



**Hewlett Packard  
Enterprise**

# **CHAPEL TUTORIAL**

**ChapelCon 2024**  
June 5, 2024



# WELCOME & INTRODUCING THE SPEAKERS

---



**Jade Abraham**



**Michael Ferguson**



**Daniel Fedorin**



# TODAY'S OUTLINE

---

## Why use Chapel for Parallel Computing?

## Introducing the Heat Diffusion Problem

- Learn about variables, ranges, arrays, and for loops

## Faster Heat Diffusion with Parallel Computing

- Learn about forall loops, race conditions, locales, on statement, implicit communication

## Heat Diffusion on Multiple Locales

- Learn about GPUs, domains, distributions, blockDist, cyclicDist, counting implicit communication, stencilDist

## Summary



# PARALLEL COMPUTING WITHIN REACH

Chapel is a language for efficient and accessible parallel computing

- Parallel computing allows programs to run much, much faster
- Trouble is, most languages are designed for sequential computing

## Sequential and Parallel



### **Sequential**

Grow a single seed before planting anything else



### **Parallel**

Grow plants simultaneously



# USER-FRIENDLY PARALLEL COMPUTING

---

The Chapel language is designed for easier expression of parallel programs

Write understandable parallel programs across the spectrum of hardware:

- Laptops, workstations, GPUs, compute clusters, supercomputers

**Result:** Chapel is easier to learn than competing technologies

- In one group, new students could contribute **8 times faster** to a simulation [1]
- In two studies, it is **more productive than Python** for parallel applications [2,3]



# **ACHIEVE PERFORMANCE AT ANY SCALE**

---

Chapel enables performance from laptops to supercomputers

Benchmarks in Chapel meet or beat performance in other languages [[3](#),[4](#),[5](#),[6](#)]

Applications written in Chapel have attained new levels of performance

- on supercomputers [[7](#)]
- on desktops and workstations [[6](#),[8](#)]

Chapel is helping users with real-world simulation and analysis tasks [[9](#),[10](#),[11](#)]



# REFERENCES

---

- [1] Laurendeau, Éric . HPC Lessons from 30 Years of Practice in CFD Towards Aircraft Design and Analysis. Keynote presentation at CHI UW 2021. [Link to Video](#).
- [2] Jan Gmys, Tiago Carneiro, Nouredine Melab, El-Ghazali Talbi, Daniel Tuytens. A comparative study of high-productivity high-performance programming languages for parallel metaheuristics. Swarm and Evolutionary Computation, 2020.. [Link to PDF](#).
- [3] Diehl, P., Morris, M., Brandt, S.R., Gupta, N., Kaiser, H. Benchmarking the Parallel 1D Heat Equation Solver in Chapel, Charm++, C++, HPX, Go, Julia, Python, Rust, Swift, and Java. Euro-Par 2023: Parallel Processing Workshops. Available at [Link to PDF](#).
- [4] [Computer Languages Benchmarks Game website](#) and [link to a post summarizing the Chapel results](#)
- [5] [Performance Highlights page on the Chapel website](#)
- [6] [Blog post: Comparing Standard Library Sorts: The Impact of Parallelism](#)
- [7] Scaling results from the Arkouda Sort described in [Chapel 2.0: Scalable and Productive Computing for All](#)
- [8] Dias, Nelson. From C and Python to Chapel as My Main Programming Language. CHI UW 2022 talk. [Link to Slides](#). [Link to Video](#).
- [9] Scott Bachman, Rebecca Green, Anna Bakker, Helen Fox, Sam Purkis and Ben Harshbarger. High-Performance Programming and Execution of a Coral Biodiversity Mapping Algorithm Using Chapel. [Link to Slides](#). [Link to Video](#).
- [10] [Arkouda](#) See specifically [this section](#) of Bill Reus's NJIT Data Science talk.
- [11] The CHAMPS team has published many papers based on their multi-physics software written in Chapel . [Advancements in CHAMPS for Multi-Layer Ice Accretion on Aircraft](#) is one recent example.



# PROBLEM INTRODUCTION: HEAT DIFFUSION

---

- Modeling heat diffusion is an important problem in physical modeling
- The goal is to determine how heat propagates through a material
  - This is determined by the laws of physics
  - Modeled by a differential equation
- There are many different ways to solve differential equations
- We will take the straightforward approach: *direct simulation*
- Techniques similar to direct simulation are used in production-grade Chapel software like CHAMPS
  - CHAMPS is a 3D Multi-Physics Simulation written in Chapel



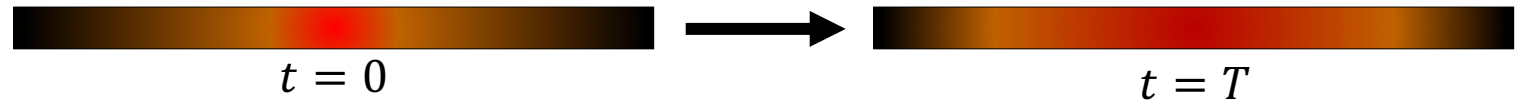
Image Credit: [https://commons.wikimedia.org/wiki/File:Blacksmith\\_working.jpg](https://commons.wikimedia.org/wiki/File:Blacksmith_working.jpg)



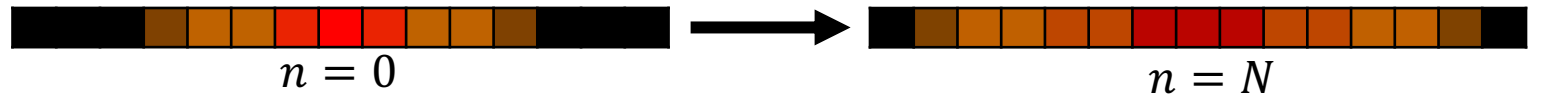


# 1D HEAT EQUATION EXAMPLE

**Differential equation:**  $\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$

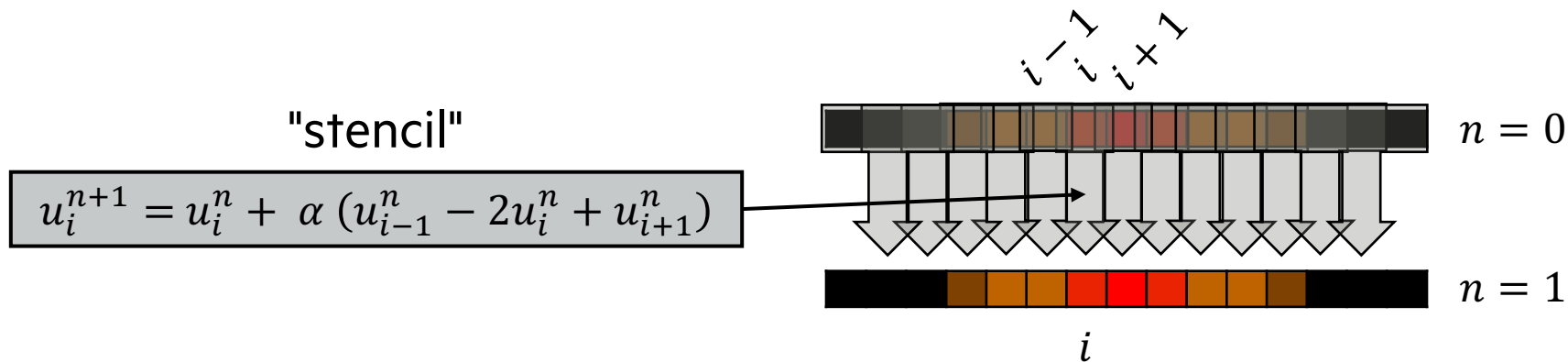


**Discretized (finite difference) equation:**  $u_i^{n+1} = u_i^n + \alpha (u_{i-1}^n - 2u_i^n + u_{i+1}^n)$



## Implementation:

- For each point, apply a “stencil” to the previous state (can be done in parallel!)
- Store the result separately to avoid changing the outcome for other points



# 1D HEAT EQUATION EXAMPLE

**Discretized (finite difference) equation:**  $u_i^{n+1} = u_i^n + \alpha (u_{i-1}^n - 2u_i^n + u_{i+1}^n)$

- where  $i \in \Omega \subset \mathbb{R}^1$  are discrete points in space, and  $(n, n + 1, \dots)$  are discrete instances in time

```
1  const omega = {0..
```

# 1D HEAT EQUATION EXAMPLE

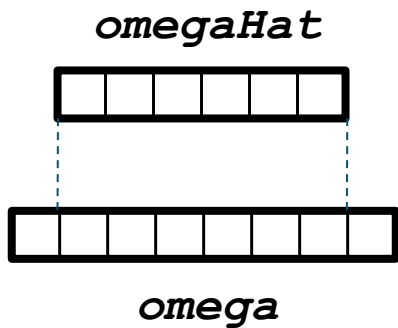
**Discretized (finite difference) equation:**  $u_i^{n+1} = u_i^n + \alpha (u_{i-1}^n - 2u_i^n + u_{i+1}^n)$

- where  $i \in \Omega \subset \mathbb{R}^1$  are discrete points in space, and  $(n, n + 1, \dots)$  are discrete instances in time

## Finite difference algorithm:

Declare ranges that correspond to # of discrete pieces

- 'nx' is the number of such pieces
- We don't have to name the ranges, but it helps



```
1  const omega = {0..<nx},
2      omegaHat = omega.expand(-1);
3  var u: [omega] real = 1.0;
4  u[nx/4..3*nx/4] = 2.0;
5  var un = u;
6  for 1..N {
7      un <=> u;
8      for i in omegaHat do
9          u[i] = un[i] + alpha *
10             (un[i-1] - 2*un[i] + un[i+1]);
11 }
```

# 1D HEAT EQUATION EXAMPLE

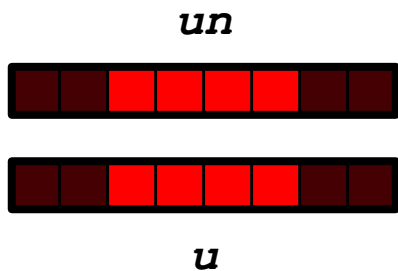
**Discretized (finite difference) equation:**  $u_i^{n+1} = u_i^n + \alpha (u_{i-1}^n - 2u_i^n + u_{i+1}^n)$

- where  $i \in \Omega \subset \mathbb{R}^1$  are discrete points in space, and  $(n, n + 1, \dots)$  are discrete instances in time

## Finite difference algorithm:

Define arrays for current time step and next time step

- ‘omega’ determines indices we can access in ‘u’
  - ‘u[i]’ is good if  $0 \leq i < nx$
  - Otherwise, access is out of bounds
- Array value at each index determines temperature



```
1  const omega = {0..<nx},
2      omegaHat = omega.expand(-1);
3  var u: [omega] real = 1.0;
4  u[nx/4..3*nx/4] = 2.0;
5  var un = u;
6  for 1..N {
7      un <=> u;
8      for i in omegaHat do
9          u[i] = un[i] + alpha *
10             (un[i-1] - 2*un[i] + un[i+1]);
11 }
```

# 1D HEAT EQUATION EXAMPLE

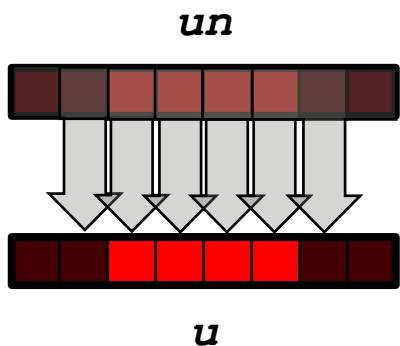
**Discretized (finite difference) equation:**  $u_i^{n+1} = u_i^n + \alpha (u_{i-1}^n - 2u_i^n + u_{i+1}^n)$

- where  $i \in \Omega \subset \mathbb{R}^1$  are discrete points in space, and  $(n, n + 1, \dots)$  are discrete instances in time

## Finite difference algorithm:

For each time step, apply discretized equation

- Use 'omegaHat' to exclude boundaries
- Between each time step, swap arrays
  - 'next' array becomes 'current' array
  - previous 'current' array now scratch space for 'next' one



```
1  const omega = {0..
```

# 1D HEAT EQUATION EXAMPLE

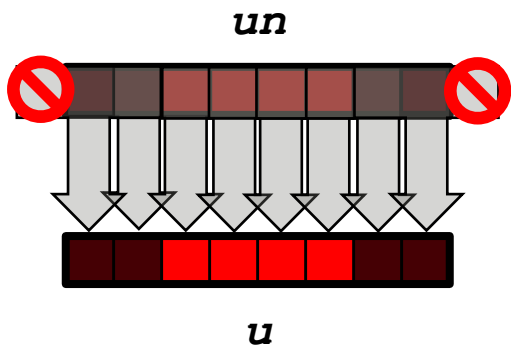
**Discretized (finite difference) equation:**  $u_i^{n+1} = u_i^n + \alpha (u_{i-1}^n - 2u_i^n + u_{i+1}^n)$

- where  $i \in \Omega \subset \mathbb{R}^1$  are discrete points in space, and  $(n, n + 1, \dots)$  are discrete instances in time

## Finite difference algorithm:

For each time step, apply discretized equation

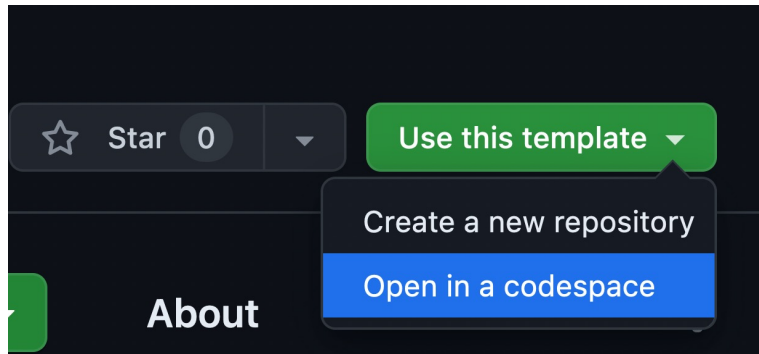
- Use ‘omegaHat’ to exclude boundaries
- Between each time step, swap arrays
  - ‘next’ array becomes ‘current’ array
  - ‘previous’ current array now scratch space for ‘next’ one



```
1  const omega = {0.. $nx$ },
2      omegaHat = omega.expand(-1);
3  var u: [ $omega$ ] real = 1.0;
4  u[ $nx/4..3*nx/4$ ] = 2.0;
5  var un = u;
6  for 1.. $N$  {
7      un <=> u;
8      for i in omega do
9          u[i] = un[i] + alpha *
10             (un[i-1] - 2*un[i] + un[i+1]);
11 }
```

# HANDS-ON: GET THE CODE

- Go to: <https://github.com/DanilaFe/chapelcon-2024-tutorial>
- Click **Use this template**, then **Open in a codespace**



- After some time, this will load a Visual Studio Code Environment
- The file is called '01-serial-heat-diffusion.chpl', and should open by default
- Use the following commands in **Terminal** to compile & run the program, and open the generated image

```
chpl 01-heat-1D-serial.chpl --fast
./01-heat-1D-serial
./01-heat-1D-serial --image=true --N=1000 --nx=256
```

# HANDS-ON: RUN HEAT DIFFUSION

- Previously, you used the following commands in **Terminal** to compile & run the program

```
chpl 01-heat-1D-serial.chpl --fast
./01-heat-1D-serial
```

- How does this scale to larger problem sizes?
  - The 'config' keyword on 'nx', 'n', and 'alpha' allows these constants to be specified from the command line.
  - To do so, simply pass '--nx <size>' to adjust the number of elements in the simulation array.
  - Try running with various problem sizes:

```
time ./01-heat-1D-serial --nx 100
time ./01-heat-1D-serial --nx 1000
time ./01-heat-1D-serial --nx 10000
```





# HANDS-ON: WHAT DOES '--FAST' DO?

- Among other things, '--fast' disables various runtime checks.
- These checks make it easier to diagnose bugs. Let's introduce a bug:

```
-7   for i in omegaHat do
+7   for i in omega do
```

- Compile and run with '--fast':

```
chpl 02-heat-1D-buggy.chpl --fast
./02-heat-1D-buggy
```

(???) doesn't even report an error, just accesses undefined memory!

- Compile without '--fast':

```
chpl 02-heat-1D-buggy.chpl
./02-heat-1D-buggy
```

halt reached - array index out of bounds  
note: index was -1 but array bounds are 0..99999



# 1D HEAT EQUATION EXAMPLE: VARIABLES

```
★1  const omega = {0..nx},
★2      omegaHat = omega.expand(-1);
★3  var u: [omega] real = 1.0;
    4  u[nx/4..3*nx/4] = 2.0;
★5  var un = u;
    6  for 1..N {
    7      un <=> u;
    8      for i in omegaHat do
    9          u[i] = un[i] + alpha *
10              (un[i-1] - 2*un[i] + un[i+1]);
11  }
```

## Meaning

- var/const: execution-time variable/constant
- param: compile-time constant
- No init-expr  $\Rightarrow$  initial value is the type's default
- No type  $\Rightarrow$  type is taken from init-expr

# 1D HEAT EQUATION EXAMPLE: RANGES

```
★1  const omega = {0..<nx},
    2      omegaHat = omega.expand(-1);
    3  var u: [omega] real = 1.0;
★4  u[nx/4..3*nx/4] = 2.0;
    5  var un = u;
★6  for 1..N {
    7      un <=> u;
    8      for i in omegaHat do
    9          u[i] = un[i] + alpha *
   10              (un[i-1] - 2*un[i] + un[i+1]);
   11  }
```

## Meaning

- Regular sequence of integers
  - low <= high: low, low+1, low+2, ..., high
  - low > high: degenerate (an empty range)
  - low or high unspecified: unbounded in that direction

## Examples

```
1..6           // 1, 2, 3, 4, 5, 6
6..1           // empty
3..            // 3, 4, 5, 6, 7, ...
1..<6          // 1, 2, 3, 4, 5
1..6 by -1     // 6, 5, 4, 3, 2, 1
```

# 1D HEAT EQUATION EXAMPLE: ARRAY

```
1  const omega = {0.. $\langle$ nx},
2      omegaHat = omega.expand(-1);
★3  var u: [omega] real = 1.0;
4  u[nx/4.. $3$ *nx/4] = 2.0;
5  var un = u;
6  for 1..N {
7      un <=> u;
8      for i in omegaHat do
9          u[i] = un[i] + alpha *
10             (un[i-1] - 2*un[i] + un[i+1]);
11 }
```

## Meaning

- '[D] t': stores an element of type 't' for each index in domain 'D'
  - E.g., '[omega] real'
  - The domain defines valid indices into array (and more)
- Chapel has array literals, too (not in code block)
  - '[5, 3, 9]': represent the array with elements 5, 3, and 9

# 1D HEAT EQUATION EXAMPLE: FOR LOOPS

```
1  const omega = {0.. $\langle$ nx},
2      omegaHat = omega.expand(-1);
3  var u: [omega] real = 1.0;
4  u[nx/4..3*nx/4] = 2.0;
5  var un = u;
★6  for 1..N {
7      un <=> u;
★8  for i in omegaHat do
9      u[i] = un[i] + alpha *
10         (un[i-1] - 2*un[i] + un[i+1]);
11 }
```

## Meaning

- Executes loop body serially, once per loop iteration
- An example is **‘for i in 1..10’**
- Declares a new variable ‘i’
  - ‘i’ is an ‘int’ because ‘1..10’ is a range of ‘int’s.
  - You can loop over a range, array, iterator, iterable object, ...
- **‘do’** is a cleaner syntax for single-statement loops

# 1D HEAT EQUATION EXAMPLE: WHAT'S NEXT?

```
1  const omega = {0..<nx},
2      omegaHat = omega.expand(-1);
3  var u: [omega] real = 1.0;
4  u[nx/4..3*nx/4] = 2.0;
5  var un = u;
6  for 1..N {
7      un <=> u;
8      for i in omegaHat do
9          u[i] = un[i] + alpha *
10             (un[i-1] - 2*un[i] + un[i+1]);
11 }
```

## Next Steps

- We now have a working implementation of heat diffusion
- Does it make the most out of a modern computer?
  - **No!**
- Parallelism can lead to significant perf improvements
- Most modern CPUs have more than one processor core
- To fully leverage your CPU — and go fast — you need to use all its cores in parallel!
- How do we make use of Chapel's parallel programming support?

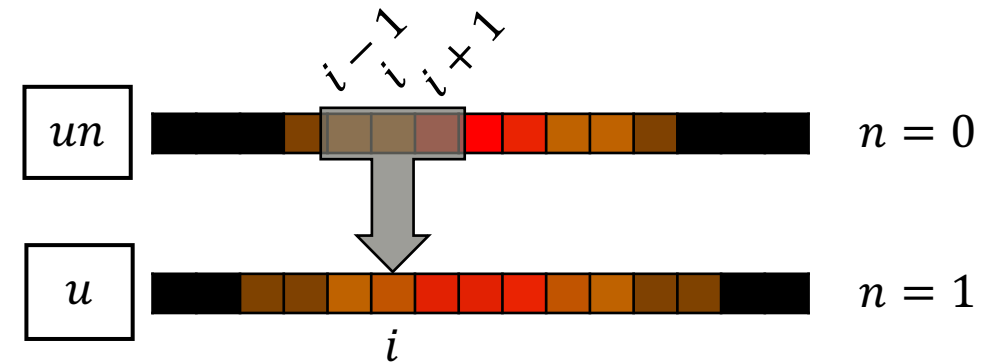
**USING  
PARALLELISM TO  
IMPROVE  
PERFORMANCE**



# 1D HEAT EQUATION EXAMPLE

- Note that in this case, the stencil can be applied to the entire array in parallel
  - each value in  $u^{n+1}$  (the ‘u’ array in the code) depends strictly on values in  $u^n$  (the ‘un’ array in the code)
- Chapel’s ‘forall’ loops are a quick, high-level way to get data parallelism

```
1  const omega = {0.. $\text{nx}$ },
2      omegaHat = omega.expand(-1);
3  var u: [ $\text{omega}$ ] real = 1.0;
4  u[ $\text{nx}/4..3*\text{nx}/4$ ] = 2.0;
5  var un = u;
6  for 1.. $N$  {
7      un <=> u;
★8  forall i in omegaHat do
9      u[i] = un[i] + alpha *
10         (un[i-1] - 2*un[i] + un[i+1]),
11  }
```



Switched the inner ‘for’ loop to a ‘forall’, which automatically runs the loop in parallel when possible.

The rest of the code is unchanged!





# HANDS-ON: RUN HEAT DIFFUSION

- We provide a different version of the program that can do either serial or parallel execution

```
chpl 03-heat-1D.chpl --fast
```

- How does this scale to larger problem sizes?
  - The 'config' keyword on 'nx', 'n', and 'alpha' allows these constants to be specified from the command line.
  - To do so, simply pass '--nx <size>' to adjust the number of elements in the simulation array.
  - Measure the execution times from the previous, serial version:

```
time ./01-heat-1D-serial --nx 100
time ./01-heat-1D-serial --nx 1000
time ./01-heat-1D-serial --nx 10000
```

- Then, compare them to the parallel execution times with the 'forall' loop

```
time ./03-heat-1D --nx 100
time ./03-heat-1D --nx 1000
time ./03-heat-1D --nx 10000
```

# 1D HEAT EQUATION EXAMPLE: FORALL LOOPS

```
1  const omega = {0.. $\langle$ nx},
2      omegaHat = omega.expand(-1);
3  var u: [omega] real = 1.0;
4  u[nx/4..3*nx/4] = 2.0;
5  var un = u;
6  for 1..N {
7      un <=> u;
★8  forall i in omegaHat do
9      u[i] = un[i] + alpha *
10         (un[i-1] - 2*un[i] + un[i+1]);
11 }
```

## Forall loops: Central concept for data parallel computation

- Like for-loops, but parallel
- Implementation details determined by iterand (e.g., '1..N')
  - specifies number of tasks, which tasks run which iterations, ...
  - in practice, typically uses a number of tasks appropriate for target hardware
- In essence: 'forall' runs the parallel iterator of the iterand.
  - Chapel's built-in arrays, ranges, etc. all have parallel iterators

## Forall loops assert...

- ...parallel is allowed:** OK to execute iterations simultaneously
- ...order independence:** iterations could occur in any order
- ...serializability:** all iterations could be executed by one task
  - e.g., can't have synchronization dependences between iterations

# RACE CONDITIONS

- Chapel allows data races
- ‘forall’ loops assert order independence, so writes to ‘u’ can happen in any order
- For synchronization, Chapel provides ‘atomic’ and ‘sync’ types, as well as barriers
  - We will not be covering these in this tutorial

**Safe:** all writes are to different elements of ‘u’

```
forall i in omegaHat do
    u[i] = un[i] + alpha *
        (un[i-1] - 2*un[i] + un[i+1]);
```

**Unsafe:** all writes are to the same index. Final value determined by order of writes (tasks are *racing*)

```
forall i in omegaHat do
    u[1] = un[i] + alpha *
        (un[i-1] - 2*un[i] + un[i+1]);
```



# 1D HEAT EQUATION: WHAT'S NEXT, AGAIN?

---

- Our implementation is parallel, making full use the CPU cores
- To get more computing power, we might want to use more hardware, or different types of hardware
  - Both personal computers and high-scale supercomputers increasingly ship with **GPUs**
  - GPUs excel at certain forms of parallel programming, and can lead to a significant speedup
  - To make the most use of your whole computer, you might want to use CPUs and GPUs
- An alternative approach to solving larger problems is to **connect more computers** together
- This can help divide-and-conquer work to solve problems faster, or to tackle larger problems
  - If one computer can go fast, can 20 computers go 20x faster?
  - Some problems are so big they simply can't fit in a single computer's memory!
- Chapel can support both GPU and multi-node programming using a single shared concept: *locales*



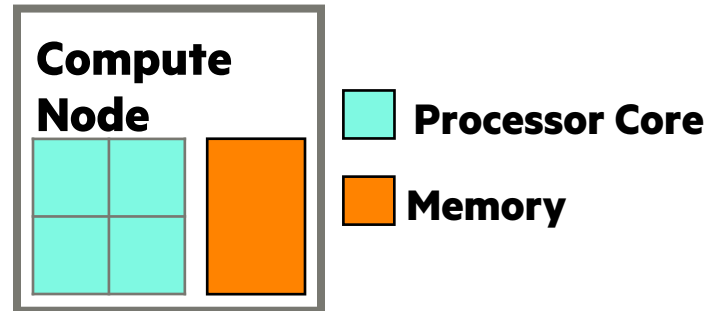
# INTRODUCTION TO LOCALES

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

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

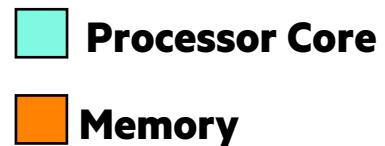
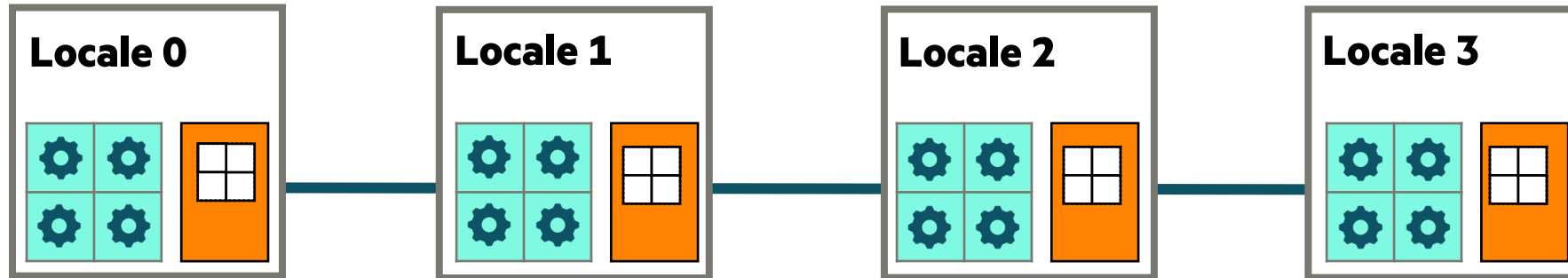
For now, think of each locale as a compute node or a GPU

So far, we've only executed on one locale



# KEY CONCERNS FOR SCALABLE PARALLEL COMPUTING

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



# GETTING STARTED WITH LOCALES

---

Chapel provides built-in locale variables

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

Locale methods support queries about the target system:

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



# BASIC FEATURES FOR LOCALITY

04-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;  
}
```

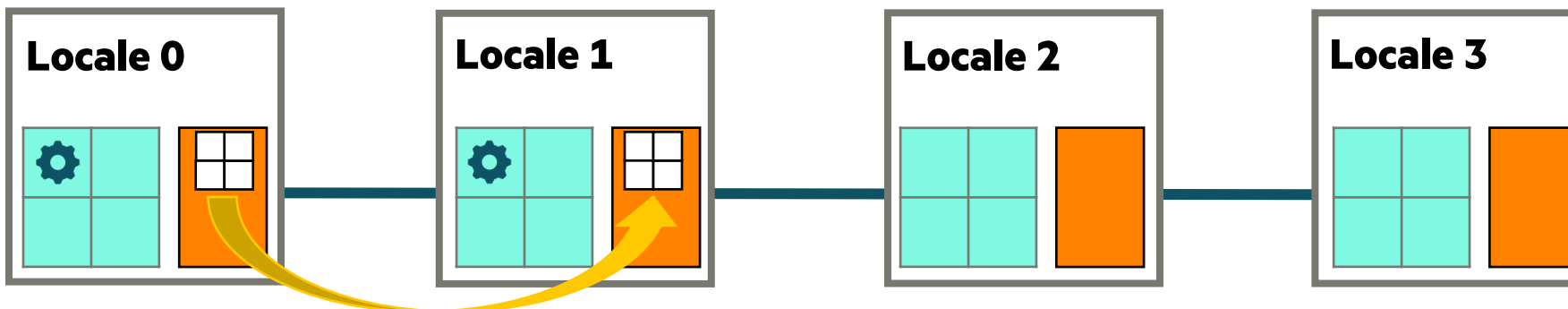
All Chapel programs begin running as a single task on locale 0

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**



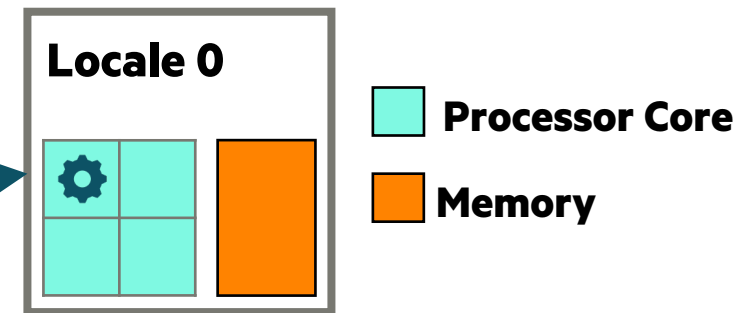


# INTRODUCTION TO LOCALES

Serial 'for' loops do not bring in additional locales or cores.

**This is a serial, local computation**

```
1  const omega = {0..<nx},
2      omegaHat = omega.expand(-1);
3  var u: [omega] real = 1.0;
4  u[nx/4..3*nx/4] = 2.0;
5  var un = u;
6  for 1..N {
7      un <=> u;
8      for i in omegaHat do
9          u[i] = un[i] + alpha *
10             (un[i-1] - 2*un[i] + un[i+1]);
11 }
```

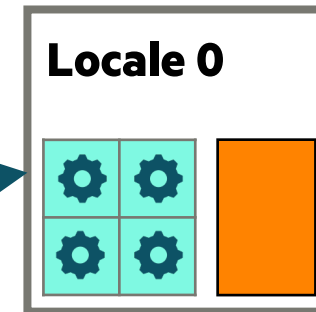



# INTRODUCTION TO LOCALES

By switching to a 'forall' loop, we made use of all available CPU cores

**This is a parallel, but still local computation**

```
1  const omega = {0..<nx},
2      omegaHat = omega.expand(-1);
3  var u: [omega] real = 1.0;
4  u[nx/4..3*nx/4] = 2.0;
5  var un = u;
6  for 1..N {
7      un <=> u;
8      forall i in omegaHat do
9          u[i] = un[i] + alpha *
10             (un[i-1] - 2*un[i] + un[i+1]);
11 }
```



 **Processor Core**  
 **Memory**

# INTRODUCTION TO LOCALES

```
1  const omega = {0.. $\text{nx}$ },
2      omegaHat = omega.expand(-1);
3  var u: [omega] real = 1.0;
4  u[nx/4..3*nx/4] = 2.0;
5  var un = u;
6  for 1..N {
7      un <=> u;
8      forall i in omegaHat do
9          u[i] = un[i] + alpha *
10             (un[i-1] - 2*un[i] + un[i+1]);
11 }
```

- So far, both the serial and parallel version use only the starting locale.
  - The ‘for’ loop uses only one core on the locale
  - The ‘forall’ loop uses all cores on the locale
- How are the concepts of locales and ‘on’ statements used to program GPUs and multi-node systems?

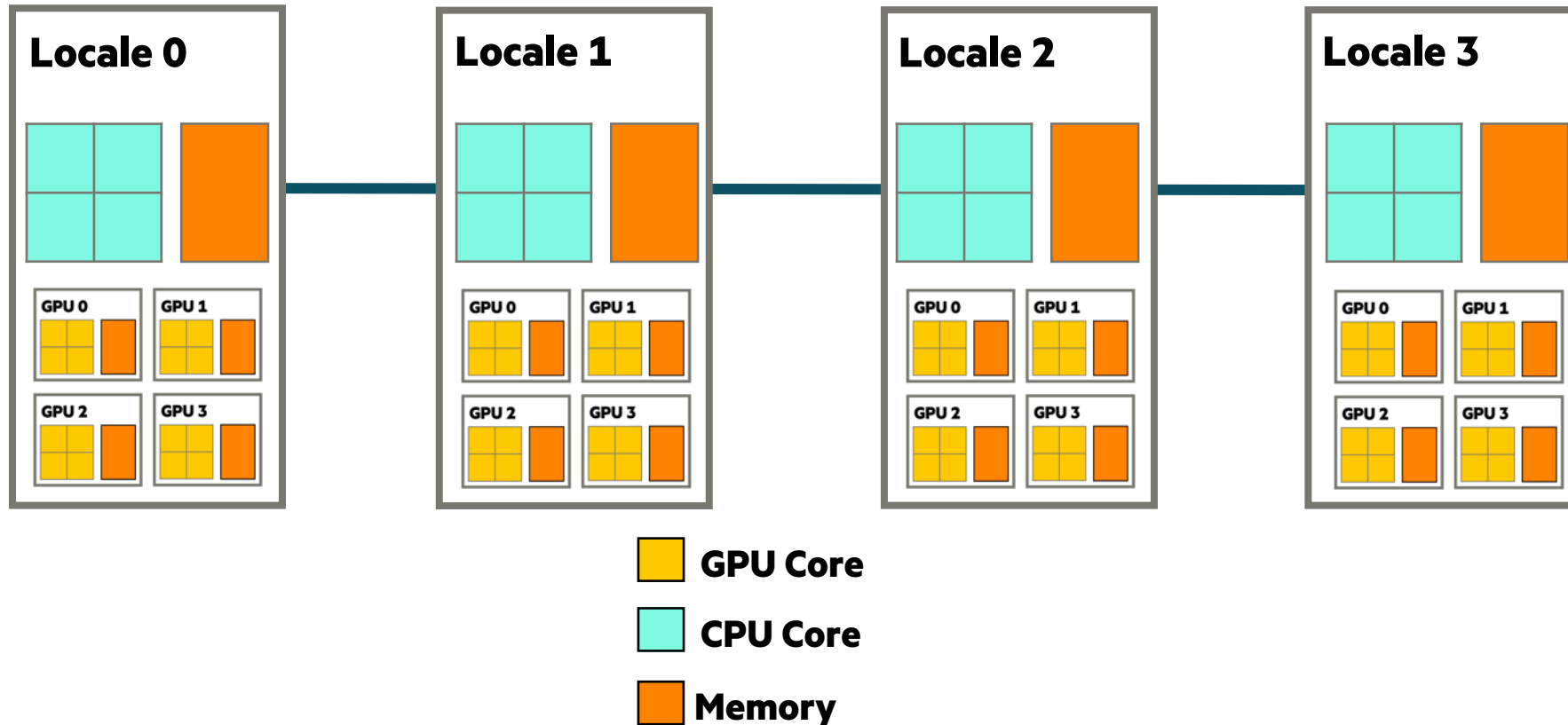
**IMPROVING  
PERFORMANCE  
USING MULTIPLE  
LOCALES**



# PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS

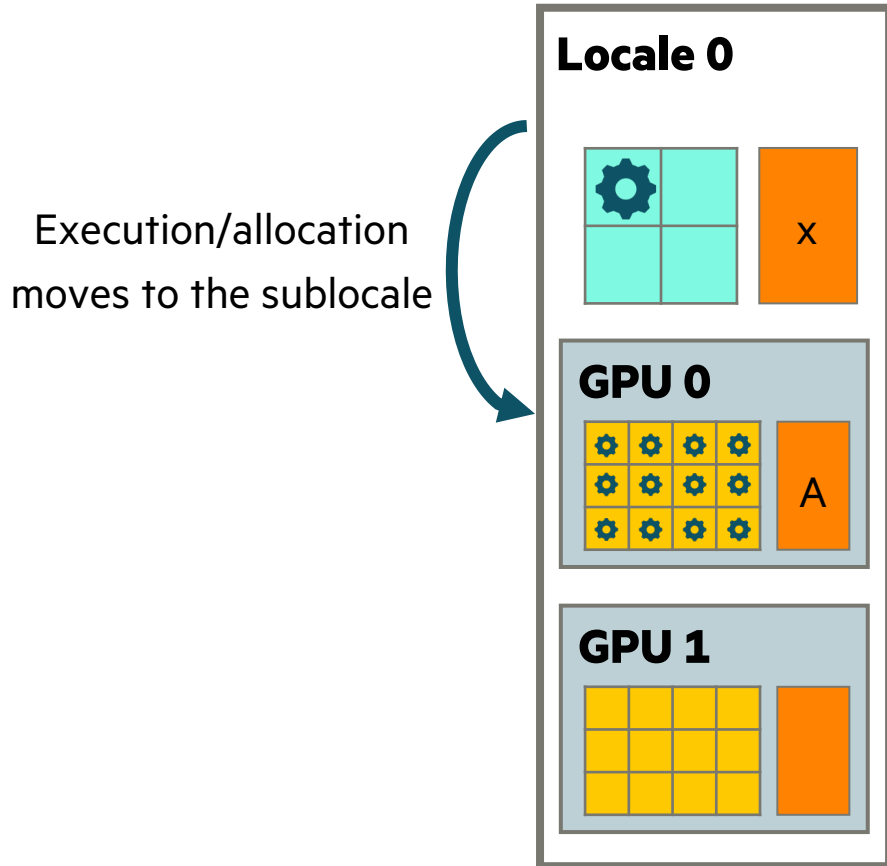
Complicating matters, compute nodes now often have GPUs with their own processors and memory

We represent these as *sub-locales* in Chapel



# PARALLELISM AND LOCALITY IN THE CONTEXT OF GPUS

 CPU Core    GPU Core    Memory



```
1 var x = 10;
```

```
2
```

```
3
```



```
4 on here.gpus[0] {
```

```
5     var A = [1, 2, 3, 4, 5, ...];
```

```
6     forall a in A do a += 1;
```

```
7 }
```

```
8
```



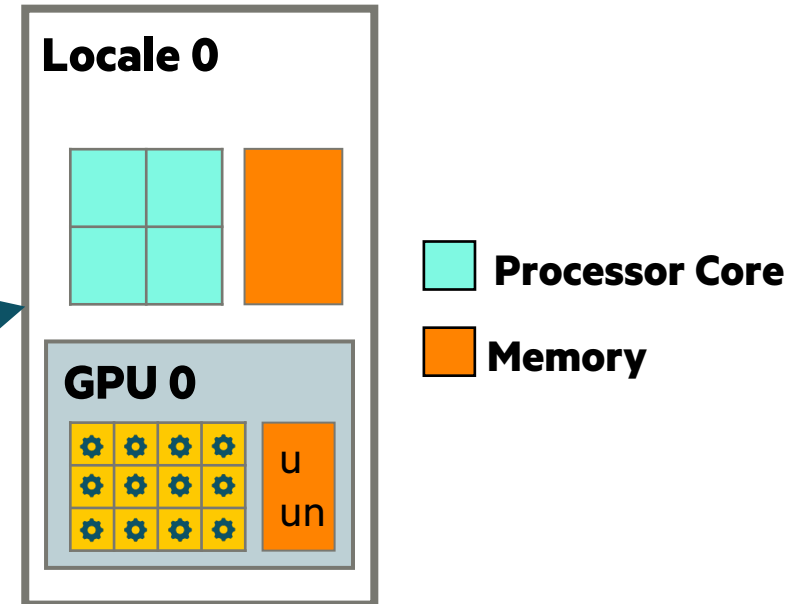
```
9 writeln(x);
```

# THE 1D HEAT DIFFUSION PROBLEM ON GPUS

Putting the problem into an 'on' clause for GPUs is enough to make it run on the GPU

Parallel, performs N GPU kernel launches

```
1  on here.gpus[0] {
2    const omega = {0.. $nx$ },
3        omegaHat = omega.expand(-1);
4    var u: [omega] real = 1.0;
5    u[ $nx/4..3*nx/4$ ] = 2.0;
6    var un = u;
7    for 1..N {
8      un <=> u;
9      forall i in omegaHat do
10         u[i] = un[i] + alpha *
11           (un[i-1] - 2*un[i] + un[i+1]);
12     }
13 }
```



# THE 1D HEAT DIFFUSION PROBLEM ON GPUS

```
1  on here.gpus[0] {
2      const omega = {0.. $\text{nx}$ },
3          omegaHat = omega.expand(-1);
4      var u: [omega] real = 1.0;
5      u[nx/4.. $3 \cdot \text{nx}/4$ ] = 2.0;
6      var un = u;
7      for 1..N {
8          un <=> u;
9          forall i in omegaHat do
10             u[i] = un[i] + alpha *
11                 (un[i-1] - 2*un[i] + un[i+1]);
12         }
13     }
```

- Now, we've used a sub-locale that represents a GPU
- However, we still only used a single locale at a time
- How can we make use of multiple locales to divide-and-conquer the heat diffusion problem?



# 1D HEAT EQUATION EXAMPLE: DOMAINS

```
★1  const omega = {0.. $\langle$ nx},
★2      omegaHat = omega.expand(-1);
3   var u: [omega] real = 1.0;
4   u[nx/4..3*nx/4] = 2.0;
5   var un = u;
6   for 1..N {
7     un <=> u;
8     for i in omegaHat do
9       u[i] = un[i] + alpha *
10          (un[i-1] - 2*un[i] + un[i+1]);
11 }
```

## Meaning

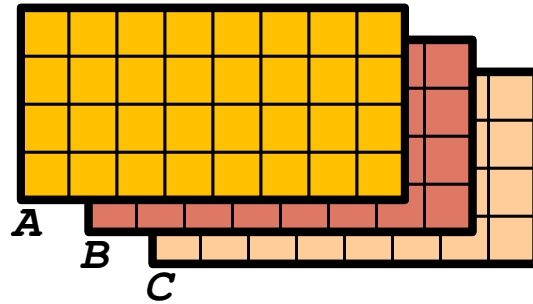
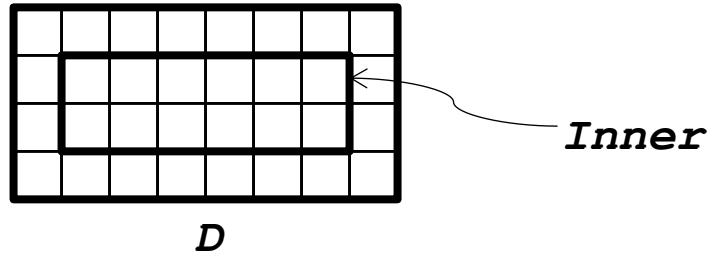
- Domains are first-class index sets
- They are the fundamental Chapel concept for data parallelism
- Useful for declaring arrays and computing with them

## Examples

```
const omega = {0.. $\langle$ nx};

const m = 4, n = 8;
const D = {1..m, 1..n};
const Inner = D.expand(-1);
```

# 1D HEAT EQUATION EXAMPLE: DOMAINS



## Meaning

- Domains are first-class index sets
- They are the fundamental Chapel concept for data parallelism
- Useful for declaring arrays and computing with them

## Examples

```
const omega = {0..<nx};  
  
const m = 4, n = 8;  
const D = {1..m, 1..n};  
const Inner = D.expand(-1);
```

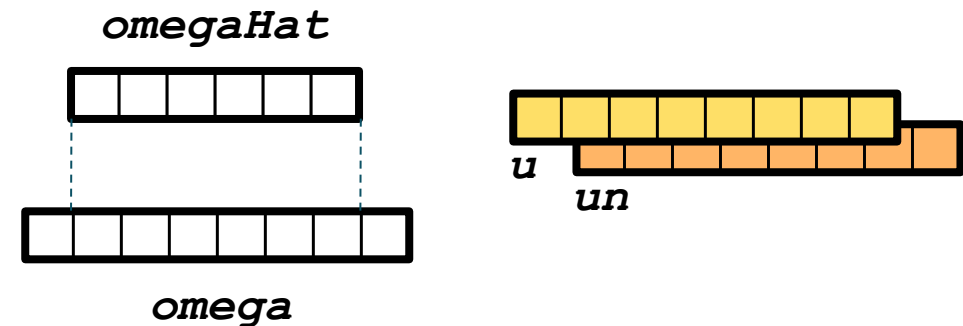
```
var A, B, C: [D] real;
```

# 1D HEAT EQUATION EXAMPLE: DOMAINS

```
★1  const omega = {0.. $\langle$ nx},
2      omegaHat = omega.expand(-1);
3  var u: [omega] real = 1.0;
4  u[nx/4.. $3$ *nx/4] = 2.0;
5  var un = u;
6  for 1.. $N$  {
7      un <=> u;
8      for i in omegaHat do
9          u[i] = un[i] + alpha *
10             (un[i-1] - 2*un[i] + un[i+1]);
11 }
```

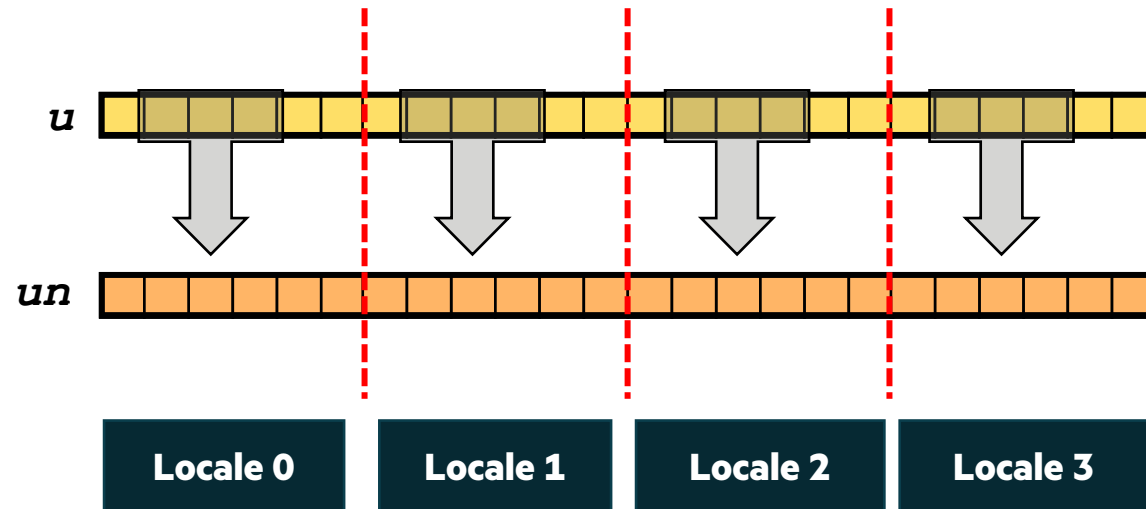
## Meaning

- 'omega' is the array size
- 'omegaHat' excludes the boundaries to avoid OOB access
- 'u' has domain 'omega', 'un' inherits it



# DISTRIBUTING THE 1D HEAT EQUATION

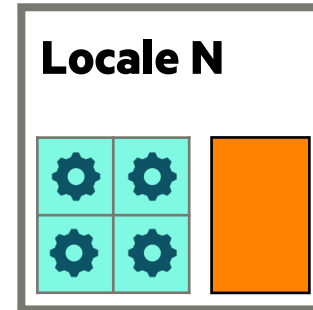
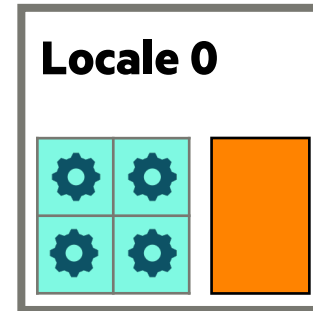
To use multiple locales, we could distribute  $u$  and  $un$  in chunks across multiple locales  
– taking advantage of their memory and compute resources




```
const omega = blockDist.createDomain({0..<nx});
```

# 1D HEAT EQUATION EXAMPLE: BLOCKDIST

```
1  const omega =
★2      blockDist.createDomain({0..<nx}),
3      omegaHat = omega.expand(-1);
4  var u: [omega] real = 1.0;
5  u[nx/4..3*nx/4] = 2.0;
6  var un = u;
7  for 1..N {
8      un <=> u;
9      forall i in omegaHat do
10         u[i] = un[i] + alpha *
11             (un[i-1] - 2*un[i] + un[i+1]);
12 }
```



 Processor Core  
 Memory

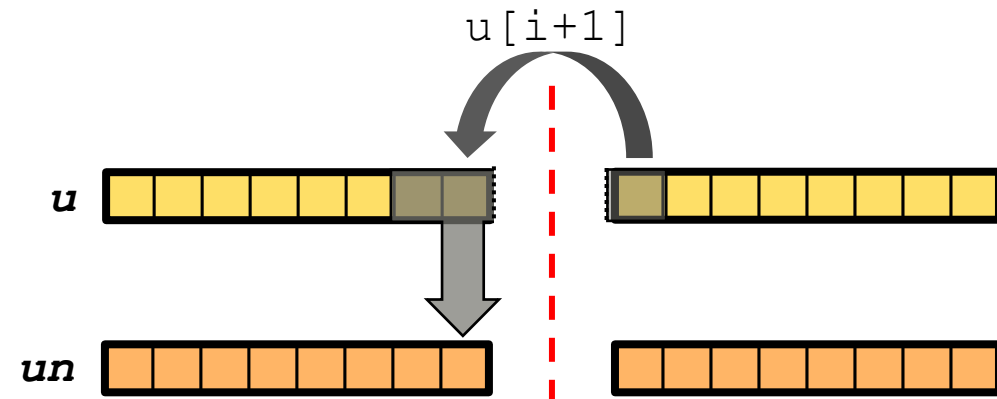


# 1D HEAT EQUATION EXAMPLE: BLOCKDIST

```
1  const omega =
★2      blockDist.createDomain({0..<nx}),
3      omegaHat = omega.expand(-1);
4  var u: [omega] real = 1.0;
5  u[nx/4..3*nx/4] = 2.0;
6  var un = u;
7  for 1..N {
8      un <=> u;
9      forall i in omegaHat do
10         u[i] = un[i] + alpha *
11             (un[i-1] - 2*un[i] + un[i+1]);
12 }
```

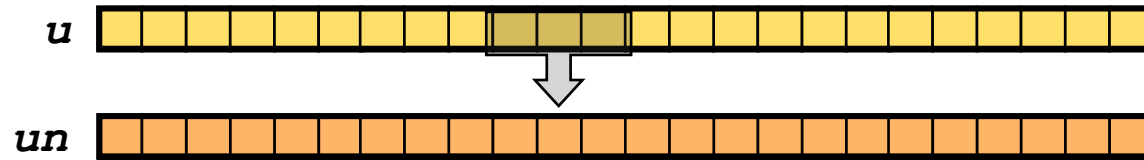
## Why does this work?

- 'omega' is block-distributed
- 'omegaHat' inherits 'omega's distribution
- Thus, 'u' is block-distributed
- 'un' inherits 'u's domain (and distribution)
- 'omegaHat' invokes 'blockDist's parallel/distr. iterator
  - the body of the loop is automatically split across multiple tasks on each locale
- Communication occurs automatically when a loop references a value stored on a remote locale

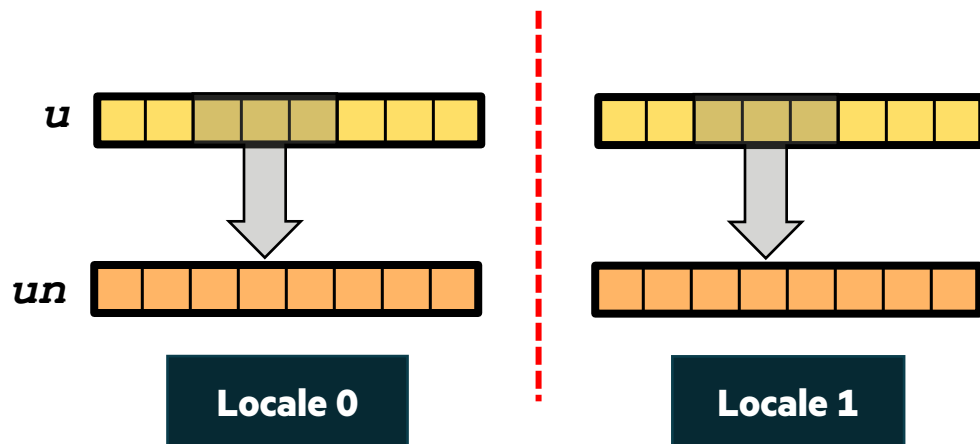


# MORE ON DISTRIBUTIONS

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



...to the target locales' memory and processors:



Domain Distributions specify...

- ...mapping of indices to locales
- ...layout of domains / arrays in memory
- ...parallel iteration strategies
- ...other core operations on arrays / domains

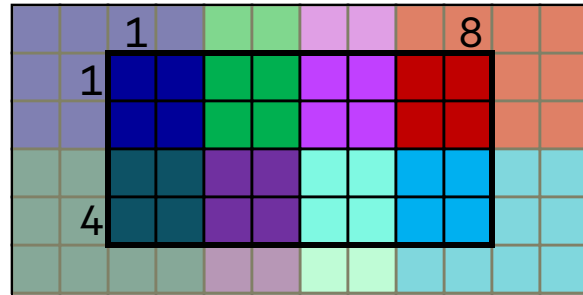
Locale 2

Locale 3



# SAMPLE DOMAIN DISTRIBUTIONS: BLOCK AND CYCLIC

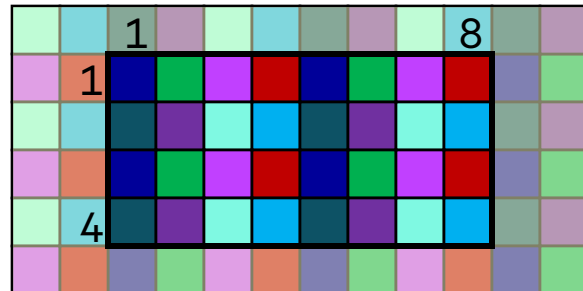
```
var Dom = blockDist.createDomain({1..4, 1..8});
```



*distributed to*



```
var Dom = cyclicDist.createDomain({1..4, 1..8});
```



*distributed to*





# 2D HEAT EQUATION EXAMPLE

## 2D and 3D stencil codes are more common and practical

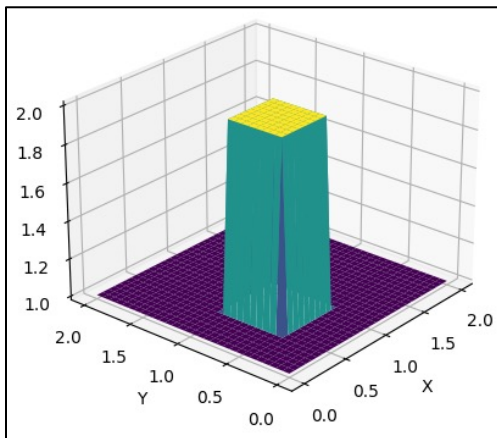
- They also present more interesting considerations for parallelization and distribution

### 2D heat / diffusion PDE:

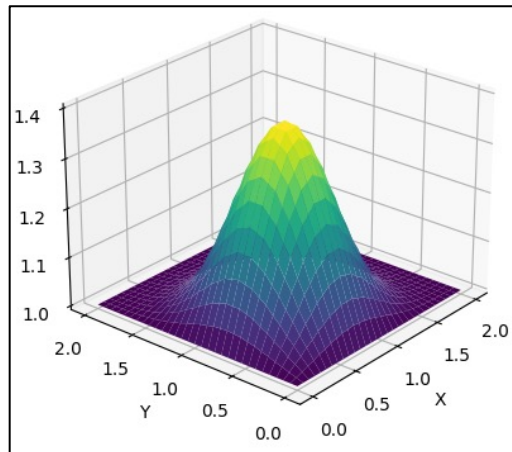
$$\frac{\partial u}{\partial t} = \alpha \Delta u = \alpha \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$

### Discretized (finite-difference) form:

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



$n = 0$



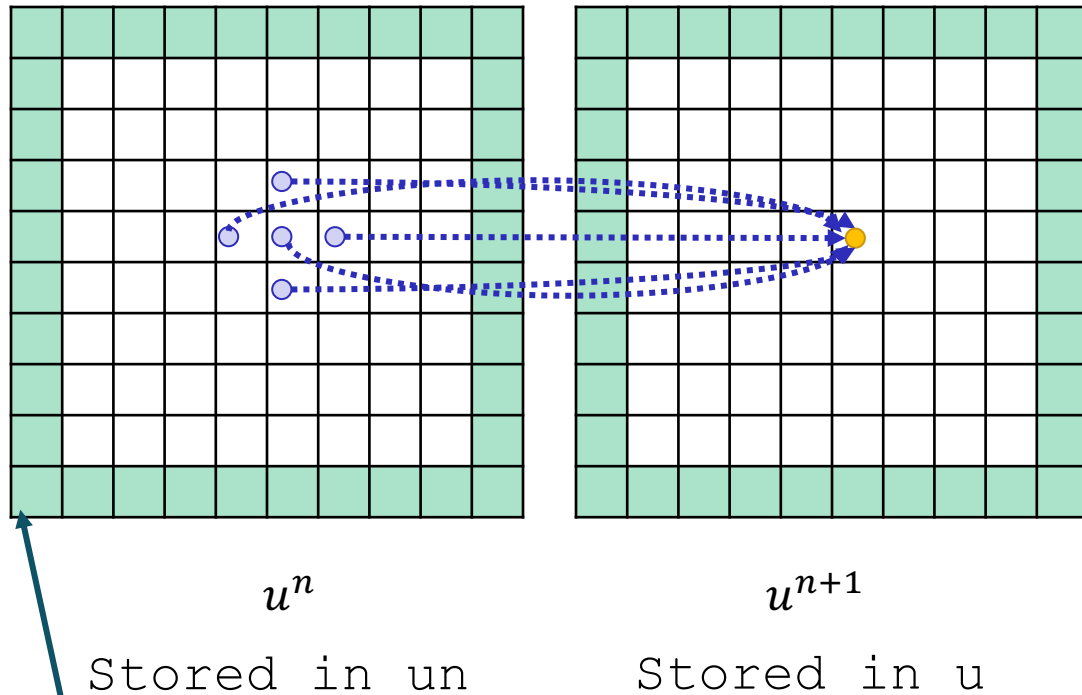
$n = N$

```

1  const omega = {0..

```

# PARALLEL 2D HEAT EQUATION



Fixed boundary values

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

```
7 ...
8   forall (i, j) in omegaHat do
9       u[i, j] = un[i, j] + alpha * (
10           un[i-1, j] + un[i, j-1] +
11           un[i+1, j] + un[i, j+1] -
12           4 * un[i, j]);
13 ...
```

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



# BLOCK DISTRIBUTED & PARALLEL 2D HEAT EQUATION

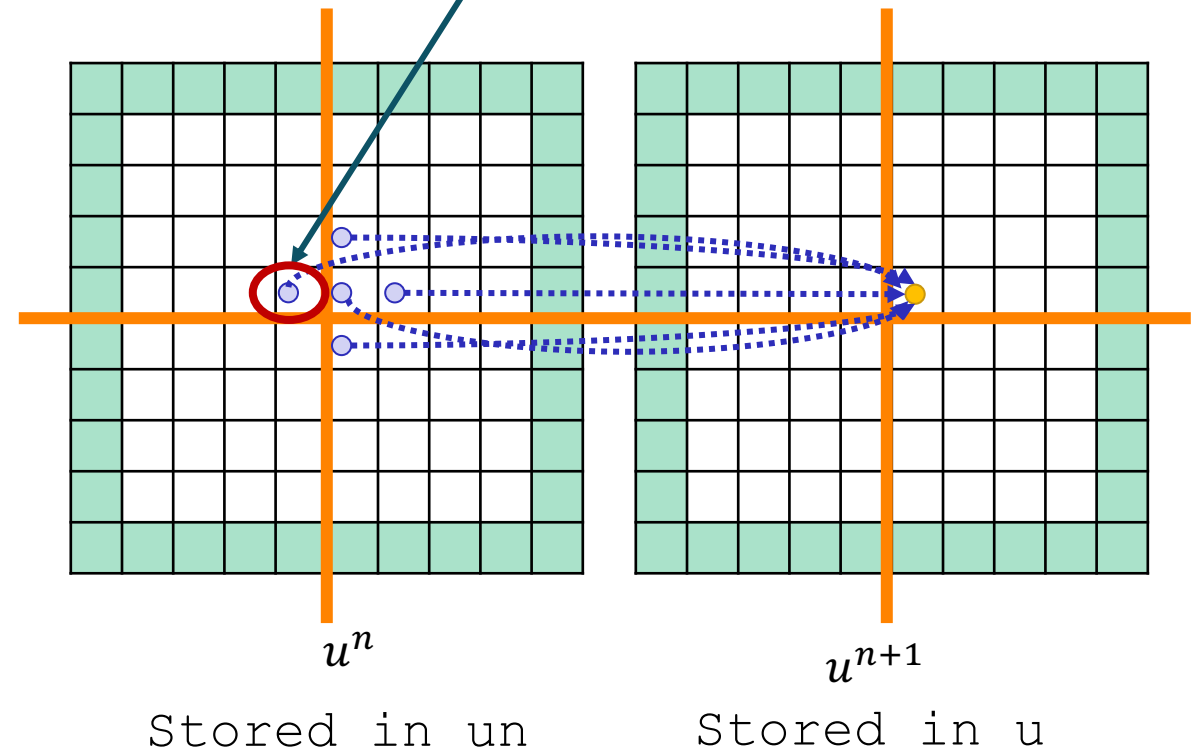
- Declaring distributed domains with the block distribution

```
const Omega = blockDist.createDomain(0.. $\langle$ nx, 0.. $\langle$ ny),  
      OmegaHat = Omega.expand(-1);
```

Array access across locale boundaries automatically invokes communication

- Distributed & Parallel loop over 'OmegaHat'

```
for 1.. $\langle$ 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]);  
}
```



# COMM DIAGNOSTICS

The 'CommDiagnostics' module provides functions for tracking comm between locales

- the following is a common pattern:

```
use CommDiagnostics;
...
startCommDiagnostics();
potentiallyCommHeavyOperation();
stopCommDiagnostics();
...
printCommDiagnosticsTable();
```

- which results in a table summarizing comm counts between the **start** and **stop** calls, e.g.,

locale	get	put	execute_on	execute_on_nb
-----:	--:	--:	-----:	-----:
0	10	0	6	12
1	105	5	0	0
2	105	4	0	0
3	105	7	0	0

- Compiling with '--no-cache-remote' before collecting comm diagnostics is recommended



# HANDS ON: HEAT 2D COMM DIAGNOSTICS RESULTS

- Gather comm diagnostics for 2D block dist.
  - 09-heat-2D-block.chpl

- *Compilation:*

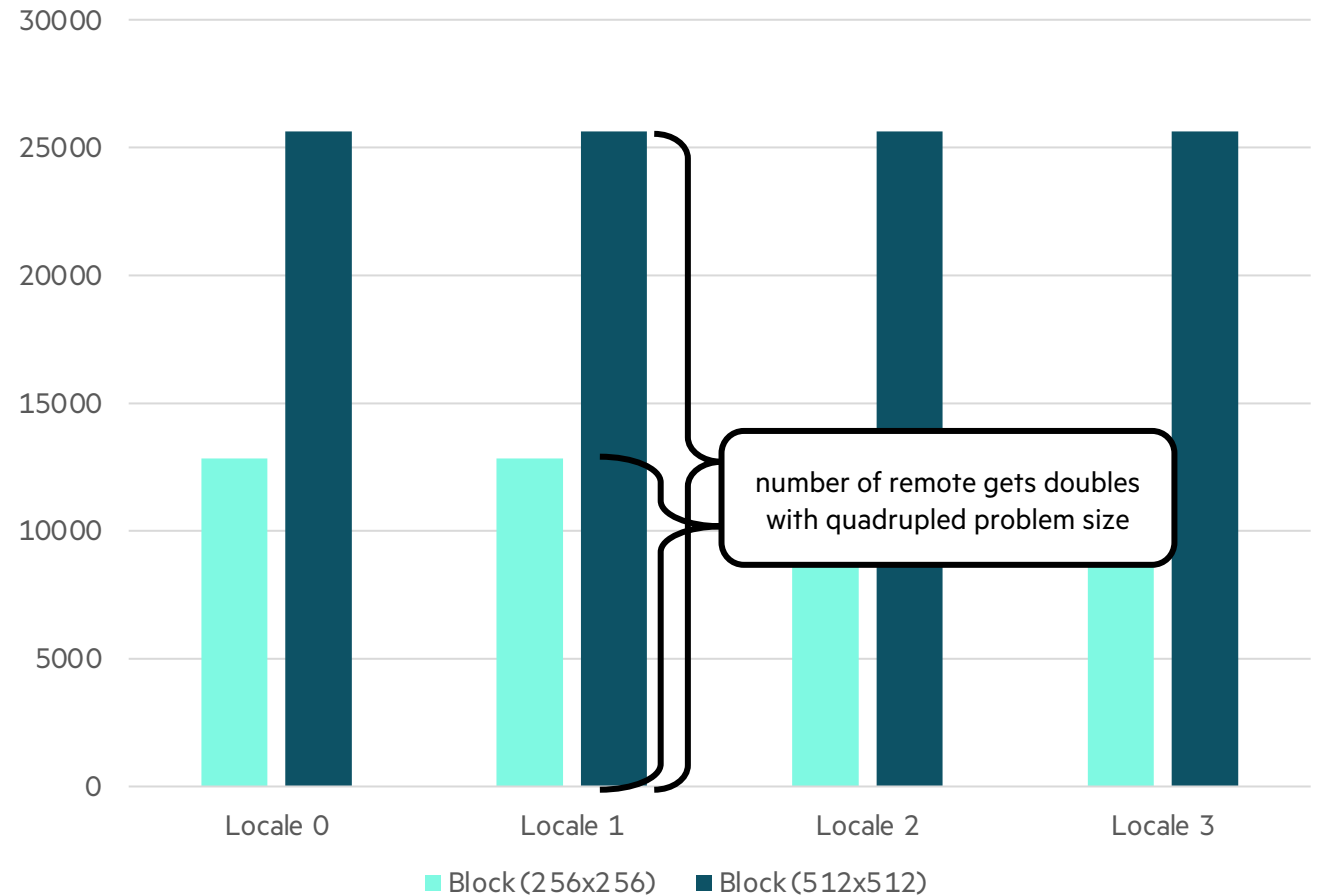
```
CHPL_COMM=gasnet
chpl 09-heat-2D-block.chpl --fast
--no-cache-remote
```

- *Execution:*

```
./09-heat-2D-block -n14 --N=100
--RunCommDiag=true --nx=256 --ny=256
./09-heat-2D-block -n14 --N=100
--RunCommDiag=true --nx=512 --ny=512
```

- **Block:** number of gets scales with size
  - But communication is slow!

Number of Gets on 4 Locales – Block vs. Stencil



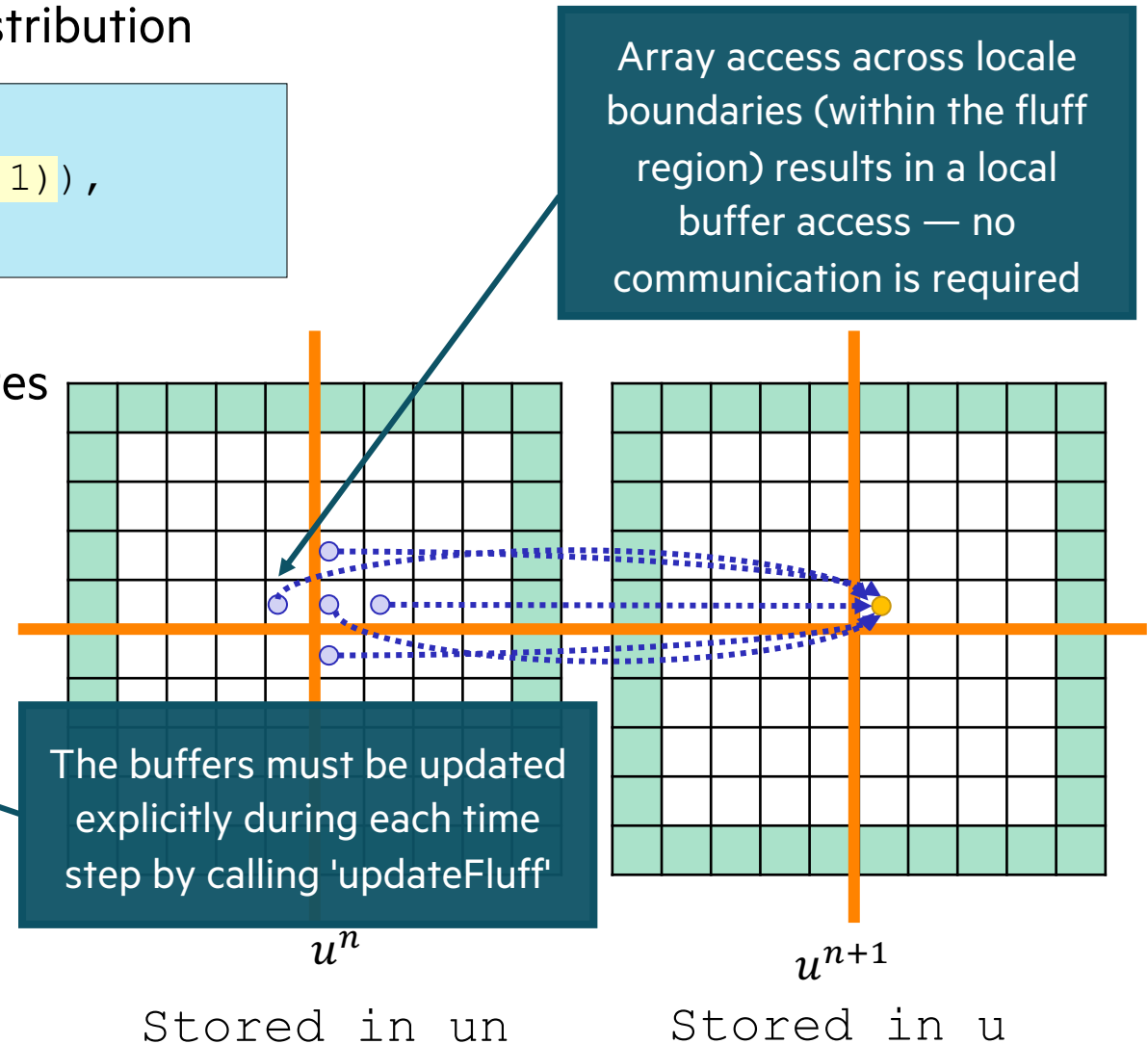
# STENCIL DISTRIBUTED & PARALLEL 2D HEAT EQUATION

- Declaring distributed domains with the stencil distribution

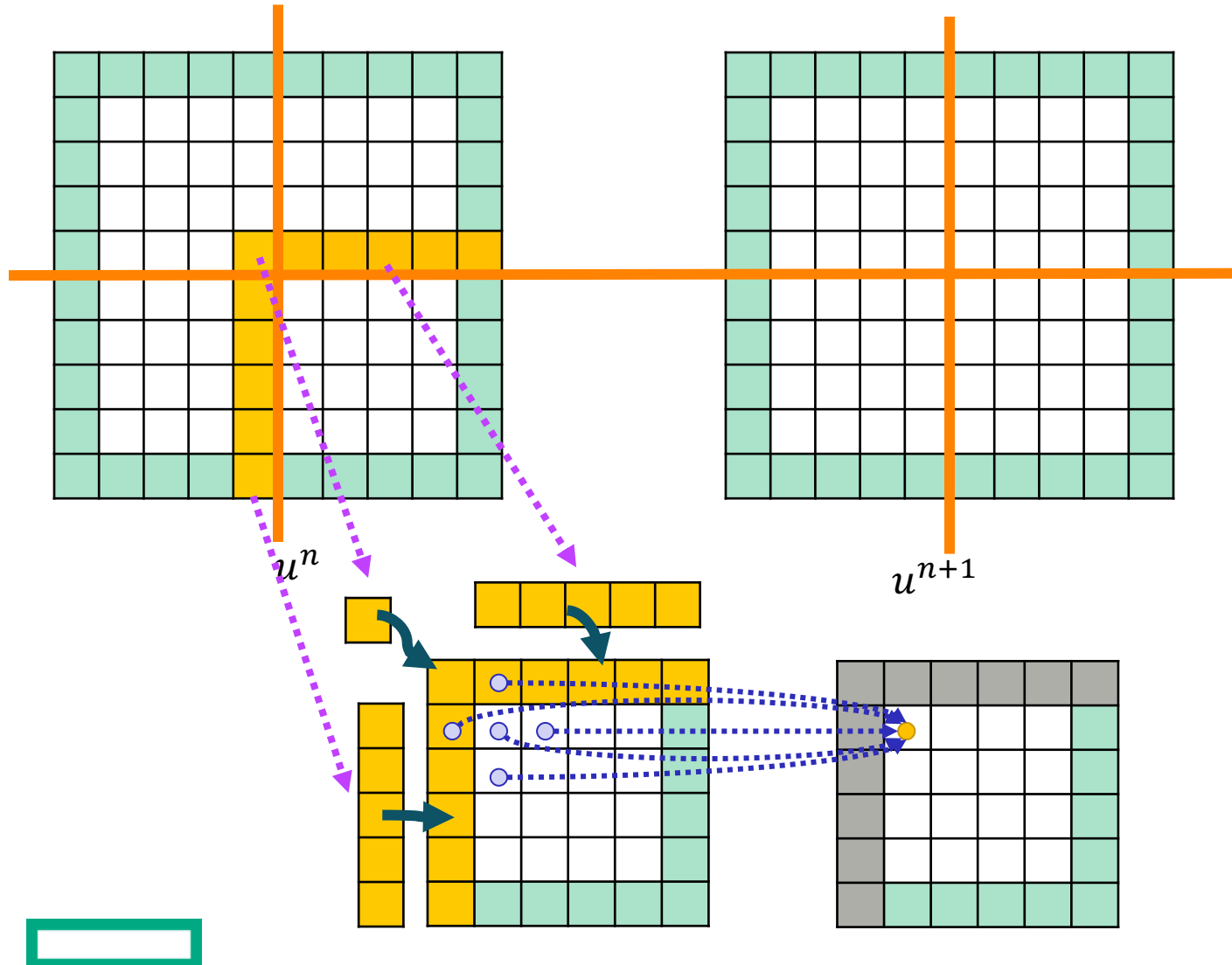
```
const Omega = stencilDist.createDomain(  
    {0..  
nx, 0..  
ny}, fluff=(1,1)),  
OmegaHat = Omega.expand(-1);
```

- 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]);  
}
```



# STENCIL DISTRIBUTED & PARALLEL 2D HEAT EQUATION



- 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

# HANDS ON: HEAT 2D COMM DIAGNOSTICS RESULTS

- Comparing comm diagnostics for:

- 09-heat-2D-block.chpl
- 10-heat-2D-stencil.chpl

- *Compilation:*

```
CHPL_COMM=gasnet
chpl 10-heat-2D-stencil.chpl --fast
--no-cache-remote
```

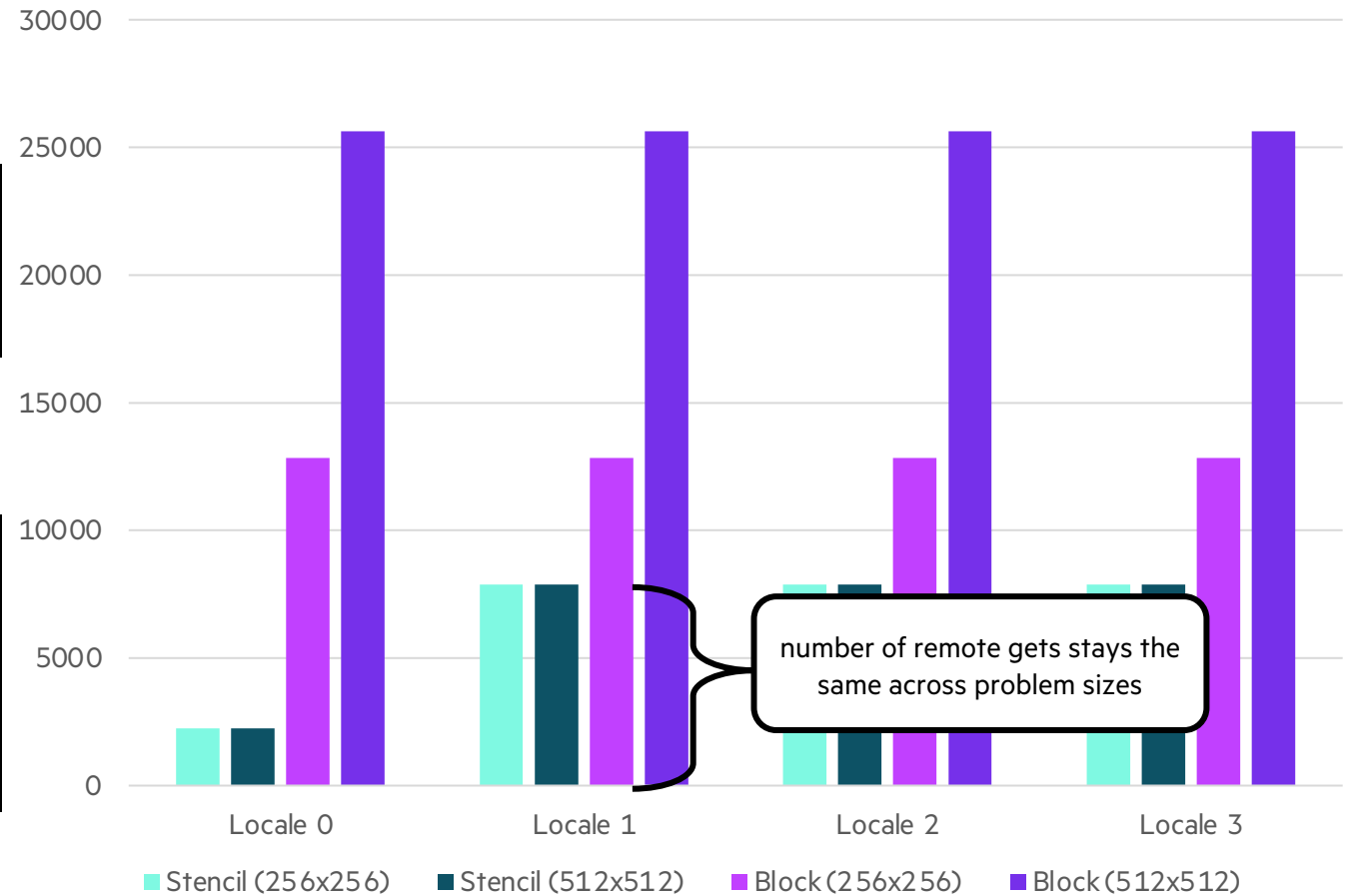
- *Execution:*

```
./09-heat-2D-block -n14 --N=100
--RunCommDiag=true --nx=512 --ny=512

./10-heat-2D-stencil -n14 --N=100
--RunCommDiag=true --nx=512 --ny=512
```

- **Block:** number of gets scales with size
- **Stencil:** static number of gets per iteration

Number of Gets on 4 Locales – Block vs. Stencil





# SUMMARY

---

We've used direct simulation of a heat diffusion problem to introduce parallel computing in Chapel

## Key ideas:

- Parallel computing is key to performance on modern hardware
- Chapel has powerful language features to make parallel computing more user-friendly
  - 'forall' supports easy expression of data parallelism and even distributed execution
  - 'on' supports moving execution to a different 'locale'
  - distributed domains and arrays make it easy to use storage across many locales (compute nodes)
- Chapel supports parallelism across a spectrum of hardware: laptops, GPUs, supercomputers

See <https://github.com/DanilaFe/chapelcon-2024-tutorial> for more resources

