

# Chapel Overview

## CSCE 569 Parallel Computing

Department of Computer Science and Engineering

Yonghong Yan

yanyh@cse.sc.edu

<https://passlab.github.io/CSCE569/>

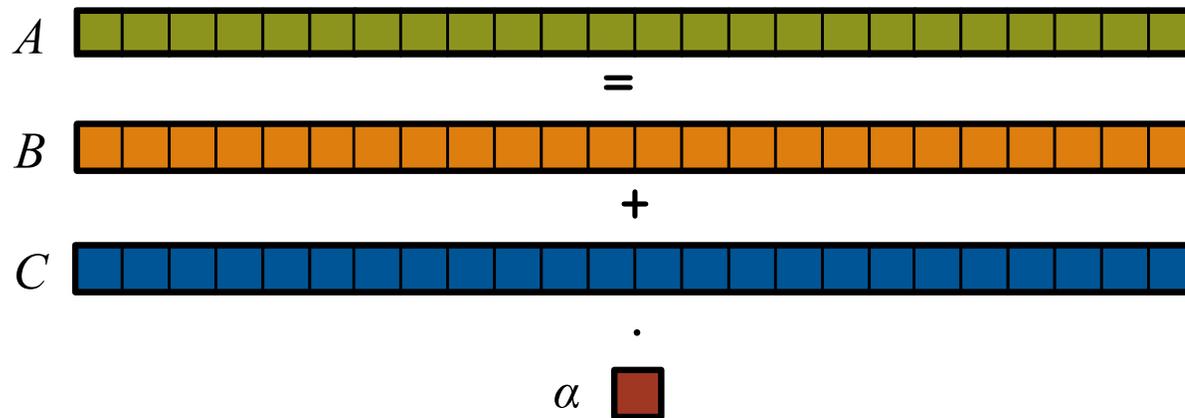
**Slides adapted from presentation by  
Brad Chamberlain, Chapel Team, Cray Inc.  
Intel Extreme Scale Technical Review Meeting  
November 11<sup>th</sup>, 2014**

# 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:**

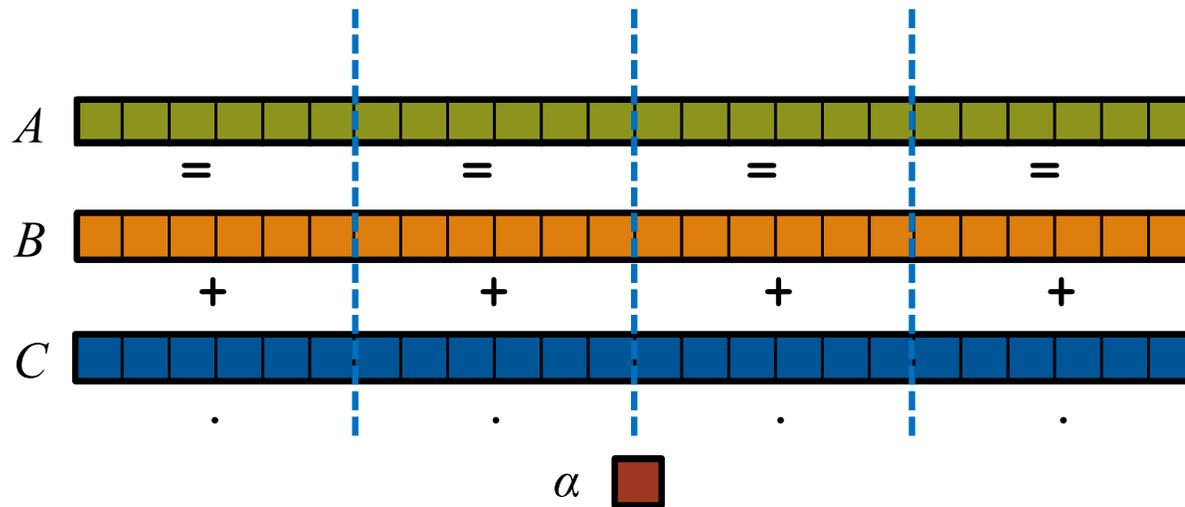


# STREAM Triad: a trivial parallel computation

**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:**

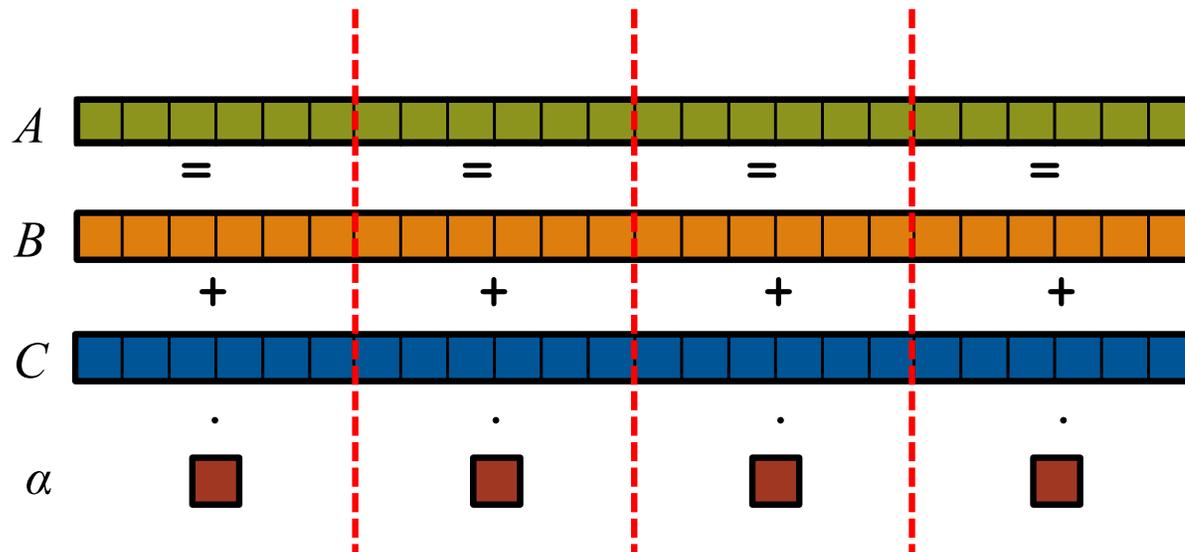


# STREAM Triad: a trivial parallel computation

**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):**

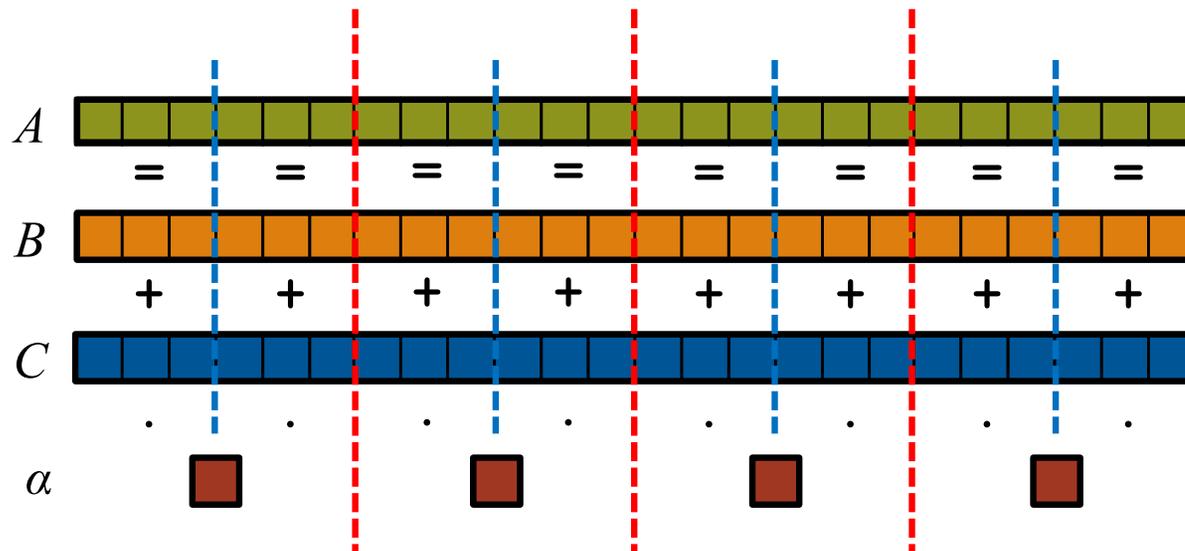


# STREAM Triad: a trivial parallel computation

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

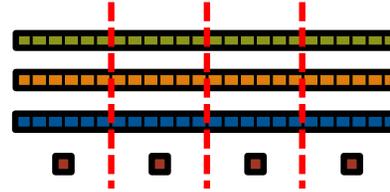
**Compute:**  $\forall i \in 1..m, A_i = B_i + \alpha \cdot C_i$

**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;  
}
```

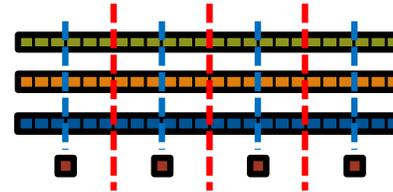
```
int HPCC_Stream(HPCC_Params *params, int doIO) {  
    register int j;  
    double scalar;  
  
    VectorSize = HPCC_LocalVectorSize( params, 3,  
                                        sizeof(double), 0 );  
  
    a = HPCC_XMALLOC( double, VectorSize );  
    b = HPCC_XMALLOC( double, VectorSize );  
    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

```

#ifndef _OPENMP
#include <omp.h>
#endif

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

int HPCCC_StarStream(HPCCC_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 = HPCCC_Stream( params, 0 == myRank );
    MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm );
    return errCount;
}

int HPCCC_LocalStream(HPCCC_Params *params, int myRank) {
    double scalar;
    VectorSize = HPCCC_LocalVectorSize( params, 3, sizeof(double), 0 );

    a = HPCCC_XMALLOC( double, VectorSize );
    b = HPCCC_XMALLOC( double, VectorSize );
    c = HPCCC_XMALLOC( double, VectorSize );

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

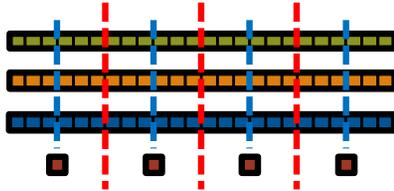
    #ifndef _OPENMP
    #pragma omp parallel for
    #endif
    for (j=0; j<VectorSize; j++) {
        b[j] = 2.0;
        c[j] = 0.0;
    }

    scalar = 3.0;

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

    HPCCC_free(c);
    HPCCC_free(b);
    HPCCC_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);
    if( N % dimBlock.x != 0 ) dimGrid

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

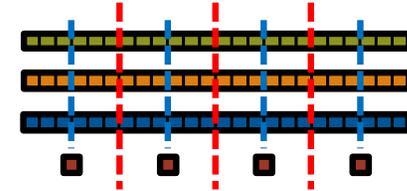
    scalar=3.0f;
    STREAM_Triad<<<dimGrid,dimBlock>>>(d_b, d_c, d_a, scalar, N);
    cudaThreadSynchronize();

    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/threads	iteration/task
Instruction-level vectors/threads	pragmas	iteration
GPU/accelerator	CUDA/OpenCL/OpenACC	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;
}

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;
}

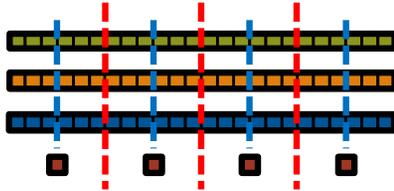
#ifdef _OPENMP
#pragma omp parallel for
#endif
for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 0.0;
}

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);
    if( N % dimBlock.x != 0 ) dimGrid

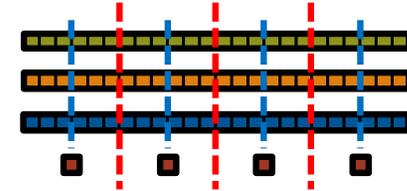
    set_array<<<dimGrid,dimBlock>>>(d_b, .5f, N);
    set_array<<<dimGrid,dimBlock>>>(d_c, .5f, N);

    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*

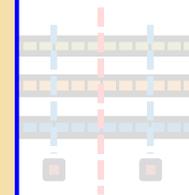
# STREAM Triad: Chapel

## MPI + OpenMP

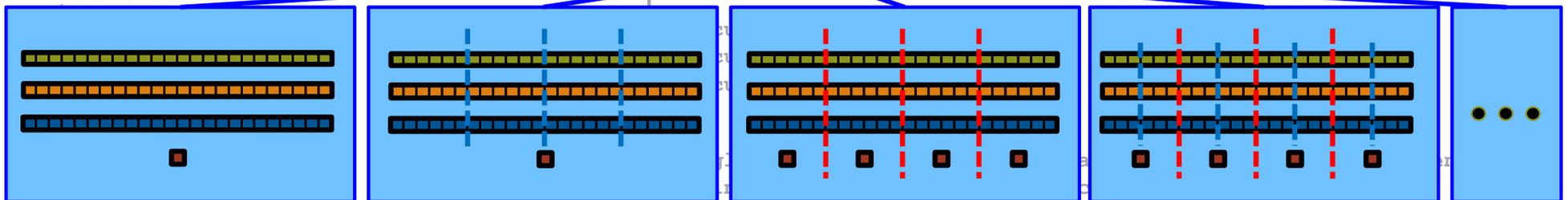
```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif
static int VectorSize;
static double *a, *b, *c;
int HPCC_StarStream(HPCC_Params *pa
    int myRank, commSize;
    int rv, errCount;
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Comm_size( comm, &commSize );
    MPI_Comm_rank( comm, &myRank );
    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 );
        }
    }
    #pragma omp parallel for
    #endif
    for (
    a[
    HPCC
    HPCC
    HPCC
    retu
}
```

## 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.

COMPUTE | STORE | ANALYZE

Copyright 2014 Cray Inc.



# Outline

## ✓ Motivation

## ➤ Chapel Background and Themes

## ● Survey of Chapel Concepts

- Quick run-through of basic concepts
- Slightly more detail on advanced/research-y concepts

## ➤ Project Status and Resources



# What is Chapel?



- **An emerging parallel programming language started from DARPA HPCS: High Productivity Computing Systems program**
  - 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?

## 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:

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



COMPUTE | STORE | ANALYZE

Copyright 2014 Cray Inc.

# 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

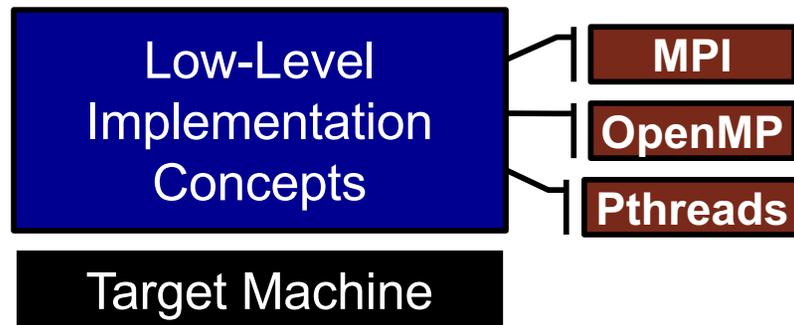
- **Styles:** data-parallel, task-parallel, concurrency, nested, ...
- **Levels:** model, function, loop, statement, expression

...target any parallelism available in the hardware

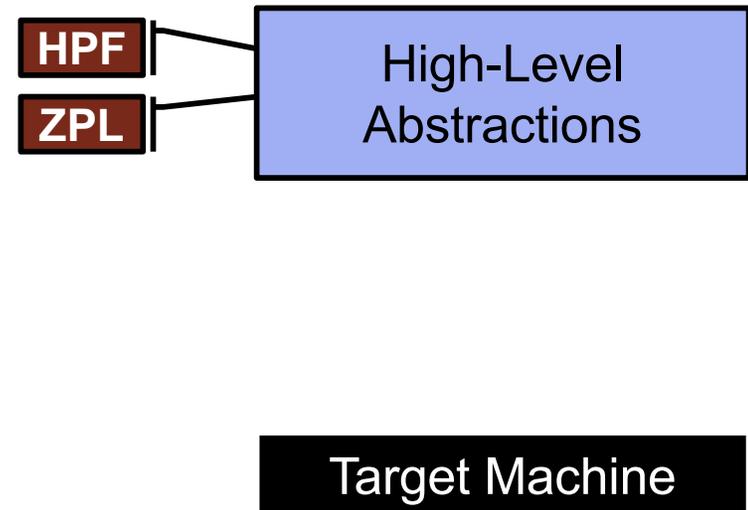
- **Types:** machines, nodes, cores, instructions

Type of HW Parallelism	Programming Model	Unit of Parallelism
Inter-node	Chapel	executable/task
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?”*



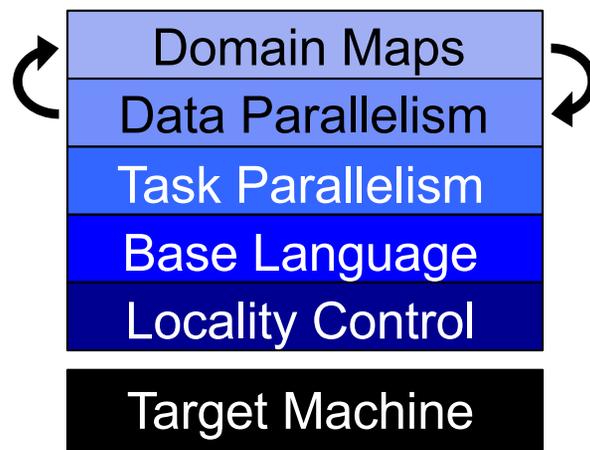
*“Why don’t I have more control?”*

### 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

### Chapel aims to narrow this gulf:

- 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



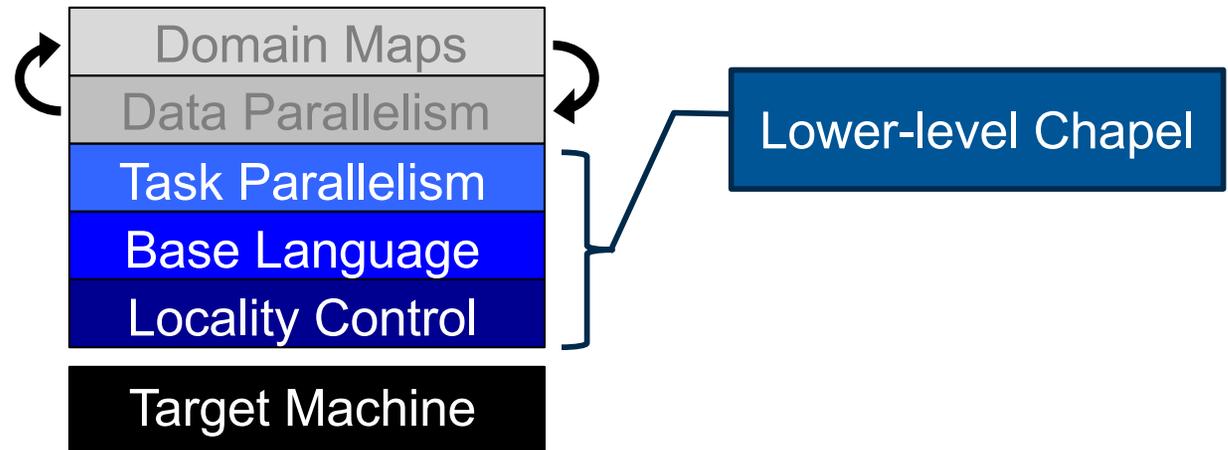
# Outline

- ✓ Motivation
- ✓ Chapel Background and Themes
- Survey of Chapel Concepts
- Project Status and Resources

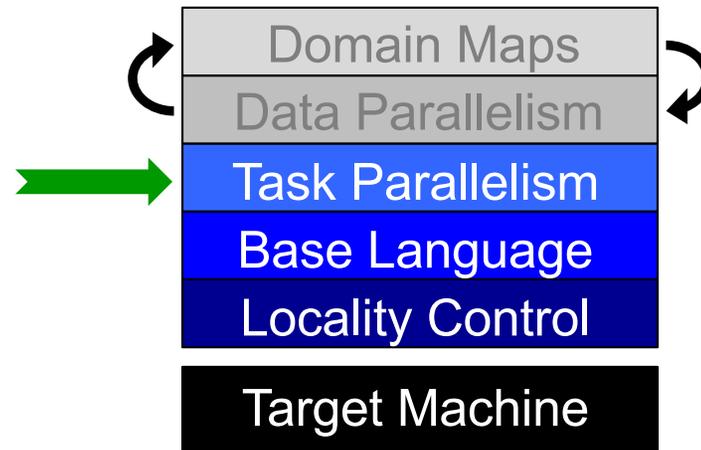


# Lower-Level Features

*Chapel language concepts*



# Task Parallel Features



# Task Parallelism: Begin Statements

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

## Possible outputs:

```
hello world  
goodbye
```

```
goodbye  
hello world
```

# Sample Task Parallel Feature: Coforall Loops

```
coforall t in 0..#numTasks { // create a task per iteration
  writeln("Hello from task ", t, " of ", numTasks);
} // wait for its tasks to complete before proceeding

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

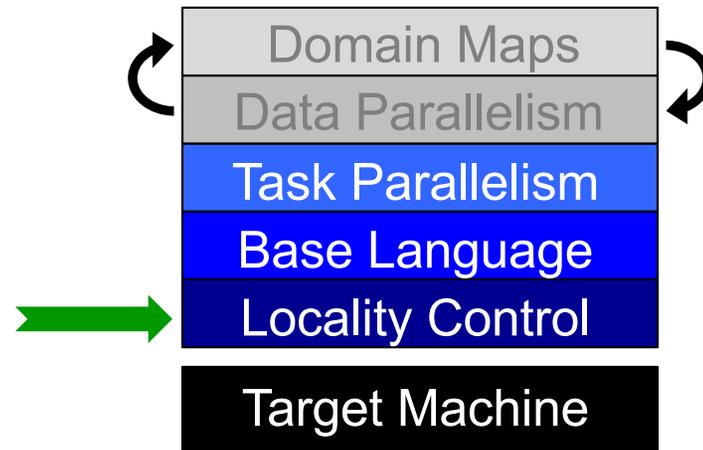
## Sample output:

```
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
```

# Task Parallelism: Data-Driven Synchronization

- 1) ***atomic variables***: support atomic operations (as in C++)
  - e.g., compare-and-swap; atomic sum, mult, etc.
- 2) ***single-assignment variables***: reads block until assigned
- 3) ***synchronization variables***: store full/empty state
  - by default, reads/writes block until the state is full/empty

# Locality Features



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");
```

```
begin on A[i,j] do  
    bigComputation(A);  
  
begin on node.left do  
    search(node.left);
```

# Chapel and PGAS

- Chapel is a PGAS language...

...but unlike most, it's not restricted to SPMD

⇒ never think in terms of “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

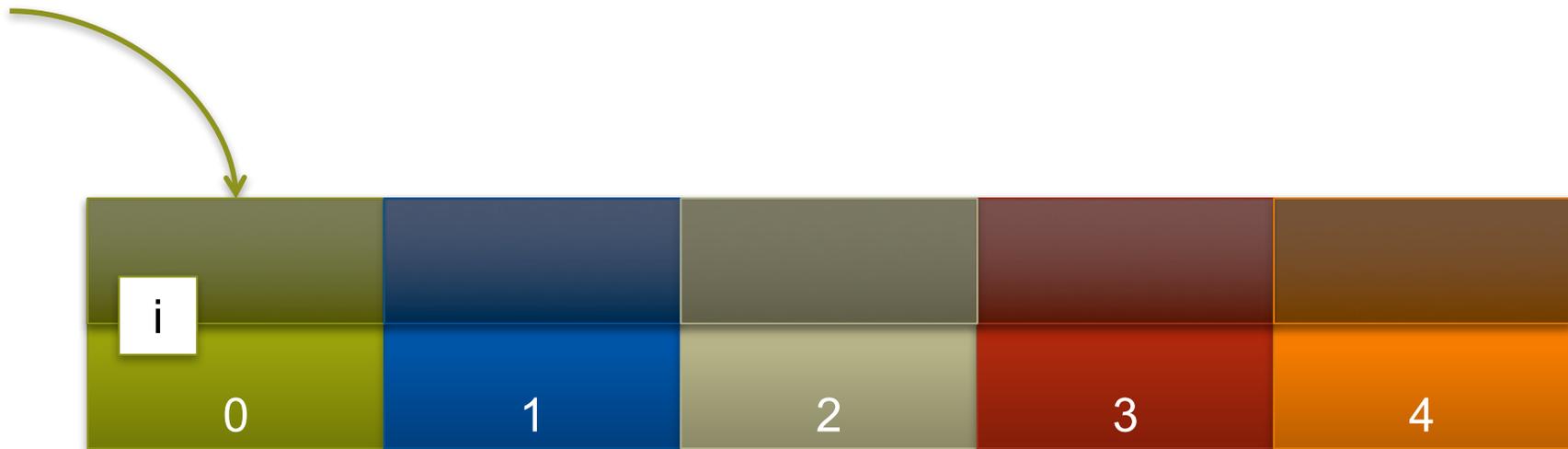


*Locales* (think: “compute nodes”)

COMPUTE | STORE | ANALYZE

# Chapel: Scoping and Locality

```
var i: int;
```



*Locales* (think: “compute nodes”)

COMPUTE | STORE | ANALYZE

# Chapel: Scoping and Locality

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



*Locales* (think: “compute nodes”)

COMPUTE | STORE | ANALYZE

# Chapel: Scoping and Locality

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

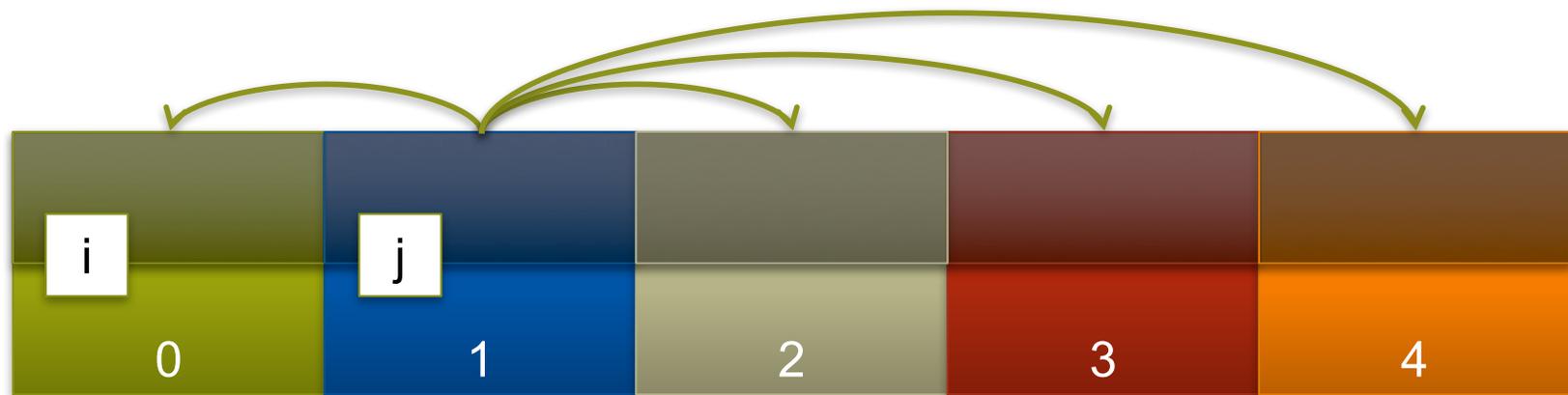


*Locales* (think: “compute nodes”)

COMPUTE | STORE | ANALYZE

# Chapel: Scoping and Locality

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



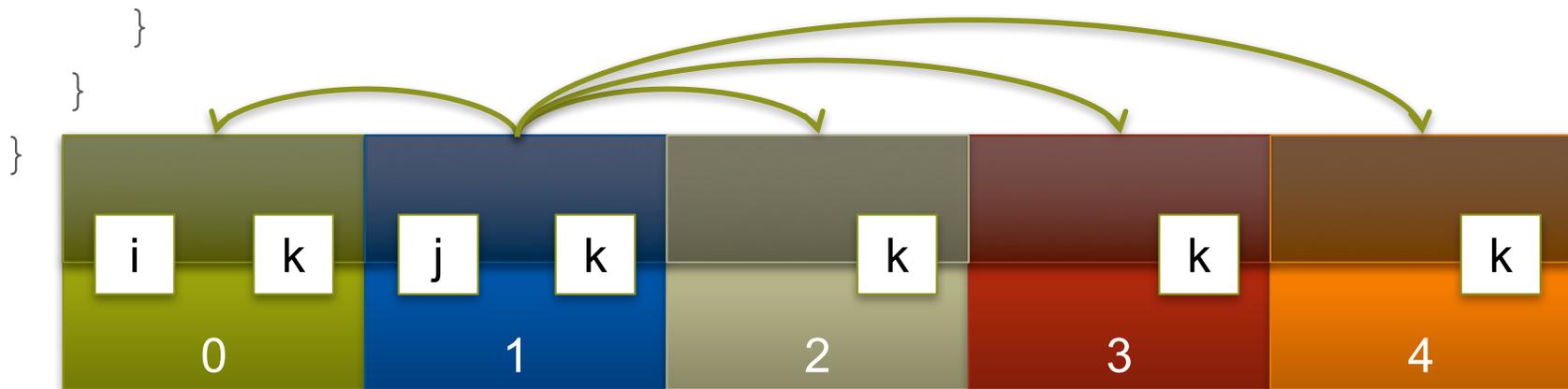
*Locales* (think: “compute nodes”)

COMPUTE | STORE | ANALYZE

# 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*

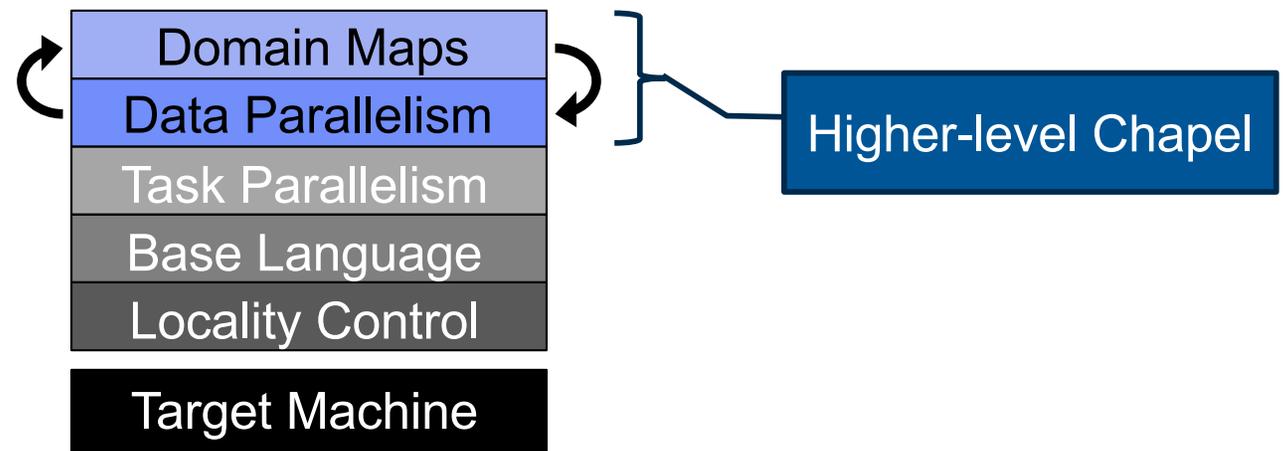


*Locales* (think: “compute nodes”)

COMPUTE | STORE | ANALYZE

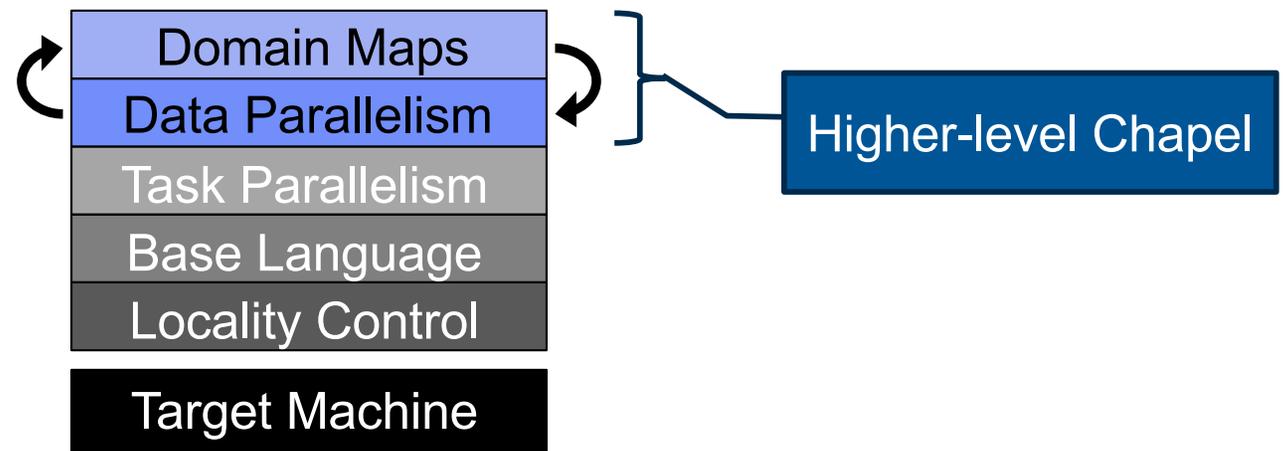
# Higher-Level Features

*Chapel language concepts*



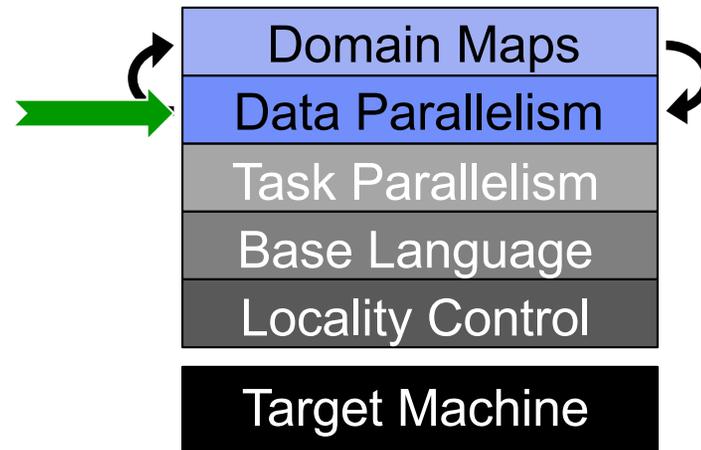
# Higher-Level Features

*Chapel language concepts*



Theme 2: Global-view  
Abstractions

# Data Parallel Features (by example)



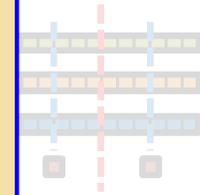
# STREAM Triad: Chapel

## MPI + OpenMP

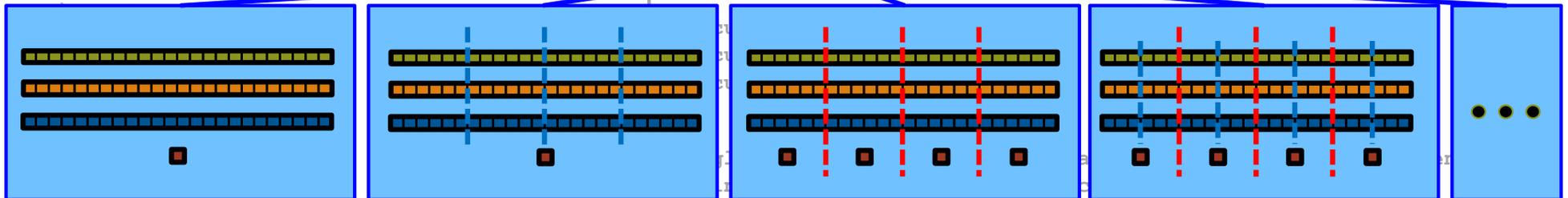
```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif
static int VectorSize;
static double *a, *b, *c;
int HPCC_StarStream(HPCC_Params *pa
    int myRank, commSize;
    int rv, errCount;
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Comm_size( comm, &commSize );
    MPI_Comm_rank( comm, &myRank );
    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 );
        }
    }
    #pragma omp parallel for
    #endif
    for
    a[
    HPCC
    HPCC
    HPCC
    retu
}
```

## 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



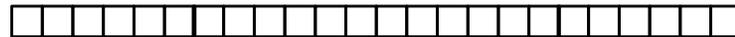
Philosophy: Good language design can tease details of locality and parallelism away from an algorithm, permitting the compiler, runtime, applied scientist, and HPC expert to each focus on their strengths.

COMPUTE | STORE | ANALYZE



# STREAM Triad in Chapel

```
const ProblemSpace = {1..m};
```



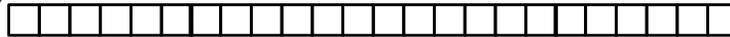
```
var A, B, C: [ProblemSpace] real;
```



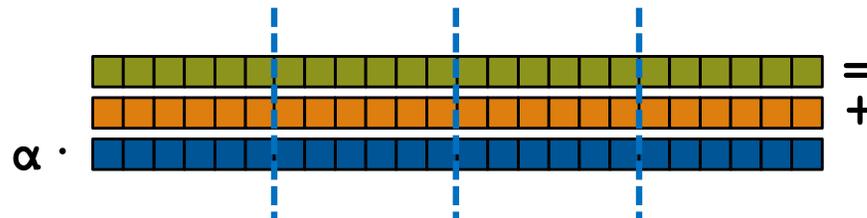
```
A = B + alpha * C;
```

# STREAM Triad: Chapel (multicore)

```
const ProblemSpace = {1..m};
```



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

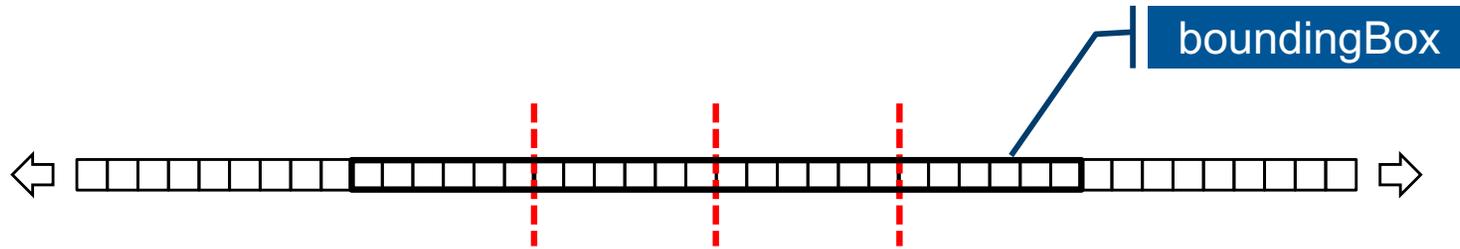


```
A = B + alpha * C;
```

No domain map specified => use default layout

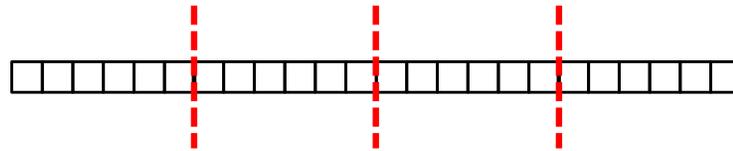
- current locale owns all domain indices and array values
- computation will execute using local processors only

# STREAM Triad: Chapel (multilocale, blocked)

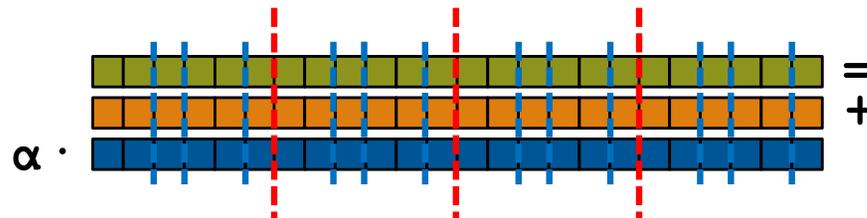


```
const ProblemSpace = {1..m}
```

```
dmapped Block(boundingBox={1..m});
```

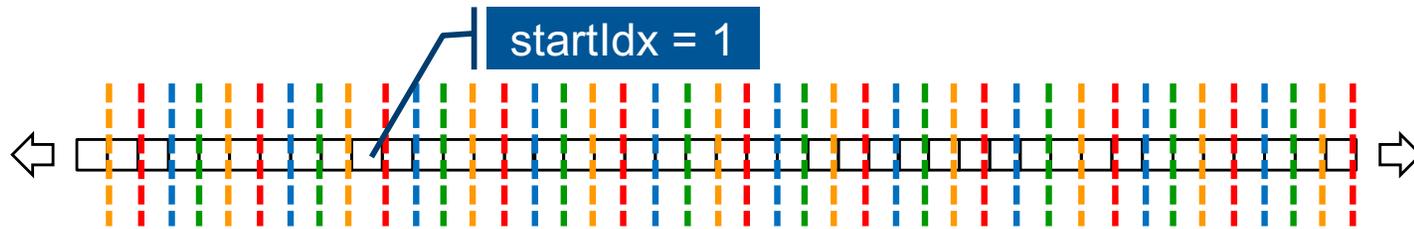


```
var A, B, C: [ProblemSpace] real;
```

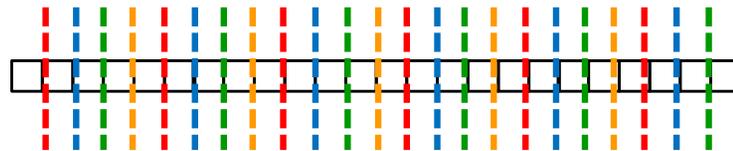


```
A = B + alpha * C;
```

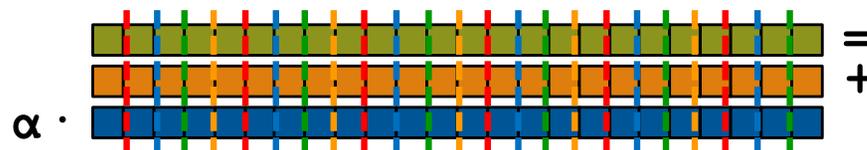
# STREAM Triad: Chapel (multilocale, cyclic)



```
const ProblemSpace = {1..m}
      dmapped Cyclic(startIdx=1);
```

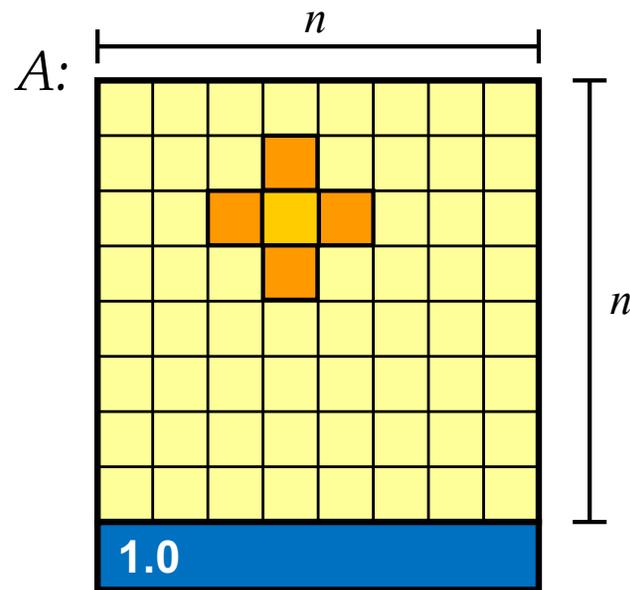


```
var A, B, C: [ProblemSpace] real;
```



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

# Data Parallelism by Example: Jacobi Iteration



repeat until max  
change  $< \epsilon$



COMPUTE | STORE | ANALYZE

# 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
```

```
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)

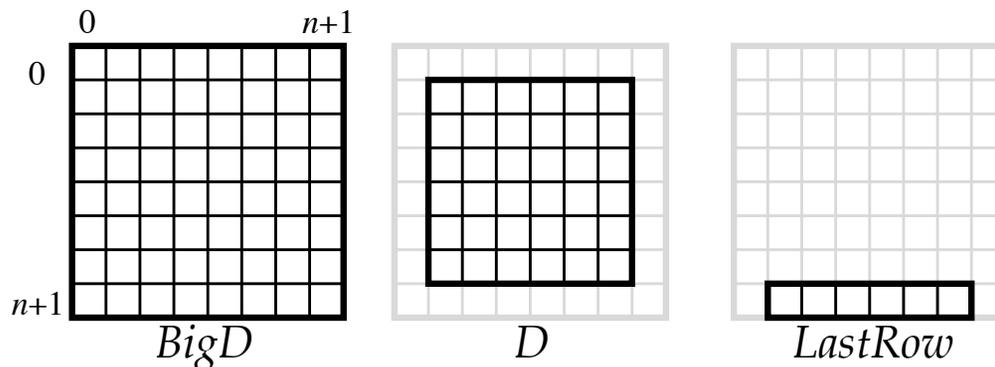
# Jacobi Iteration in Chapel

```
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)

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

$Dom1[Dom2] \Rightarrow$  computes the intersection of two domains



**.exterior()**  $\Rightarrow$  one of several built-in domain generators

# Jacobi Iteration in Chapel

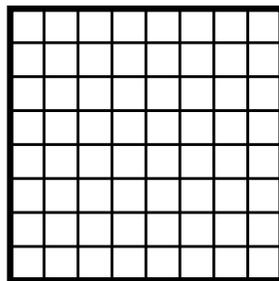
```
config const n = 6,  
            epsilon = 1.0e-5;  
  
const BigD = {0..n+1, 0..n+1},  
            D = BigD[1..n, 1..n],  
            LastRow = D.exterior(1,0);  
  
var A, Temp : [BigD] real;
```

## Declare arrays

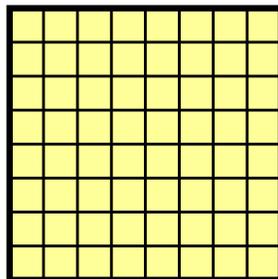
**var**  $\Rightarrow$  can be modified throughout its lifetime

**: [Dom] T**  $\Rightarrow$  array of size *Dom* with elements of type *T*

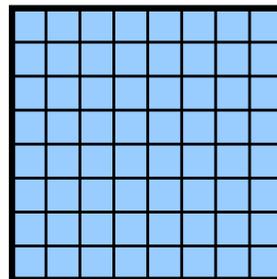
**(no initializer)**  $\Rightarrow$  values initialized to default value (0.0 for reals)



*BigD*



*A*



*Temp*

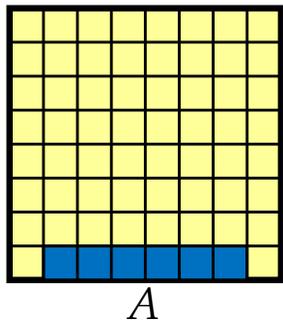
```
(i, j+1)) / 4;
```

# 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;
```

## Set Explicit Boundary Condition

**Arr[Dom]**  $\Rightarrow$  refer to array slice (“forall i in Dom do ...Arr[i]...”)



# Jacobi Iteration in Chapel

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

## Compute 5-point stencil

**forall** *ind* in *Dom*  $\Rightarrow$  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**  $\Rightarrow$  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 (shared memory)

```
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;
```

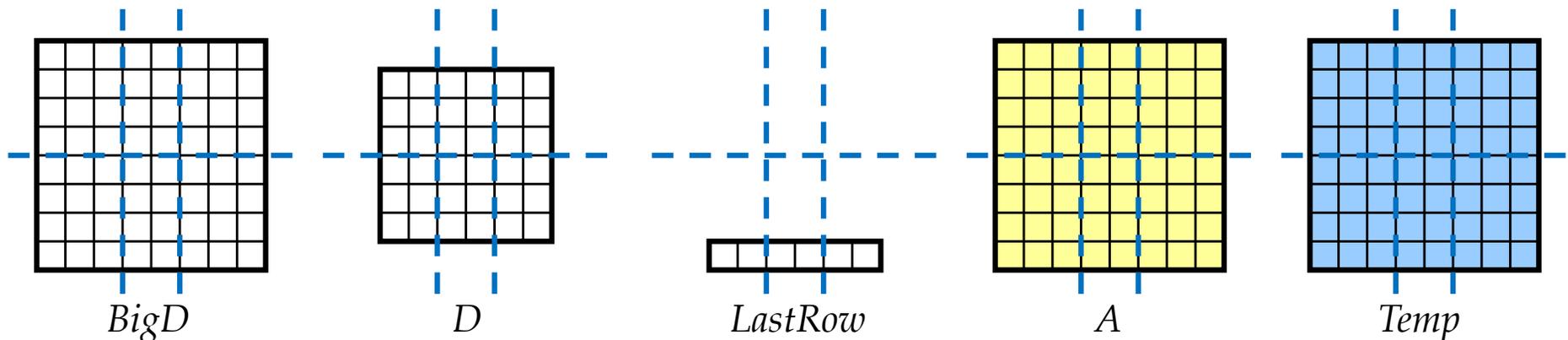
By default, domains and their arrays are mapped to a single locale.  
Any data parallelism over such domains/ arrays will be executed by the cores on that locale.  
Thus, this is a shared-memory parallel program.

```
Temp[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 (distributed memory)

```
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



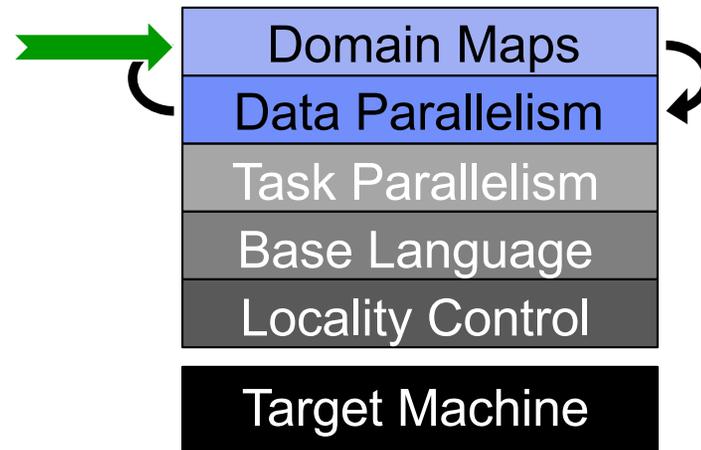
COMPUTE | STORE | ANALYZE

# 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;
```

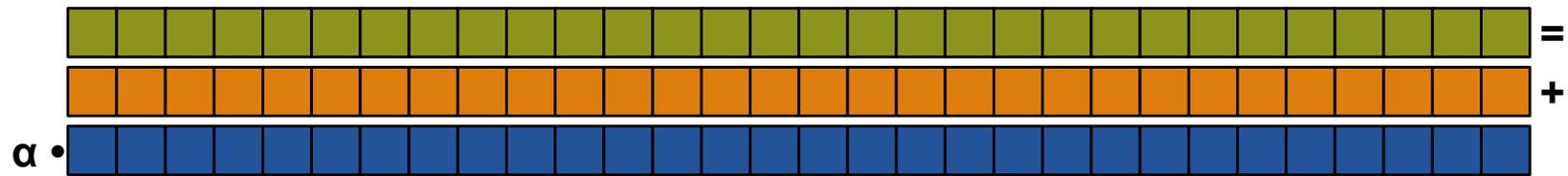


# Domain Maps



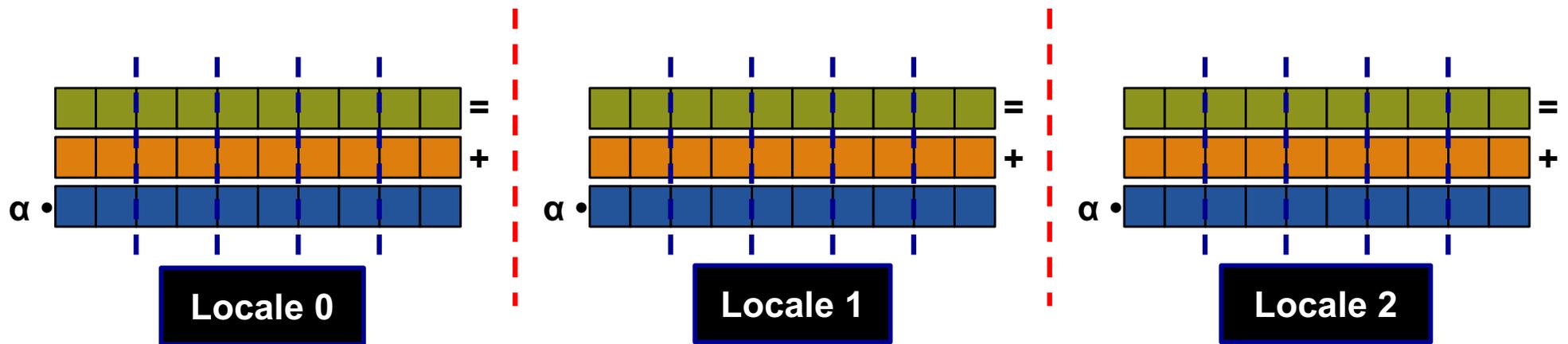
# Domain Maps

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



`A = B + alpha * C;`

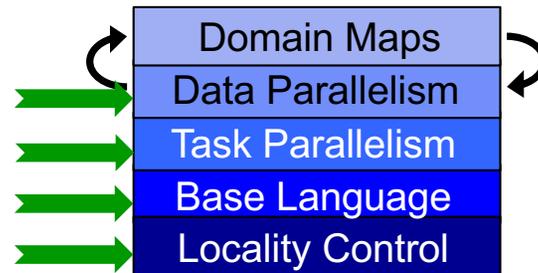
...to the target locales' memory and processors:



COMPUTE | STORE | ANALYZE

# 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

# Domain Map Descriptors

## Domain Map

**Represents:** a domain map value

**Generic w.r.t.:** index type

**State:** the domain map's representation

**Typical Size:**  $\Theta(1)$

**Required Interface:**

- create new domains

## Domain

**Represents:** a domain

**Generic w.r.t.:** index type

**State:** representation of index set

**Typical Size:**  $\Theta(1) \rightarrow \Theta(\text{numIndices})$

**Required Interface:**

- create new arrays
- queries: size, members
- iterators: serial, parallel
- domain assignment
- index set operations

## Array

**Represents:** an array

**Generic w.r.t.:** index type, element type

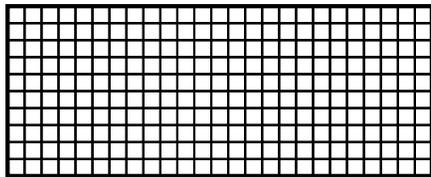
**State:** array elements

**Typical Size:**  $\Theta(\text{numIndices})$

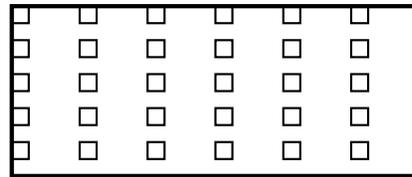
**Required Interface:**

- (re-)allocation of elements
- random access
- iterators: serial, parallel
- slicing, reindexing, aliases
- get/set of sparse “zero” values

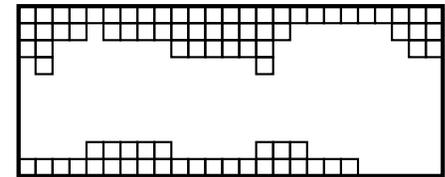
# Chapel Domain Types



*dense*



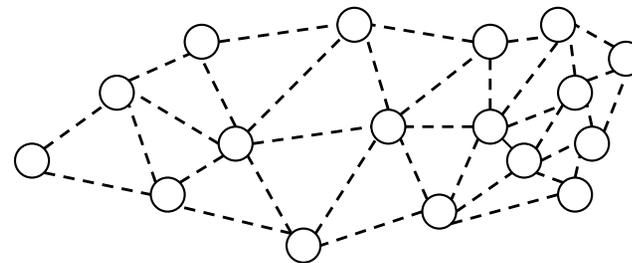
*strided*



*sparse*

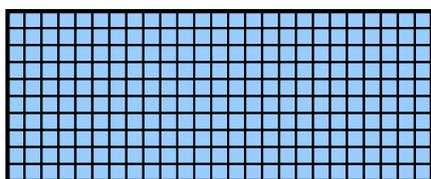


*associative*

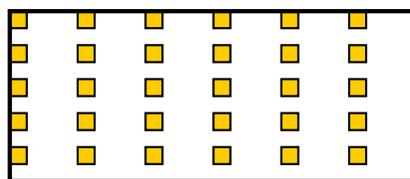


*unstructured*

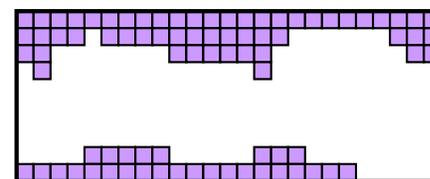
# Chapel Array Types



*dense*



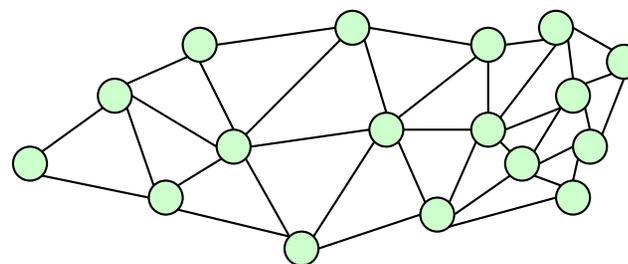
*strided*



*sparse*

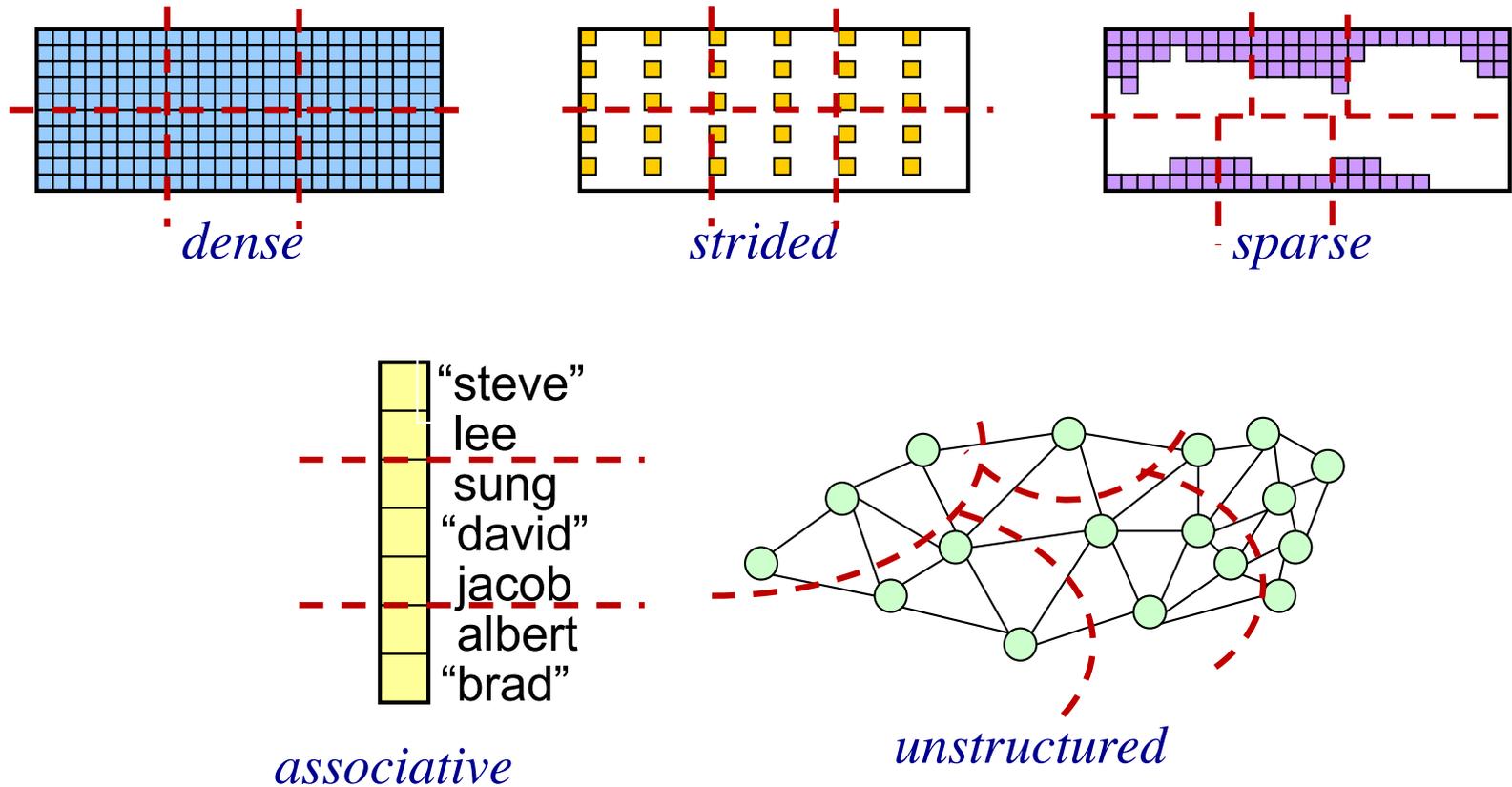


*associative*



*unstructured*

# All Domain Types Support Domain Maps



# Domain Maps Summary

- **Data locality requires mapping arrays to memory well**
  - distributions between distinct memories
  - layouts within a single memory
- **Most languages define a single data layout & distribution**
  - where the distribution is often the degenerate “everything’s local”
- **Domain maps...**
  - ...move such policies into user-space...
  - ...exposing them to the end-user through high-level declarations

```
const Elms = {0..#numElms} dmapped Block(...)
```



# Outline

- ✓ Motivation
- ✓ Chapel Background and Themes
- ✓ Survey of Chapel Concepts
- **Project Status and Resources**



# Implementation Status -- Version 1.10.0 (Oct 2014)

## Overall Status:

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

## This is a good time to:

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



# Chapel version 1.10 is now available



## ● Highlights Include:

- lighter-weight tasking via Sandia's Qthreads
- initial support for Intel Xeon Phi Knights Corner (KNC)
- renewed focus on standard libraries
- support for Lustre and cURL-based data channels
- expanded array capabilities
- improved semantic checks, bug fixes, third-party packages, ...
- significant performance improvements...

# Chapel: It's not just for HPC anymore



- **“Big data” programmers want productive languages too**
  - MapReduce, Pig, Hive, HBase have their place, but also drawbacks
  - Wouldn't a general, locality-aware parallel language be nice here too?
- **Chapel support for HDFS\*: A first step**
  - Developed by Tim Zakian (Indiana University), summer 2013
  - Summer 2014: extended support to include Lustre, cURL
- **Questions:**
  - What killer apps/demos to focus on?

\*HDFS = Hadoop Distributed File System

<http://chapel.cray.com/presentations/SC13/06-hdfs-ferguson.pdf>



COMPUTE | STORE | ANALYZE

Copyright 2014 Cray Inc.

# Interactive Chapel



- **What if you could work with Chapel interactively:**

```
chpl> var A: [1..n] real;  
OK.  
chpl> [i in 1..n] A = i / 2.0;  
OK.  
chpl> writeln(A);  
0.5 1.0 1.5 2.0 2.5 3.0  
chpl> proc foo(x) { x *= 2; }  
OK.
```

- **What if this worked not only on your desktop, but by offloading onto compute nodes as well:**

```
chpl> var myLocales = getNodes(100);  
OK.  
chpl> var MyDist = new Block({1..1000000}, myLocales);  
OK.
```

- **We've recently started an effort to implement such a capability**

# For More Information: Online Resources

## Chapel project page: <http://chapel.cray.com>

- overview, papers, presentations, language spec, ...

## Chapel GitHub page: <https://github.com/chapel-lang>

- download 1.10.0 release, browse source repository

## Chapel SourceForge page: <https://sourceforge.net/projects/chapel/>

- join community mailing lists; alternative release download site

## Mailing Aliases:

- [chapel\\_info@cray.com](mailto:chapel_info@cray.com): contact the team at Cray
- [chapel-announce@lists.sourceforge.net](mailto:chapel-announce@lists.sourceforge.net): list for announcements only
- [chapel-users@lists.sourceforge.net](mailto:chapel-users@lists.sourceforge.net): user-oriented discussion list
- [chapel-developers@lists.sourceforge.net](mailto:chapel-developers@lists.sourceforge.net): developer discussion
- [chapel-education@lists.sourceforge.net](mailto:chapel-education@lists.sourceforge.net): educator discussion
- [chapel-bugs@lists.sourceforge.net](mailto:chapel-bugs@lists.sourceforge.net): public bug forum



# For More Information: Suggested Reading

## Overview Papers:

- [\*A Brief Overview of Chapel\*](#), Chamberlain (pre-print of a chapter for *A Brief Overview of Parallel Programming Models*, edited by Pavan Balaji, to be published by MIT Press in 2014).
  - *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*



# For More Information: Lighter Reading

## Blog Articles:

- [Chapel: Productive Parallel Programming](#), Chamberlain, [Cray Blog](#), May 2013.
  - *a short-and-sweet introduction to Chapel*
- [Why Chapel?](#) ([part 1](#), [part 2](#), [part 3](#)), Chamberlain, [Cray Blog](#), June-October 2014.
  - *a current series of articles answering common questions about why we are pursuing Chapel in spite of the inherent challenges*
- [\[Ten\] Myths About Scalable Programming Languages](#) ([index available here](#)), Chamberlain, [IEEE TCSC Blog](#), April-November 2012.
  - *a series of technical opinion pieces designed to combat standard arguments against the development of high-level parallel languages*

