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
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
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


[4] Computer Languages Benchmarks Game website and link to a post summarizing the Chapel results

[5] Performance Highlights page on the Chapel website


[7] Scaling results from the Arkouda Sort described in Chapel 2.0: Scalable and Productive Computing for All


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

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

```chpl
const omega = {0..<nx},
omegaHat = omega.expand(-1);
var u: [omega] real = 1.0;
u[nx/4..3*nx/4] = 2.0;
var un = u;
for 1..N {
    un <=> u;
    for i in omegaHat do
        u[i] = un[i] + alpha * (un[i-1] - 2*un[i] + un[i+1]);
}
```
Discretized (finite difference) equation: \( u_i^{n+1} = u_i^n + \alpha (u_{i-1}^n - 2u_i^n + u_{i+1}^n) \)

- where \( i \in \Omega \subseteq \mathbb{R}^1 \) are discrete points in space, and \((n, n + 1, \ldots)\) 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

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

```chapel
const omega = {0..<nx},
omegaHat = omega.expand(-1);
var u: [omega] real = 1.0;
u[nx/4..3*nx/4] = 2.0;
var un = u;
for 1..N {
  un <=> u;
  for i in omegaHat do
    u[i] = un[i] + alpha *
    (un[i-1] - 2*un[i] + un[i+1]);
}
```
Discretized (finite difference) equation: \( u^{n+1}_i = u^n_i + \alpha (u^n_{i-1} - 2u^n_i + u^n_{i+1}) \)
- where \( i \in \Omega \subset \mathbb{R}^1 \) are discrete points in space, and \((n, n+1, \ldots)\) 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

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

1D HEAT EQUATION EXAMPLE
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 \( i \in \Omega \subseteq \mathbb{R} \) are discrete points in space, and \((n, n + 1, ...)\) 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

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

```
const omega = {0..<nx},
omegaHat = omega.expand(-1);

var u: [omega] real = 1.0;
u[nx/4..3*nx/4] = 2.0;

var un = u;
for 1..N {
    un <=> u;
    for i in omegaHat do
        u[i] = un[i] + alpha * (un[i-1] - 2*un[i] + un[i+1]);
}
```

**Meaning**

- var/const: execution-time variable/constant
- param: compile-time constant
- No init-expr ⇒ initial value is the type's default
- No type ⇒ type is taken from init-expr
1D HEAT EQUATION EXAMPLE: RANGES

```chpl
const omega = {0..<nx},
omegaHat = omega.expand(-1);

var u: [omega] real = 1.0;
u[nx/4..3*nx/4] = 2.0;

var un = u;

for 1..N {
    un <=> u;
    for i in omegaHat do
        u[i] = un[i] + alpha * (un[i-1] - 2*un[i] + un[i+1]);
}
```

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

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

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

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?

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

Image Credit: https://commons.wikimedia.org/wiki/File:Pouring_Liquid_Gold.jpg
• 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

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

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

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’

\[
\text{forall } i \text{ in } \omegaHat \text{ do } \\
\text{ } u[i] = \text{un}[i] + \text{alpha} \ast (\text{un}[i-1] - 2*\text{un}[i] + \text{un}[i+1]);
\]

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

\[
\text{forall } i \text{ in } \omegaHat \text{ do } \\
\text{ } u[1] = \text{un}[i] + \text{alpha} \ast (\text{un}[i-1] - 2*\text{un}[i] + \text{un}[i+1]);
\]
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**.
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

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.

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;
}

Locale 0

Locale 1

Locale 2

Locale 3
INTRODUCTION TO LOCALES

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

This is a serial, local computation

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

Locale 0
By switching to a ‘forall’ loop, we made use of all available CPU cores.

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

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

**Locale 0**

- Processors
- Memory
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
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

Execution/allocation moves to the sublocale

Locale 0

GPU 0

GPU 1

var x = 10;

on here.gpus[0] {
    var A = [1, 2, 3, 4, 5, ...];
    forall a in A do a += 1;
}
writeln(x);
Putting the problem into an ‘on’ clause for GPUs is enough to make it run on the GPU

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

Parallel, performs N GPU kernel launches
THE 1D HEAT DIFFUSION PROBLEM ON GPUS

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?

```chpl
on here.gpus[0] {
    const omega = {0..<nx},
        omegaHat = omega.expand(-1);
    var u: [omega] real = 1.0;
    u[nx/4..3*nx/4] = 2.0;
    var un = u;
    for 1..N {
        un <=> u;
        forall i in omegaHat do
            u[i] = un[i] + alpha * (un[i-1] - 2*un[i] + un[i+1]);
    }
}
```
1D HEAT EQUATION EXAMPLE: DOMAINS

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

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
```chapel
const omega = {0..<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

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

var A, B, C: [D] real;
```
1D HEAT EQUATION EXAMPLE: DOMAINS

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

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.

```chpl
const omega = blockDist.createDomain({0..<nx});
```
1D HEAT EQUATION EXAMPLE: BLOCKDIST

```chapel
const omega =
    blockDist.createDomain({0..<nx}),
omegaHat = omega.expand(-1);
var u: [omega] real = 1.0;
u[nx/4..3*nx/4] = 2.0;
var un = u;
for 1..N {
    un <=< u;
    forall i in omegaHat do
        u[i] = un[i] + alpha * (un[i-1] - 2*un[i] + un[i+1]);
}
```
1D HEAT EQUATION EXAMPLE: BLOCKDIST

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
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
SAMPLE DOMAIN DISTRIBUTIONS: BLOCK AND CYCLIC

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

![Block Distribution Diagram]

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

![Cyclic Distribution Diagram]
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:

\[
u_{i,j}^{n+1} = u_{i,j}^n + \alpha \left( u_{i+1,j}^n + u_{i-1,j}^n - 4u_{i,j}^n + u_{i,j+1}^n + u_{i,j-1}^n \right)
\]
PARALLEL 2D HEAT EQUATION

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

\[
\begin{align*}
\forall (i, j) \in \omegaHat & \quad \text{do} \\
    u[i, j] &= \text{un}[i, j] + \alpha \times ( \\
    & \quad \text{un}[i-1, j] + \text{un}[i, j-1] + \\
    & \quad \text{un}[i+1, j] + \text{un}[i, j+1] - \\
    & \quad 4 \times \text{un}[i, j]) \\
\end{align*}
\]

\[
u_{i,j}^{n+1} = u_{i,j}^n + \alpha \left(u_{i-1,j}^n + u_{i,j-1}^n + u_{i+1,j}^n + u_{i,j+1}^n - 4u_{i,j}^n\right)
\]
BLOCK DISTRIBUTED & PARALLEL 2D HEAT EQUATION

• Declaring distributed domains with the block distribution

```cpp
const Omega = blockDist.createDomain(0..<nx, 0..<ny),
              OmegaHat = Omega.expand(-1);
```

• Distributed & Parallel loop over 'OmegaHat'

```cpp
for 1..nt {
    u <=> un;

    forall (i, j) in OmegaHat do
        u[i, j] = un[i, j] + alpha * (un[i-1, j] + un[i, j-1] +
                                        un[i+1, j] + un[i, j+1] -
                                        4 * un[i, j]);
}
```

Array access across locale boundaries automatically invokes communication.
The 'CommDiagnostics' module provides functions for tracking comm between locales

- the following is a common pattern:

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

- which results in a table summarizing comm counts between the `start` and `stop` calls, e.g.,
```
<table>
<thead>
<tr>
<th>locale</th>
<th>get</th>
<th>put</th>
<th>execute_on</th>
<th>execute_on_nb</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>------</td>
<td>--</td>
<td>--</td>
<td>----------</td>
<td>--------------</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>0</td>
<td>6</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>105</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>105</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>105</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```

- 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](chart)

- Number of remote gets doubles with quadrupled problem size.
Declaring distributed domains with the stencil distribution

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

Distributed & Parallel loop including buffer updates

```chpl
for l..nt {
    u <=> un;
    un.updateFluff();
    forall (i, j) in OmegaHat do
        u[i, j] = un[i, j] + alpha * (un[i-1, j] + un[i, j-1] +
                                    un[i+1, j] + un[i, j+1] -
                                    4 * un[i, j]);
}
```

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

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

Stored in un

Stored in u
• 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

number of remote gets stays the same across problem sizes
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