Chapel:
Productive, Multiresolution Parallel Programming

Brad Chamberlain, Chapel Team, Cray Inc.

ATPESC: Argonne Training Program for Exascale Computing

August 6th, 2015

Chapel:
HPC Programmers Deserve Nice Things Too

Brad Chamberlain, Chapel Team, Cray Inc.

ATPESC: Argonne Training Program for Exascale Computing

August 6th, 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.

Chapel Motivation

Q: Why doesn't parallel programming have an equivalent to Python / Matlab / Java / C++ / (your favorite programming language here) ?

- one that makes it easy to quickly get codes up and running
- one that is portable across system architectures and scales
- one that bridges the HPC, data analysis, and mainstream communities

A: We believe this is due not to any particular technical challenge, but rather a lack of sufficient…

…long-term efforts
…resources
…community will
…co-design between developers and users
…patience

Chapel is our attempt to change this

What is Chapel?

- **An emerging parallel programming language**
  - Design and development led by Cray Inc.
    - in collaboration with academia, labs, industry; domestically & internationally
- **A work-in-progress**
- **Goal: Improve productivity of parallel programming**

What does "Productivity" mean to you?

Recent Graduates:

"something similar to what I used in school: Python, Matlab, Java, …"

Seasoned HPC Programmers:

"that sugary stuff that I don't need because I ~~was born to suffer~~ want full control to ensure performance"

Computational Scientists:

"something that lets me express my parallel computations without having to wrestle with architecture-specific details"

Chapel Team:

"something that lets computational scientists express what they want, without taking away the control that HPC programmers want, implemented in a language as attractive as recent graduates want."

Chapel's Implementation

- **Being developed as open source at GitHub**
  - Licensed as Apache v2.0 software

- **Portable design and implementation, targeting:**
  - multicore desktops and laptops
  - commodity clusters and the cloud
  - HPC systems from Cray and other vendors
  - *in-progress:* manycore processors, CPU+accelerator hybrids, …

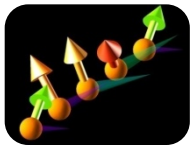
Sustained Performance Milestones

1 GF – 1988: Cray Y-MP; 8 Processors

- Static finite element analysis
- Fortran77 + Cray autotasking + vectorization

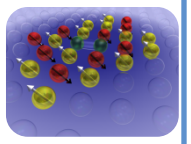
1 TF – 1998: Cray T3E; 1,024 Processors

- Modeling of metallic magnet atoms
- Fortran + MPI (Message Passing Interface)



1 PF – 2008: Cray XT5; 150,000 Processors

- Superconductive materials
- C++/Fortran + MPI + vectorization



1 EF – ~20__: Cray ____; ~10,000,000 Processors

- TBD
- TBD: C/C++/Fortran + MPI + OpenMP/OpenACC/CUDA/OpenCL?

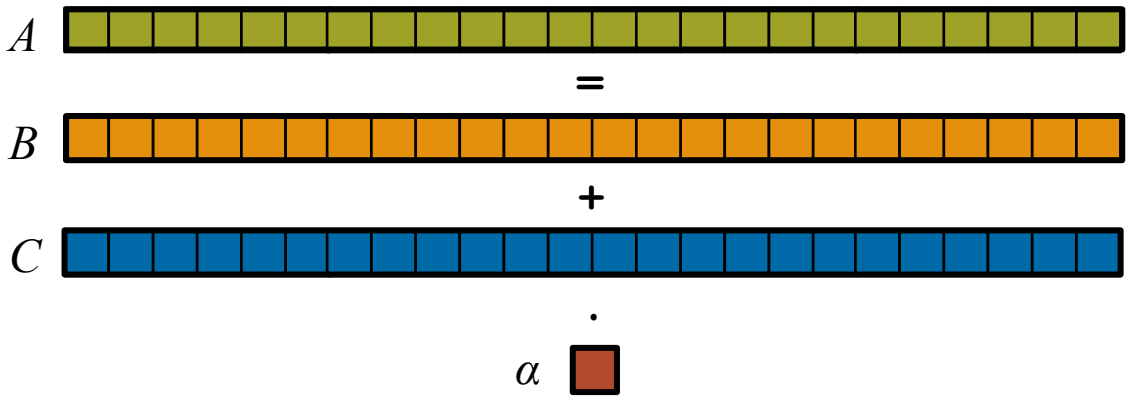
Or, perhaps something completely different?

STREAM Triad: a trivial parallel computation

Given: m-element vectors A, B, C

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

In pictures:

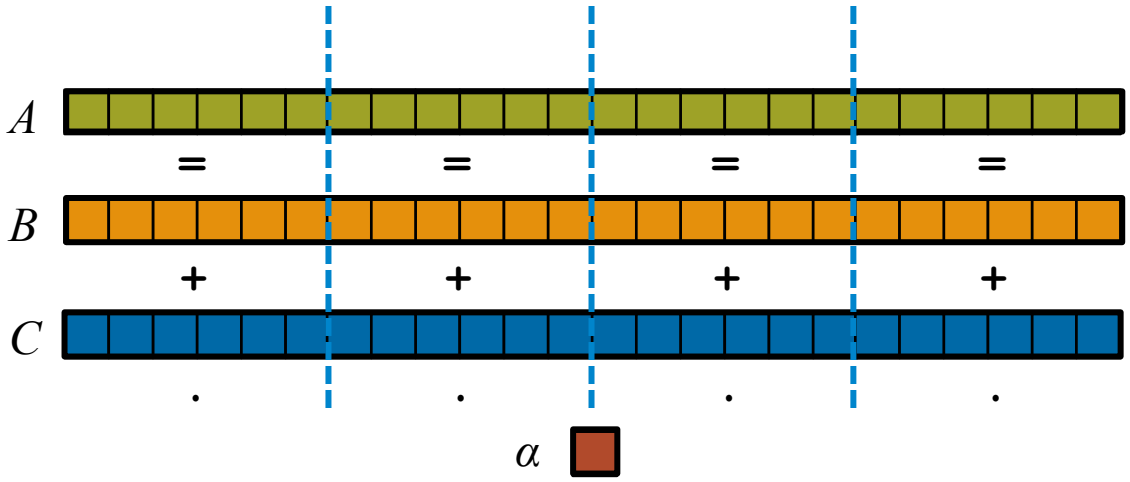


STREAM Triad: a trivial parallel computation

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:

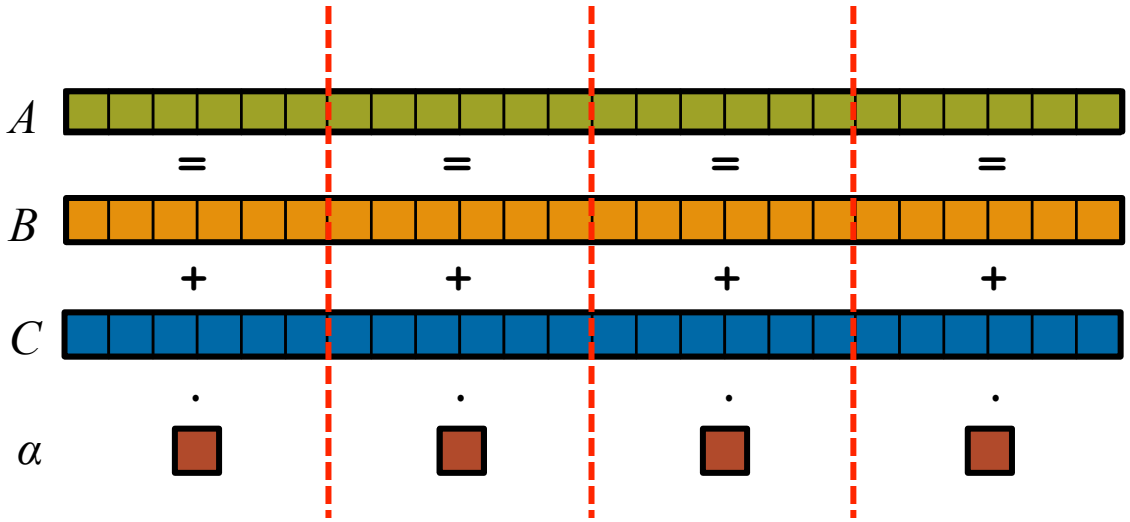


STREAM Triad: a trivial parallel computation

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):

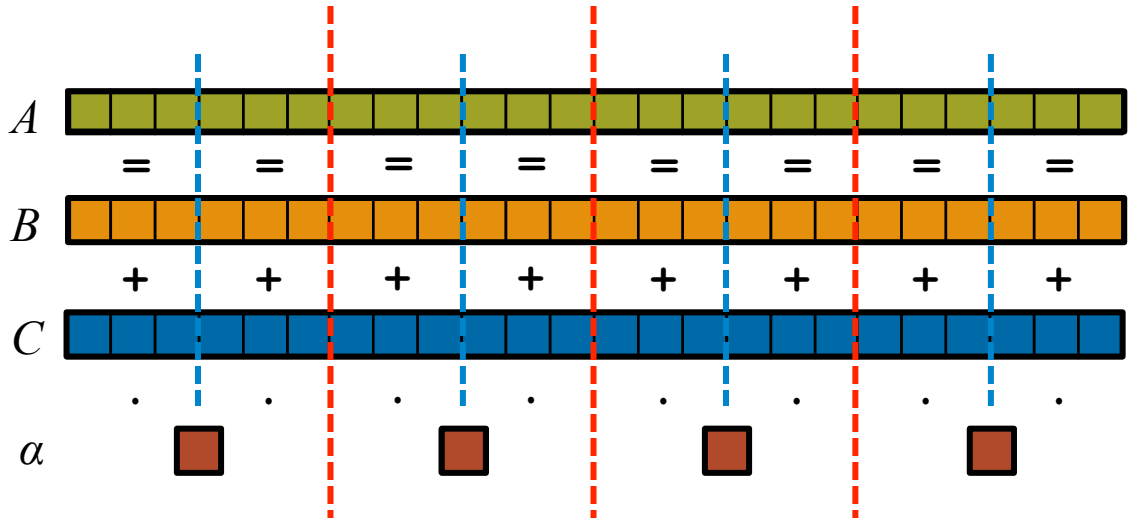


STREAM Triad: a trivial parallel computation

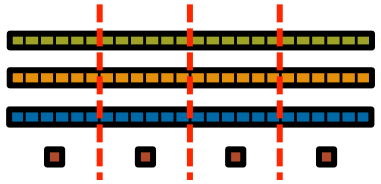
Given: m-element vectors A, B, C

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

In pictures, in parallel (distributed memory multicore):



STREAM Triad: MPI



MPI

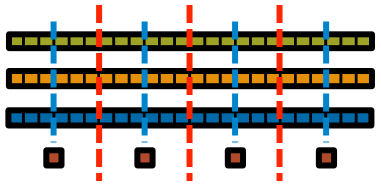
```
#include <hpcc.h>



static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Comm_size( comm, &commSize );
  MPI_Comm_rank( comm, &myRank );

  rv = HPCC_Stream( params, 0 == myRank);
  MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM,
    0, comm );

  return errCount;
}

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 );
```

```
if (!a || !b || !c) {
  if (c) HPCC_free(c);
  if (b) HPCC_free(b);
  if (a) HPCC_free(a);
  if (doIO) {
    fprintf( outFile, "Failed to allocate memory (%d).
 \n", VectorSize );
    fclose( outFile );
  }
  return 1;
}
```

```
for (j=0; j<VectorSize; j++) {
  b[j] = 2.0;
  c[j] = 0.0;
}

scalar = 3.0;
```

```
for (j=0; j<VectorSize; j++)
  a[j] = b[j]+scalar*c[j];

HPCC_free(c);
HPCC_free(b);
HPCC_free(a);
```

STREAM Triad: MPI+OpenMP

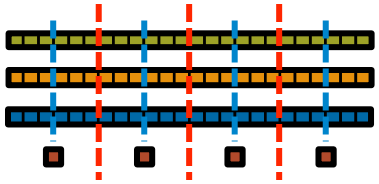


MPI + OpenMP

```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Comm_size( comm, &commSize );
  MPI_Comm_rank( comm, &myRank );

  rv = HPCC_Stream( params, 0 == myRank);
  MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM,
    0, comm );

  return errCount;
}

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 );

  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
      fprintf( outFile, "Failed to allocate memory (%d).
    \n", VectorSize );
      fclose( outFile );
    }
    return 1;
  }

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 0.0;
  }

  scalar = 3.0;

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];

  HPCC_free(c);
  HPCC_free(b);
  HPCC_free(a);
```

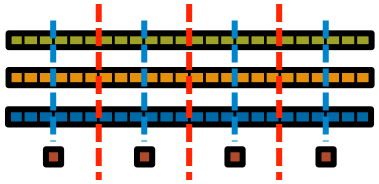
STREAM Triad: MPI+OpenMP vs. CUDA

MPI + OpenMP



```
#ifdef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Comm_size( comm, &commSize );
  MPI_Comm_rank( comm, &myRank );

  rv = HPCC_Stream( params, 0 == myRank);
  MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm );

  return errCount;
}

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 );

  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
      fprintf( outFile, "Failed to allocate memory (%d).\n", VectorSize );
      fclose( outFile );
    }
    return 1;
  }

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 0.0;
  }

  scalar = 3.0;

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];

  HPCC_free(c);
  HPCC_free(b);
  HPCC_free(a);

  return 0;
}
```

CUDA



```
#define N       2000000

int main() {
  float *d_a, *d_b, *d_c;
  float scalar;

  cudaMalloc((void**)&d_a, sizeof(float)*N);
  cudaMalloc((void**)&d_b, sizeof(float)*N);
  cudaMalloc((void**)&d_c, sizeof(float)*N);

  dim3 dimBlock(128);
  dim3 dimGrid(N/dimBlock.x );
  if( N % dimBlock.x != 0 ) dimGrid.x+=1;

  set_array<<<dimGrid,dimBlock>>>(d_b, .5f, N);
  set_array<<<dimGrid,dimBlock>>>(d_c, .5f, N);

  scalar=3.0f;
  STREAM_Triad<<<dimGrid,dimBlock>>>(d_b, d_c, d_a, scalar,  N);
  cudaThreadSynchronize();

  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);


__global__ void set_array(float *a,  float value, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) a[idx] = value;
}

__global__ void STREAM_Triad( float *a, float *b, float *c,
                              float scalar, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) c[idx] = a[idx]+scalar*b[idx];
}
```

HPC suffers from too many distinct notations for expressing parallelism and locality

Why so many programming models?

HPC has traditionally given users…

…low-level, *control-centric* programming models

…ones that are closely tied to the underlying hardware

…ones that support only a single type of parallelism

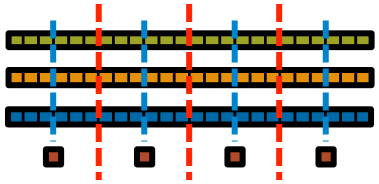
Type of HW Parallelism	Programming Model	Unit of Parallelism
Inter-node	MPI	executable
Intra-node/multicore	OpenMP / pthreads	iteration/task
Instruction-level vectors/threads	pragmas	iteration
GPU/accelerator	Open[MP|CL|ACC] / CUDA	SIMD function/task

benefits: lots of control; decent generality; easy to implement

downsides: lots of user-managed detail; brittle to changes

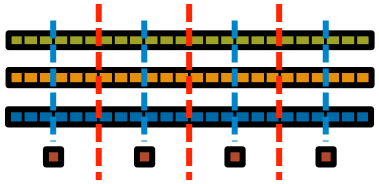
Rewinding a few slides…

MPI + OpenMP



```
#ifdef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Comm_size( comm, &commSize );
  MPI_Comm_rank( comm, &myRank );

  rv = HPCC_Stream( params, 0 == myRank);
  MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm );

  return errCount;
}

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 );

  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
      fprintf( outFile, "Failed to allocate memory (%d).\n", VectorSize );
      fclose( outFile );
    }
    return 1;
  }

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 0.0;
  }

  scalar = 3.0;

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];

  HPCC_free(c);
  HPCC_free(b);
  HPCC_free(a);

  return 0;
}
```

CUDA



```
#define N       2000000

int main() {
  float *d_a, *d_b, *d_c;
  float scalar;

  cudaMalloc((void**)&d_a, sizeof(float)*N);
  cudaMalloc((void**)&d_b, sizeof(float)*N);
  cudaMalloc((void**)&d_c, sizeof(float)*N);

  dim3 dimBlock(128);
  dim3 dimGrid(N/dimBlock.x );
  if( N % dimBlock.x != 0 ) dimGrid.x+=1;

  set_array<<<dimGrid,dimBlock>>>(d_b, .5f, N);
  set_array<<<dimGrid,dimBlock>>>(d_c, .5f, N);

  scalar=3.0f;
  STREAM_Triad<<<dimGrid,dimBlock>>>(d_b, d_c, d_a, scalar,  N);
  cudaThreadSynchronize();

  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);


__global__ void set_array(float *a,  float value, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) a[idx] = value;
}

__global__ void STREAM_Triad( float *a, float *b, float *c,
                              float scalar, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) c[idx] = a[idx]+scalar*b[idx];
}
```

HPC suffers from too many distinct notations for expressing parallelism and locality

STREAM Triad: Chapel

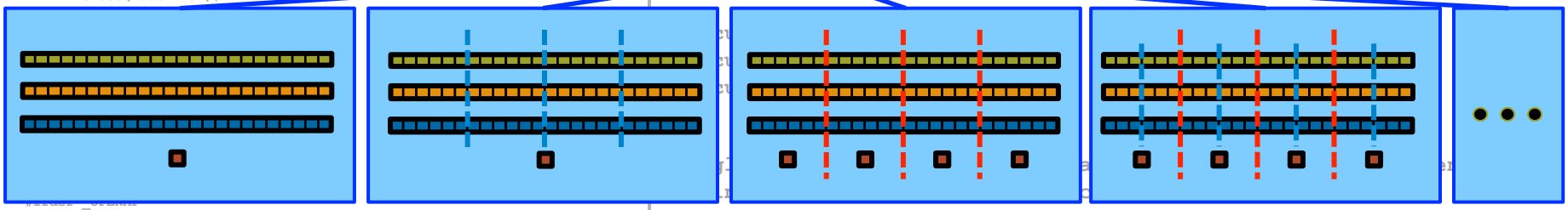
MPI + OpenMP

```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *pa
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Comm_size( comm, &commSize );
  MPI_Comm_rank( comm, &myRank );

  rv = HPCC_Stream( params, 0 == myR
  MPI_Reduce( &rv, &errCount, 1, MPI

  return errCount;
}

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

  VectorSize = HPCC_LocalVectorSize(

  a = HPCC_XMALLOC( double, VectorSi
  b = HPCC_XMALLOC( double, VectorSi
  c = HPCC_XMALLOC( double, VectorSi

  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
      fprintf( outFile, "Failed to allocate memory (%d).\n", VectorSize );
      fclose( outFile );
```

Chapel

```
config const m = 1000,
             alpha = 3.0;

const ProblemSpace = {1..m} dmapped …;

var A, B, C: [ProblemSpace] real;

B = 2.0;
C = 3.0;

A = B + alpha * C;
```

the special sauce



Philosophy: Good language design can tease details of locality and parallelism away from an algorithm, permitting the compiler, runtime, applied scientist, and HPC expert to each focus on their strengths.

Outline

- ✓ Motivation
- ➢ Chapel Background and Themes
- Survey of Chapel Concepts
- Project Status and Next Steps

- **This evening:** Chapel hands-on session

Motivating Chapel Themes

1) General Parallel Programming
2) Global-View Abstractions
3) Multiresolution Design
4) Control over Locality/Affinity
5) Reduce HPC ↔ Mainstream Language Gap

Motivating Chapel Themes

1) General Parallel Programming
2) Global-View Abstractions
3) Multiresolution Design
4) Control over Locality/Affinity
5) Reduce HPC ↔ Mainstream Language Gap

1) General Parallel Programming

With a unified set of concepts...

...express any parallelism desired in a user's program

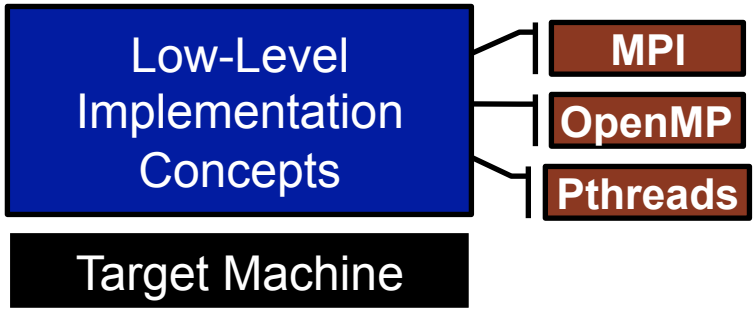
- **Styles:** data-parallel, task-parallel, concurrency, nested, …
- **Levels:** model, function, loop, statement, expression

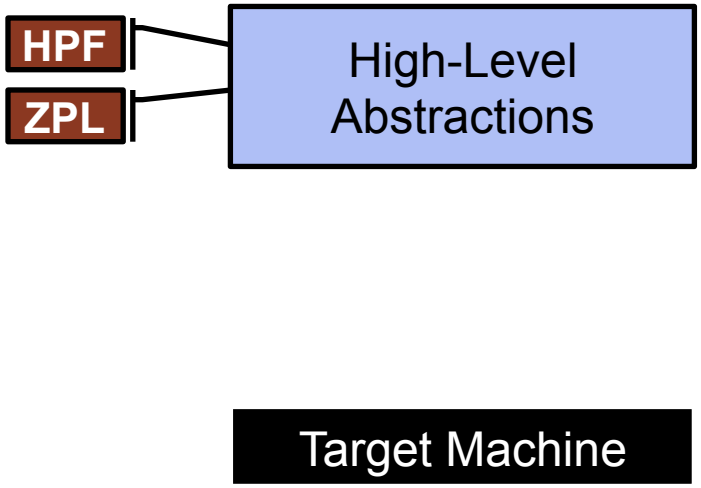
...target any parallelism available in the hardware

- **Types:** machines, nodes, cores, instruction

Type of HW Parallelism	Programming Model	Unit of Parallelism
Inter-node	Chapel	task (or executable)
Intra-node/multicore	Chapel	iteration/task
Instruction-level vectors/threads	Chapel	iteration
GPU/accelerator	Chapel	SIMD function/task

3) Multiresolution Design: Motivation





"Why is everything so tedious/difficult?"
"Why don't my programs port trivially?"

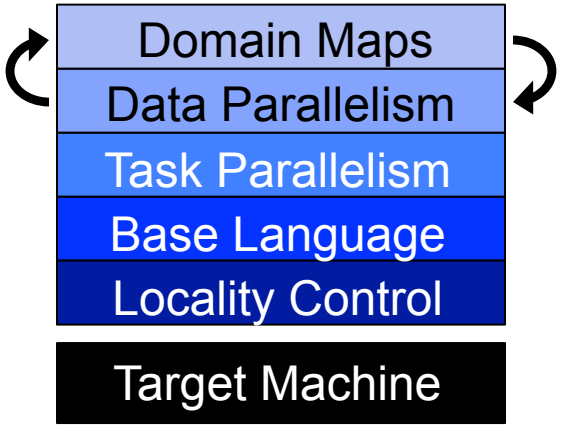
"Why don't I have more control?"

3) Multiresolution Design

Multiresolution Design: **Support multiple tiers of features**

- higher levels for programmability, productivity
- lower levels for greater degrees of control

Chapel language concepts



- build the higher-level concepts in terms of the lower
- permit the user to intermix layers arbitrarily

5) Reduce HPC ↔ Mainstream Language Gap

Consider:

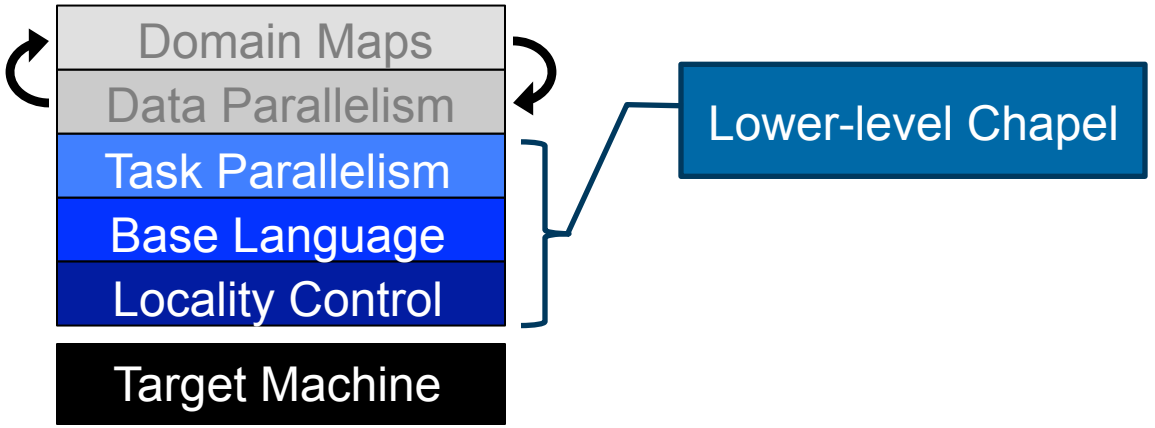
- Students graduate with training in Java, Matlab, Python, etc.
- Yet HPC programming is dominated by Fortran, C/C++, MPI

We'd like to narrow this gulf with Chapel:

- to leverage advances in modern language design
- to better utilize the skills of the entry-level workforce...
- ...while not alienating the traditional HPC programmer
  - e.g., support object-oriented programming, but make it optional

Outline

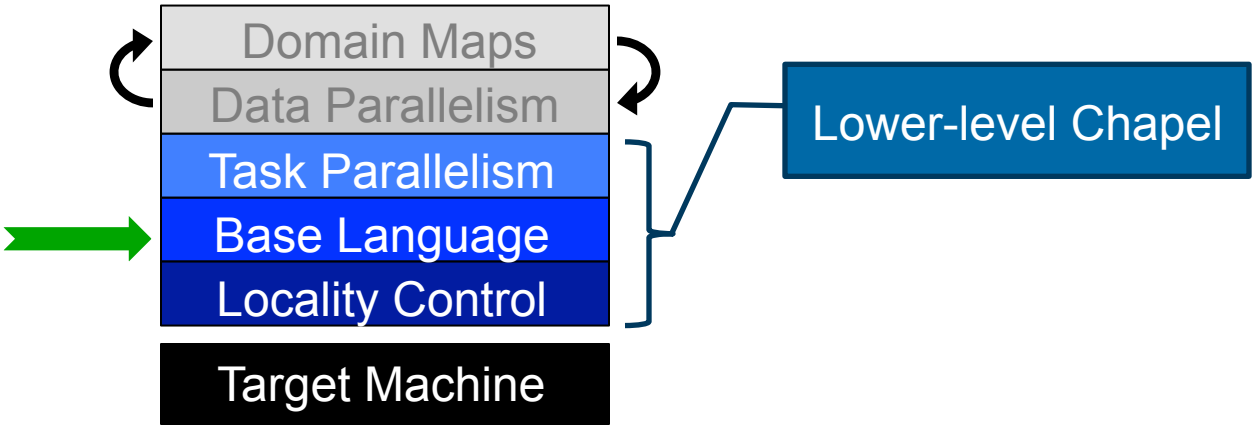
- ✓ Motivation
- ✓ Chapel Background and Themes
- ➢ Survey of Chapel Concepts



- Project Status and Next Steps

Outline

- ✓ Motivation
- ✓ Chapel Background and Themes
- ➢ Survey of Chapel Concepts



- Project Status and Next Steps

Static Type Inference

```
const pi = 3.14,              // pi is a real
      coord = 1.2 + 3.4i,     // coord is a complex…
      coord2 = pi*coord,      // …as is coord2
      name = "brad",          // name is a string
      verbose = false;        // verbose is boolean

proc addem(x, y) {            // addem() has generic arguments
  return x + y;               //   and an inferred return type
}

var sum = addem(1, pi),                  // sum is a real
    fullname = addem(name, "ford");      // fullname is a string

writeln((sum, fullname));
```

```
(4.14, bradford)
```

Range Types and Algebra

```
const r = 1..10;

printVals(r);
printVals(r # 3);
printVals(r by 2);
printVals(r by -2);
printVals(r by 2 # 3);
printVals(r # 3 by 2);
printVals(0.. #n);

proc printVals(r) {
  for i in r do
    write(r, " ");
  writeln();
}
```

```
1 2 3 4 5 6 7 8 9 10
1 2 3
1 3 5 7 9
10 8 6 4 2
1 3 5
1 3
0 1 2 3 4 … n-1
```

Iterators

```
iter fibonacci(n) {
  var current = 0,
      next = 1;
  for 1..n {
    yield current;
    current += next;
    current <=> next;
  }
}
```

```
for f in fibonacci(7) do
  writeln(f);
```

```
0
1
1
2
3
5
8
```

```
iter tiledRMO(D, tilesize) {
  const tile = {0..#tilesize,
                0..#tilesize};
  for base in D by tilesize do
    for ij in D[tile + base] do
      yield ij;
}
```

```
for ij in tiledRMO({1..m, 1..n}, 2) do
  write(ij);
```

```
(1,1) (1,2) (2,1) (2,2)
(1,3) (1,4) (2,3) (2,4)
(1,5) (1,6) (2,5) (2,6)
…
(3,1) (3,2) (4,1) (4,2)
```

Zippered Iteration

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

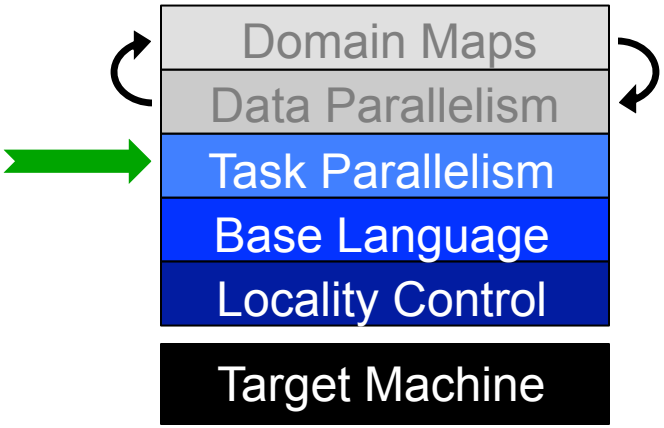
```
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
…
```

Other Base Language Features

- tuple types and values
- rank-independent programming features
- interoperability features
- compile-time features for meta-programming
  - e.g., compile-time functions to compute types, parameters
- OOP (value- and reference-based)
- argument intents, default values, match-by-name
- overloading, where clauses
- modules (for namespace management)
- …

Outline

- ✓ Motivation
- ✓ Chapel Background and Themes
- ➢ Survey of Chapel Concepts



- Project Status and Next Steps

Defining our Terms

Task: a unit of computation that can/should execute in parallel with other tasks

Task Parallelism: a style of parallel programming in which parallelism is driven by programmer-specified tasks

(in contrast with):

Data Parallelism: a style of parallel programming in which parallelism is driven by computations over collections of data elements or their indices

Task Parallelism: Begin Statements

```
// create a fire-and-forget task for a statement
begin writeln("hello world");
writeln("goodbye");
```

Possible outputs:

```
hello world
goodbye
```

```
goodbye
hello world
```

Task Parallelism: Coforall Loops

```
// create a task per iteration
coforall t in 0..#numTasks {
  writeln("Hello from task ", t, " of ", numTasks);
} // implicit join of the numTasks tasks here

writeln("All tasks done");
```

Sample output:

```
Hello from task 2 of 4
Hello from task 0 of 4
Hello from task 3 of 4
Hello from task 1 of 4
All tasks done
```

Other Task Parallel Concepts

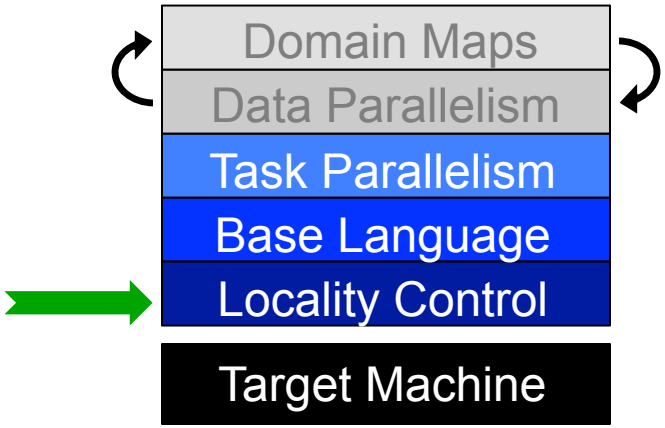
- **cobegins:** create tasks using compound statements

- **atomic variables:** support atomics ops, similar to modern C++
- **sync/single variables:** support producer/consumer patterns

- **sync statements:** join unstructured tasks
- **serial statements:** conditionally squash parallelism

Outline

- ✓ Motivation
- ✓ Chapel Background and Themes
- ➢ Survey of Chapel Concepts



Theme 4: Control over Locality/Affinity

- Project Status and Next Steps

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 = …;
```

- 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!");
```

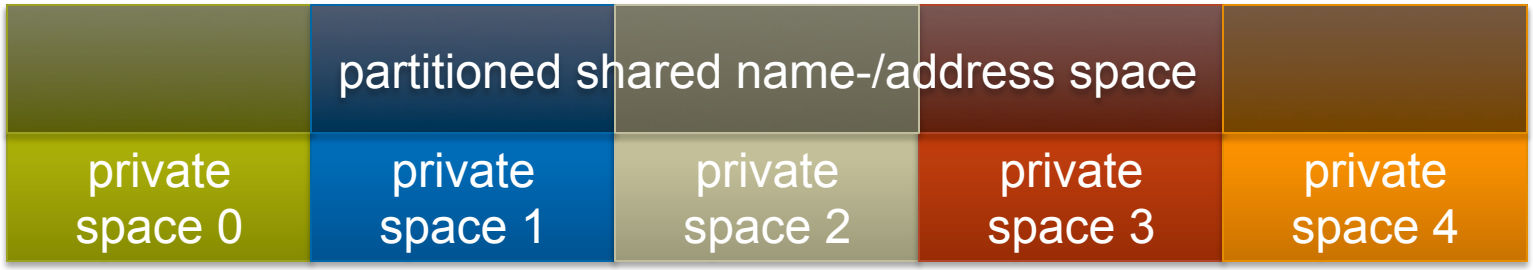
- This is a **distributed** and **parallel** program:

```
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

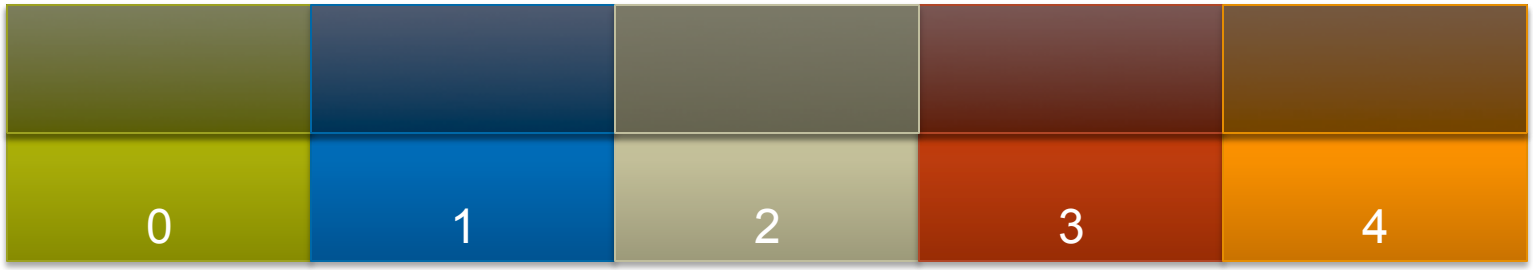
(Or perhaps: partitioned global namespace languages)

- **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



Chapel and PGAS

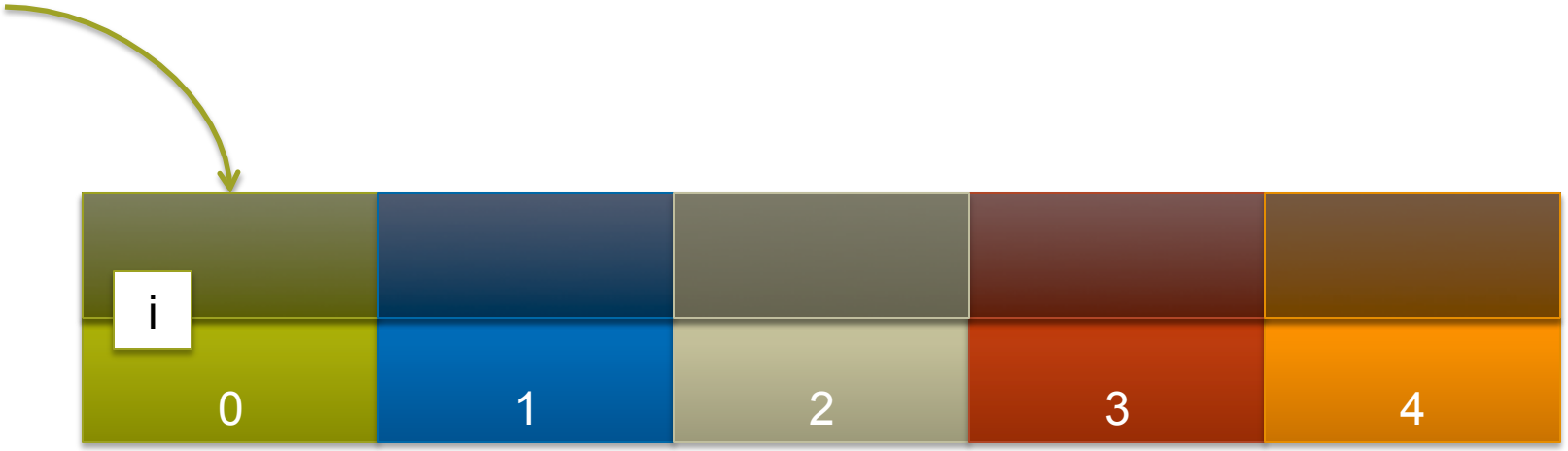
- **Chapel is PGAS, but unlike most, it's not inherently SPMD**
  - ⇒ never think about "the other copies of the program"
  - ⇒ "global name/address space" comes from lexical scoping
    - as in traditional languages, each declaration yields one variable
    - variables are stored on the locale where the task declaring it is executing



Locales (think: "compute nodes")

Chapel: Scoping and Locality

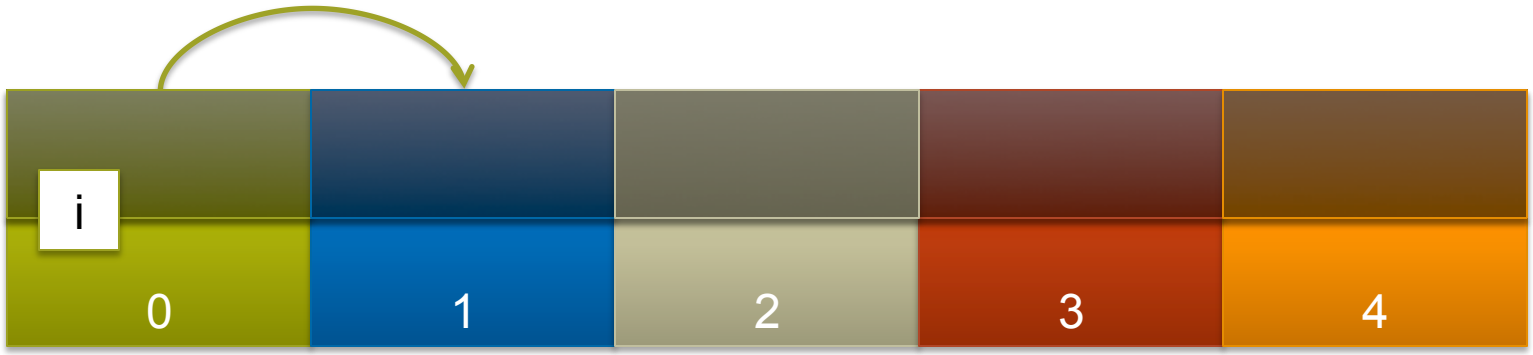
```
var i: int;
```



Locales (think: “compute nodes”)

Chapel: Scoping and Locality

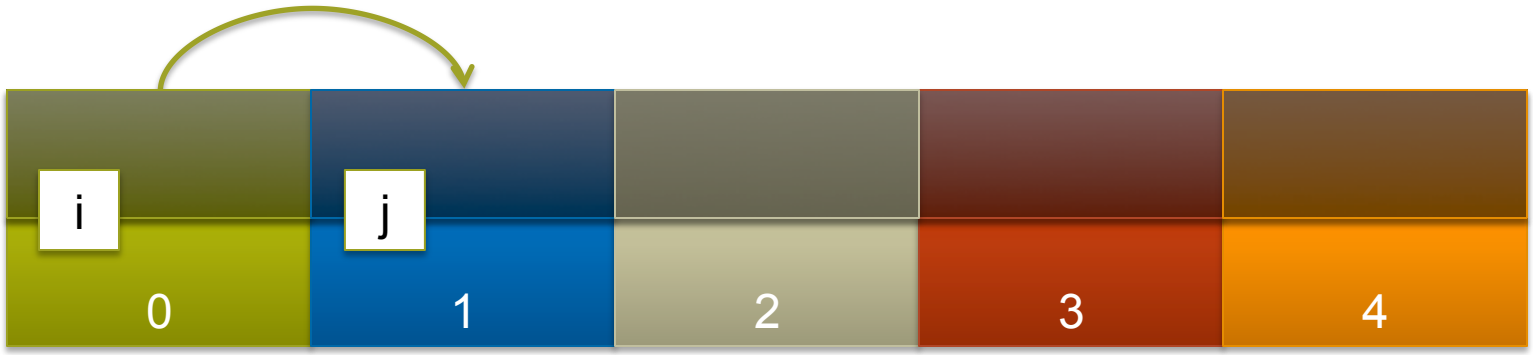
```
var i: int;
on Locales[1] {
```



Locales (think: “compute nodes”)

Chapel: Scoping and Locality

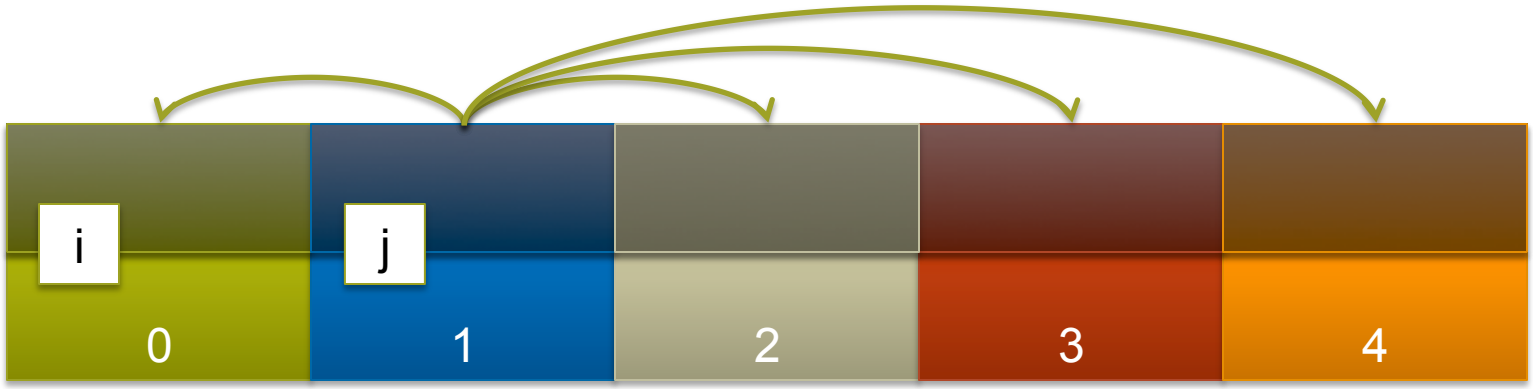
```
var i: int;
on Locales[1] {
  var j: int;
```



Locales (think: “compute nodes”)

Chapel: Scoping and Locality

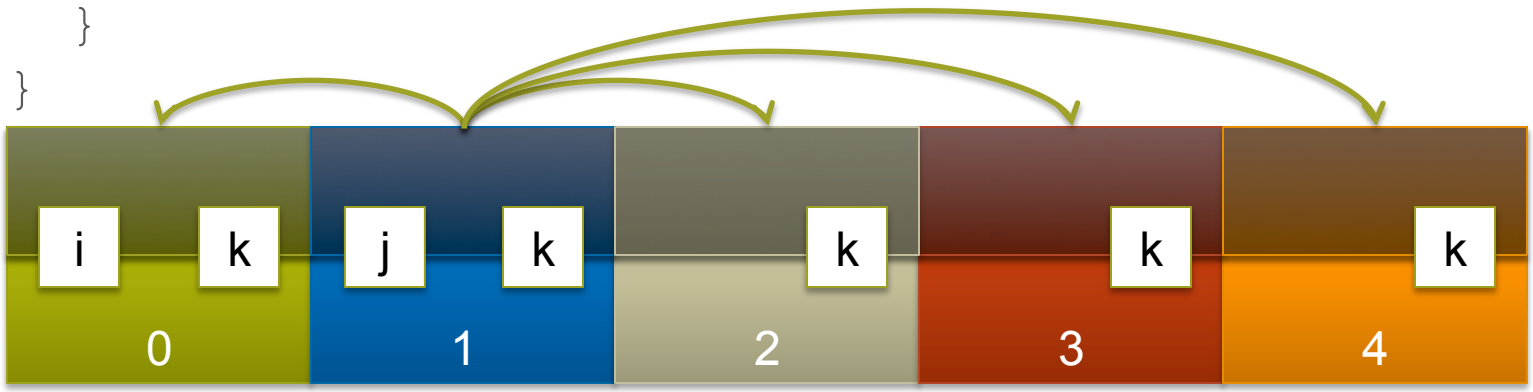
```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
```



Locales (think: “compute nodes”)

Chapel: Scoping and Locality

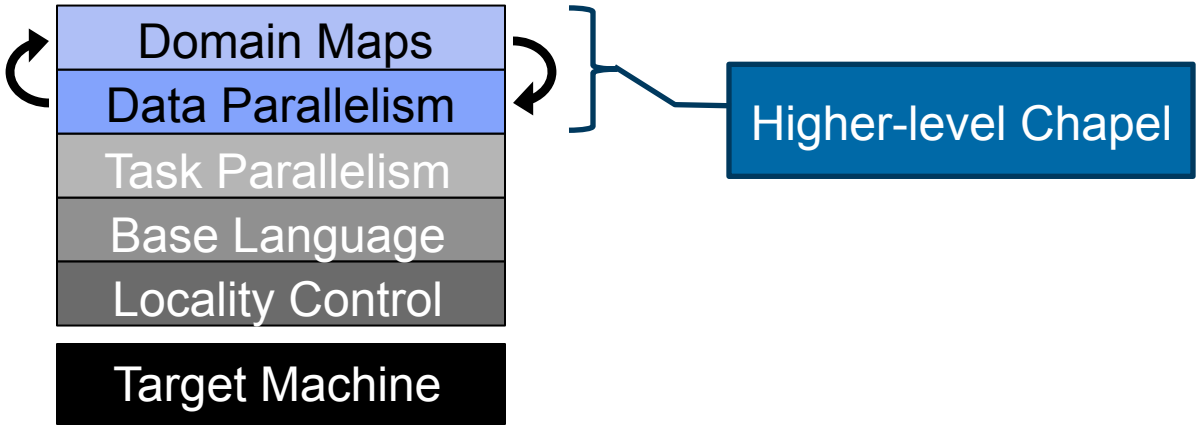
```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      var k: int;

      // within this scope, i, j, and k can be referenced;
      // the implementation manages the communication for i and j
    }
  }
}
```



Locales (think: "compute nodes")

Outline

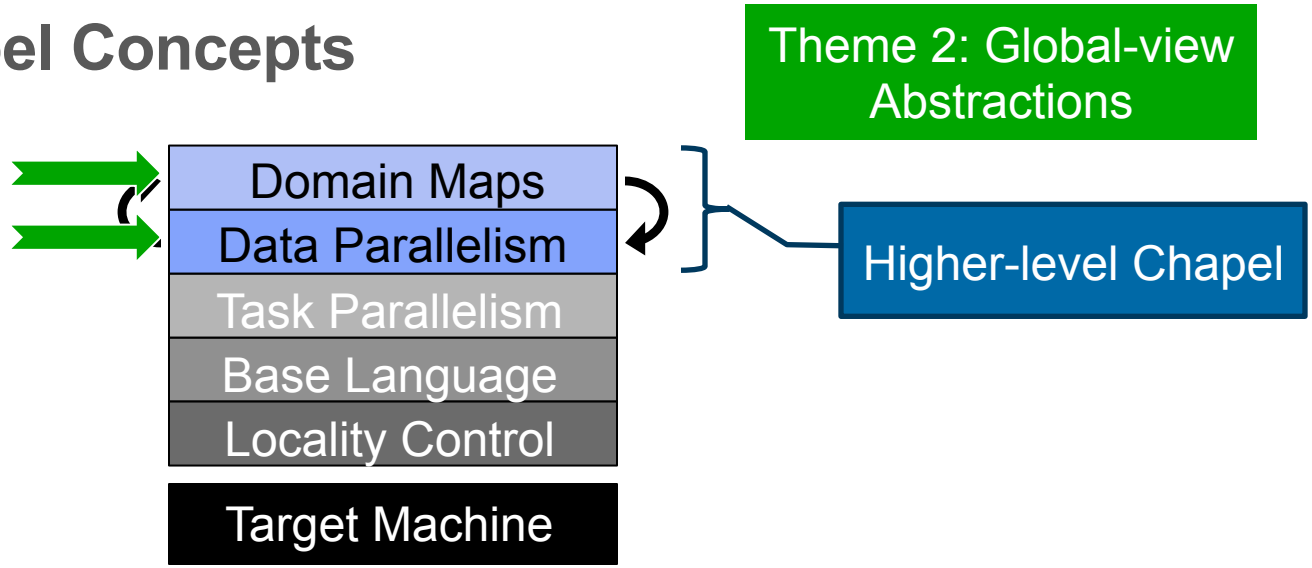
- ✓ Motivation
- ✓ Chapel Background and Themes
- ➢ Survey of Chapel Concepts



- Project Status and Next Steps

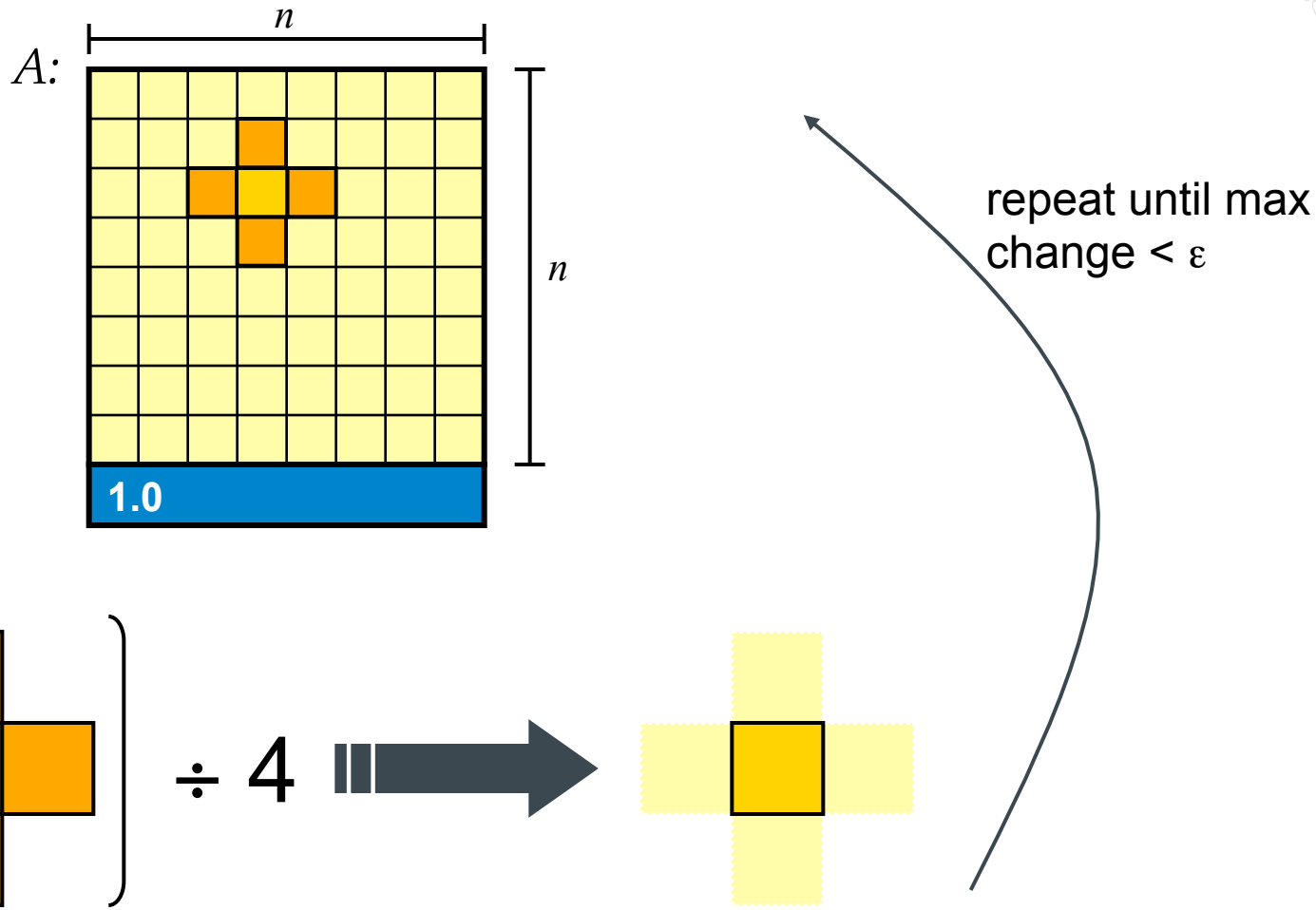
Outline

- ✓ Motivation
- ✓ Chapel Background and Themes
- ➢ Survey of Chapel Concepts



- Project Status and Next Steps

Data Parallelism by Example: Jacobi Iteration



Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
          D = BigD[1..n, 1..n],
   LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
          D = BigD[1..n, 1..n],
   LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[La

do {
  fo
                                    ]) / 4;
  co
  A[
} wh

writ
```

Declare program parameters

const ⇒ can't change values after initialization

config ⇒ can be set on executable command-line

```
prompt> jacobi --n=10000 --epsilon=0.0001
```

note that no types are given; they're inferred from initializers

n ⇒ **default integer** (64 bits)

epsilon ⇒ **default real floating-point** (64 bits)

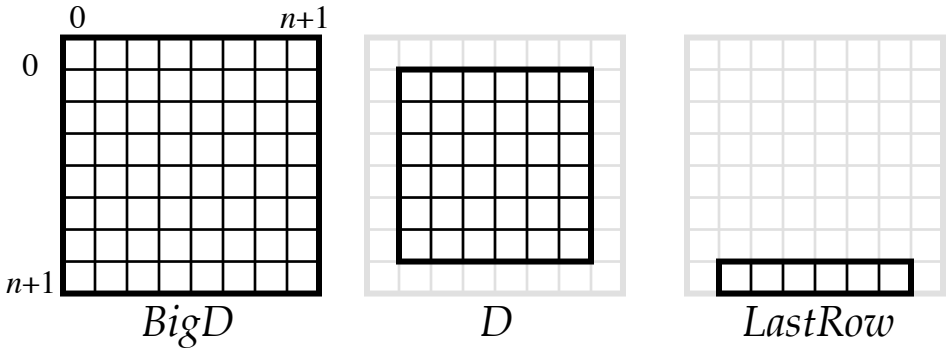
Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
          D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);
```

Declare domains (first class index sets)

{lo..hi, lo2..hi2} ⇒ 2D rectangular domain, with 2-tuple indices

Dom1[Dom2] ⇒ computes the intersection of two domains



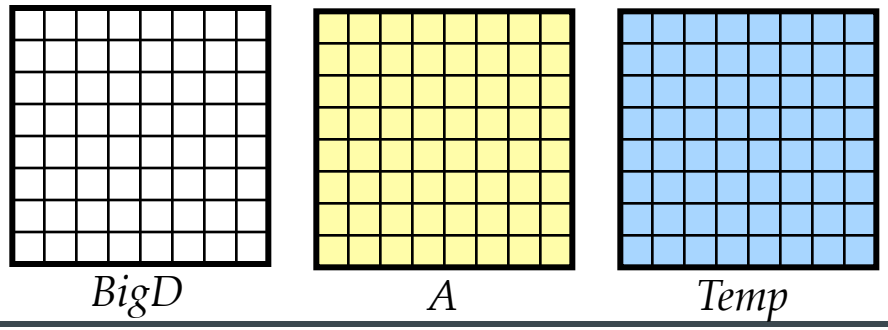
.exterior() ⇒ one of several built-in domain generators

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
          D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;
```

Declare arrays

var ⇒ can be modified throughout its lifetime
: [*Dom*] T ⇒ array of size *Dom* with elements of type *T*
(no initializer) ⇒ values initialized to default value (0.0 for reals)

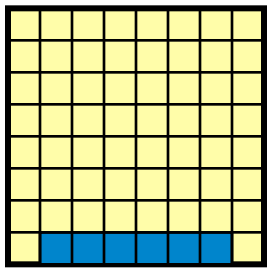


Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
          D = BigD[1..n, 1..n],
   LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do

                                          j+1]) / 4;

}

wr
```

Set Explicit Boundary Condition

Arr[Dom] ⇒ refer to array slice (“forall i in Dom do …Arr[i]…”)



A

Jacobi Iteration in Chapel

```
config const n = 6,
```

Compute 5-point stencil

forall ***ind*** **in** ***Dom*** ⇒ parallel forall expression over *Dom*'s indices, binding them to *ind*
(here, since *Dom* is 2D, we can de-tuple the indices)

```
do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;
```

Compute maximum change

***op* reduce** ⇒ collapse aggregate expression to scalar using *op*

Promotion: *abs()* and – are scalar operators; providing array operands results in parallel evaluation equivalent to:

```
forall (a,t) in zip(A,Temp) do abs(a - t)
```

```
do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
      D = BigD[1..n, 1..n],
```

Copy data back & Repeat until done

uses slicing and whole array assignment
standard *do…while* loop construct

```
do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
          D = BigD[1..n, 1..n],
   LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Write array to console

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
          D = BigD[1..n, 1..n],
   LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;
```

By default, domains and their arrays are mapped to a single locale.
Any data parallelism over such domains/ arrays will be executed by the cores on that locale.
Thus, this is a shared-memory parallel program.

```
    Temp[1,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

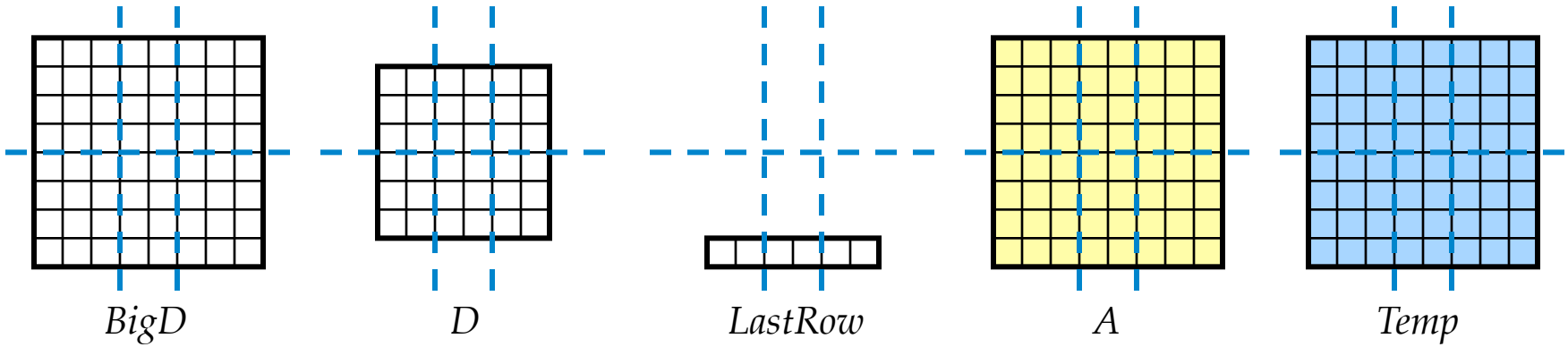
Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1} dmapped Block({1..n, 1..n}),
          D = BigD[1..n, 1..n],
   LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;
```

With this simple change, we specify a mapping from the domains and arrays to locales

Domain maps describe the mapping of domain indices and array elements to *locales*

- specifies how array data is distributed across locales
- specifies how iterations over domains/arrays are mapped to locales



Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1} dmapped Block({1..n, 1..n}),
          D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
  forall (i,j) in D do
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);

use BlockDist;
```

STREAM Triad: Chapel

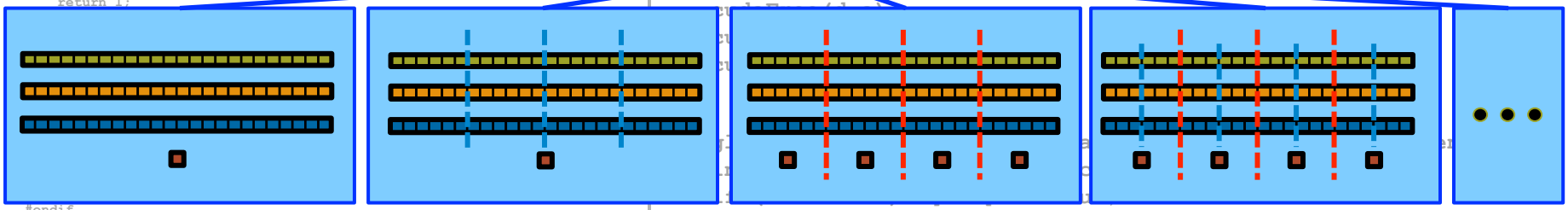
MPI + OpenMP

```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Comm_size( comm, &commSize );
  MPI_Comm_rank( comm, &myRank );

  rv = HPCC_Stream( params, 0 == myR
  MPI_Reduce( &rv, &errCount, 1, MPI

  return errCount;
}

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

  VectorSize = HPCC_LocalVectorSize(

  a = HPCC_XMALLOC( double, VectorSi
  b = HPCC_XMALLOC( double, VectorSi
  c = HPCC_XMALLOC( double, VectorSi

  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
      fprintf( outFile, "Failed to a
      fclose( outFile );
    }
    return 1;
```

Chapel

```
config const m = 1000,
             alpha = 3.0;

const ProblemSpace = {1..m} dmapped …;

var A, B, C: [ProblemSpace] real;

B = 2.0;
C = 3.0;

A = B + alpha * C;
```

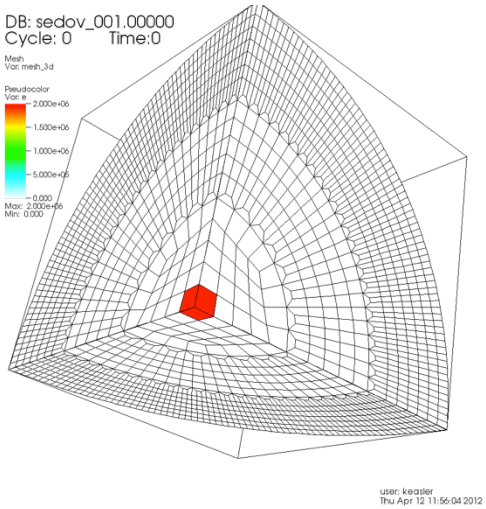
the special sauce

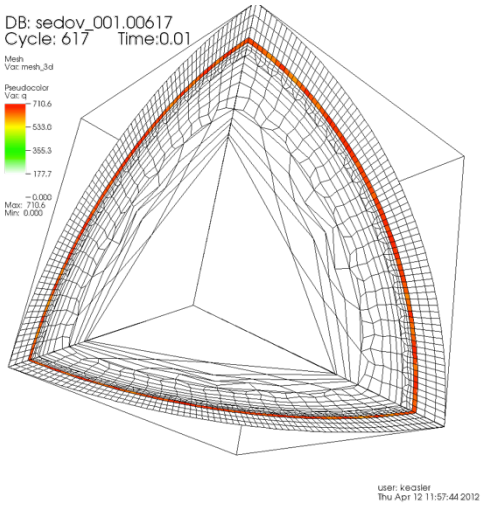


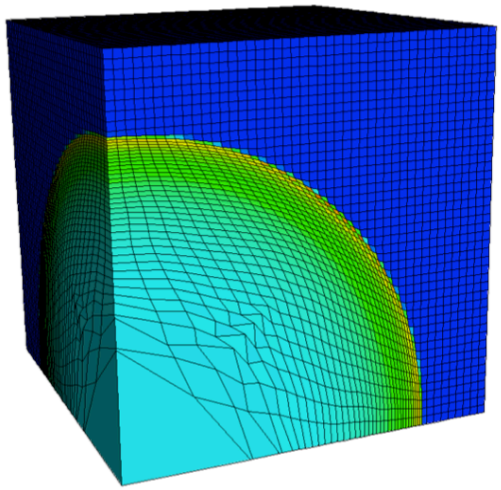
Philosophy: Good language design can tease details of locality and parallelism away from an algorithm, permitting the compiler, runtime, applied scientist, and HPC expert to each focus on their strengths.

LULESH: a DOE Proxy Application

Goal: Solve one octant of the spherical Sedov problem (blast wave) using Lagrangian hydrodynamics for a single material



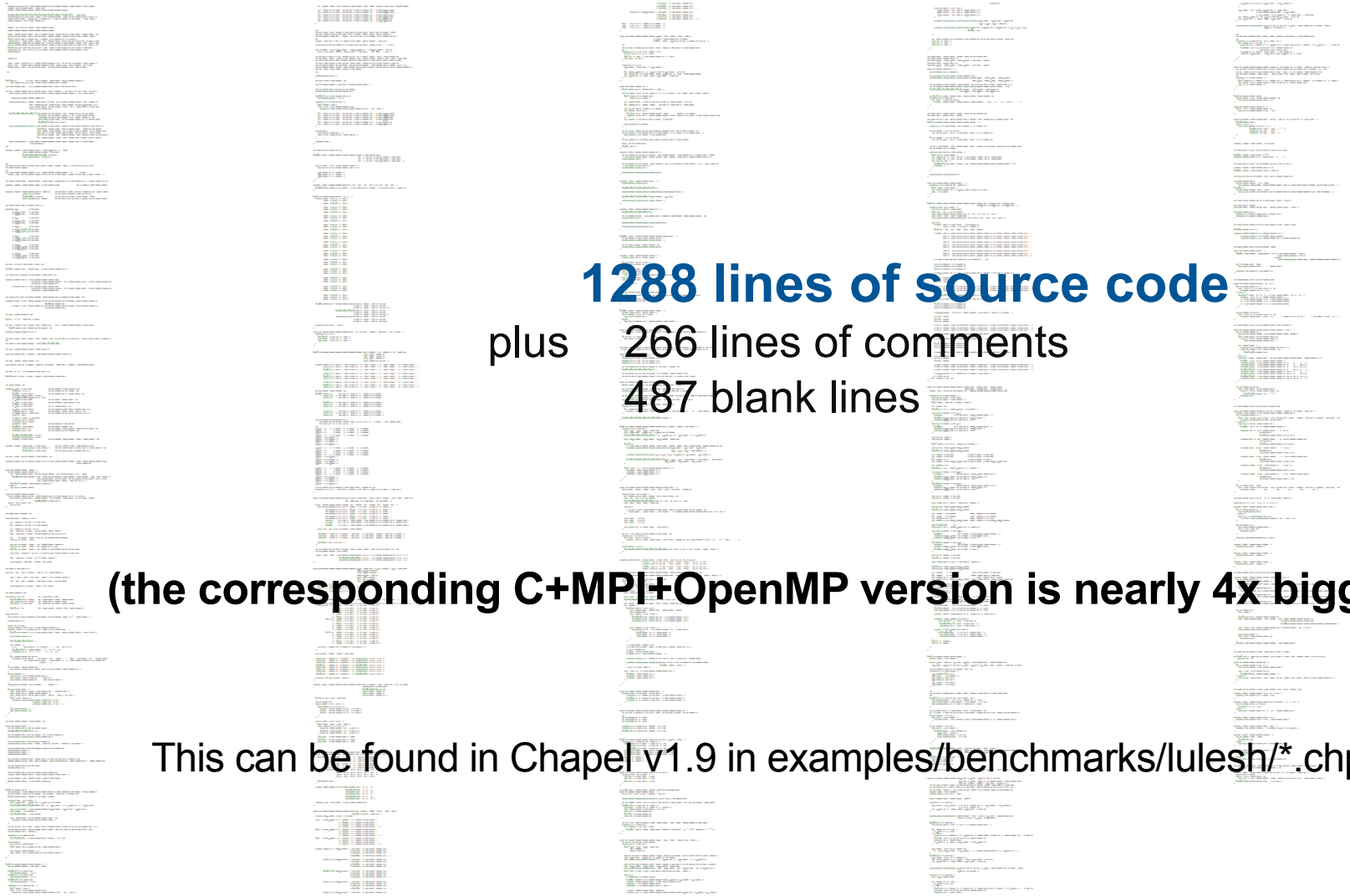




pictures courtesy of Rob Neely, Bert Still, Jeff Keasler, LLNL

LULESH in Chapel

LULESH in Chapel



1288 lines of source code

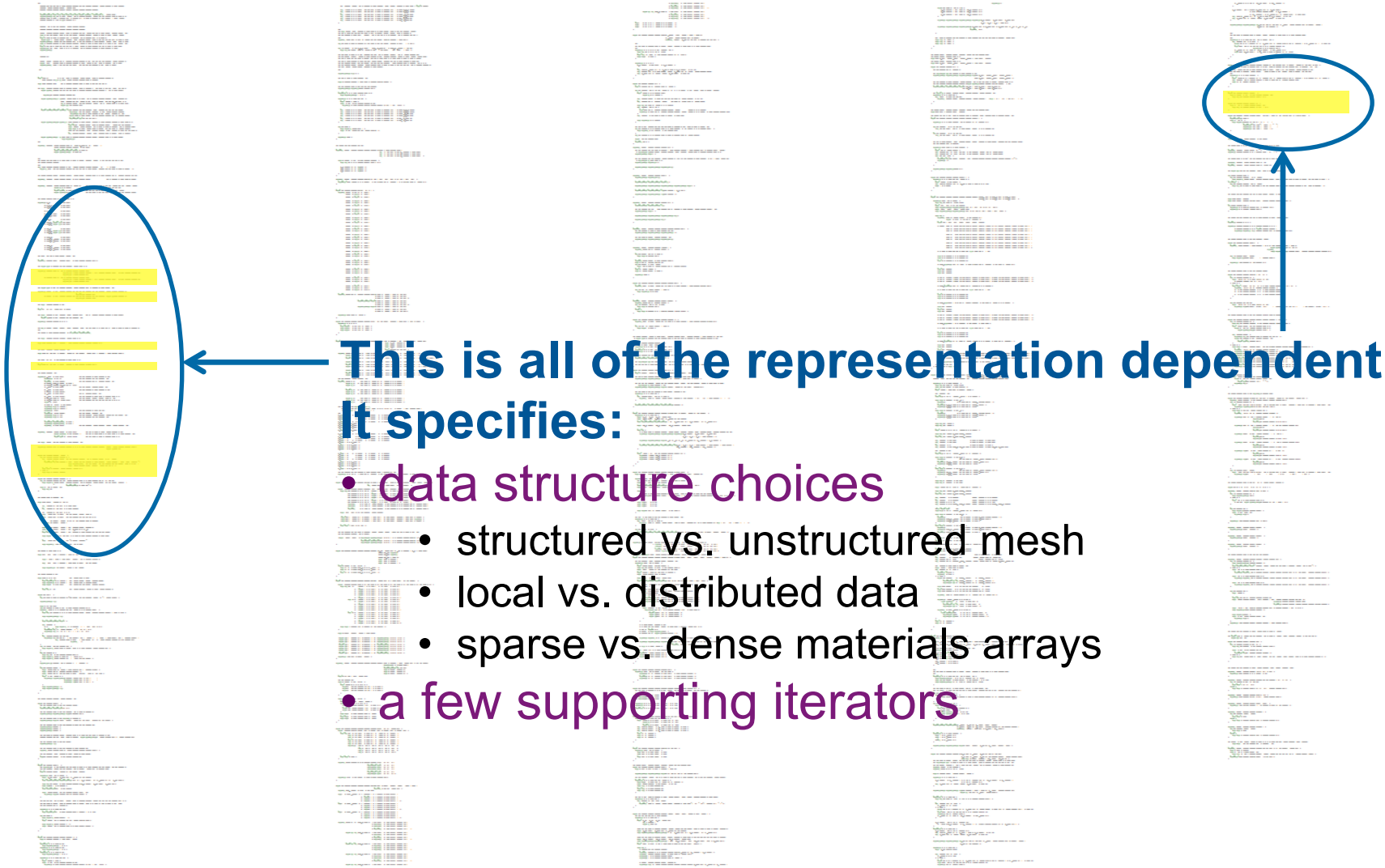
plus 266 lines of comments

487 blank lines

(the corresponding C+MPI+OpenMP version is nearly 4x bigger)

This can be found in Chapel v1.9 in examples/benchmarks/lulesh/*.chpl

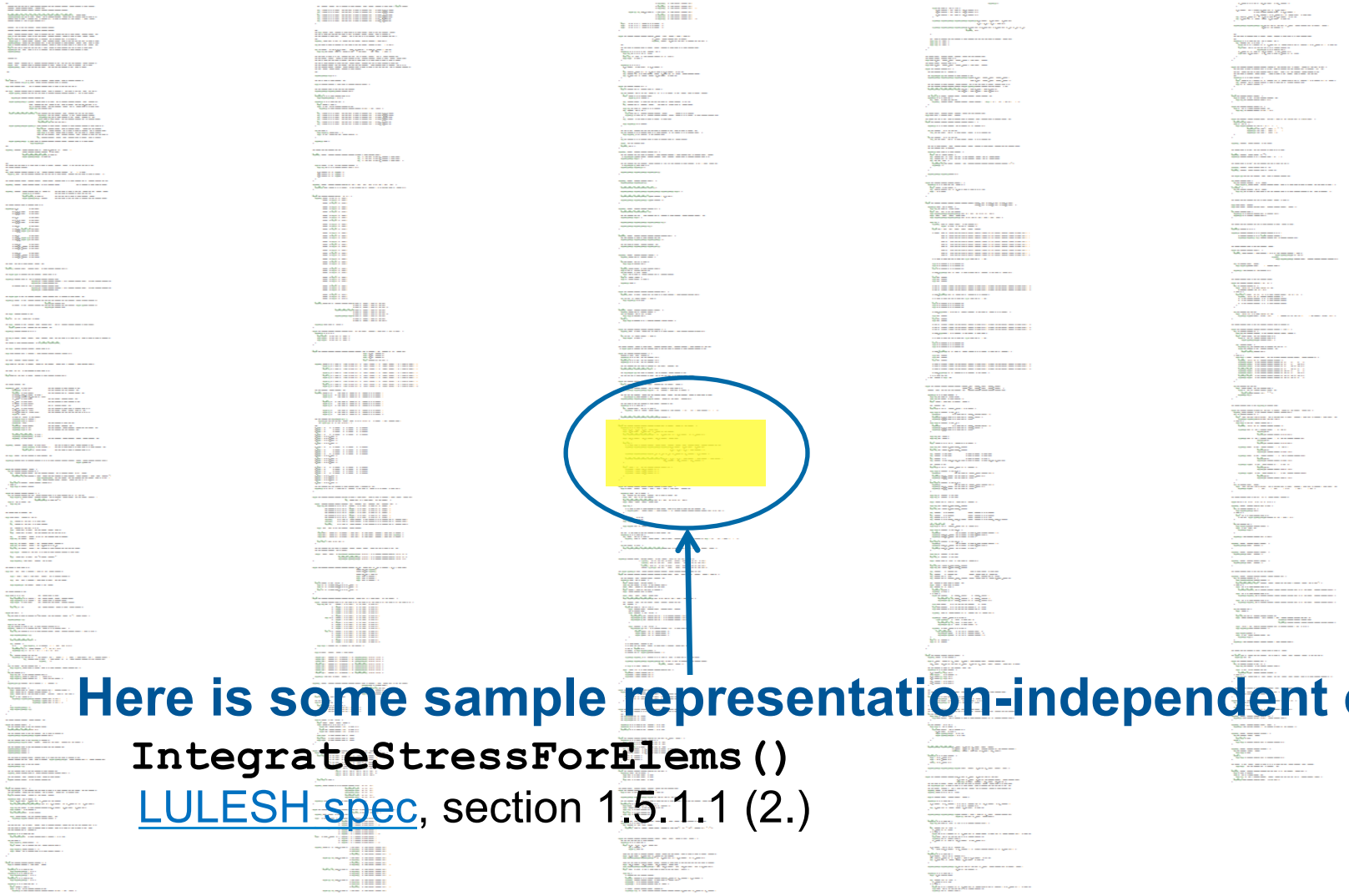
LULESH in Chapel



This is all of the representation dependent code. It specifies:

- data structure choices
  - structured vs. unstructured mesh
  - local vs. distributed data
  - sparse vs. dense materials arrays
- a few supporting iterators

LULESH in Chapel



Here is some sample representation-independent code

`IntegrateStressForElems()`

LULESH spec, section 1.5.1.1 (2.)

Representation-Independent Physics

```
proc IntegrateStressForElems(sigxx, sigyy, sigzz, determ) {
  forall k in Elems {
    var b_x, b_y, b_z: 8*real;
    var x_local, y_local, z_local: 8*real;
    localizeNeighborNodes(k, x, x_local, y, y_local, z, z_local);

    var fx_local, fy_local, fz_local: 8*real;

    local {
      /* Volume calculation involves extra work for numerical consistency. */
      CalcElemShapeFunctionDerivatives(x_local, y_local, z_local,
                                       b_x, b_y, b_z, determ[k]);

      CalcElemNodeNormals(b_x, b_y, b_z, x_local, y_local, z_local);

      SumElemStressesToNodeForces(b_x, b_y, b_z, sigxx[k], sigyy[k], sigzz[k],
                                  fx_local, fy_local, fz_local);
    }

    for (noi, t) in elemToNodesTuple(k) {
      fx[noi].add(fx_local[t]);
      fy[noi].add(fy_local[t]);
      fz[noi].add(fz_local[t]);
    }
  }
}
```

parallel loop over elements

collect nodes neighboring this element; localize node fields

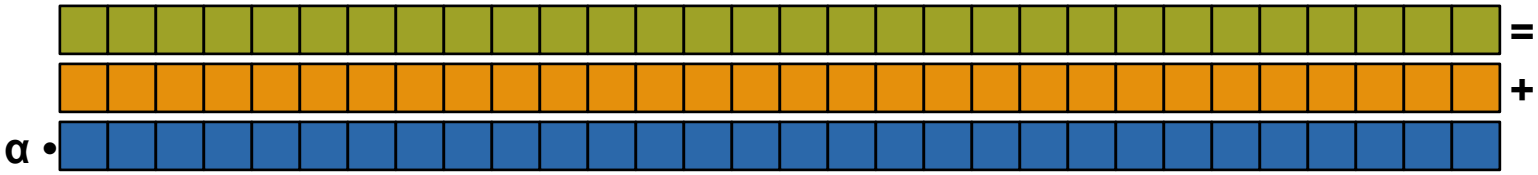
update node forces from element stresses

Because of domain maps, this code is independent of:

- structured vs. unstructured mesh
- shared vs. distributed data
- sparse vs. dense representation

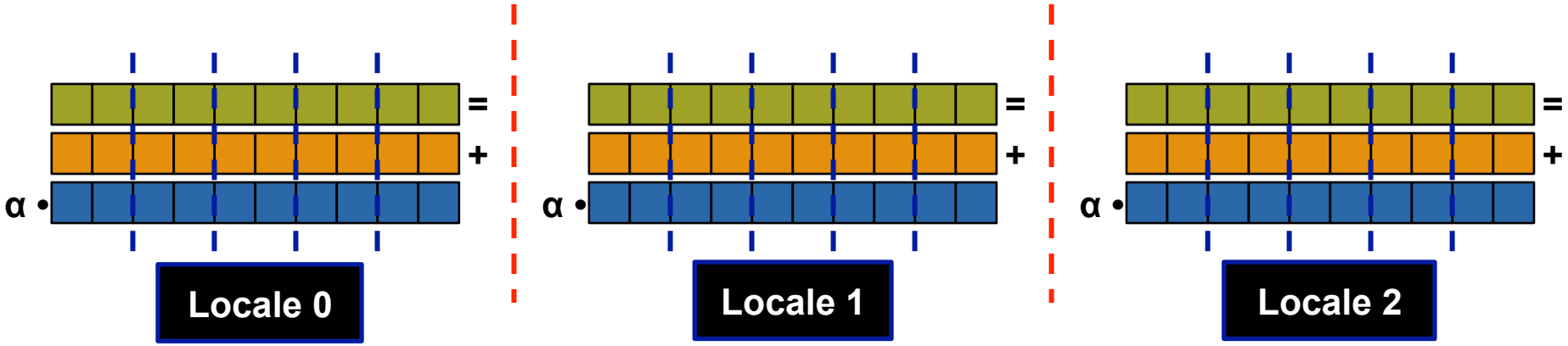
Domain Maps

Domain maps are "recipes" that instruct the compiler how to map the global view of a computation…




```
A = B + alpha * C;
```

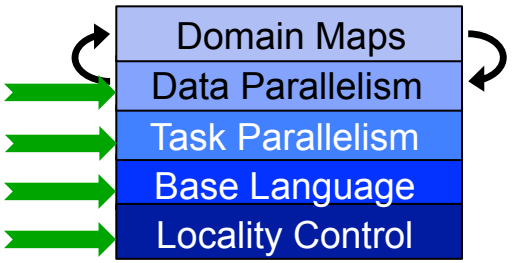
…to the target locales' memory and processors:



Chapel's Domain Map Philosophy

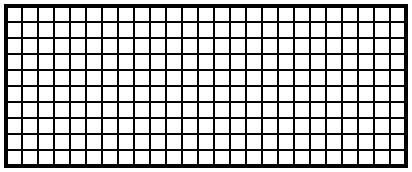
1. **Chapel provides a library of standard domain maps**
   - to support common array implementations effortlessly

2. **Expert users can write their own domain maps in Chapel**
   - to cope with any shortcomings in our standard library

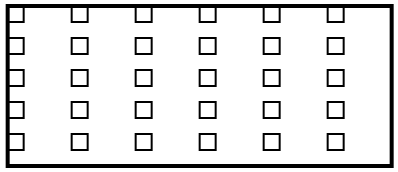


3. **Chapel's standard domain maps are written using the same end-user framework**
   - to avoid a performance cliff between "built-in" and user-defined cases

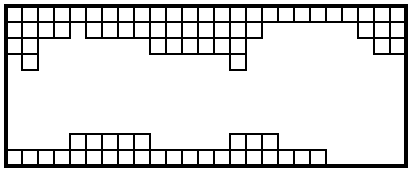
Chapel Domain Types



dense

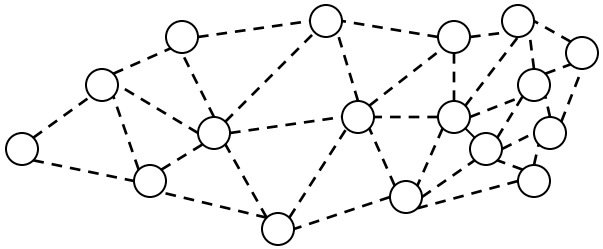


strided



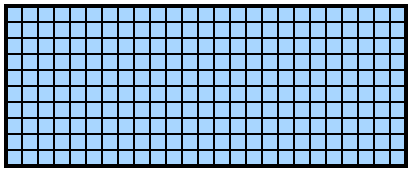
sparse

associative

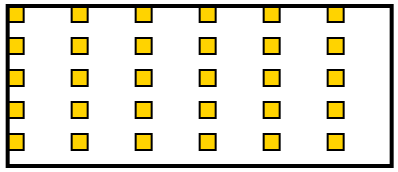


unstructured

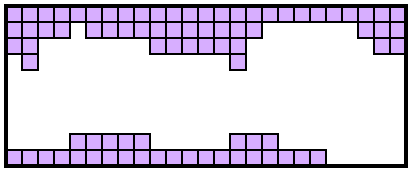
Chapel Array Types



dense

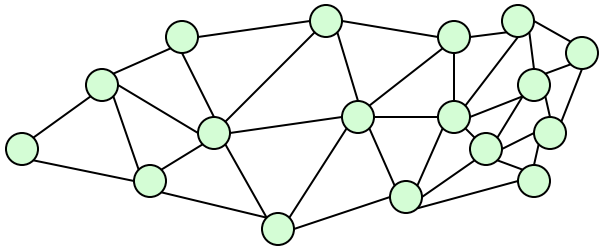


strided



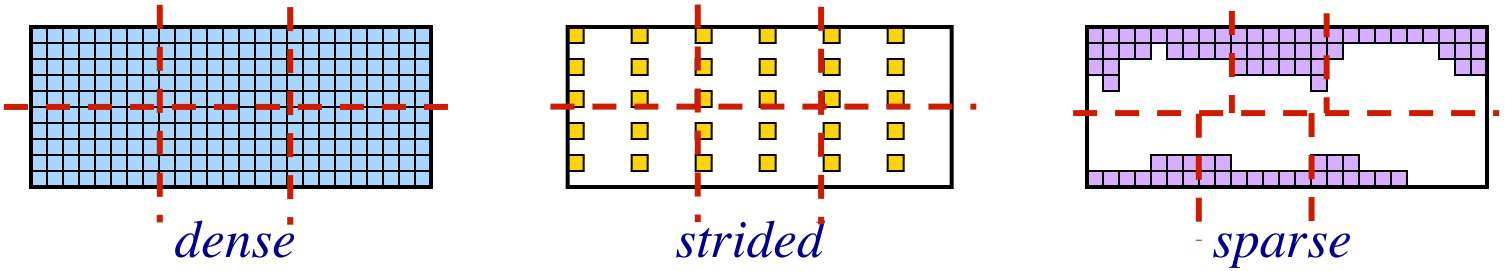
sparse

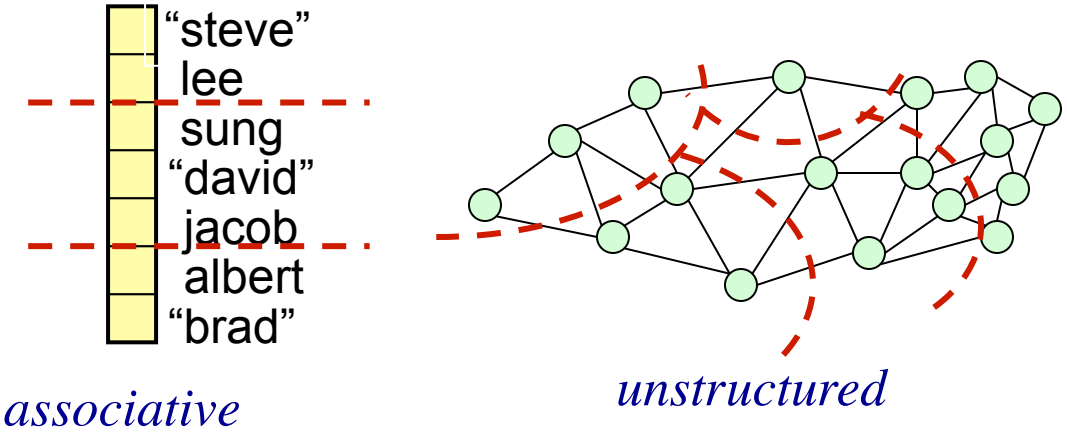
associative



unstructured

All Domain Types Support Domain Maps





For More Information on Domain Maps

HotPAR'10: *User-Defined Distributions and Layouts in Chapel: Philosophy and Framework*
Chamberlain, Deitz, Iten, Choi; June 2010

CUG 2011: *Authoring User-Defined Domain Maps in Chapel*
Chamberlain, Choi, Deitz, Iten, Litvinov; May 2011

Chapel release:

- Current domain maps:
  $CHPL_HOME/modules/dists/*.chpl
  layouts/*.chpl
  internal/Default*.chpl
- Technical notes detailing the domain map interface for implementers:
  $CHPL_HOME/doc/technotes/README.dsi

Two Other Thematically Similar Features

1) **parallel iterators:** Permit users to specify the parallelism and work decomposition used by forall loops
   - including zippered forall loops

2) **locale models:** Permit users to model the target architecture and how Chapel should be implemented on it
   - e.g., how to manage memory, create tasks, communicate, …

Like domain maps, these are…

…written in Chapel by expert users using lower-level features
- e.g., task parallelism, on-clauses, base language features, …

…available to the end-user via higher-level abstractions
- e.g., forall loops, on-clauses, lexically scoped PGAS memory, …

Summary

HPC programmers deserve better programming models

Higher-level programming models can help insulate algorithms from parallel implementation details

- yet, without necessarily abdicating control
- Chapel does this via its multiresolution design
  - domain maps, parallel iterators, and locale models are all examples
  - avoids locking crucial policy decisions into the language definition

We believe Chapel can greatly improve productivity

…for current and emerging HPC architectures

…for HPC users and mainstream uses of parallelism at scale

Outline

- ✓ Motivation
- ✓ Chapel Background and Themes
- ✓ Survey of Chapel Concepts
- ➢ **Project Status and Next Steps**

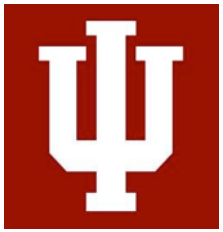
Chapel's 5-year push

- **Based on positive user response to Chapel under HPCS, Cray undertook a five-year effort to improve it**
  - we've just started our third year
- **Focus Areas:**
  1. Improving **performance** and scaling
  2. **Fixing** immature aspects of the language and implementation
     - e.g., strings, memory management, error handling, …
  3. **Porting** to emerging architectures
     - Intel Xeon Phi, accelerators, heterogeneous processors and memories, …
  4. Improving **interoperability**
  5. Growing the Chapel user and developer **community**
     - including non-scientific computing communities
  6. Exploring transition of Chapel **governance** to a neutral, external body

The Chapel Team at Cray (Spring 2015)

Chapel is a Collaborative, Community Effort

Lawrence Berkeley
National Laboratory



(and many others as well…)

http://chapel.cray.com/collaborations.html

A Year in the Life of Chapel

- **Two major releases per year** (April / October)
  - **~a month later:** detailed release notes
- **SC** (Nov)
  - annual **Lightning Talks BoF** featuring talks from the community
  - annual **CHUG (Chapel Users Group) happy hour**
  - plus tutorials, panels, BoFs, posters, educator sessions, exhibits, …
- **CHIUW:** Chapel Implementers and Users Workshop (May/June)
  - CHIUW 2014 held at IPDPS (Phoenix, AZ)
  - CHIUW 2015 held at PLDI/FCRC (Portland, OR)
- **Talks, tutorials, research visits, blog posts, …** (year-round)

Implementation Status -- Version 1.11.0 (Apr 2015)

Overall Status:

- **User-facing Features:** generally in good shape
  - some receiving additional attention (e.g., strings, OOP, errors)
- **Multiresolution Features:** in use today
  - their interfaces are likely to continue evolving over time
- **Error Messages:** not always as helpful as one would like
  - correct code tends to work well, incorrect code can be puzzling
- **Performance:** hit-or-miss depending on the idioms used
  - ultimately, Chapel will support competitive performance
  - effort to-date has focused primarily on correctness

This is a great time to:

- Try out the language and compiler
- Use Chapel for non-performance-critical projects
- Give us feedback to improve Chapel
- Use Chapel for parallel programming education

Chapel and Education

- **When teaching parallel programming, I like to cover:**
  - data parallelism
  - task parallelism
  - concurrency
  - synchronization
  - locality/affinity
  - deadlock, livelock, and other pitfalls
  - performance tuning
  - …

- **I don't think there's been a good language out there…**
  - for teaching *all* of these things
  - for teaching *some* of these things well at all
  - ***until now:*** We believe Chapel can play a crucial role here

(see http://chapel.cray.com/education.html for more information and http://cs.washington.edu/education/courses/csep524/13wi/ for my use of Chapel in class)

Suggested Reading

Overview Papers:

- *A Brief Overview of Chapel*, Chamberlain (early draft of a chapter for *A Brief Overview of Parallel Programming Models*, edited by Pavan Balaji, to be published by MIT Press in 2015).
  - *a detailed overview of Chapel's history, motivating themes, features*

- *The State of the Chapel Union* [slides], Chamberlain, Choi, Dumler, Hildebrandt, Iten, Litvinov, Titus. CUG 2013, May 2013.
  - *a higher-level overview of the project, summarizing the HPCS period*

Lighter Reading

Blog Articles:

- *Chapel: Productive Parallel Programming*, Cray Blog, May 2013.
  - *a short-and-sweet introduction to Chapel*

- *Why Chapel?* (part 1, part 2, part 3), Cray Blog, June-October 2014.
  - *a recent series of articles answering common questions about why we are pursuing Chapel in spite of the inherent challenges*

- *[Ten] Myths About Scalable Programming Languages*, IEEE TCSC Blog (index available on chapel.cray.com "blog articles" page), April-November 2012.
  - *a series of technical opinion pieces designed to combat standard arguments against the development of high-level parallel languages*

Online Resources

Project page: http://chapel.cray.com

- overview, papers, presentations, language spec, …

GitHub page: https://github.com/chapel-lang

- download Chapel; browse source repository; contribute code

Facebook page: https://www.facebook.com/ChapelLanguage

Community Resources

SourceForge page: https://sourceforge.net/projects/chapel/

- hosts community mailing lists
  (also serves as an alternate release download site to GitHub)

Mailing Aliases:

write-only:

- chapel_info@cray.com: contact the team at Cray

read-only:

- chapel-announce@lists.sourceforge.net: read-only announcement list

read/write:

- chapel-users@lists.sourceforge.net: user-oriented discussion list
- chapel-developers@lists.sourceforge.net: developer discussion
- chapel-education@lists.sourceforge.net: educator discussion
- chapel-bugs@lists.sourceforge.net: public bug forum

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, URIKA, and YARCDATA. 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.

CRAY
THE SUPERCOMPUTER COMPANY

http://chapel.cray.com chapel_info@cray.com http://sourceforge.net/projects/chapel/