Productive Programming in Chapel: A Computation-Driven Introduction

Short Introduction to Locality

Michael Ferguson and Lydia Duncan
Cray Inc,

SC15 November 15th, 2015

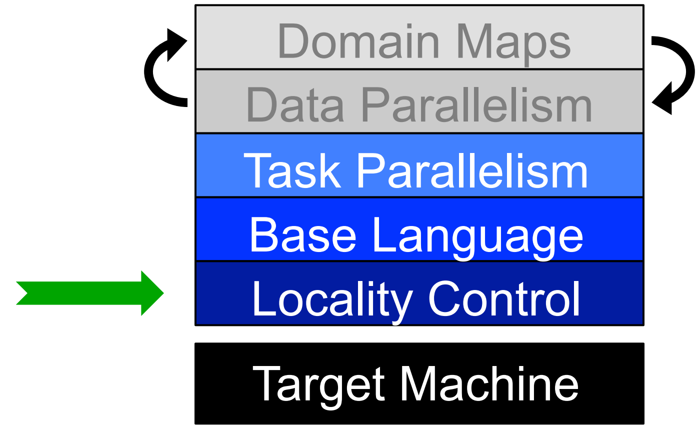
Safe Harbor Statement

This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.

Outline

- ✓ Motivation
- ✓ Chapel Background and Themes
- ✓ Learning the Base Language with n-body
- ✓ Short Introduction to Task Par
- ✓ Hands-On 1: Hello World
- ➤ **Short Introduction to Locality**
- Data Parallelism with Jacobi
- Hands-On 2: Mandelbrot
- Project Status, Next Steps

Theme 4: Control over Locality/Affinity



The Locale Type

Definition:

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

Typically: A compute node (multicore processor or SMP)

Getting started with locales

- Specify # of locales when running Chapel programs

```
% a.out --numLocales=8
```

```
% a.out -nl 8
```

- Chapel provides built-in locale variables

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

Locales

L0	L1	L2	L3	L4	L5	L6	L7

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

Locale Operations

- Locale methods support queries about the target system:

```
proc locale.physicalMemory(…) { … }
proc locale.numCores { … }
proc locale.id { … }
proc locale.name { … }
```

- *On-clauses* support placement of computations:

```
writeln("on locale 0");

on Locales[1] do
  writeln("now on locale 1");

writeln("on locale 0 again");
```

```
on A[i,j] do
  bigComputation(A);

on node.left do
  search(node.left);
```

Parallelism and Locality: Orthogonal in Chapel

- This is a **parallel**, but local program:

```
begin writeln("Hello world!");
writeln("Goodbye!");
```

- This is a **distributed**, but serial program:

```
writeln("Hello from locale 0!");
on Locales[1] do writeln("Hello from locale 1!");
writeln("Goodbye from locale 0!");
```

- This is a **distributed** and **parallel** program:

```
begin on Locales[1] do writeln("Hello from locale 1!");
on Locales[2] do begin writeln("Hello from locale 2!");
writeln("Goodbye from locale 0!");
```

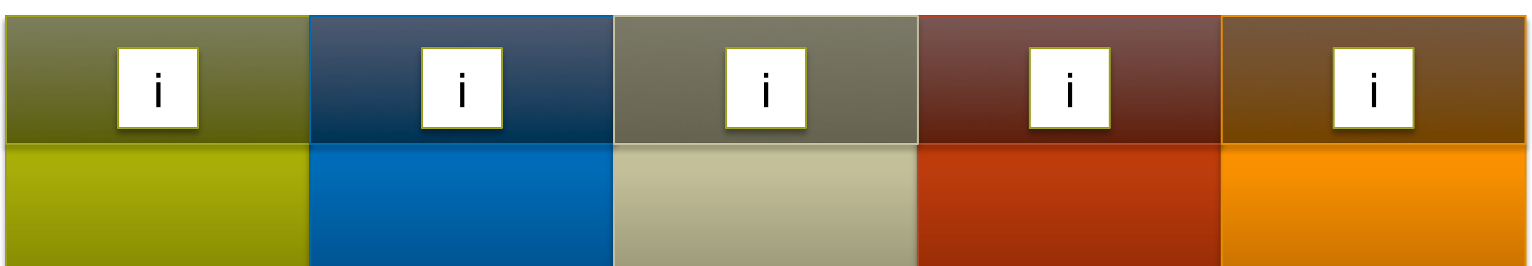
Partitioned Global Address Space (PGAS) Languages

(Or perhaps: partitioned global namespace languages)

- **abstract concept:**
  - support a shared namespace on distributed memory
    - permit parallel tasks to access remote variables by naming them
  - establish a strong sense of ownership
    - every variable has a well-defined location
    - local variables are cheaper to access than remote ones
- **traditional PGAS languages have been SPMD in nature**
  - best-known examples: Co-Array Fortran, UPC

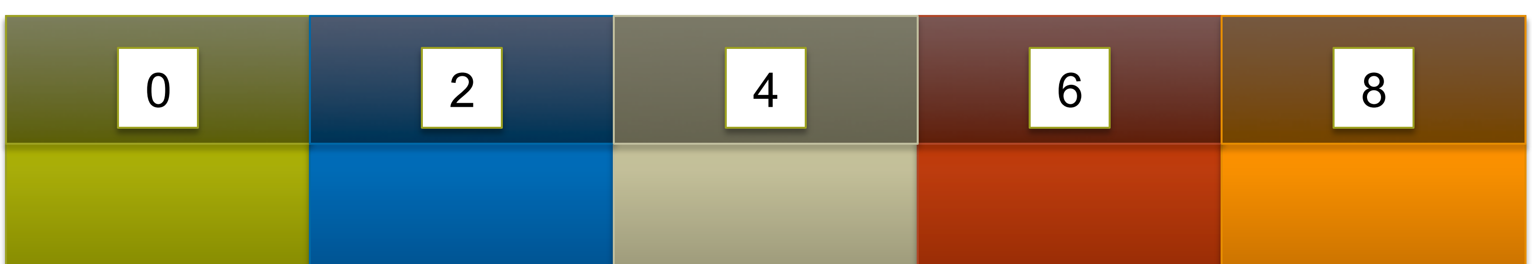
SPMD PGAS Languages (using a pseudo-language, not Chapel)

```
proc main() {
  var i(*): int;          // declare a shared variable i
```



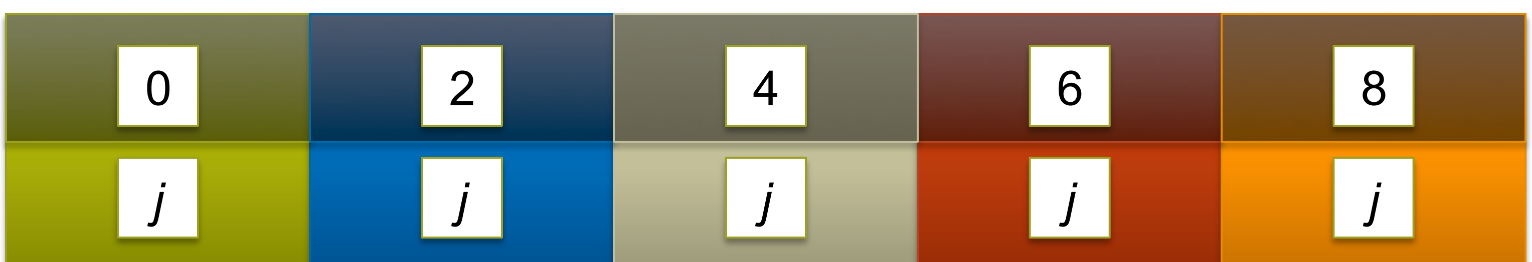
SPMD PGAS Languages (using a pseudo-language, not Chapel)

```
proc main() {
  var i(*): int;          // declare a shared variable i
  i = 2*this_image();     // each image initializes its copy
```



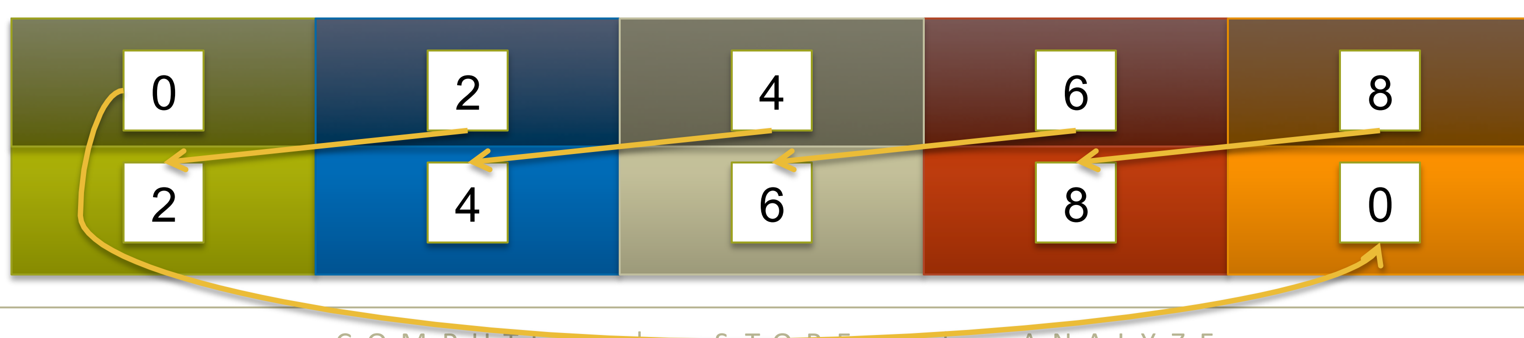
SPMD PGAS Languages (using a pseudo-language, not Chapel)

```
proc main() {
  var i(*): int;          // declare a shared variable i
  i = 2*this_image();     // each image initializes its copy
  var j: int;             // declare a private variable j
```



SPMD PGAS Languages (using a pseudo-language, not Chapel)

```
proc main() {
  var i(*): int;          // declare a shared variable i
  i = 2*this_image();     // each image initializes its copy
  var j: int;             // declare a private variable j
  j = i( (this_image()+1) % num_images() );
    // ^^ access our neighbor's copy of i
    // communication implemented by compiler + runtime

    // How did we know our neighbor had an i?
    // Because it's SPMD – we're all running the same
    // program.  (Simple, but restrictive)
```

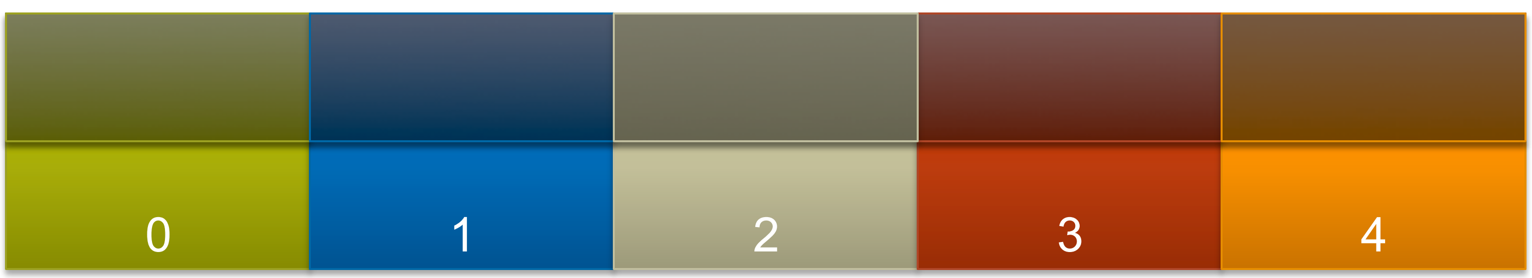


Chapel and PGAS

- **PGAS: Partitioned Global Address Space**
  - support a shared namespace on distributed memory
  - but allow reasoning about locality
- **Chapel is PGAS, but unlike most, it's not inherently SPMD**

  ⇒ never think about "the other copies of the program"

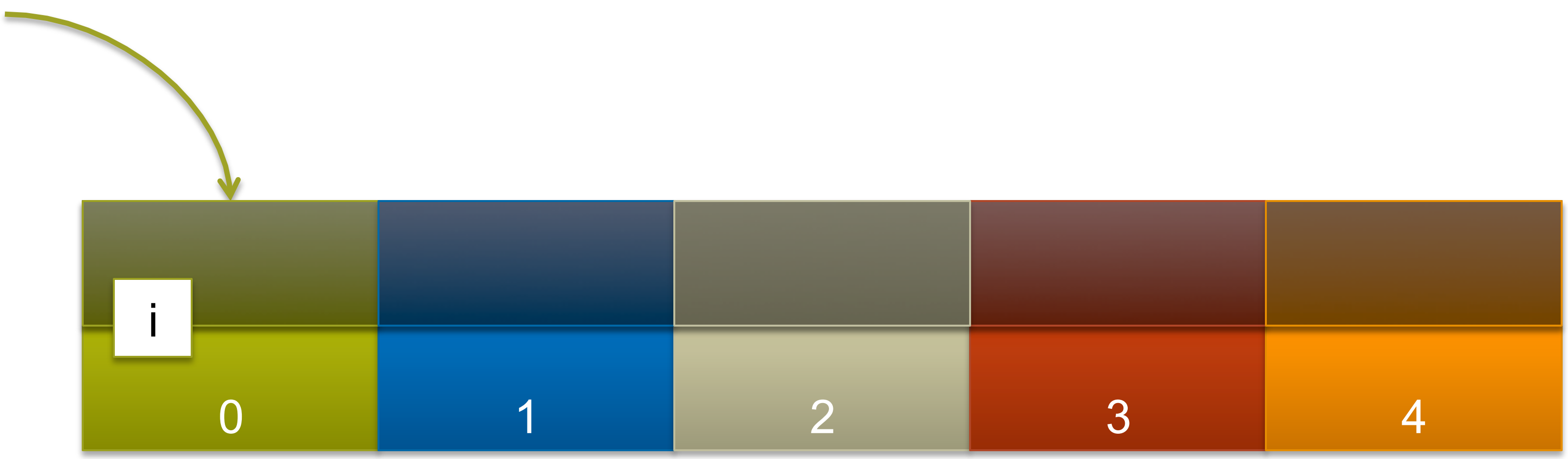
  ⇒ "global name/address space" comes from lexical scoping
  - as in traditional languages, each declaration yields one variable
  - variables are stored on the locale where the task declaring it is executing



Locales (think: "compute nodes")

Chapel: Scoping and Locality

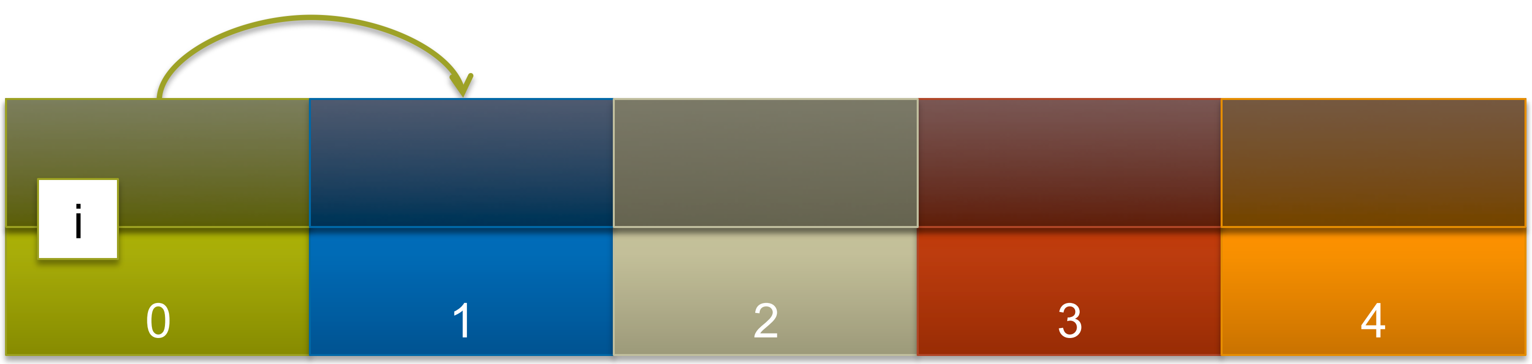
```
var i: int;
```



Locales (think: "compute nodes")

Chapel: Scoping and Locality

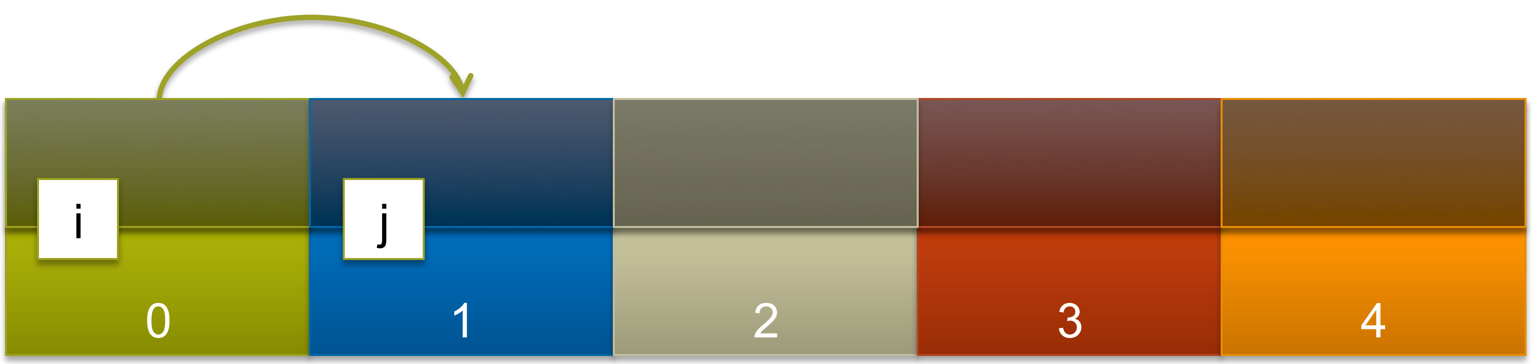
```
var i: int;
on Locales[1] {
```



Locales (think: “compute nodes”)

Chapel: Scoping and Locality

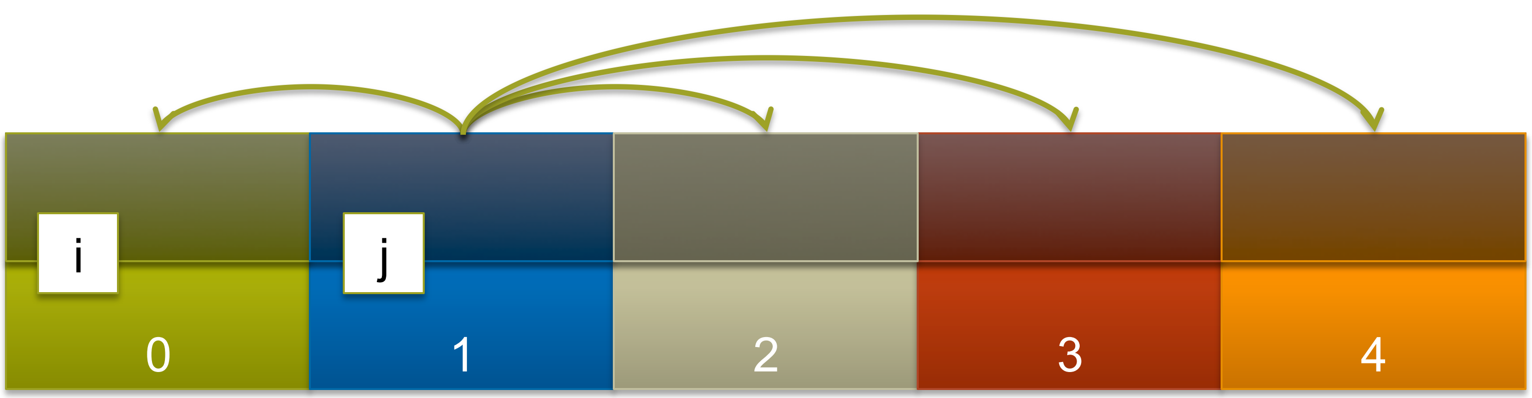
```
var i: int;
on Locales[1] {
  var j: int;
```



Locales (think: "compute nodes")

Chapel: Scoping and Locality

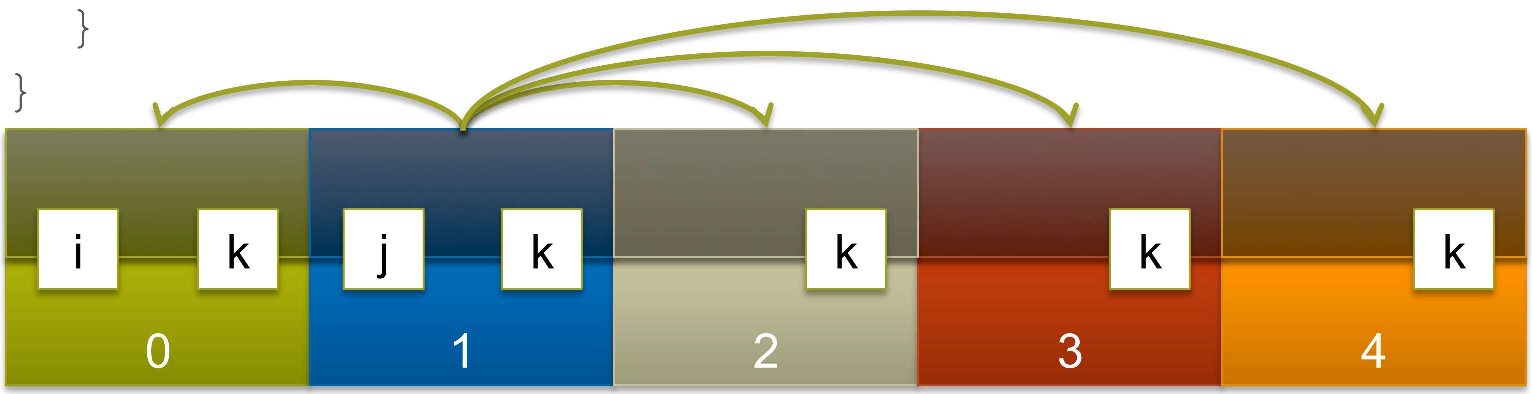
```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
```



Locales (think: “compute nodes”)

Chapel: Scoping and Locality

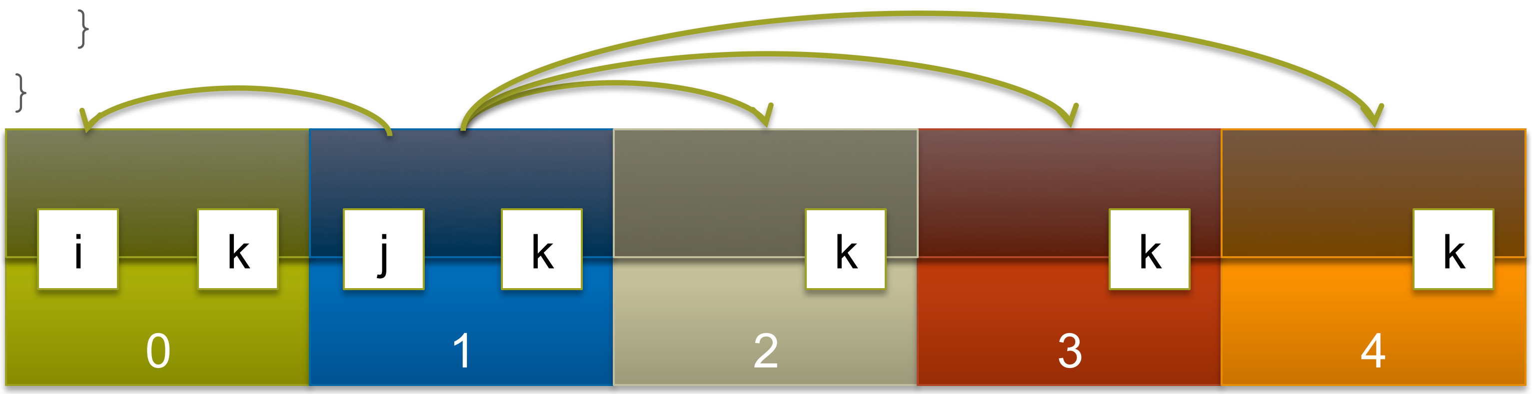
```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      var k: int;
      // within this scope, i, j, and k can be referenced;
      // the implementation manages the communication for i and j
    }
  }
}
```



Locales (think: "compute nodes")

Chapel: Scoping and Locality

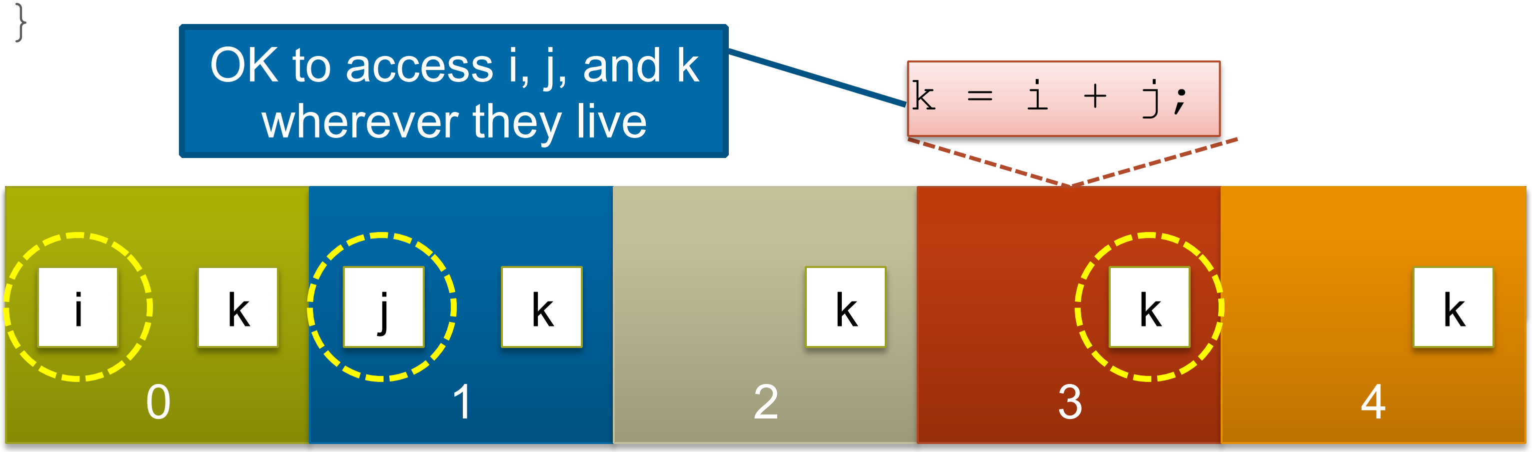
```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      var k: int;
      // within this scope, i, j, and k can be referenced;
      // the implementation manages the communication for i and j
      k = i + j;
    }
  }
}
```



Locales (think: "compute nodes")

Chapel: Scoping and Locality

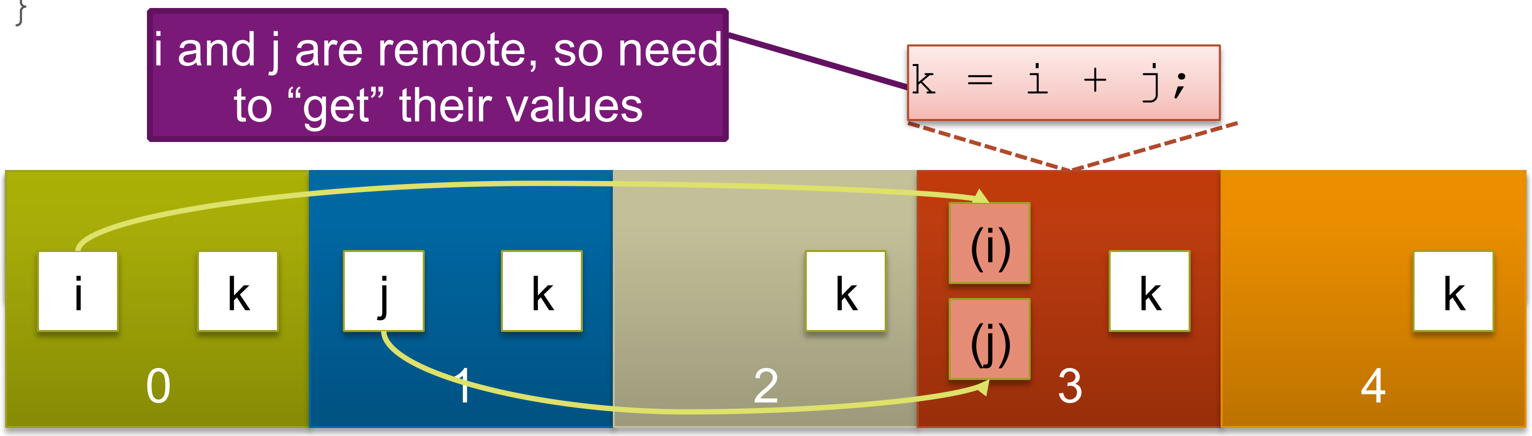
```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      var k: int;
      k = i + j;
    }
  }
}
```



Images / Threads / Locales / Places / etc. (think: "compute nodes")

Chapel: Scoping and Locality

```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      var k: int;
      k = i + j;
    }
  }
}
```



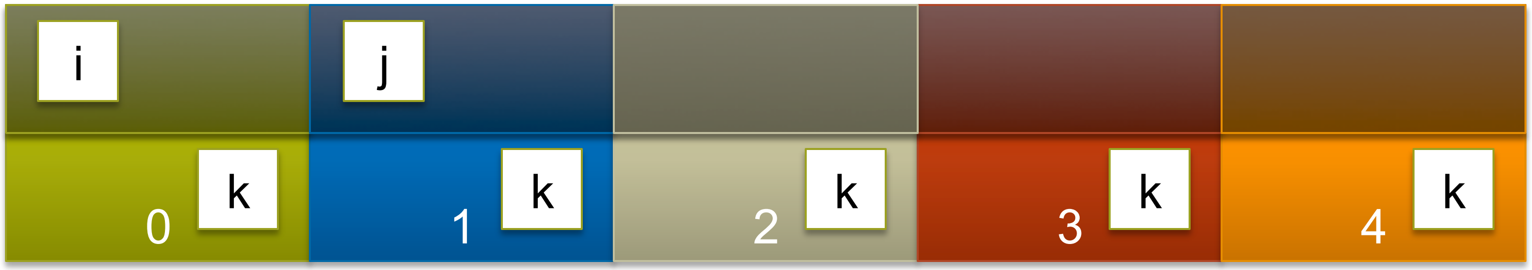
Images / Threads / Locales / Places / etc. (think: "compute nodes")

Chapel and PGAS: Public vs. Private

How public a variable is depends only on scoping

- who can see it?
- who actually bothers to refer to it non-locally?

```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      var k = i + j;
    }
  }
}
```



Locales (think: "compute nodes")

Querying a Variable's Locale

- **Syntax**

```
locale-query-expr:
  expr . locale
```

- **Semantics**
  - Returns the locale on which *expr* is stored
- **Example**

```
var i: int;
on Locales[1] {
  var j: int;
  writeln((i.locale.id, j.locale.id));  // outputs (0,1)
}
```

Here

- **Built-in locale variable**

```
const here: locale;
```

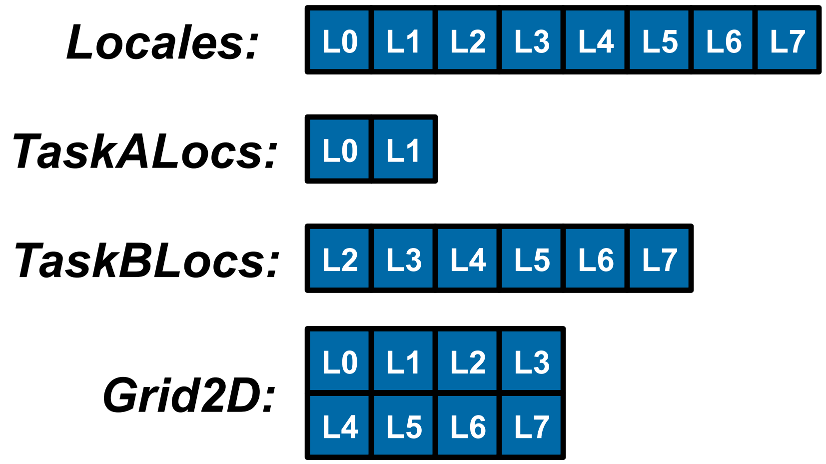
- **Semantics**
  - Refers to the locale on which the task is executing
- **Example**

```
writeln(here.id);      // outputs 0
on Locales[1] do
  writeln(here.id);    // outputs 1

on myC do
  if (here == Locales[0]) then …
```

Rearranging Locales

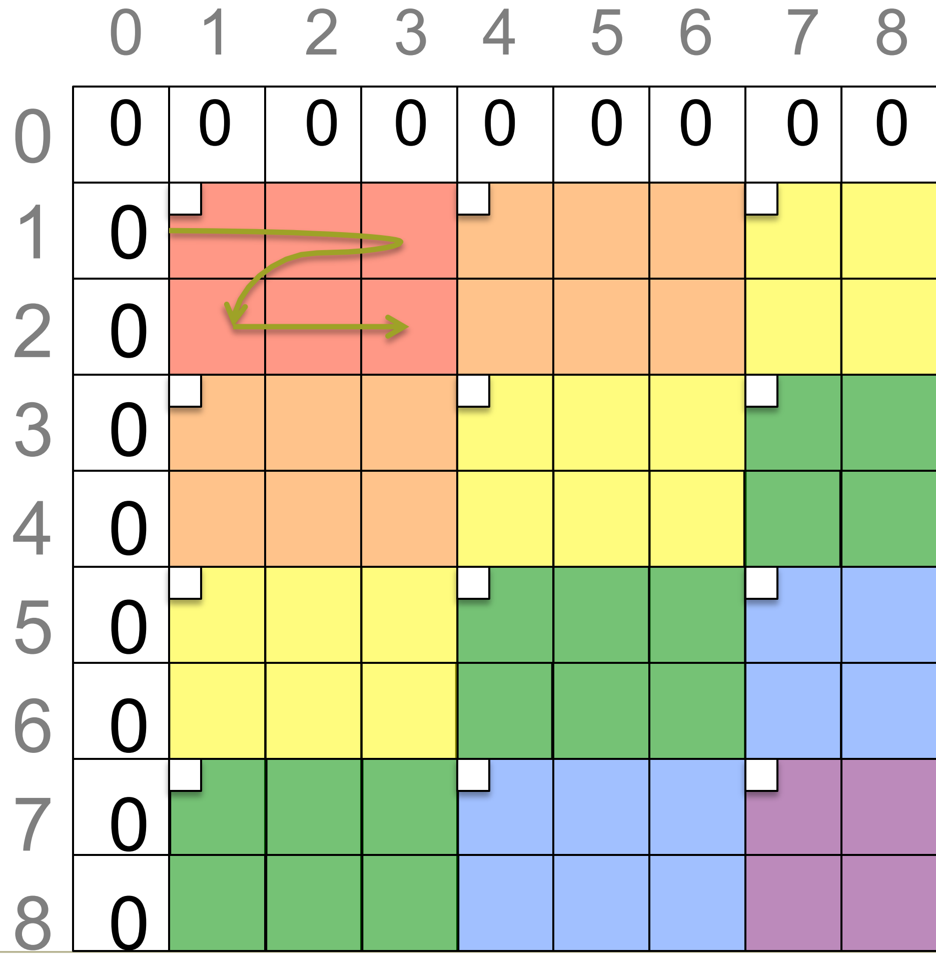
Create locale views with standard array operations:

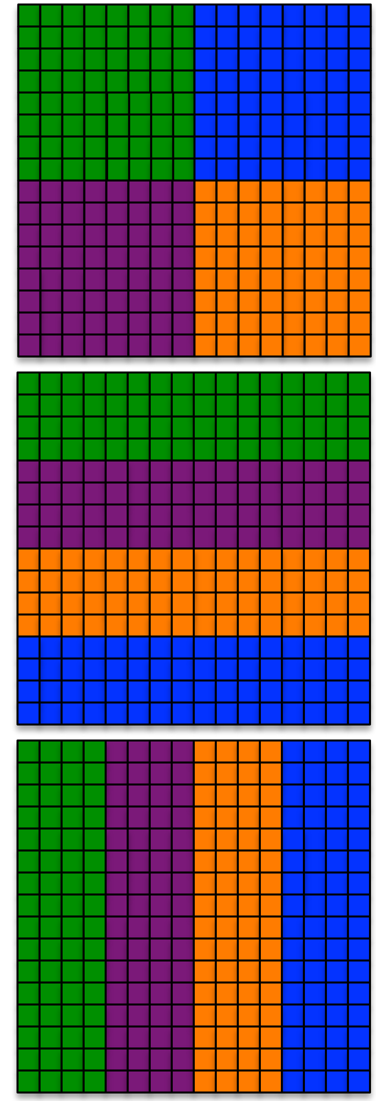
```
var TaskALocs = Locales[0..1];
var TaskBLocs = Locales[2..];

var Grid2D = reshape(Locales, {1..2, 1..4});
```



Distributed Smith-Waterman

Now, what about distributed memory?

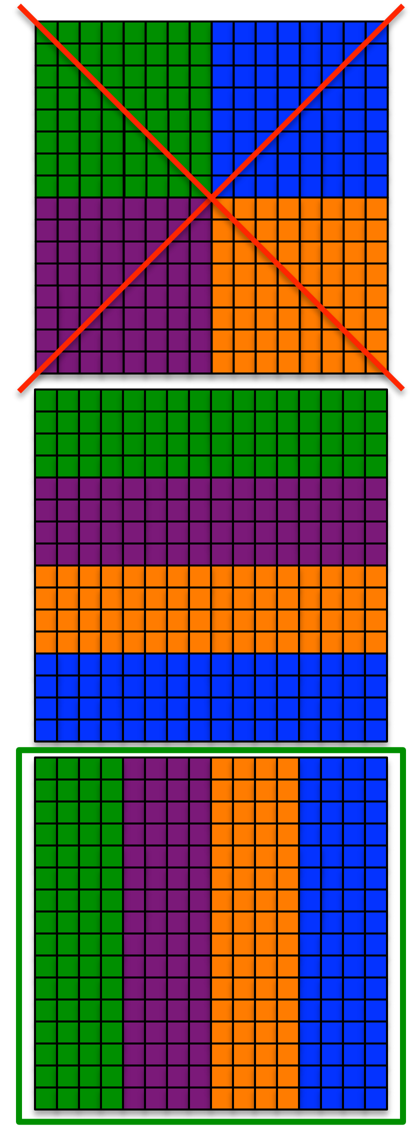




Distributed Smith-Waterman

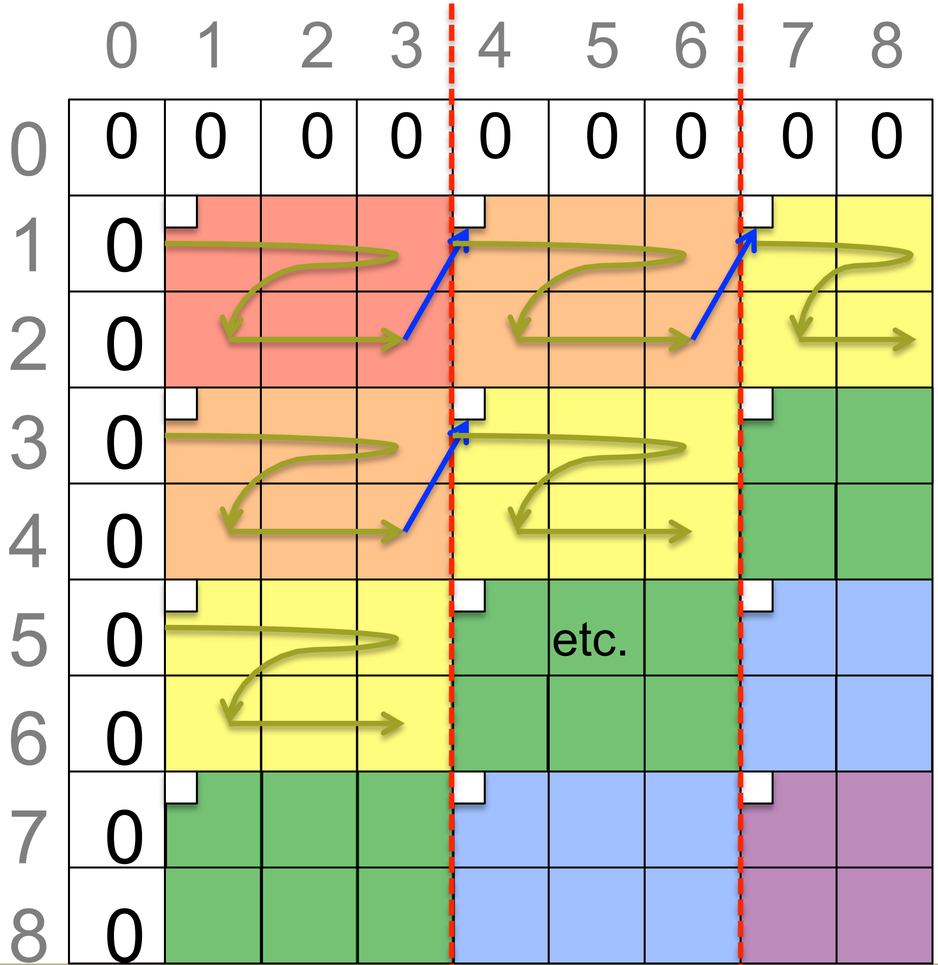
Now, what about distributed memory?

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0								
2	0								
3	0								
4	0								
5	0								
6	0								
7	0								
8	0								

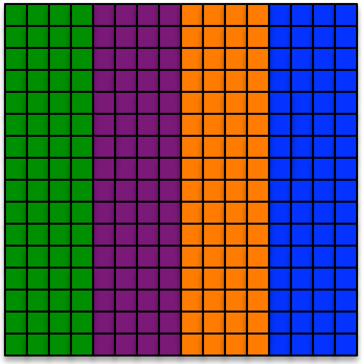


Distributed Smith-Waterman

Now, what about distributed memory?



Advantages:
- Good cache behavior: Nice fat blocks of data touchable in memory order
- Pipeline parallelism: Good utilization once pipeline is filled



Distributed Smith-Waterman

Distributed Chunked Data-Driven Task-Parallel Approach:

Reshape the 1D Locales array into a 2D column

Block-distribute the data space across the column of locales

Compute each chunk on the locale that owns its initial element

```
const Hspace = {0..n, 0..n};
const LocaleGrid = Locales.reshape({0..#numLocales, 0..0});
const DistHSpace = Hspace dmapped Block(Hspace, LocaleGrid);
var H: [DistHSpace] int;

proc computeH(H: [] int) {
  const ProbSpace = H.domain.translate(1,1);
  const StrProbSpace = ProbSpace by (rowsPerChunk, colsPerChunk);
  var NeighborsDone: [StrProbSpace] atomic int;
…
  proc computeHHelp(x,y) {
    on H[x,y] {
      for (i,j) in ProbSpace[x..#rowsPerChunk, y..#colsPerChunk] do
        H[i,j] = f(H[i-1,j-1], H[i-1,j], H[i,j-1]);
    const eastReady =  NeighborsDone[x,            y+colsPerChunk].fetchAdd(1);
    …etc…
    if (eastReady == 2)  then begin computeHHelp(x,          y+colsPerChunk);
    …etc…
} } }
```

Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

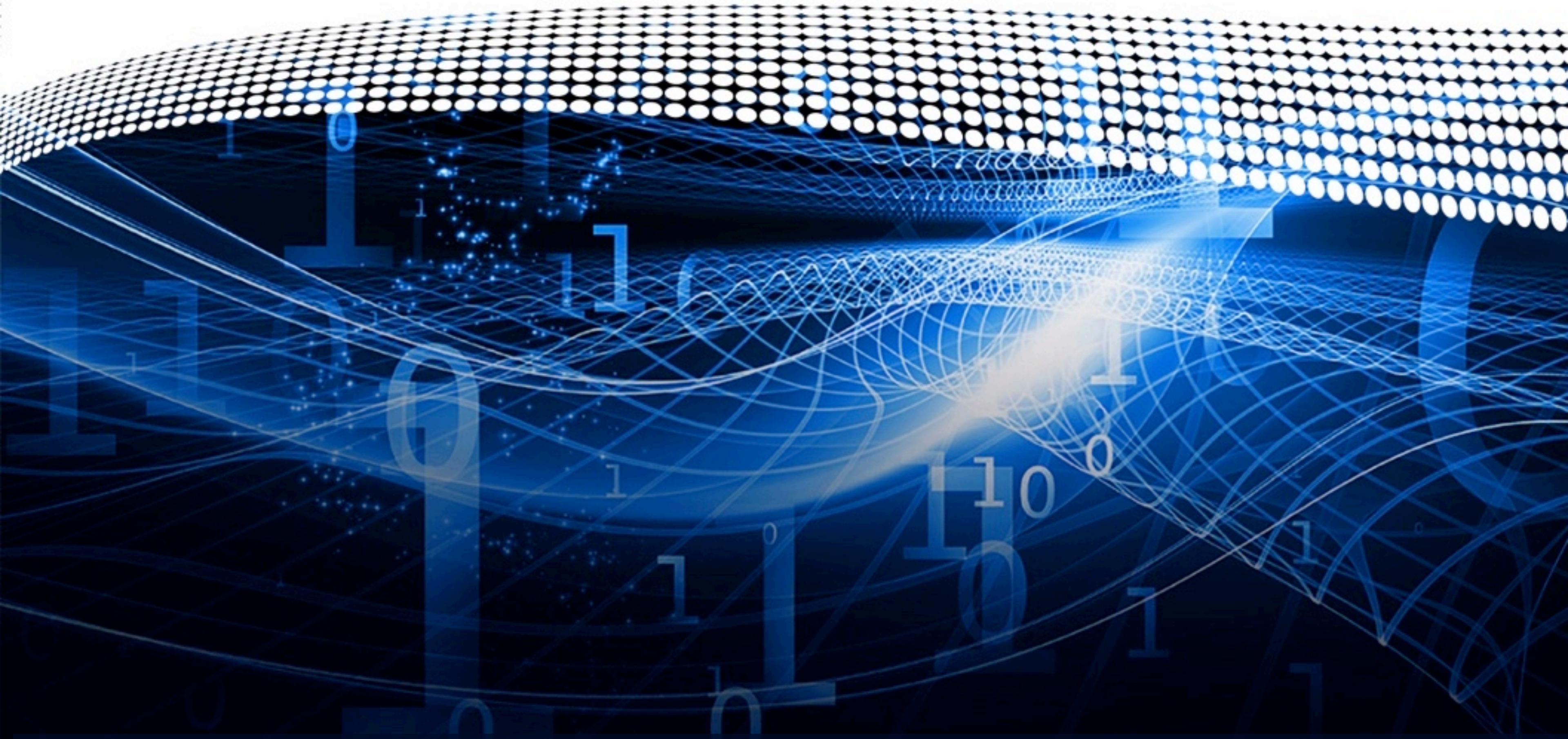
Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, and URIKA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.

COMPUTE | STORE | ANALYZE



http://chapel.cray.com chapel_info@cray.com http://github.com/chapel-lang/chapel/