WHAT IS CHAPEL?

Chapel: A modern parallel programming language
- portable & scalable
- open-source & collaborative

Goals:
- Support general parallel programming
- Make parallel programming at scale far more productive
  - Python-like support for rapid prototyping
  - yet with the performance, scalability, portability of Fortran/C/C++, MPI, OpenMP, CUDA, ...
WHY CREATE A NEW LANGUAGE?

• Because parallel programmers deserve better
  • the state of the art for HPC programming is a mash-up of libraries, pragmas, and extensions
  • SPMD-based models are restrictive compared to having a global namespace and asynchrony
  • parallelism and locality are concerns that deserve first-class language features

[Image Source: Kathy Yelick’s (UC Berkeley, LBNL) CHIUW 2018 keynote: Why Languages Matter More Than Ever, used with permission]
Imagine having a programming language for parallel computing that was as...

...**programmable** as Python

...yet also as...

...**fast** as Fortran

...**scalable** as MPI

...**GPU-ready** as CUDA/OpenMP/OpenCL/OpenACC/...

...**portable** as C

...**fun** as [your favorite programming language]

**This is our motivation for Chapel**
OUTLINE

• Introductory Content
  • What is Chapel?
    • Chapel Characteristics
    • Chapel Benchmarks & Apps
    • Chapel vs. Standard Practice
• Further Details: Chapel Features
  • Base Language Features
  • Task-Parallelism & Locality
  • Data-Parallelism
• Wrap-up
CHAPEL CHARACTERISTICS
KEY CHARACTERISTICS OF CHAPEL

- compiled: to generate the best performance possible
- statically typed: to avoid simple errors after hours of execution
- interoperable: with C, Fortran, Python, ...
- portable: runs on laptops, clusters, the cloud, supercomputers
- open-source: to reduce barriers to adoption and leverage community contributions
WHAT DO CHAPEL PROGRAMS LOOK LIKE?

**helloTaskPar.chpl**: print a message from each core in the system

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

```bash
> chpl helloTaskPar.chpl
> ./helloTaskPar --numLocales=4
Hello from task 1 of 4 on n1032
Hello from task 4 of 4 on n1032
Hello from task 1 of 4 on n1034
Hello from task 2 of 4 on n1032
Hello from task 1 of 4 on n1033
Hello from task 3 of 4 on n1034
```

**fillArray.chpl**: declare and parallel-initialize a distributed array

```chapel
use CyclicDist;
config const n = 1000;
const 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);
```

```bash
> chpl fillArray.chpl
> ./fillArray --n=5 --numLocales=4
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
```
**CHAPEL RELEASES**

**Q: What is provided in a Chapel release?**

**A:** Chapel releases contain...

...**the Chapel compiler** (`chpl`): translates Chapel source code into optimized executables

...**runtime libraries:** help map Chapel programs to a system’s capabilities (e.g., processors, network, memory, …)

...**library modules:** provide standard algorithms, data types, capabilities, ...

...**documentation:** also available online at: [https://chapel-lang.org/docs/](https://chapel-lang.org/docs/)

...**sample programs:** primers, benchmarks, etc.

**Q: How often is Chapel released? When is the next one?**

**A:** Chapel is released every 3 months

- version 1.27.0 was released June 30, 2022
- version 1.28.0 is scheduled for September 17, 2022
Our team consists of:
- 19 full-time employees
- 3 summer interns
- our director

We also have:
- a visiting scholar joining soon
- an open position

CHAPEL BENCHMARKS AND APPLICATIONS
FOR DESKTOP BENCHMARKS, CHAPEL IS COMPACT AND FAST

[plot generated by summarizing data from https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html as of May 10, 2022]
FOR DESKTOP BENCHMARKS, CHAPEL IS COMPACT AND FAST (ZOOMED)

[plot generated by summarizing data from https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html as of May 10, 2022]
FOR HPC BENCHMARKS, CHAPEL TENDS TO BE CONCISE, CLEAR, AND COMPETITIVE

**STREAM TRIAD: C + MPI + OPENMP**

```chapel
use BlockDist;
config const m = 1000, alpha = 3.0;
const Dom = {1..m} dmapped ...;
var A, B, C: [Dom] real;
B = 2.0;
C = 1.0;
A = B + alpha * C;
```

**HPCC RA: MPI KERNEL**

```chapel
forall (_, r) in zip(Updates, RAStream()) do
T[r & indexMask].xor(r);
```
**FLAGSHIP CHAPEL APPLICATIONS**

**CHAMPS: 3D Unstructured CFD**
Éric Laurendeau, Simon Bourgault-Côté, Matthieu Parenteau, et al.
École Polytechnique Montréal

**Arkouda: NumPy at Massive Scale**
Mike Merrill, Bill Reus, et al.
US DoD

**CrayAI: Distributed Machine Learning**
Hewlett Packard Enterprise

**ChplUltra: Simulating Ultralight Dark Matter**
Nikhil Padmanabhan, J. Luna Zagorac, et al.
Yale University / University of Auckland

**ChOp: Chapel-based Optimization**
Tiago Carneiro, Nouredine Melab, et al.
INRIA Lille, France

Your application here?

(images provided by their respective teams and used with permission)
CHAMPS SUMMARY

What is it?
- 3D unstructured CFD framework for airplane simulation
- ~120k lines of Chapel written from scratch in ~3 years

Who wrote it?
- Professor Éric Laurendeau’s students + postdocs at Polytechnique Montreal

Why Chapel?
- performance and scalability competitive with MPI + C++
- students found it far more productive to use
HPC Lessons From 30 Years of Practice in CFD Towards Aircraft Design and Analysis

LAB HISTORY AT POLYTECHNIQUE

- **NSCODE** (2012 - early 2020):
  - Shared memory 2D/2.5D structured multi-physics solver written in C/Python
  - ~800 C/header files: ~120k lines of code
  - Run by Python interface using f2py (f90 APIs)
  - Difficult to maintain at the end or even to merge new developments

- **(U)VLM** (2012 - now):
  - ~5-6 versions in different languages (Matlab, Fortran, C++, Python, Chapel)
  - The latest version in Chapel is integrated in CHAMPS

- **EULER2D** (early 2019):
  - Copy in Chapel of a small version of NSCODE as benchmark between C and Chapel that illustrated the Chapel language potential
  - ~10 Chapel files: ~1750 lines of code

- **CHAMPS** (mid 2019 - now):
  - Distributed memory 3D/2D unstructured multi-physics solver written in Chapel
  - ~120 Chapel files: ~48k lines of code

https://youtu.be/wD-a_KyB8al?t=1904

(images provided by the CHAMPS team and used with permission)
HPC Lessons From 30 Years of Practice in CFD Towards Aircraft Design and Analysis (June 4, 2021)

“To show you what Chapel did in our lab... [our previous framework] ended up 120k lines. And my students said, ‘We can't handle it anymore. It’s too complex, we lost track of everything.’ And today, they went from 120k lines to 48k lines, so 3x less.

But the code is not 2D, it’s 3D. And it’s not structured, it’s unstructured, which is way more complex. And it’s multi-physics... So, I've got industrial-type code in 48k lines.”

“[Chapel] promotes the programming efficiency ... We ask students at the master’s degree to do stuff that would take 2 years and they do it in 3 months. So, if you want to take a summer internship and you say, ‘program a new turbulence model,’ well they manage. And before, it was impossible to do.”

“So, for me, this is like the proof of the benefit of Chapel, plus the smiles I have on my students everyday in the lab because they love Chapel as well. So that’s the key, that’s the takeaway.”

• Talk available online: https://youtu.be/wD-a_KyB8al?t=1904 (hyperlink jumps to the section quoted here)
**RECENT CHAMPS HIGHLIGHTS**

- **CHAMPS 2.0** was released this year
  - added many new capabilities and improvements
  - grew from ~48k to ~120k lines
- Team gave 5–6 talks at **2022 AIAA AVIATION** in June
- While on sabbatical this year, Éric presented at ONERA, DLR, U. de Strasbourg, T. U. Braunschweig
- Participated in the **4th AIAA High-lift Prediction Workshop** and **1st AIAA Ice Prediction Workshop**
  - Generating comparable results to high-profile sites: Boeing, Lockheed Martin, NASA, JAXA, Georgia Tech, ...

(images taken from Éric Laurendeau’s SIAM PP22 talk, *A Case Study on the Impact of Chapel within an Academic Computational Aerodynamic Laboratory*, with permission)
CURRENT FLAGSHIP CHAPEL APPLICATIONS

CHAMPS: 3D Unstructured CFD
Éric Laurendeau, Simon Bourgault-Côté, Matthieu Parenteau, et al.
École Polytechnique Montréal

Arkouda: NumPy at Massive Scale
Mike Merrill, Bill Reus, et al.
US DoD

CrayAI: Distributed Machine Learning
Hewlett Packard Enterprise

ChplUltra: Simulating Ultralight Dark Matter
Nikhil Padmanabhan, J. Luna Zagorac, et al.
Yale University / University of Auckland

ChOp: Chapel-based Optimization
Tiago Carneiro, Nouredine Melab, et al.
INRIA Lille, France

Your application here?

(images provided by their respective teams and used with permission)
**Motivation:** Say you’ve got...

...HPC-scale data science problems to solve

...a bunch of Python programmers

...access to HPC systems

How will you leverage your Python programmers to get your work done?
ARKOUDA’S HIGH-LEVEL APPROACH

Arkouda Client
(written in Python)

Arkouda Server
(written in Chapel)

User writes Python code in Jupyter, making familiar NumPy/Pandas calls
ARKOUDA SUMMARY

What is it?
• A Python library supporting a key subset of NumPy and Pandas for Data Science
  – Uses a Python-client/Chapel-server model to get scalability and performance
  – Computes massive-scale results (multi-TB-scale arrays) within the human thought loop (seconds to a few minutes)
• ~22k lines of Chapel, largely written in 2019, continually improved since then

Who wrote it?
• Mike Merrill, Bill Reus, et al., US DoD
• Open-source: https://github.com/Bears-R-Us/arkouda

Why Chapel?
• high-level language with performance and scalability
• close to Pythonic
  – enabled writing Arkouda rapidly
  – doesn’t repel Python users who look under the hood
• ports from laptop to supercomputer
## Arkouda Performance Compared to NumPy

<table>
<thead>
<tr>
<th>benchmark</th>
<th>NumPy 0.75 GB</th>
<th>Arkouda (serial) 0.75 GB 1 core, 1 node</th>
<th>Arkouda (parallel) 0.75 GB 36 cores x 1 node</th>
<th>Arkouda (distributed) 384 GB 36 cores x 512 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>argsort</td>
<td>0.03 GiB/s</td>
<td>0.05 GiB/s 1.66x</td>
<td>0.50 GiB/s 16.7x</td>
<td>55.12 GiB/s 1837.3x</td>
</tr>
<tr>
<td>coargsort</td>
<td>0.03 GiB/s</td>
<td>0.07 GiB/s 2.3x</td>
<td>0.50 GiB/s 16.7x</td>
<td>29.54 GiB/s 984.7x</td>
</tr>
<tr>
<td>gather</td>
<td>1.15 GiB/s</td>
<td>0.45 GiB/s 0.4x</td>
<td>13.45 GiB/s 11.7x</td>
<td>539.52 GiB/s 469.1x</td>
</tr>
<tr>
<td>reduce</td>
<td>9.90 GiB/s</td>
<td>11.66 GiB/s 1.2x</td>
<td>118.57 GiB/s 12.0x</td>
<td>43683.00 GiB/s 4412.4x</td>
</tr>
<tr>
<td>scan</td>
<td>2.78 GiB/s</td>
<td>2.12 GiB/s 0.8x</td>
<td>8.90 GiB/s 3.2x</td>
<td>741.14 GiB/s 266.6x</td>
</tr>
<tr>
<td>scatter</td>
<td>1.17 GiB/s</td>
<td>1.12 GiB/s 1.0x</td>
<td>13.77 GiB/s 11.8x</td>
<td>914.67 GiB/s 781.8x</td>
</tr>
<tr>
<td>stream</td>
<td>3.94 GiB/s</td>
<td>2.92 GiB/s 0.7x</td>
<td>24.58 GiB/s 6.2x</td>
<td>6266.22 GiB/s 1590.4x</td>
</tr>
</tbody>
</table>
ARKOUDA ARGSORT AT MASSIVE SCALE

- Ran on a large Apollo system, summer 2021
  - 73,728 cores of AMD Rome
  - 72 TiB of 8-byte values
  - 480 GiB/s (2.5 minutes elapsed time)
  - ~100 lines of Chapel code

Close to world-record performance—quite likely a record for performance/SLOC
OUTLINE/TIME CHECK

• Introductory Content
  • What is Chapel?
  • Chapel Characteristics
  • Chapel Benchmarks & Apps
  • Chapel vs. Standard Practice
• Further Details: Chapel Features
  • Base Language Features
  • Task-Parallelism & Locality
  • Data-Parallelism
• Wrap-up
CHAPEL VS. STANDARD PRACTICE:
PARALLELISM + LOCALITY,
SPMD VS. GLOBAL-VIEW
KEY CONCERNS FOR SCALABLE PARALLEL COMPUTING

1. **parallelism**: What tasks should run simultaneously?
2. **locality**: Where should tasks run? Where should data be allocated?
STREAM TRIAD: A TRIVIAL CASE OF PARALLELISM + LOCALITY

**Given:** $m$-element vectors $A, B, C$

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

**In pictures:**

![Diagram showing the computation of $A_i = B_i + \alpha \cdot C_i$.]
STREAM TRIAD: A TRIVIAL CASE OF PARALLELISM + LOCALITY

**Given:** \( m \)-element vectors \( A, B, C \)

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

**In pictures, in parallel (shared memory / multicore):**

[Diagram of A, B, C vectors with parallel operations and \( \alpha \) value]
**Given:** \( m \)-element vectors \( A, B, C \)

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

**In pictures, in parallel (distributed memory):**
STREAM TRIAD: A TRIVIAL CASE OF PARALLELISM + LOCALITY

**Given:** $m$-element vectors $A$, $B$, $C$

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

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

![Diagram showing vector addition and scalar multiplication in parallel]
Many Disparate Notations for Expressing Parallelism + Locality

STREAM TRIAD IN CONVENTIONAL HPC PROGRAMMING MODELS

```c
#include <hpcc.h>

int HPCC_Stream(HPCC_Params *params, int doIO) {
    register int j;
    double scalar;

    VectorSize = HPCC_LocalVectorSize( params, 3,
                                       sizeof(double), 0 );
    a = HPCC_XMALLOC( double, VectorSize );
    b = HPCC_XMALLOC( double, VectorSize );
    c = HPCC_XMALLOC( double, VectorSize );

    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Comm_size( comm, &commSize );
    MPI_Comm_rank( comm, &myRank );
    rv = HPCC_Stream( params, 0 == myRank );
    MPI_Cancel( &rv );

    /* HPCC_Stream + doIO */
    for (j=0; j<VectorSize; j++) {
        a[j] = b[j]+scalar*c[j];
    }
    HPCC_free( c );
    HPCC_free( b );
    HPCC_free( a );
    return 0;
}
```
Many Disparate Notations for Expressing Parallelism + Locality

STREAM TRIAD IN CONVENTIONAL HPC PROGRAMMING MODELS

Note: This is a trivial parallel computation—imagine the additional complexity for something more realistic...

Challenge: Can we do better?
use BlockDist;
config const m = 1000,
        alpha = 3.0;
const Dom = {1..m} dmapped ...;
var A, B, C: [Dom] real;
B = 2.0;
C = 1.0;
A = B + alpha * C;

forall (_, r) in zip(Updates, RAStream()) do
    T[r & indexMask].xor(r);

CHAPEL SUPPORTS GLOBAL-VIEW / POST-SPMD PROGRAMMING

- “Apply a 3-point stencil to a vector”

\[
\begin{align*}
& (\text{Global-View}) \\
& + \text{ ( } \frac{3}{2} \text{ )/2} \\
& = \text{ ( ( } \frac{3}{2} \text{ )/2} \text{ )}
\end{align*}
\]
CHAPEL SUPPORTS GLOBAL-VIEW / POST-SPMD PROGRAMMING

• “Apply a 3-point stencil to a vector”

Global-View

\[
\frac{1}{2} \left( \begin{array}{c} \text{ภ} \\ \text{ภ} \\ \text{ภ} \end{array} \right) + \left( \begin{array}{c} \text{ภ} \\ \text{ภ} \\ \text{ภ} \end{array} \right) + \left( \begin{array}{c} \text{ภ} \\ \text{ภ} \\ \text{ภ} \end{array} \right)
\]

= \left( \begin{array}{c} \text{ภ} \\ \text{ภ} \\ \text{ภ} \end{array} \right)

SPMD

\[
\frac{1}{2} \left( \begin{array}{c} \text{ภ} \\ \text{ภ} \\ \text{ภ} \end{array} \right) + \left( \begin{array}{c} \text{ภ} \\ \text{ภ} \\ \text{ภ} \end{array} \right) + \left( \begin{array}{c} \text{ภ} \\ \text{ภ} \\ \text{ภ} \end{array} \right)
\]

= \left( \begin{array}{c} \text{ภ} \\ \text{ภ} \\ \text{ภ} \end{array} \right) 

= \left( \begin{array}{c} \text{ภ} \\ \text{ภ} \\ \text{ภ} \end{array} \right) 

= \left( \begin{array}{c} \text{ภ} \\ \text{ภ} \\ \text{ภ} \end{array} \right)
CHAPEL SUPPORTS GLOBAL-VIEW / POST-SPMD PROGRAMMING

“Apply a 3-point stencil to a vector”

**Global-View Chapel code**

```chapel
proc main() {
    var n = 1000;
    const D = {1..n} dmapped ...;
    var A, B: [D] real;
    forall i in D[2..n-1] do
        B[i] = (A[i-1] + A[i+1])/2;
}
```

**SPMD pseudocode (MPI-esque)**

```chapel
proc main() {
    var n = 1000;
    var p = numProcs(),
        me = myProc(),
        myN = n/p,
        myLo = 1,
        myHi = myN;
    var A, B: [0..myN+1] real;
    if (me < p-1) {
        send(me+1, A[myN]);
        recv(me+1, A[myN+1]);
    } else
        myHi = myN-1;
    if (me > 0) {
        send(me-1, A[1]);
        recv(me-1, A[0]);
    } else
        myLo = 2;
    forall i in myLo..myHi do
        B[i] = (A[i-1] + A[i+1])/2;
}
```
TWO QUICK SIDEBARS TO ROUND OUT THIS SECTION

1. Doing SPMD programming in Chapel
2. Illustrating Chapel’s global namespace
• Being a general-purpose language, Chapel doesn’t preclude you from writing SPMD patterns in Chapel:

```chapel
coforall loc in Locales do
    on loc do
        myMain();

proc myMain() {
    // ... write your SPMD computation here ...
}
```

SIDEBAR 1: CHAPEL SUPPORTS SPMD PROGRAMMING AS WELL
Note 1: Variables are allocated on the locale where the task is running
Note 2: Tasks can refer to visible variables, whether local or remote.

verbatim
config const verbose = false;
var total = 0,
    done = false;
…
on Locales[1] {
    if !done {
        if verbose then
            writeln("Adding locale 1’s contribution");
            total += computeMyContribution();
    }
}

SIDEBAR 2: CHAPEL’S GLOBAL NAMESPACE

Code runs on locale 1, but refers to values stored on locale 0.
OUTLINE/TIME CHECK

• Introductory Content
  • What is Chapel?
  • Chapel Characteristics
  • Chapel Benchmarks & Apps
  • Chapel vs. Standard Practice

• Further Details: Chapel Features
  • Base Language Features
  • Task-Parallelism & Locality
  • Data-Parallelism

• Wrap-up
FURTHER DETAILS:
OVERVIEW OF CHAPEL FEATURES
CHAPEL FEATURE AREAS

Chapel language concepts

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target System
BASE LANGUAGE

Chapel language concepts

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target System

“Lower-level” Chapel
Our first program shows a stylized way of computing \( n \) values of the Fibonacci sequence in Chapel...

- This is admittedly an artificial example, but you might imagine replacing it with the code required to...
  - traverse your data structure
  - iterate in a tiled manner over your array
  - or any other iteration pattern that you’d like to parameterize, reuse, or abstract away from your primary computations

The Fibonacci Sequence:

- First two items:
  - 0
  - 1

- Successive terms found by adding the previous two terms
  - 1 \((0 + 1)\)
  - 2 \((1 + 1)\)
  - 3 \((1 + 2)\)
  - 5 \((2 + 3)\)
  - 8 \((3 + 5)\)
  - ...

A TOY COMPUTATION: THE FIBONACCI SEQUENCE
FIBONACCI ITERATION

```chapl
config const n = 10;

for f in fib(n) do
    writeln(f);

iter fib(x) {
    var current = 0,
    next = 1;

    for i in 1..x {
        yield current;
        current += next;
        current <=> next;
    }
}
```

```
prompt> chpl fib.chpl
prompt>
```

prompt> ./fib
0
1
1
2
3
5
8
13
21
34
48
FIBONACCI ITERATION

```chapel
config const n = 10;

for f in fib(n) do writeln(f);

iter fib(x) {
    var current = 0, next = 1;
    for i in 1..x {
        yield current;
        current += next;
        current <=> next;
    }
}
```

Drive this loop by invoking `fib(n)`

prompt> chpl fib.chpl
prompt> ./fib

```plaintext
0
1
1
2
3
5
8
13
21
34
```
Fibonacci Iteration

```chapl
config const n = 10;

for f in fib(n) do
  writeln(f);

iter fib(x) {
  var current = 0,
    next = 1;

  for i in 1..x {
    yield current;
    current += next;
    current <=> next;
  }
}
```

Execute the loop’s body for that value

‘yield’ this expression back to the loop’s index variable

```
prompt> chpl fib.chpl
prompt> ./fib
0
1
1
2
3
5
8
13
21
34
50
```
FIBONACCI ITERATION

```chpl
config const n = 10;

for f in fib(n) do
  writeln(f);

iter fib(x) {
  var current = 0,
  next = 1;

  for i in 1..x {
    yield current;
    current += next;
    current <=> next;
  }
}
```

prompt> chpl fib.chpl
prompt> ./fib
0
1
1
2
3
5
8
13
21
34
FIBONACCI ITERATION

```chpl
config const n = 10;

for f in fib(n) do writeln(f);

iter fib(x) {
    var current = 0,
        next = 1;

    for i in 1..x {
        yield current;
        current += next;
        current <=> next;
    }
}
```

Config[urable] declarations support command-line overrides

```
prompt> chpl fib.chpl
prompt> ./fib --n=1000
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
```

FIBONACCI ITERATION

```chpl
config const n = 10;

for f in fib(n) do
    writeln(f);

iter fib(x) {
    var current = 0,
        next = 1;
    for i in 1..x {
        yield current;
        current += next;
        current <=> next;
    }
}
```

Static type inference for:
- constants / variables
- arguments
- return types

Explicit typing also supported

prompt> chpl fib.chpl
prompt> ./fib --n=1000
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
...
FIBONACCI ITERATION

```chapel
fib.chpl

config const n: int = 10;

for f in fib(n) do
    writeln(f);

iter fib(x: int): int {
    var current: int = 0,
        next: int = 1;

    for i in 1..x {
        yield current;
        current += next;
        current <=> next;
    }
}
```

```
prompt> chpl fib.chpl
prompt> ./fib --n=1000
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
...
```

Explicit typing also supported
FIBONACCI ITERATION

```chapel
config const n = 10;

for (i, f) in zip(0..<n, fib(n)) do
  writeln("fib #", i, " is ", f);

iter fib(x) {
  var current = 0,
  next = 1;

  for i in 1..x {
    yield current;
    current += next;
    current <=> next;
  }
}
```

prompt> chpl fib.chpl
prompt> ./fib --n=1000
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
fib #7 is 13
fib #8 is 21
fib #9 is 34
fib #10 is 55
fib #11 is 89
fib #12 is 144
fib #13 is 233
fib #14 is 377
...
FIBONACCI ITERATION

fib.chpl

```chpl
config const n = 10;

for (i, f) in zip(0..<n, fib(n)) do
    writeln("fib ", i, " is ", f);

iter fib(x) {
    var current = 0,
    next = 1;

    for i in 1..x {
        yield current;
        current += next;
        current <=> next;
    }
}
```

prompt> chpl fib.chpl
prompt> ./fib --n=1000
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
fib #7 is 13
fib #8 is 21
fib #9 is 34
fib #10 is 55
fib #11 is 89
fib #12 is 144
fib #13 is 233
fib #14 is 377
...
OTHER BASE LANGUAGE FEATURES

- **Various basic types:** `bool(w), int(w), uint(w), real(w), imag(w), complex(w), enums, tuples`
- **Object-oriented programming**
  - Value-based records (like C structs supporting methods, generic fields, etc.)
  - Reference-based classes (somewhat like Java classes or C++ pointers-to-classes)
    - Nilable vs. non-nilable variants
    - Memory-management strategies (shared, owned, borrowed, unmanaged)
    - Lifetime checking
- **Error-handling**
- **Generic programming / polymorphism**
- **Compile-time meta-programming**
- **Modules** (supporting namespaces)
- **Procedure overloading / filtering**
- **Arguments:** default values, intents, name-based matching, type queries
  - and more…
TASK PARALLELISM AND LOCALITY CONTROL

Chapel language concepts

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control

“Lower-level” Chapel

Target System
**THE LOCALE: CHAPEL’S KEY FEATURE FOR LOCALITY**

- **locale**: a unit of the target architecture that can run tasks and store variables
  - Think “compute node” on a typical HPC system

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

**Locales array:**

Locale 0

Locale 1

Locale 2

Locale 3

User’s program starts running as a single task on locale 0
const numTasks = here.maxTaskPar;
coforall tid in 1..numTasks do
  writef("Hello from task %n of %n on %s\n", tid, numTasks, here.name);
const numTasks = here.maxTaskPar;
coforall tid in 1..numTasks do
  printf("Hello from task %n of %n on %s\\n", tid, numTasks, here.name);

‘here’ refers to the locale on which this code is currently running

how many parallel tasks can my locale run at once?

what’s my locale’s name?
**TASK-PARALLEL “HELLO WORLD”**

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

A 'coforall' loop executes each iteration as an independent task.
**TASK-PARALLEL “HELLO WORLD”**

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

So far, this is a shared-memory program

Nothing refers to remote locales, explicitly or implicitly
const numTasks = here.maxTaskPar;
coforall tid in 1..numTasks do
  writef("Hello from task %n of %n on %s
", tid, numTasks, here.name);
**TASK-PARALLEL “HELLO WORLD” (DISTRIBUTED VERSION)**

```chpl
coforall loc in Locales {
  on loc {
    const numTasks = here.maxTaskPar;
    coforall tid in 1..numTasks do
      writef("Hello from task %n of %n on %s\n",
             tid, numTasks, here.name);
  }
}
```
**TASK-PARALLEL “HELLO WORLD” (DISTRIBUTED VERSION)**

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

**Locales array:**

- Locale 0
- Locale 1
- Locale 2
- Locale 3

The array of locales we're running on (introduced a few slides back)
**TASK-PARALLEL “HELLO WORLD” (DISTRIBUTED VERSION)**

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

- **create a task per locale on which the program is running**
- **have each task run ‘on’ its locale**
- **then print a message per core, as before**

**Example Output**

```
prompt> chpl helloTaskPar.chpl
prompt> ./helloTaskPar -numLocales=4
Hello from task 1 of 4 on n1032
Hello from task 4 of 4 on n1032
Hello from task 1 of 4 on n1034
Hello from task 2 of 4 on n1032
Hello from task 1 of 4 on n1033
Hello from task 3 of 4 on n1034
Hello from task 1 of 4 on n1035
...```

---

**helloTaskPar.chpl**

create a task per locale on which the program is running

have each task run ‘on’ its locale

then print a message per core, as before
**TASK-PARALLEL “HELLO WORLD” (DISTRIBUTED VERSION)**

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

```
prompt> chpl helloTaskPar.chpl
prompt> ./helloTaskPar -numLocales=4
Hello from task 1 of 4 on n1032
Hello from task 4 of 4 on n1032
Hello from task 1 of 4 on n1034
Hello from task 2 of 4 on n1032
Hello from task 1 of 4 on n1033
Hello from task 3 of 4 on n1034
Hello from task 1 of 4 on n1035
...
```
Domain Maps
Data Parallelism
Task Parallelism
Base Language
Locality Control
Target System
DATA PARALLELISM AND DOMAIN MAPS

Chapel language concepts

<table>
<thead>
<tr>
<th>Domain Maps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Parallelism</td>
</tr>
<tr>
<td>Task Parallelism</td>
</tr>
<tr>
<td>Base Language</td>
</tr>
<tr>
<td>Locality Control</td>
</tr>
<tr>
<td>Target System</td>
</tr>
</tbody>
</table>

Higher-level Chapel
DATA-PARALLEL ARRAY FILL

```chapel
config const n = 1000;

const 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);
```
config const \( n = 1000 \);

const \( D = \{1..n, 1..n\} \);

var \( A: [D] \text{ real}; \)

forall \((i, j)\) in \( D \) do
\[
A[i,j] = i + (j - 0.5)/n;
\]

writeln(A);
DATA-PARALLEL ARRAY FILL

```chapel
config const n = 1000;

const 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);
```

*fillArray.chpl*

- declare a domain, a first-class index set
- declare an array over that domain
- iterate over the domain's indices in parallel, assigning to the corresponding array elements
# DATA-PARALLEL ARRAY FILL

```chpl
cfg const n = 1000;

const 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);
```

So far, this is a shared-memory program

Nothing refers to remote locales, explicitly or implicitly
DATA-PARALLEL ARRAY FILL

```chpl
config const n = 1000;

const 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);
```

So far, this is a shared-memory program

Nothing refers to remote locales, explicitly or implicitly
DATA-PARALLEL ARRAY FILL

```chapel
config const n = 1000;

const 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);
```
DATA-PARALLEL ARRAY FILL (DISTRIBUTED VERSION)

```chapel
fillArray.chpl

use CyclicDist;

config const n = 1000;

const 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);
```

```plaintext
A =

1.1 1.3 1.5 1.5 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
```
DATA-PARALLEL ARRAY FILL (DISTRIBUTED VERSION)

```chapel
fillArray.chpl
use CyclicDist;

config const n = 1000;

const 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);
```

apply a domain map, specifying how to implement...
...the domain's indices,
...the array's elements,
...the loop's iterations,
...on the program's locales
DATA-PARALLEL ARRAY FILL (DISTRIBUTED VERSION)

```chapel
fillArray.chpl

use CyclicDist;

config const n = 1000;

const 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);
```

Because this computation is independent of the locales, changing the number of locales or distribution doesn’t affect the output
DATA-PARALLEL ARRAY FILL (DISTIBUTED VERSION)

```chapel
use CyclicDist;

config const n = 1000;

const 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);
```

Because this computation is independent of the locales, changing the number of locales or distribution doesn’t affect the output.
DATA-PARALLEL ARRAY FILL (DISTRIBUTED VERSION)

```chpl
fillArray.chpl

use CyclicDist;

config const n = 1000;

const D = {1..n, 1..n}
    dmapped Cyclic(startIdx = (1,1));
var A: [D] real;

forall (i, j) in D do
    A[i, j] = i*10 + j + (here.id+1)/10.0;

writeln(A);
```

If we make it sensitive to the locales, the output varies with the distribution details
DATA-PARALLEL ARRAY FILL (DISTRIBUTED VERSION)

```chpl
fillArray.chpl

use CyclicDist;

config const n = 1000;

const D = {1..n, 1..n}
    dmapped Cyclic(startIdx = (1,1));
var A: [D] real;

forall (i,j) in D do
    A[i,j] = i*10 + j + (here.id+1)/10.0;

writeln(A);
```

If we make it sensitive to the locales, the output varies with the distribution details.
use CyclicDist;

config const n = 1000;

const D = {1..n, 1..n}
    dmapped Cyclic(startIdx = (1,1));
var A: [D] real;

forall (i,j) in D do
    A[i,j] = i*10 + j + (here.id+1)/10.0;

writeln(A);
Chapel is unique among programming languages
- built-in features for scalable parallel computing make it HPC-ready
- supports clean, concise code relative to conventional approaches
- ports and scales from laptops to supercomputers

Chapel is being used for productive parallel computing at scale
- users are reaping its benefits in practical, cutting-edge applications
- applicable to domains as diverse as physical simulations and data science

If you’re interested in taking Chapel for a spin, let us know!
- we’re happy to work with users and user groups to help ease the learning curve
- we’re discussing holding a day-long tutorial for you with hands-on, pending interest
CHAPEL RESOURCES

Chapel homepage: [https://chapel-lang.org](https://chapel-lang.org)
- (points to all other resources)

Social Media:
- Twitter: [@ChapelLanguage](https://twitter.com/ChapelLanguage)
- Facebook: [@ChapelLanguage](https://facebook.com/ChapelLanguage)
- YouTube: [http://www.youtube.com/c/ChapelParallelProgrammingLanguage](http://www.youtube.com/c/ChapelParallelProgrammingLanguage)

Community Discussion / Support:
- Discourse: [https://chapel.discourse.group/](https://chapel.discourse.group/)
- Gitter: [https://gitter.im/chapel-lang/chapel](https://gitter.im/chapel-lang/chapel)
- Stack Overflow: [https://stackoverflow.com/questions/tagged/chapel](https://stackoverflow.com/questions/tagged/chapel)
- GitHub Issues: [https://github.com/chapel-lang/chapel/issues](https://github.com/chapel-lang/chapel/issues)
THANK YOU

https://chapel-lang.org
@ChapelLanguage
BACKUP SLIDES: CHAPEL ON GPUS
CHAPEL ON GPUS

Background:

- GPUs have become a key feature in many HPC systems
- We have long described Chapel’s goal as being “any parallel algorithm on any parallel hardware”
- Yet, historically, Chapel releases have only supported GPUs via interoperability
  - i.e., call GPU code written in CUDA, OpenCL, OpenMP, ... as an extern routine

What’s New?

- Lots of progress in the past year...
TARGETING GPUs WITH CHAPEL WAS POSSIBLE FOR THE FIRST TIME, BUT VERY LOW-LEVEL:

```chapel
pragma "codegen for GPU"
export proc add_nums(A: c_ptr(real(64))){
    A[0] = A[0]+5;
}

var funcPtr = createFunction();
var A = [1, 2, 3, 4, 5];
__primitive("gpu kernel launch", funcPtr,
    <grid and block size>,...,c_ptrTo(A), ...);
writeln(A);
```

```
extern {
    #define FATBIN_FILE "chpl__gpu.fatbin"
    double createFunction(){
        fatbinBuffer = <read FATBIN_FILE into buffer>
        cuModuleLoadData(&cudaModule, fatbinBuffer);
        cuModuleGetFunction(&function, cudaModule,
            "add_nums");
    }
}
```
Raised the level of abstraction significantly, yet with significant restrictions:

- only relatively simple computations
- single GPU only
- single locale only

```chapel
on here.gpus[0] {
    var A: [0..<n] int;
    foreach a in A do
        a += 1;
}
```

- 'on' statement controls the execution/allocation policy
- 'A' will be allocated in the Unified Virtual Memory
- 'foreach' will turn into a GPU kernel
CHAPEL FOR GPUS: CHAPEL 1.26.0

Improved generality: computational styles, multiple GPUs, CPU+GPU parallelism

```plaintext
cobegin {
    A[0..<cpuSize] += 1;
}

coforall subloc in 1..numGPUs do on here.getChild(subloc) {
    const myShare = cpuSize+gpuSize*(subloc-1)..#gpuSize;

    var AonThisGPU = A[myShare];
    AonThisGPU += 1;
    A[myShare] = AonThisGPU;
}
```

- Two concurrent tasks
- CPU works on its part
- GPUs work on their part and copy the result back
CHAPEL FOR GPUS: CHAPEL 1.27.0

Added support for using the GPUs of multiple locales simultaneously, improved sublocale abstractions

```plaintext
config const n=1000, alpha=0.5;

coforall loc in Locales do on loc {
  coforall gpu in here.gpus do on gpu {
    var A, B, C: [1..n] real;
    A = B + alpha * C;
  }
}
```
CHAPEL FOR GPUS: WHAT’S NEXT?

- Performance Analysis & Improvements
- Portability to additional vendors
- GPU participation in inter-node communication