s)

HPE

Parallel Hardware is Available

Parallelism can (1) speed up your code and (2) let you handle bigger problems…

…and practically anyone with a computer already has access to parallel hardware!



99.97% of surveyed computers have more than 1 CPU!

Measure the Performance of your Gaming GPU with Chapel

Posted on August 27, 2024.

Tags: GPU Programming | How-to | Windows

By: Ahmad Rezaii

If you have a GPU, you have a ton of tiny cores!

Everyone Needs Parallelism

If you are an HPC programmer, you have access to that parallel hardware, and more

- HPC systems also have multiple cores and many of them have GPUs
- Performant parallel code on an HPC system can be trickier to write
  - Compute nodes on HPC systems can have multiple network interfaces and CPUs
  - This poses other programming and performance challenges, including NUMA effects
- In an HPC context, you might also want to parallelize your workload across multiple *nodes*
  - You still want all the aforementioned forms of parallelism

Chapel brings parallelism to the table

HPE

The Format

- I don’t know your background
- I only have 30 minutes
- This will be a high-level overview advertising many features
- Specific tutorials today and tomorrow cover the details

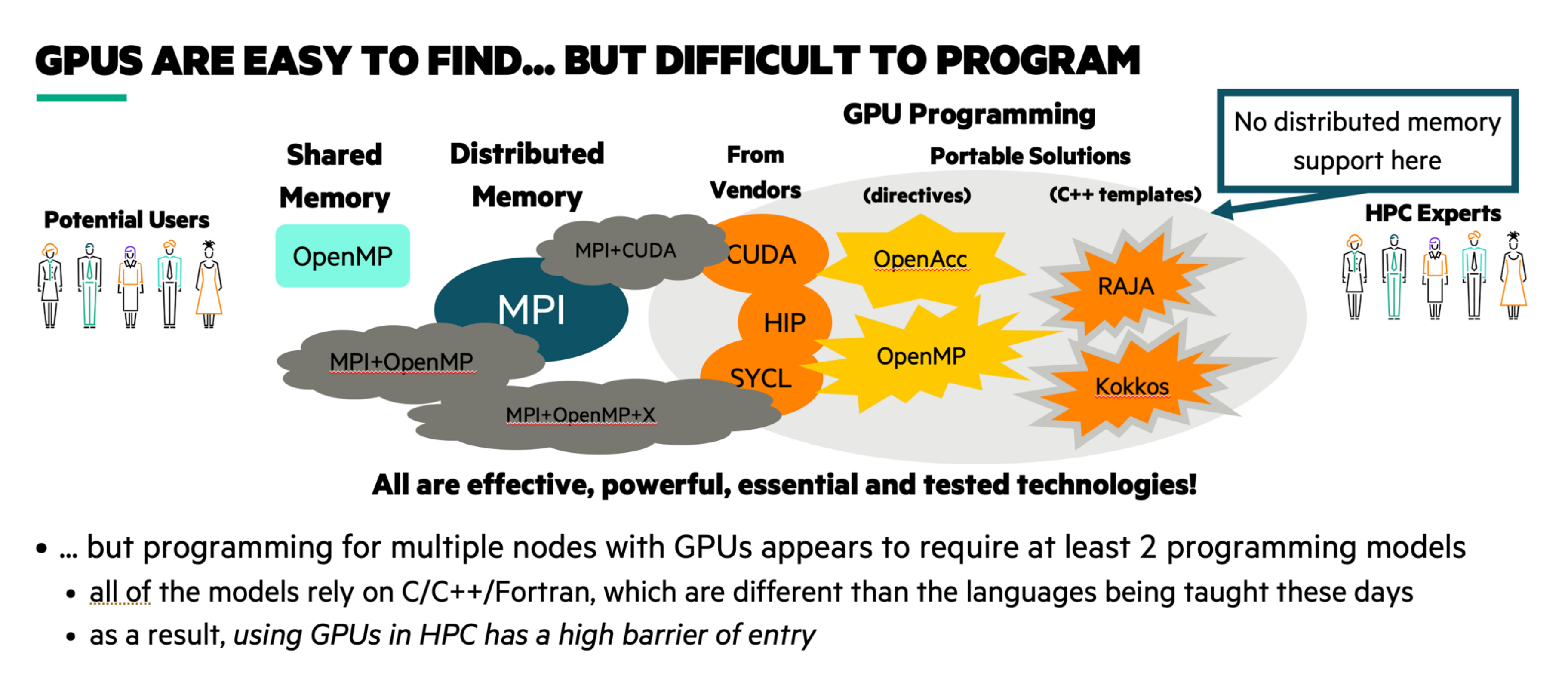
October 7: Tutorials, Day 1

Time (PDT)	
9:00 - 9:10	**Welcome/Introduction** *Brandon Neth* ▸ Description
9:10 - 9:40	**An Introduction to Chapel** *Daniel Fedorin* ▸ Description
9:40 - 10:20	**IO Demo/Exercise Session** *Lydia Duncan* ▸ Description
10:20 - 11:00	**Parallel Loops Demo/Exercise Session** *Shreyas Khandekar* ▸ Description
11:00 - 11:30	**Break**
11:30 - 12:10	**Distributions Demo/Exercise Session** *Brandon Neth* ▸ Description
12:10 - 12:50	**Aggregate Data Structures Demo/Exercise Session** *Jade Abraham* ▸ Description
12:50 - 14:00	**Free-Code Session** ▸ Description

The Parallel Programming Landscape

— Parallel programming is important, so there are many technologies to help with it.
— As I've already mentioned, there are many different types of parallel programming!
- Multi-core programming (threads?)
    - POSIX threads / pthreads in C
    - std::thread in C++
    - Rayon in Rust
    - OpenMP
- GPU Programming
    - CUDA/HIP
    - PyTorch / NumPy (wraps vendor GPU libs)
    - Kokkos
    - OpenCL
    - OpenACC
    - OpenMP
- Distributed Programming
    - MPI

The Parallel Programming Landscape



A New Language.

Chapel is designed from the ground up with two major philosophical goals, which makes it uniquely suited for writing parallel code

— **Parallelism-by-default**
While other languages provide parallelism as an extension on top of the language, or a third-party library, Chapel keeps parallelism at the forefront.

— **A multi-resolution philosophy**
Chapel provides high-level, elegant parallel programming constructs, but gives the user more control if these constructs prove insufficient.

In some ways, most of Chapel’s features are consequences of these two design goals.

Chapel as a Unified Parallel Language

Chapel's design enables it to seamlessly accommodate various parallel programming paradigms

— **Locales** describe where a computation could take place and where data could be stored
  - Locales are a fundamental building block in Chapel, supporting its parallel-by-default nature

— **Distributions** provide recipes for representing and distributing data
  - Distributed, sparse, GPU arrays are just arrays, but the language knows to treat them specially when they need it!

— **Parallel Iterators** enable data-structure-specific parallelism
  - Generic, high-level code can use the appropriate parallelization depending on the data structure provided

— **Low-level features** provide explicit control when high-level abstractions won't do
  - Explicit task parallelism, locks, atomics, barriers, etc. are all part of the standard library!

— **A Parallel Standard Library** enables uniform and on-by-default high performance
  - Summations, bulk operations, standard library sort are parallel

— **A global memory view** makes accessing remote and local data uniform, hiding implementation details
  - No need to explicitly fetch data from other places

What does Chapel look like?

Chapel

```
record myPair {
  var x: int;
  var y: string;

  proc foo() {
    writeln("(", x, ", ", y, ")");
  }
}

var p = new myPair(42, "hello");
var Ap: [1..10] myPair;
```

Python

```
class MyPair:
    def __init__(self, x: int = 0, y: string = ""):
        self.x = x
        self.y = y

    def foo(self):
        print(f"({self.x}, {self.y})”)


p = MyPair(42, “hello”)
Ap = [MyPair() for i in range(10)]
```

Chapel’s General Features

Chapel has many of the usual features of imperative languages. We’ll cover some of them today.

Input and Output
Covers reading, writing, and formatted IO

Aggregate Data Structures
Covers records, classes, and memory management

What next?

- In the remainder of the talk, I will talk at a high level about Chapel’s unique parallel features
- During the rest of the tutorial days, these features (and more!) will be discussed in-depth

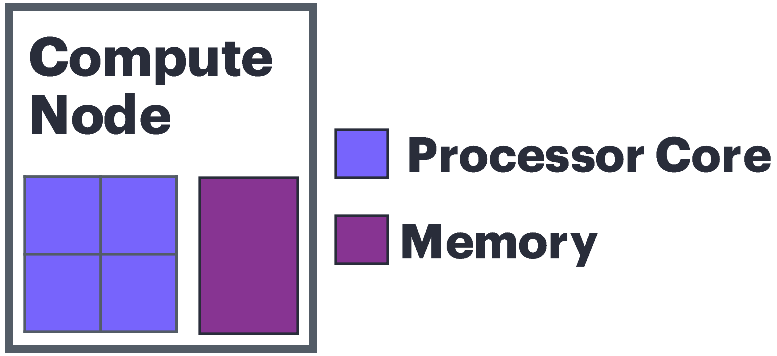
Locales

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 locale as a compute node

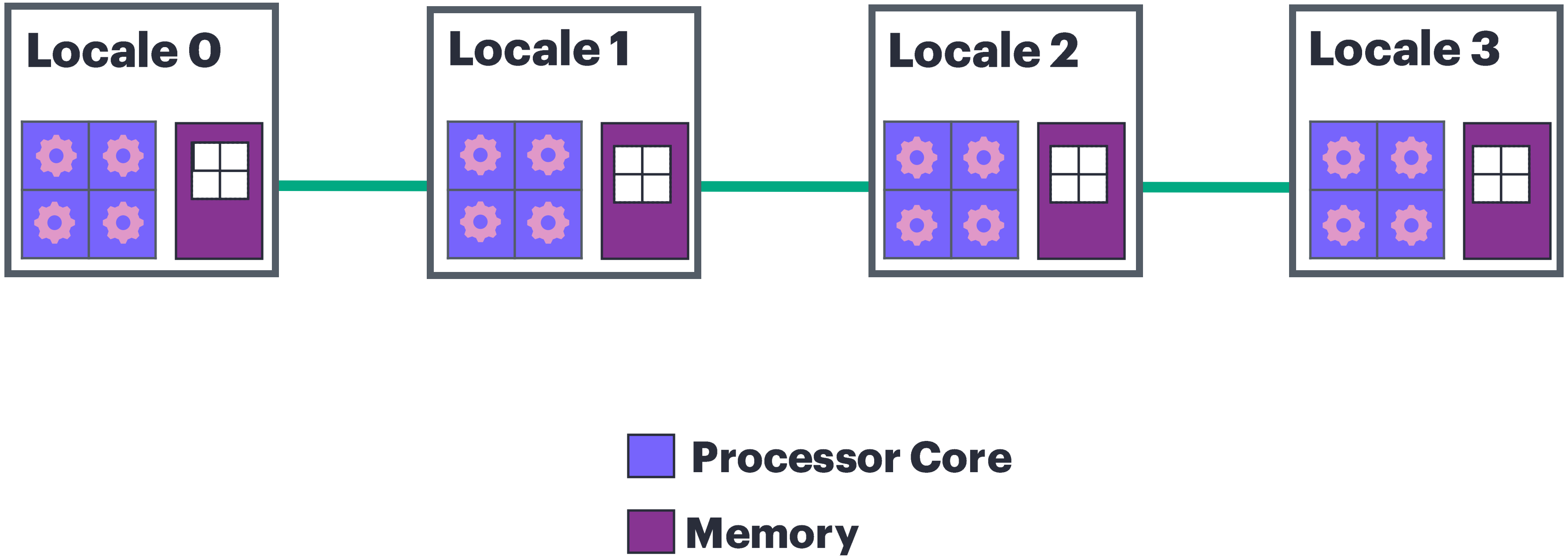


Locales

Scalable parallel computing has two major concerns:

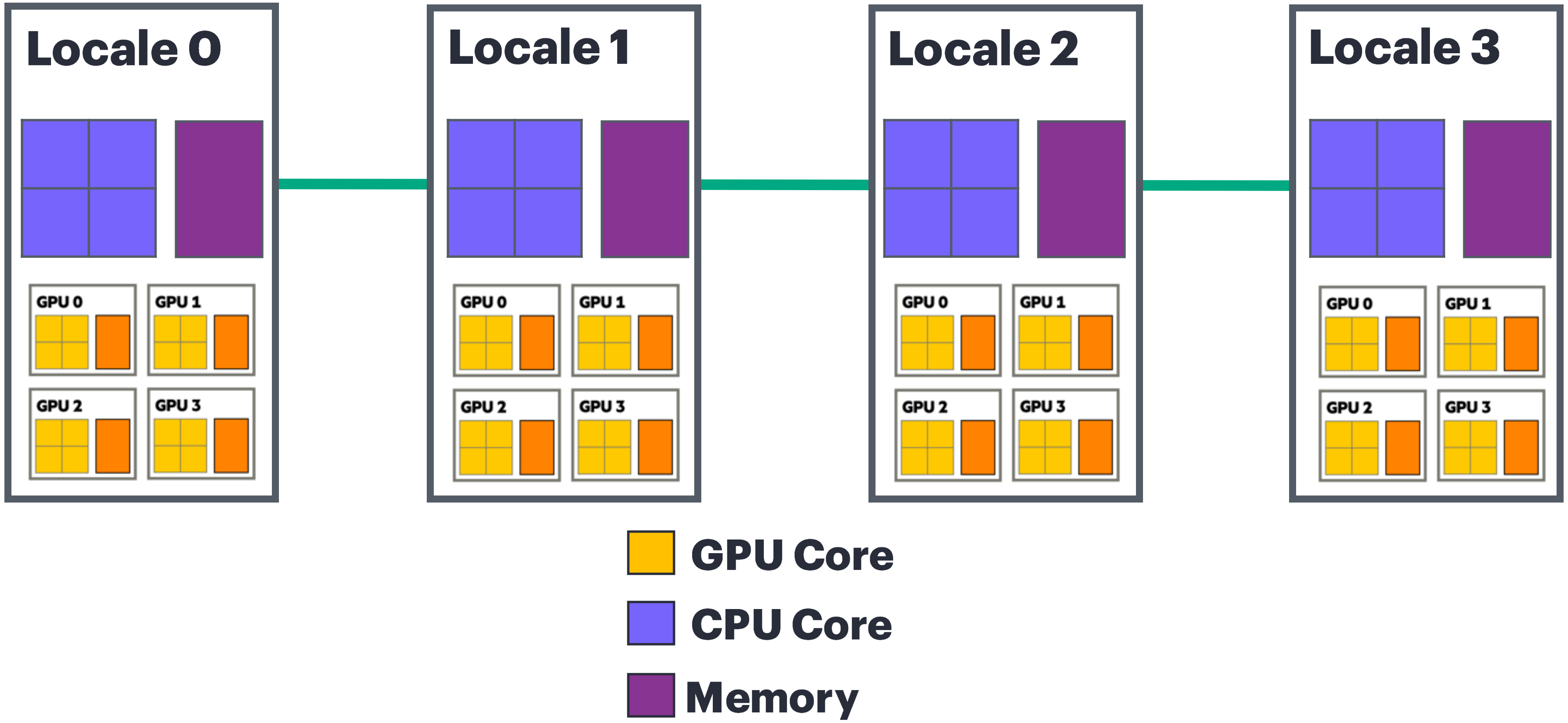
— **parallelism:** Which tasks should run simultaneously?

— **locality:** Where should tasks run? Where should data be allocated?



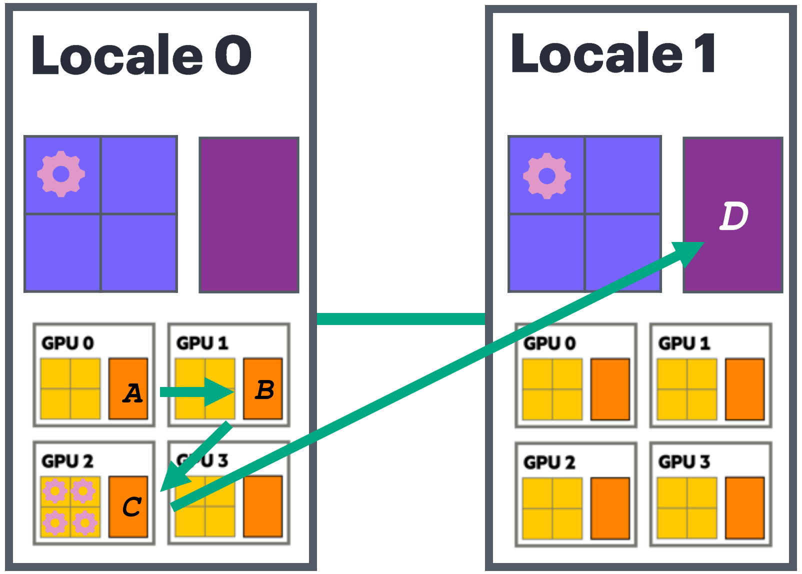
Locales for modeling GPUs

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



Locale Examples

```
on here.gpus[0] var A = [1,2,3,4,5];
on here.gpus[1] var B = A;
on here.gpus[2] {
  var C = B*B;

  on Locales[1] {
    var D = C;
  }
}
```



Locales

Locality is a central notion to Chapel.

- All code is executing on some locale (always available via the ‘here’ variable)
- All variables have a locale on which they are stored (you can write ’myVar.locale’ to retrieve it)

Arrays, Domains, and Distributions

Arrays are a core data structure for many parallel tasks.

Chapel's arrays are very general, allowing the user control of how they are indexed, stored, and iterated

A domain (like 'D1') can describe the shape of the array

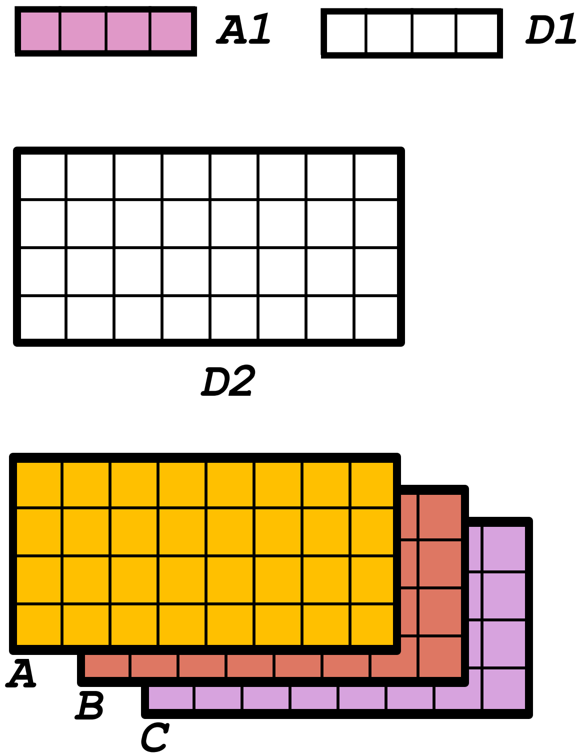
— its dimensions (1D, 2D)

— its size (1x10, 200x200)

Multiple arrays can share a domain

— So indices into 'A' are always valid for 'B' and 'C'

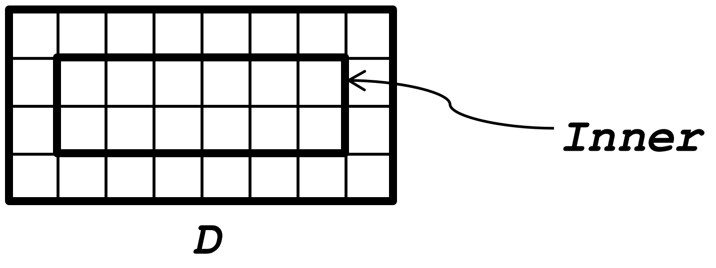
```
var A1 = [1,2,3,4];
var D1 = A.domain; // same as {0..3}

var D2 = {0..7, 0..3};
var A, B, C: [D2] int;
```



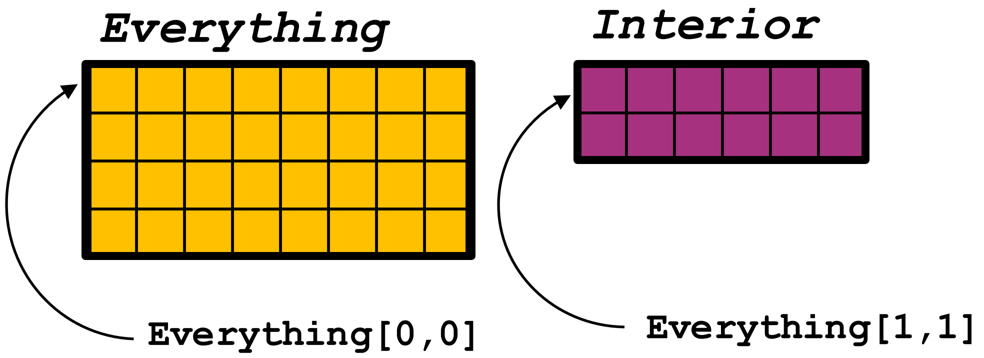
Arrays, Domains, and Distributions

A domain can also describe at what indices array elements reside

- for some problems, it's convenient to index arrays at 0, for others it isn't
- Even if 'D' is 0-indexed, it's convenient to 1-index 'Inner'
- sometimes, we can allow indices can come and go




```
var D = {0..3, 0..7};
var Inner = D.expand(-1);

var Everything: [D] int;
var Interior: [Inner] int;
```

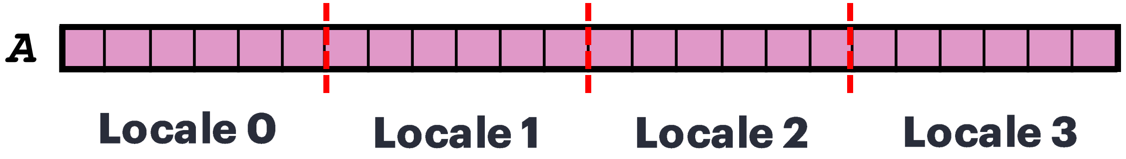


Arrays, Domains, and Distributions

Distributions describe many properties of the array, including…

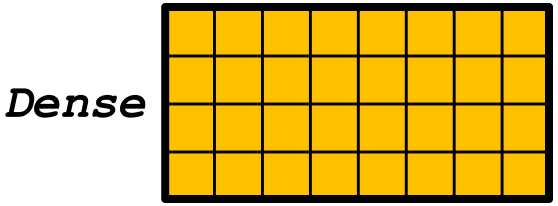
— **Where it's stored:** single locale, split in even chunks across all locales, split round-robin across all locales, etc.?

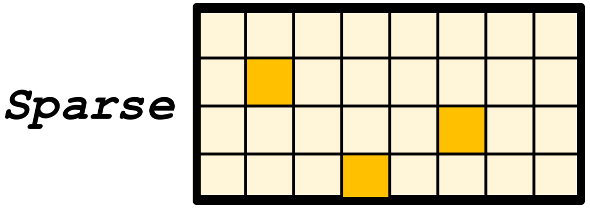
- Previously, we've seen storing variables on different locales
- But big arrays may not fit in the memory of a single node!



— **How it's stored:** by default, arrays are arranged consecutively in memory, but they don't have to be!

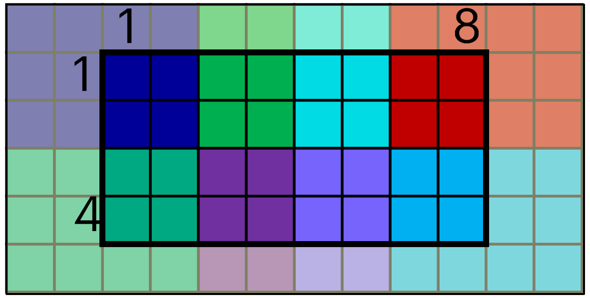
- Compressed Sparse Columns and Compressed Sparse Rows are layouts for sparse arrays





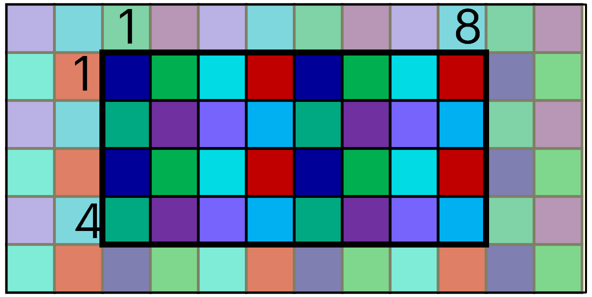
Arrays, Domains, and Distributions

`Block`



distributed to

`Cyclic`



distributed to

More on Arrays, Domains and Distributions

Brandon's demo on distributions will cover this topic in more depth

Distributions
Covers arrays, domains, and distributions

Parallel Loops

In addition to sequential ‘for’ loops, Chapel provides parallel loops

— **‘foreach’ loops**

- assert order-independence (iterations ought not to affect each other).
- could loosely correspond to vectorizable operations

```
on here.gpus[0] var A = foreach i in 1..10 do i * i;
```

— **‘forall’ loops**

- invoke the *parallel iterator* of the thing-being-iterated
- many of Chapel’s standard data structures come with parallel iterators (ranges, arrays, etc.)
- you can automatically parallelize computations, in a way that aligns with the data structure

```
forall i in zip(A.domain, B.domain) do B[i] = A[i] + 1;
```

HPE

Parallel Loops

In addition to "plain" 'for' loops, Chapel provides parallel loops

- **'foreach' loops**
  - assert order-independence (iterations ought not to affect each other).
  - could loosely correspond to vectorizable operations

```
on here.gpus[0] var A = foreach i in 1..10 do i * i;
```

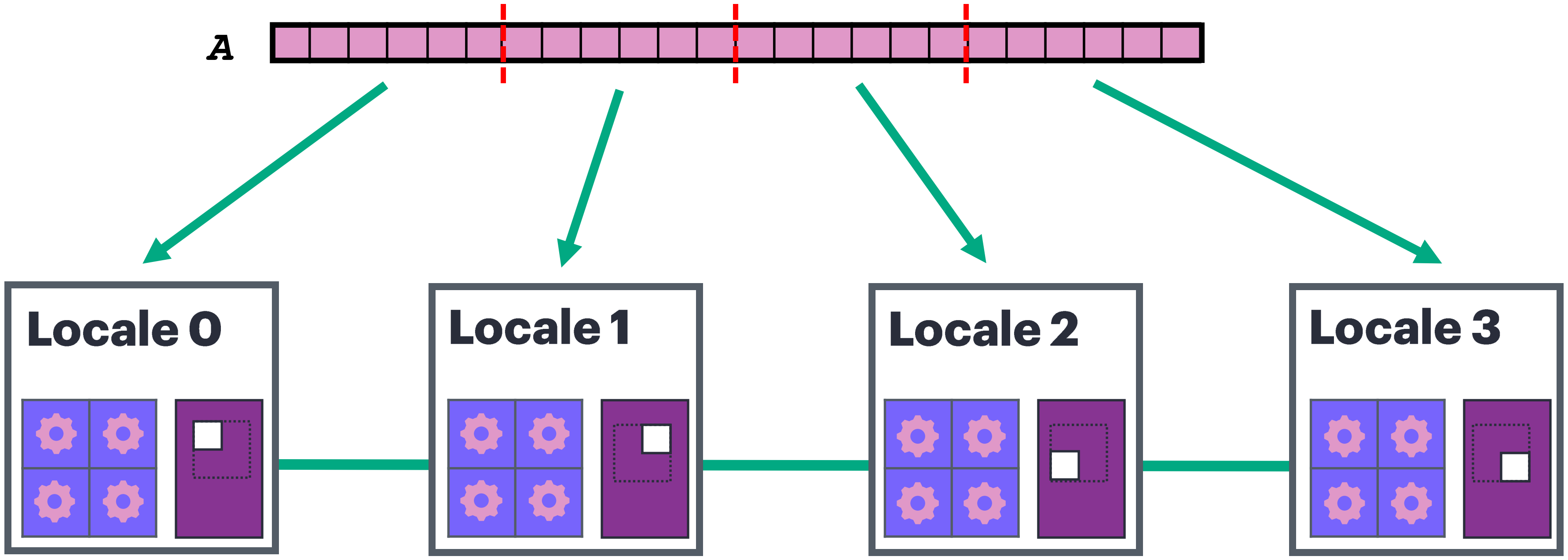
- **'forall' loops**
  - invoke the *parallel iterator* of the thing-being-iterated
  - many of Chapel's standard data structures come with parallel iterators (ranges, arrays, etc.)
  - this means you can automatically parallelize computations, in the most appropriate way

```
B = A + 1; // use promotion
```

Parallel Loops

Distributions customize the parallel iterator, so distributed arrays are processed distributedly

```
var A = blockDist.createArray(1..24, int);
forall a in A { /* ... */ }
```



Parallel Loops

'forall' loops are high-level parallelism constructs.

— "How many tasks?"
  - Data structure decides, often depending on the available hardware and load

— "Which task gets what piece of the work?"
  - Data structure decides (e.g., block distributed array gives each thread a "block")

— "What nodes / devices / domains does the code run on?"
  - Data structure decides (local is a common default, except for distributed arrays)

A lot of the time, fire and forget!

With generics, the same code can be used for a variety of parallel behaviors.

More on Parallel Loops

Shreyas' demo on parallel loops will cover this topic in more depth

Parallel Loops
Covers 'coforall', 'forall', and 'foreach' loops, and more!

Lower-Level Parallel Constructs

Chapel is a *multi-resolution* language: high-level features can give ways to low-level features

- **'coforall' loops**
  - spawn exactly one task for each iteration of the loop
  - allow for explicit control over task parallelism

```
coforall i in 1..here.maxTaskPar do foo();
```

- **'cobegin' statement**
  - executes each statement in the block in a new task

```
cobegin { foo(); bar(); }
```

- **atomics, syncs, barriers**
  - if you need various synchronization idioms

```
var x: atomic int; x.exchange(1);
```

The high-level features (parallel iterators for 'forall' loops) are written using the low-level features.

Sync and Atomic Variables

Our advent of code articles cover these lower-level features as tools for solving programming puzzles.

Day 11: Monkeying Around
Covers ‘coforall’, ‘sync’, barriers

Day 12: On the Summit
Covers ‘atomic’, ‘coforall’

Conclusion

Recap

1. Why is parallelism important?

- It can speed up your work
- It can make intractable problems, tractable
- making efficient use desktops and HPC machines requires parallelism

2. How do I use Chapel to make parallel computing productive?

- **Chapel bundles parallelism from the get-go**, giving you consistent tools to express a variety of parallel work
- **Locales** + **'on' statements** let you talk about where code should run and memory should be stored
- **Arrays, Domains** and **Distributions** offer powerful tools for storing and distributing collections of elements
- **High-level parallel loops** provide quick and easy parallelism for many data structures
- **Low-level features** like 'coforall' loops, atomics, etc., let you implement traditional parallel idioms and more