Chapel: Making Large-Scale Parallel Programming Productive

Brad Chamberlain

Cray Inc.

Google: August 10th, 2011

Sustained Performance Milestones

1 GF – 1988: Cray Y-MP; 8 Processors

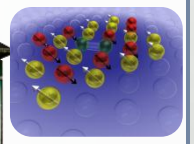
- Static finite element analysis

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

- Modeling of metallic magnet atoms

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

- Superconductive materials



1 EF – ~2018: Cray ____; ~10,000,000 Processors

- TBD

Sustained Performance Milestones

1 GF – 1988: Cray Y-MP; 8 Processors

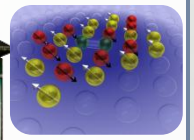
- Static finite element analysis
- Fortran77 + Cray autotasking + vectorization

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

- Modeling of metallic magnet atoms
- Fortran + MPI (?)

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

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



1 EF – ~2018: Cray ____; ~10,000,000 Processors

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

Why Do HPC Programming Models Change?

HPC has traditionally given users…

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

…ones that are closely tied to the underlying hardware

Examples:

HW Granularity	Programming Model	Unit of Parallelism
Inter-node	MPI	executable
Intra-node/multicore	OpenMP/pthreads	iteration/task
Instruction-level vectors/threads	pragmas	iteration
GPU/accelerator	CUDA/OpenCL	SIMD function

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

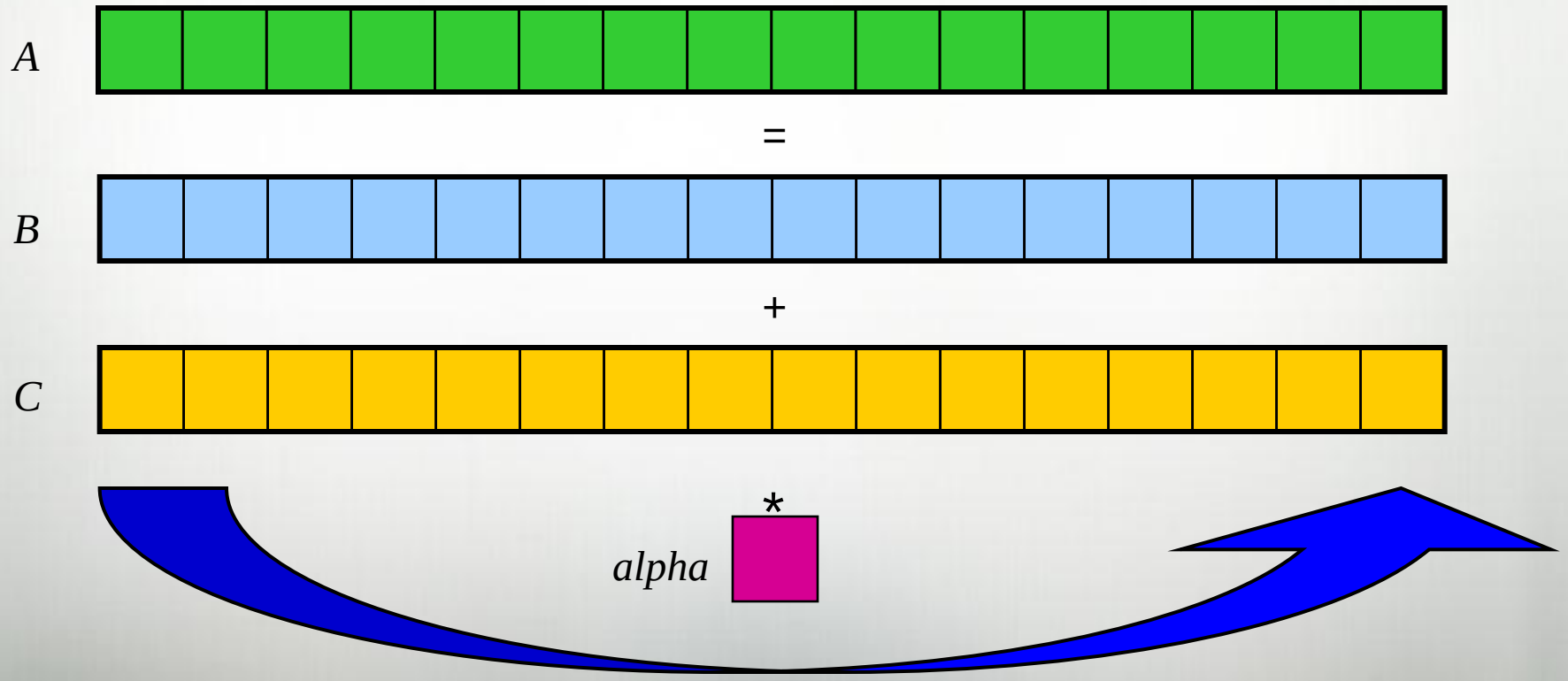
downsides: lots of user-managed detail; brittle to changes

Introduction to STREAM Triad

Given: m-element vectors A, B, C

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

Pictorially:

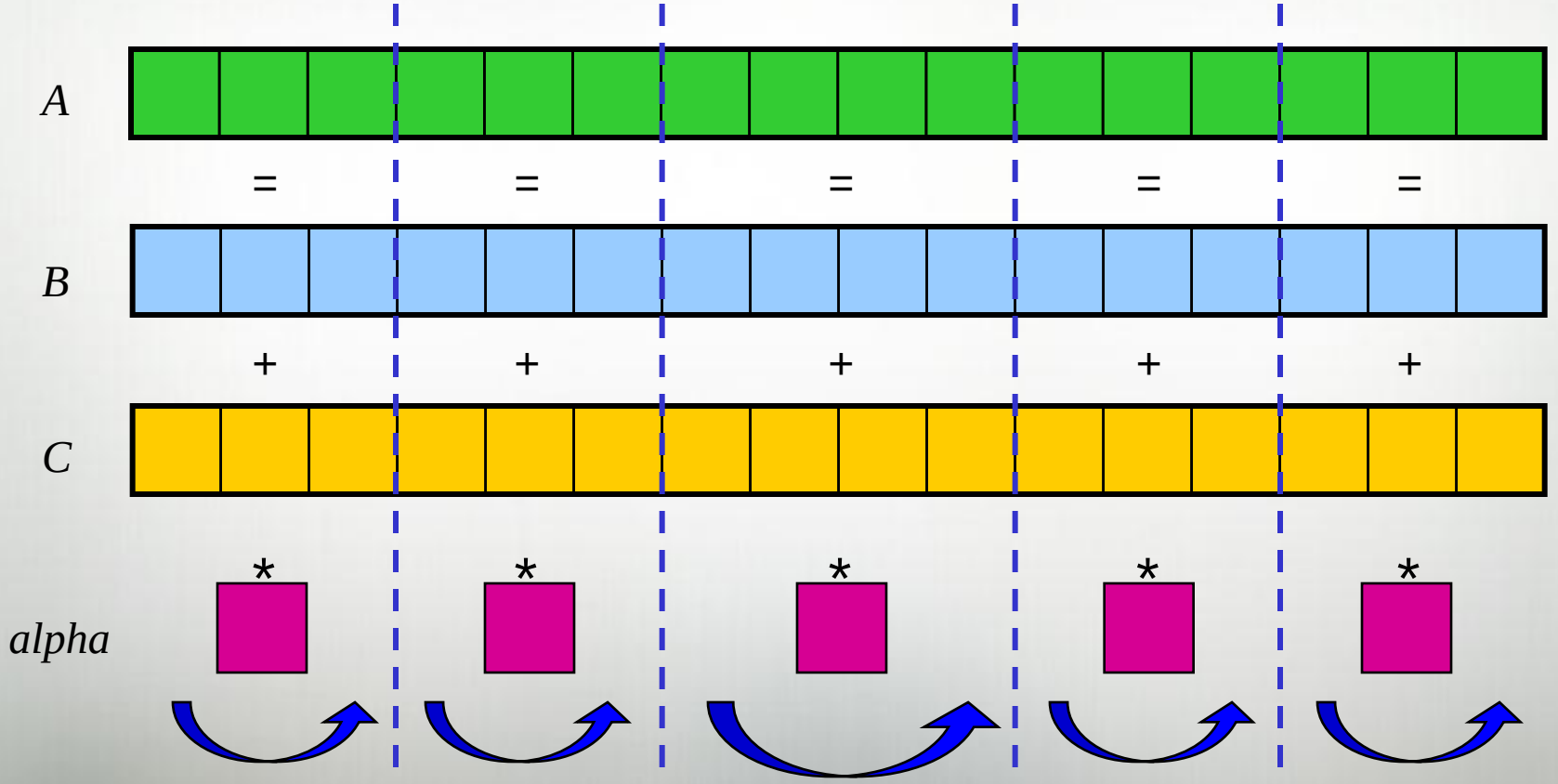


Introduction to STREAM Triad

Given: m-element vectors A, B, C

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

Pictorially (in parallel):



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);

  return 0;
}
```

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);

  return 0;
}
```

STREAM Triad: MPI+OpenMP vs. CUDA

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);

  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];
}
```

STREAM Triad: MPI+OpenMP vs. CUDA vs. 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 *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 == myRan
  MPI_Reduce( &rv, &errCount, 1, MPI_I

  return errCount;
}

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

  VectorSize = HPCC_LocalVectorSize( p

  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 all
      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;
```

```
;
;
;
```

```
N);
N);
```

```
d_c, d_a, scalar,  N);
```

```
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];
}
```

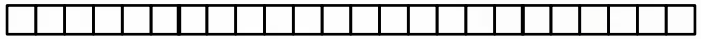
Chapel

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

const ProblemSpace = [1..m] dmapped …;

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

B = …;
C = …;

A = B + alpha * C;
```

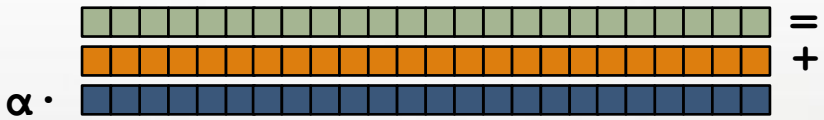
the special sauce

STREAM Triad: Chapel (multicore)

```
const ProblemSpace = [1..m];
```



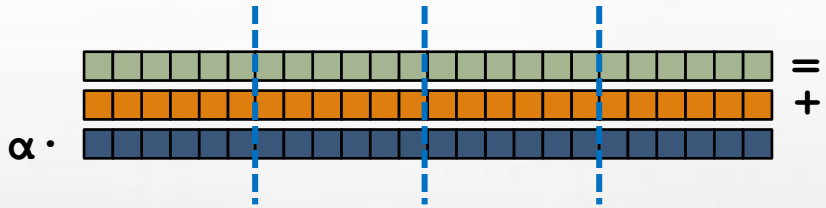
```
var A, B, C: [ProblemSpace] real;
```




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

STREAM Triad: Chapel (multicore)

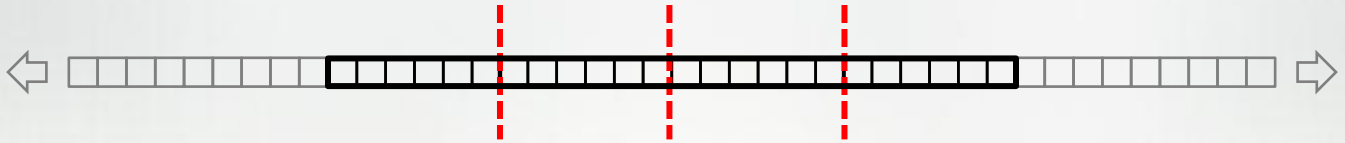
```
const ProblemSpace = [1..m];

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

A = B + alpha * C;
```



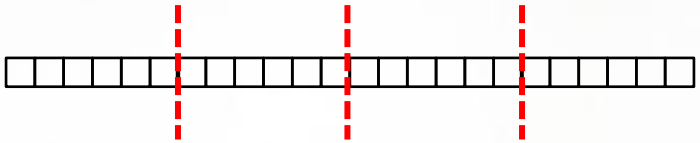
No domain map specified => use default layout

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

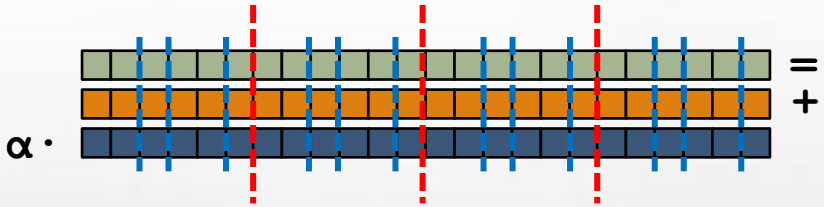
STREAM Triad: Chapel (multinode, blocked)



```
const ProblemSpace = [1..m]
                     dmapped Block(boundingBox=[1..m]);
```

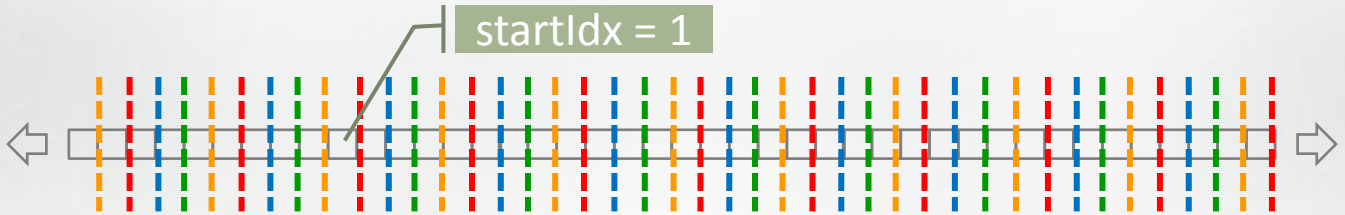


```
var A, B, C: [ProblemSpace] real;
```

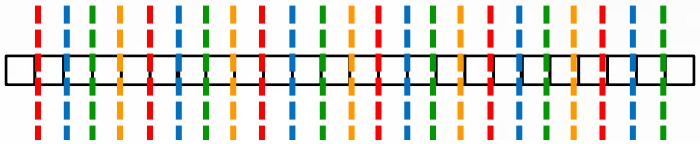



```
A = B + alpha * C;
```

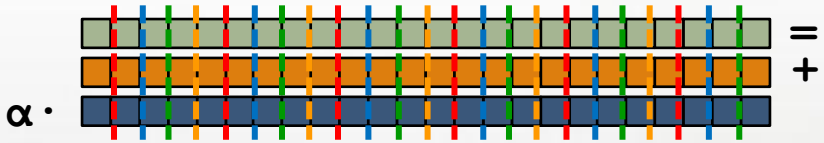
STREAM Triad: Chapel (multinode, cyclic)




```
const ProblemSpace = [1..m]
                          dmapped Cyclic(startIdx=1);
```



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




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

Some Chapel Themes Illustrated Here

General Parallel Programming

- "any parallel algorithm on any parallel hardware"

Global-View Parallel Programming

- operate on distributed arrays as if they were local

Multiresolution Parallel Programming

- high-level features for convenience/portability
- low-level features for greater control
- abstractions to span this gap & separate concerns

Outline

- ✓ Chapel Motivation
- ➢ Quick Tour of Some Chapel Features
- Advanced Chapel Features
- Project Status and Summary

What is Chapel?

- A new parallel programming language
  - Design and development led by Cray Inc.
  - Started under the DARPA HPCS program
- **Overall goal:** Improve programmer productivity
  - Improve the programmability of parallel computers
  - Match or beat the performance of current programming models
  - Support better portability than current programming models
  - Improve the robustness of parallel codes
- A work-in-progress

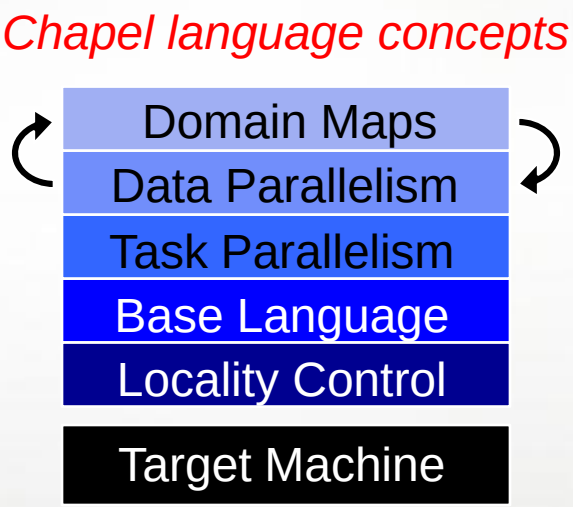
Chapel's Implementation

- Being developed as open source at SourceForge
- Licensed as BSD software
- Target Architectures:
  - multicore desktops and laptops
  - commodity clusters
  - Cray architectures
  - systems from other vendors
  - (in-progress: CPU+accelerator hybrids, manycore, …)

Chapel's Multiresolution Design

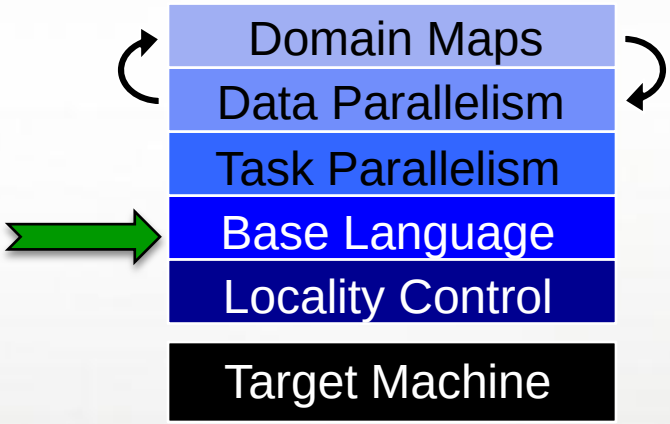
Multiresolution Design: Support multiple tiers of features

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



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

Base Language Features



Static Type Inference

```
const pi = 3.14,              // pi is a real
      coord = 1.2 + 3.4i,     // loc is a complex…
      coord2 = pi*loc,        // …as is loc2
      name = "brad",          // name is a real
      verbose = false;        // verbose is boolean


proc addem(x, y) {            // addem() is generic
  return x + y;
}

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

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

```
(4.14, bradford)
```

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(D, 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)
```

Range Types and Algebra

```
const r = 1..10;

printVals(r # 3);
printVals(r # -3);
printVals(r by 2);
printVals(r by 2 align 2);
printVals(r by -2);
printVals(r by 2 # 3);
printVals(r # 3 by 2);

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

```
1 2 3
8 9 10
1 3 5 7 9
2 4 6 8 10
10 8 6 4 2
1 3 5
1 3
```

Zipper Iteration

```
var A: [0..9] real;

for (i,j,a) in (1..10, 2..20 by 2, A) do
  a = j + i/10.0;

writeln(A);
```

```
2.1 4.2 6.3 8.4 10.5 12.6 14.7 16.8 18.9 21.0
```

Configs

```
param intSize = 32;
type elementType = real(32);
const epsilon = 0.01:elementType;
var start = 1:int(intSize);
```

Configs

```
config param intSize = 32;
config type elementType = real(32);
config const epsilon = 0.01:elementType;
config var start = 1:int(intSize);
```

```
% chpl myProgram.chpl -sintSize=64 -selementType=real
% a.out --start=2 --epsilon=0.00001
```

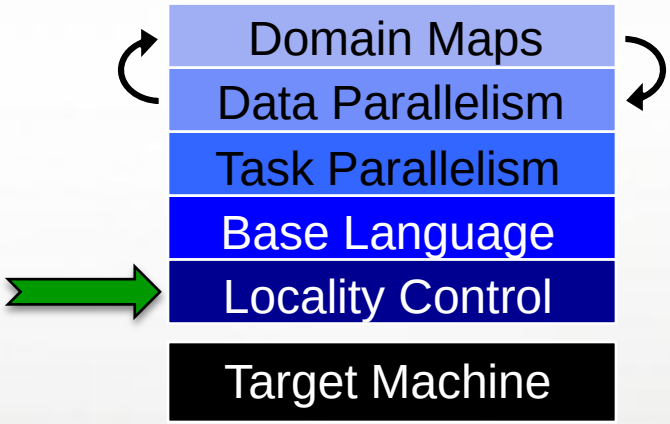
Default and Named Arguments

```
proc foo(name="joe", weight=175, age) {
  …
}

foo("brad", age=101);
```

Other Base Language Features

- tuples types
- compile-time features for meta-programming
  - e.g., compile-time functions to compute types, params
- rank-independent programming features
- value- and reference-based OOP
- overloading, where clauses
- modules (for namespace management)
- …

Locality Features



The Locale

- **Definition**
  - Abstract unit of target architecture
  - Capable of running tasks and storing variables
    - i.e., has processors and memory
  - Supports reasoning about locality
- **Properties**
  - a locale's tasks have ~uniform access to local vars
  - Other locale's vars are accessible, but at a price
- **Locale Examples**
  - A multi-core processor
  - An SMP node

Coding 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 LocaleSpace = [0..#numLocales];
const Locales: [LocaleSpace] locale;
```

Locale Operations

- Locale methods support reasoning about machine resources:

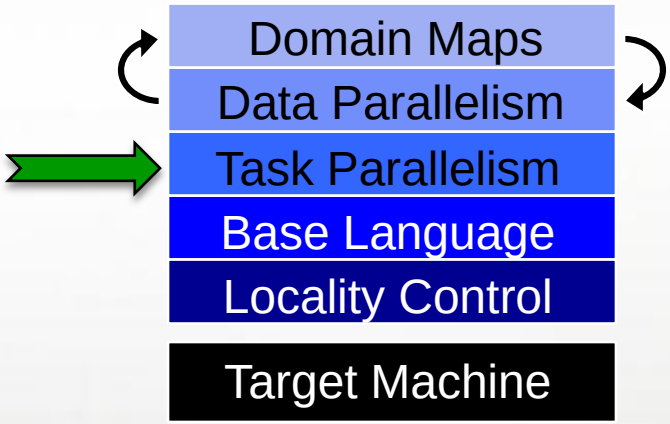
```
proc locale.physicalMemory(…) { … }
proc locale.numCores(…) { … }
proc locale.name(…) { … }
```

- *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
  begin bigComputation(A);

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

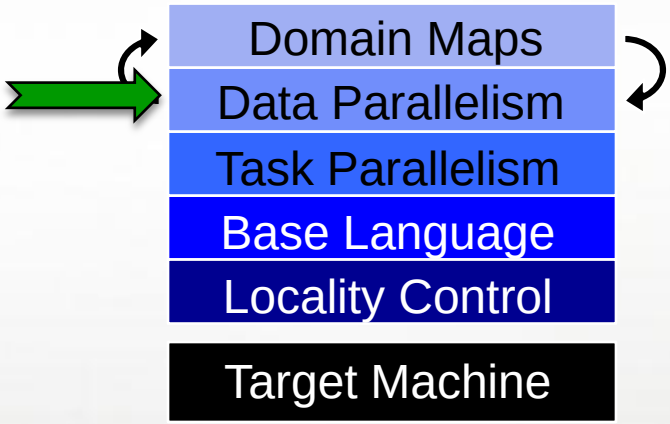
Task Parallel Features



Bounded Buffer Producer/Consumer Example

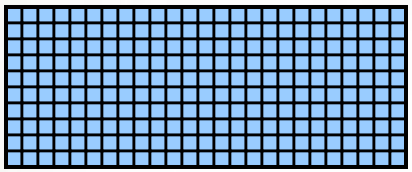
```
cobegin {
  producer();
  consumer();
}

// 'sync' types store full/empty state along with value
var buff$: [0..#buffersize] sync real;

proc producer() {
  var i = 0;
  for … {
    i = (i+1) % buffersize;
    buff$(i) = …;    // reads block until empty, leave full
} }

proc consumer() {
  var i = 0;
  while … {
    i= (i+1) % buffersize;
    …buff$(i)…;     // writes block until full, leave empty
} }
```

Data Parallel Features

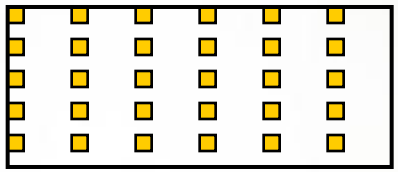


Chapel Domain/Array Types

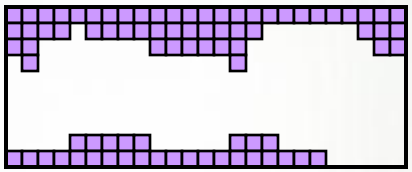
Chapel supports several types of domains and arrays:



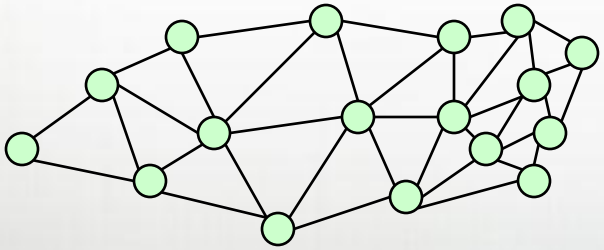
dense



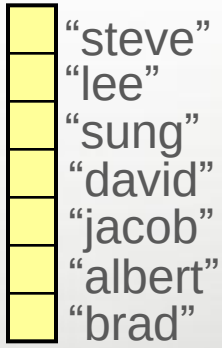
strided



sparse



unstructured



associative

Chapel Domain/Array Operations

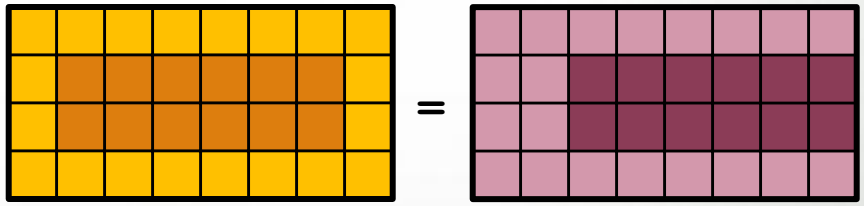
- Parallel and Serial Iteration

```
A = forall (i,j) in D do (i + j/10.0);
```

1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8
2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8
3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8
4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.8

- Array Slicing; Domain Algebra

```
A[InnerD] = B[InnerD+(0,1)];
```



- Promotion of Scalar Functions and Operators

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

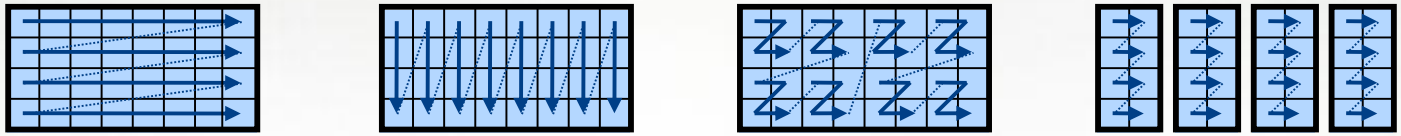
```
A = exp(B, C);
```

- And several other operations: indexing, reallocation, set operations, reindexing, aliasing, queries, …

Data Parallelism: Implementation Qs

Q1: How are arrays laid out in memory?

- Are regular arrays laid out in row- or column-major order? Or…?

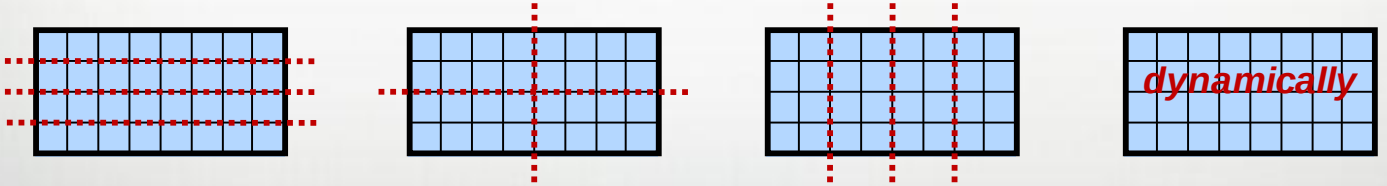


…?

- What data structure is used to store sparse arrays? (COO, CSR, …?)

Q2: How are data parallel operators implemented?

- How many tasks?
- How is the iteration space divided between the tasks?



…?

Data Parallelism: Implementation Qs

Q3: How are arrays distributed between locales?

- Completely local to one locale? Or distributed?
- If distributed… In a blocked manner? cyclically? block-cyclically? recursively bisected? dynamically rebalanced? …?

Q4: What architectural features will be used?

- Can/Will the computation be executed using CPUs? GPUs? both?
- What memory type(s) is the array stored in? CPU? GPU? texture? …?

A1: In Chapel, any of these could be the correct answer

A2: Chapel's *leader-follower iterators* and *domain maps* are designed to give the user full control over such decisions

Outline

- ✓ Chapel Motivation
- ✓ Quick Tour of Some Chapel Features
- ➢ Advanced Chapel Features
  - leader-follower iterators
  - domain maps
- Project Status and Summary

Promotion Semantics

Promoted functions/operators are defined in terms of zippered semantics in Chapel. For example:

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

is equivalent to:

```
forall (a,b,c) in (A,B,C) do
  a = b + alpha * c;
```

Benefits of Zippered Promotion Semantics

- Chained whole-array operations are implemented element-wise rather than operator-wise.

  ⇒ No temporary arrays required by semantics

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

⇏

```
T1 = alpha * C;
A = B + T1;
```

⇒ No surprises in memory requirements

⇒ Friendlier to cache utilization

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

⇒

```
forall (a,b,c) in (A,B,C) do
  a = b + alpha * c;
```

Leader/Follower Iterators

- All zippered forall loops are defined in terms of leader/follower iterators:
  - *leader iterators:* create parallelism, assign iterations to tasks
  - *follower iterators:* serially execute work generated by leader

- *Conceptually*, the Chapel compiler translates:

```
forall (a,b,c) in (A,B,C) do
  a = b + alpha * c;
```

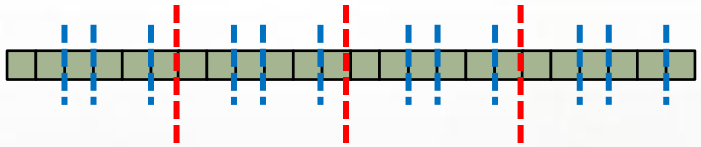
into:

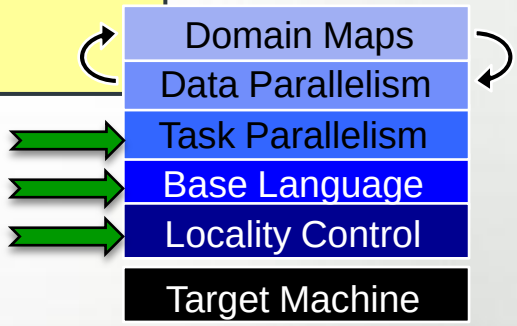
```
for work in A.lead() do
  for (a,b,c) in (A.follow(work), B.follow(work)
                  C.follow(work)) do
    a = b + alpha * c;
```

Defining Leaders and Followers

Leader iterators are defined using task/locality features:

```
iter BlockArr.lead() {
  coforall loc in Locales do
    on loc do
      coforall tid in here.numCores do
        yield computeMyBlock(loc.id, tid);
}
```





Follower iterators simply use serial features:

```
iter BlockArr.follow(work) {
  for i in work do
    yield accessElement(i);
}
```

Controlling Data Parallelism

Q: *"But what if I don't like the approach implemented by an array's leader iterator?"*

A: Several possibilities…

Controlling Data Parallelism

```
forall (b,a,c) in (B,A,C) do
  a = b + alpha * c;
```

Make something else the leader.

Controlling Data Parallelism

```
forall (a,b,c) in (dynamic(A, chunk=64), B, C) do
  a = b + alpha * c;
```

Invoke some other leader iterator explicitly (perhaps one that you wrote yourself).

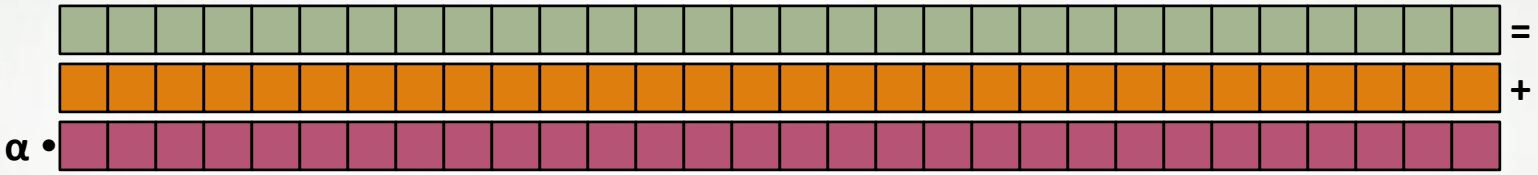
Controlling Data Parallelism

```
const ProblemSize = [1..n] dmapped BlockCyclic(start=1,
                                               blocksize=64);

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


A = B + alpha * C;
```

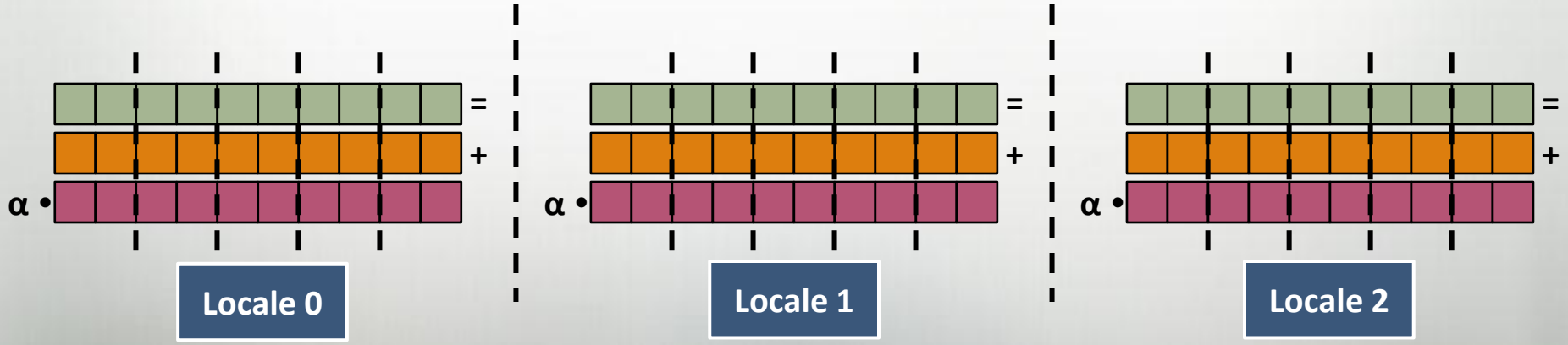
Change the array's default leader by changing its domain map (perhaps to one that you wrote yourself).

Domain Maps

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



…to the target locales’ memory and processors:



Domain Maps

Domain maps define data storage:

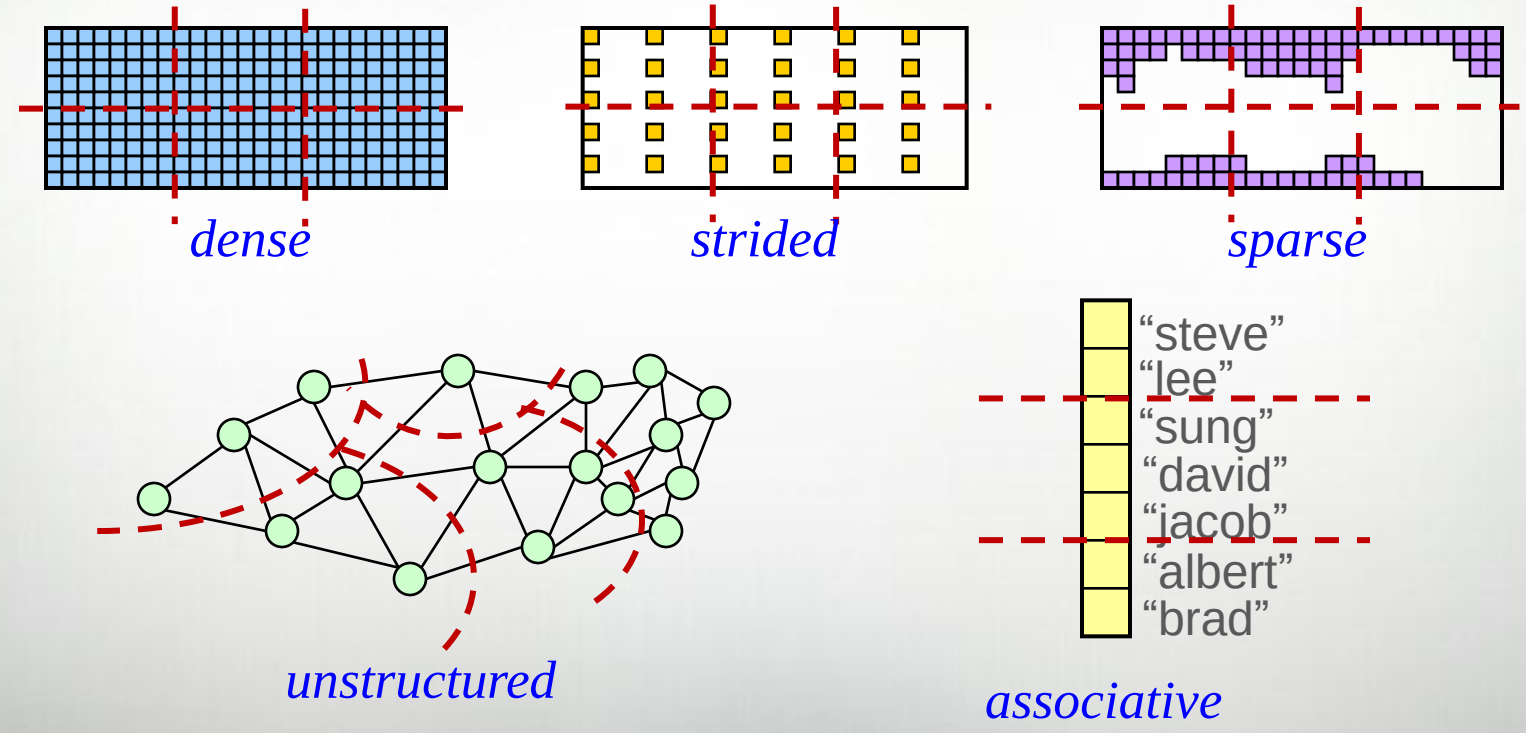
- Mapping of domain indices and array elements to locales
- Layout of arrays and index sets in each locale's memory

…as well as operations:

- random access, iteration, slicing, reindexing, rank change, …
- the Chapel compiler generates calls to these methods to implement the user's array operations

All Domain Types Support Domain Maps

All Chapel domain types support domain maps



Layouts and Distributions

Domain Maps fall into two major categories:

layouts: target a single locale (memory)

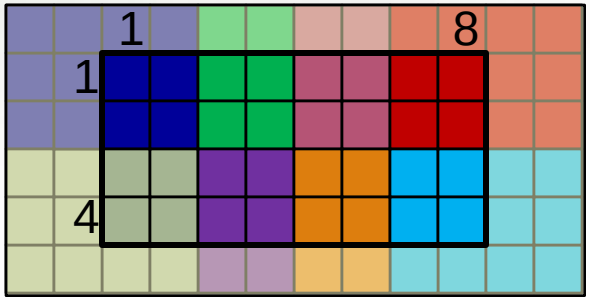
- e.g., a desktop machine or multicore node
- **examples:** row- and column-major order, tilings, compressed sparse row, space-filling curves

distributions: target distinct locales (memories)

- e.g., a distributed memory cluster or supercomputer
- **examples:** Block, Cyclic, Block-Cyclic, Recursive Bisection, …

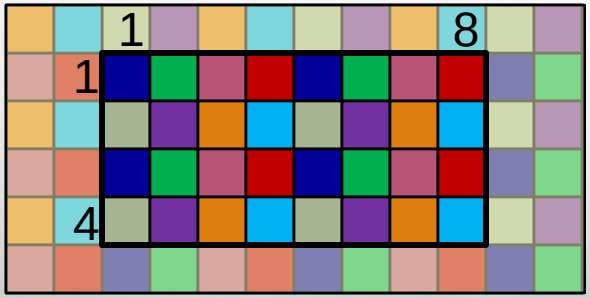
Sample Distributions: Block and Cyclic

```
var Dom = [1..4, 1..8] dmapped Block( [1..4, 1..8] );
```



distributed to

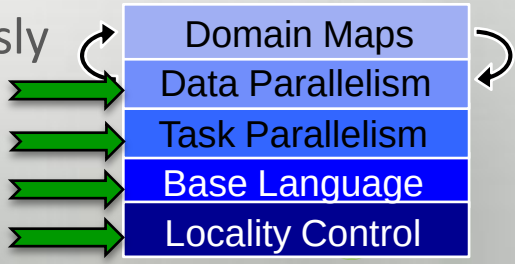

```
var Dom = [1..4, 1..8] dmapped Cyclic( startIdx=(1,1) );
```



distributed to

Chapel's Domain Map Philosophy

1. Chapel provides a library of standard domain maps
   - to support common array implementations effortlessly
2. Advanced users can write their own domain maps in Chapel
   - to cope with shortcomings in our standard library
3. Chapel's standard layouts and distributions will be written using the same user-defined domain map framework
   - to avoid a performance cliff between "built-in"/optimized domain maps and user-defined
4. Domain maps should only affect implementation and performance, not semantics
   - to support switching between domain maps effortlessly



Domain Maps Descriptors

Domain Map

Represents: a domain map value

Generic w.r.t.: index type

State: the domain map's representation

Typical Size: Θ(1)

Required Interface:

- create new domains

Domain

Represents: a domain

Generic w.r.t.: index type

State: representation of index set

Typical Size: Θ(1) → Θ(*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: Θ(*numIndices*)

Required Interface:

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

For More Information on Domain Maps

- HotPAR'10 paper/talk: *User-Defined Distributions and Layouts in Chapel: Philosophy and Framework*
- CUG'11 paper/talk: *Authoring User-Defined Domain Maps in Chapel*
- In the Chapel release…
  - Technical notes detailing domain map interface for programmers:

    $CHPL_HOME/doc/technotes/README.dsi
  - Current domain maps:

    $CHPL_HOME/modules/dists/*.chpl

    layouts/*.chpl

    internal/Default*.chpl

For More Information on Leader/Follower Iterators

- PGAS'11 submission (in review): *Composable Parallel Iterators in Chapel*
- In the Chapel release…
  - Primer-style example program:

    $CHPL_HOME/examples/primers/leaderfollower.chpl

Outline

- ✓ Chapel Motivation
- ✓ Quick Tour of Some Chapel Features
- ✓ Advanced Chapel Features
- ➢ Project Status and Summary

Status

- Everything you've heard about today works in the current compiler
  - (which is not to say that it's bug-free or feature-complete)

- Performance can still be hit or miss
  - a number of optimizations remain
    - some low-hanging, some more aggressive
  - generally speaking…

    …single-locale works better than multi-locale

    …multi-locale works best with fine-grain, demand-driven communication patterns (or embarrassingly parallel computations)

Next Steps

No-brainers:

- Performance Optimizations
- Feature Improvements/Bug Fixes
- Support Collaborations
- Develop post-HPCS strategy/funding

More advanced topics:

- Hierarchical Locales to target manycore/CPU+GPUs
  - additional hierarchy and heterogeneity warrants it
- Resiliency/Fault Tolerance

Our Team

- Cray:


Brad Chamberlain


Sung-Eun Choi


Greg Titus


Vass Litvinov


Tom Hildebrandt

- External Collaborators:


Albert Sidelnik


Jonathan Turner


Srinivas Sridharan

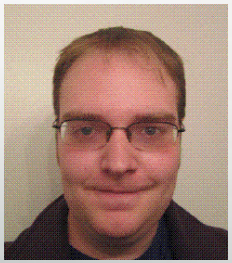
●●●

You? Your Friend/Student/ Colleague?

- Interns:


Jonathan Claridge


Hannah Hemmaplardh


Andy Stone


Jim Dinan


Rob Bocchino


Mack Joyner

●●●

Featured Collaborations

- **ORNL/Notre Dame** (Srinivas Sridharan, Jeff Vetter, Peter Kogge): Asynchronous software transactional memory over distributed memory
- **UIUC** (David Padua, Albert Sidelnik, Maria Garzarán): CPU-GPU computing
- **Sandia** (Kyle Wheeler, Rich Murphy): Chapel over Qthreads user threading
- **BSC/UPC** (Alex Duran): Chapel over Nanos++ user-level tasking
- **LTS** (Michael Ferguson): Improved I/O and strings
- **LLNL** (Tom Epperly et al.): Interoperability via Babel
- **Argonne** (Rusty Lusk, Rajeev Thakur, Pavan Balaji): Chapel over MPICH
- **CU Boulder** (Jeremy Siek, Jonathan Turner): Interfaces, concepts, generics
- **U. Oregon/Paratools Inc.** (Sameer Shende): Performance analysis with Tau
- **U. Malaga** (Rafael Asenio, Maria Gonzales, Rafael Larossa): Parallel file I/O
- **PNNL/CASS-MT** (John Feo, Daniel Chavarria): Cray XMT tuning
- (your name here?)

Potential Areas for Collaboration

- resiliency/fault tolerance
- memory management
- application studies
- tools/IDEs
- libraries
- data-intensive computation
- (your idea here?)

(see http://chapel.cray.com/collaborations.html for a more complete list)

For Further Information

- **Chapel Home Page** (papers, presentations, tutorials):
  http://chapel.cray.com
- **Chapel Project Page** (releases, mailing lists, code):
  http://sourceforge.net/projects/chapel/
- **General Questions/Info:**
  chapel_info@cray.com (or SourceForge chapel-users list)
- **Upcoming Events:**

  SC11 (November, Seattle WA):

  Monday, Nov 14th: full-day comprehensive tutorial

  Friday, Nov 18th: half-day broader engagement tutorial

  TBD: potential Chapel BOF/User's Group Workshop

CRAY
THE SUPERCOMPUTER COMPANY

http://chapel.cray.com chapel-info@cray.com http://sourceforge.net/projects/chapel/