Chapel: Striving for Productivity at Petascale, Sanity at Exascale

Brad Chamberlain

Cray Inc.

LLNL: December 14th, 2011

What is Chapel?

- A new parallel programming language
  - Design and development led by Cray Inc.
    - In collaboration with academics, labs, industry
  - Initiated 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, …)

PGAS: Partitioned Global Address Space Languages
(Or perhaps: Partitioned Global *Namespace* Languages)

Concept:

- support a shared namespace
  - "any parallel task can access any lexically visible variable"
- give each variable a well-defined affinity to a processor/node
  - "local variables are cheaper to access than remote ones"
- founding members: UPC, Co-Array Fortran, Titanium

Strengths:

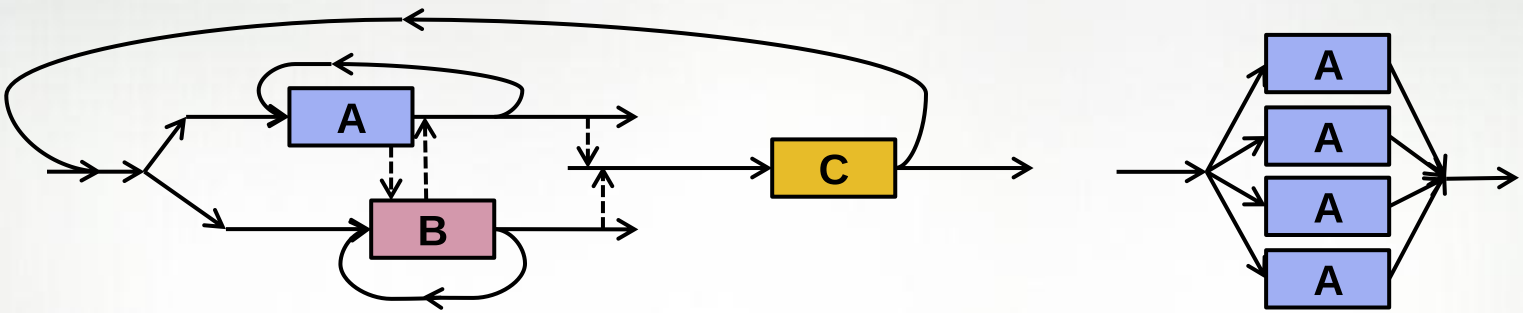
- permits users to specify *what* to transfer rather than *how*
- supports reasoning about locality/affinity to get scalability

Weaknesses (of traditional PGAS languages):

- restricted to SPMD programming and execution models
- limited support for distributed arrays

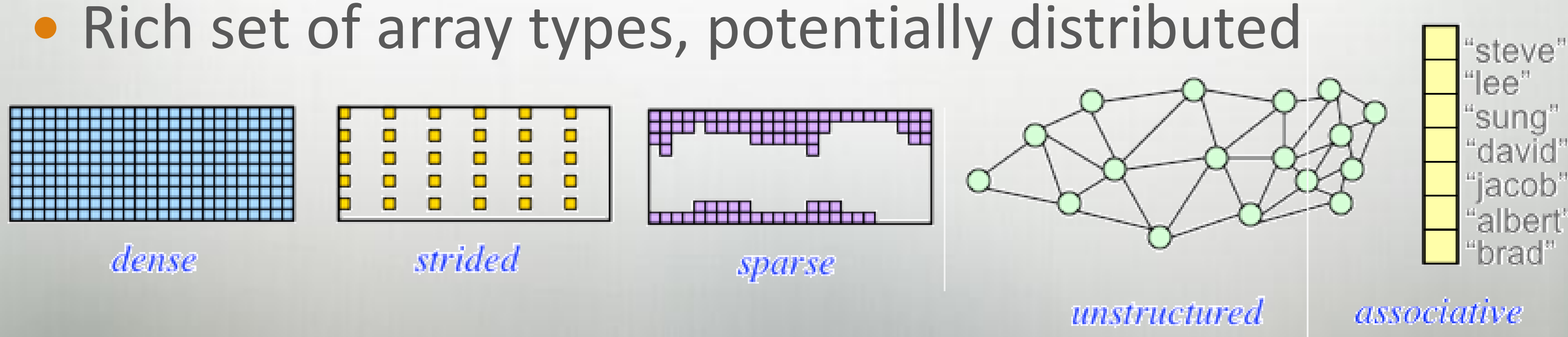
Chapel: A Next-Generation PGAS Language

- General/dynamic/multithreaded parallelism



- Distinct concepts for parallelism vs. locality
  - e.g., *cobegin* creates tasks, *locale* type represents locality

- Rich set of array types, potentially distributed



Why I'm here this week

- *Not* to try and convince you to use Chapel today

- Rather, to see how we can maximize its future utility to you

  …as Chapel matures and hardens

  …as you move to more advanced algorithms

  …as you start dealing with next-generation architectures

- And to look for near-term collaborations to help us reach that point in the best state possible

Outline

- ✓ Chapel Context
- ➢ Motivation
- Feature Tour
- Advanced Features / Research Topics
- Project Status and Overview
- Chapel and Exascale

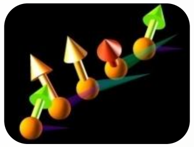
Sustained Performance Milestones

1 GF – 1988: Cray Y-MP; 8 Processors

- Static finite element analysis

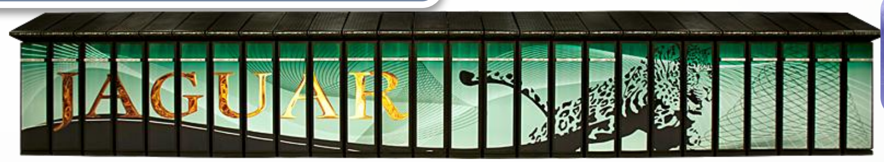
1 TF – 1998: Cray T3E; 1,024 Processors

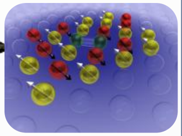
- Modeling of metallic magnet atoms



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

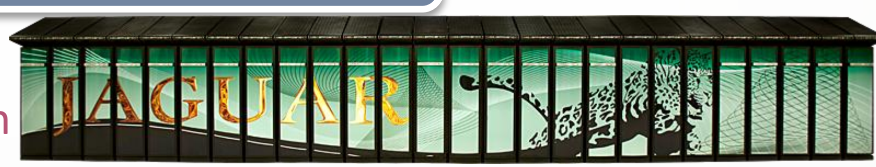
- Static finite element analysis
- Fortran77 + Cray autotasking + vectorization

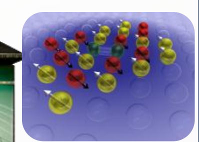
1 TF – 1998: Cray T3E; 1,024 Processors

- 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/OpenACC

Or Perhaps Something Completely Different?

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

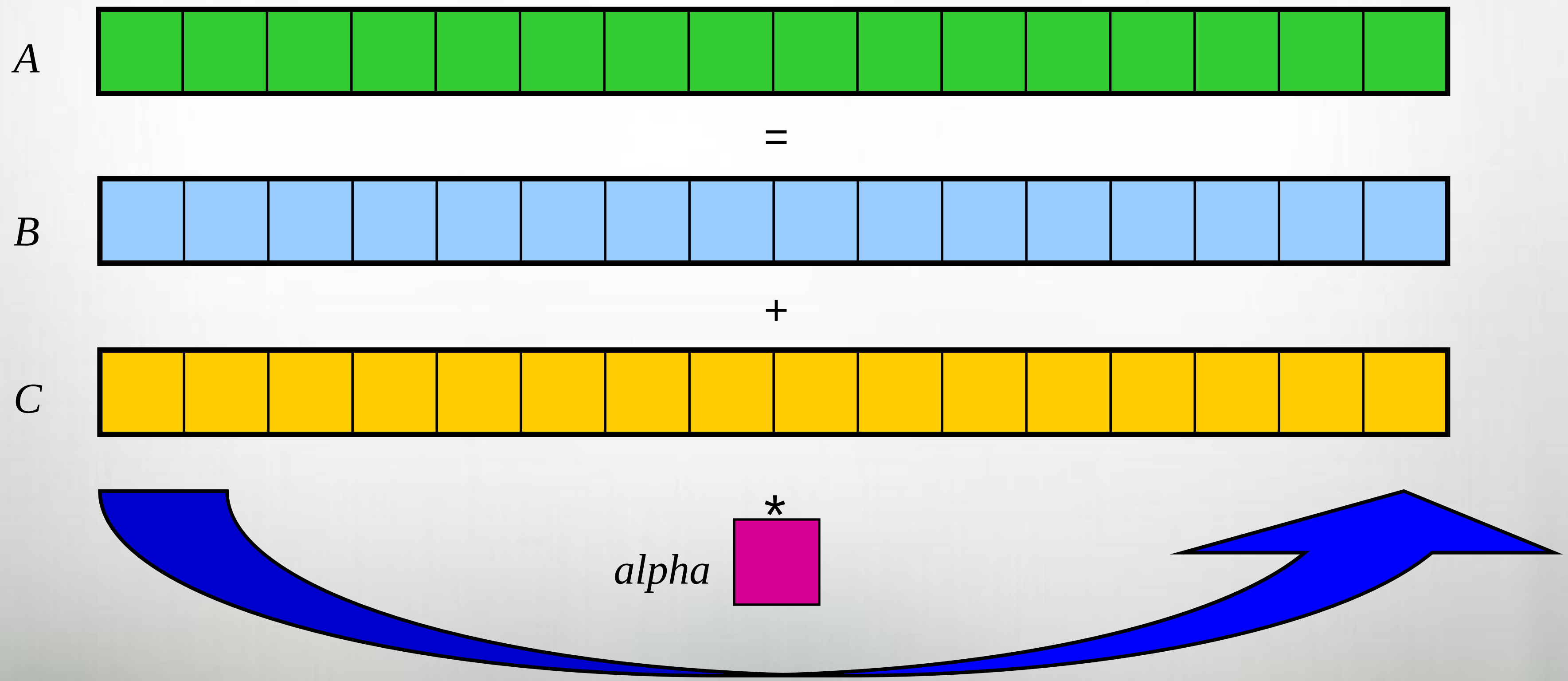
downsides: lots of user-managed detail; brittle to changes

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:

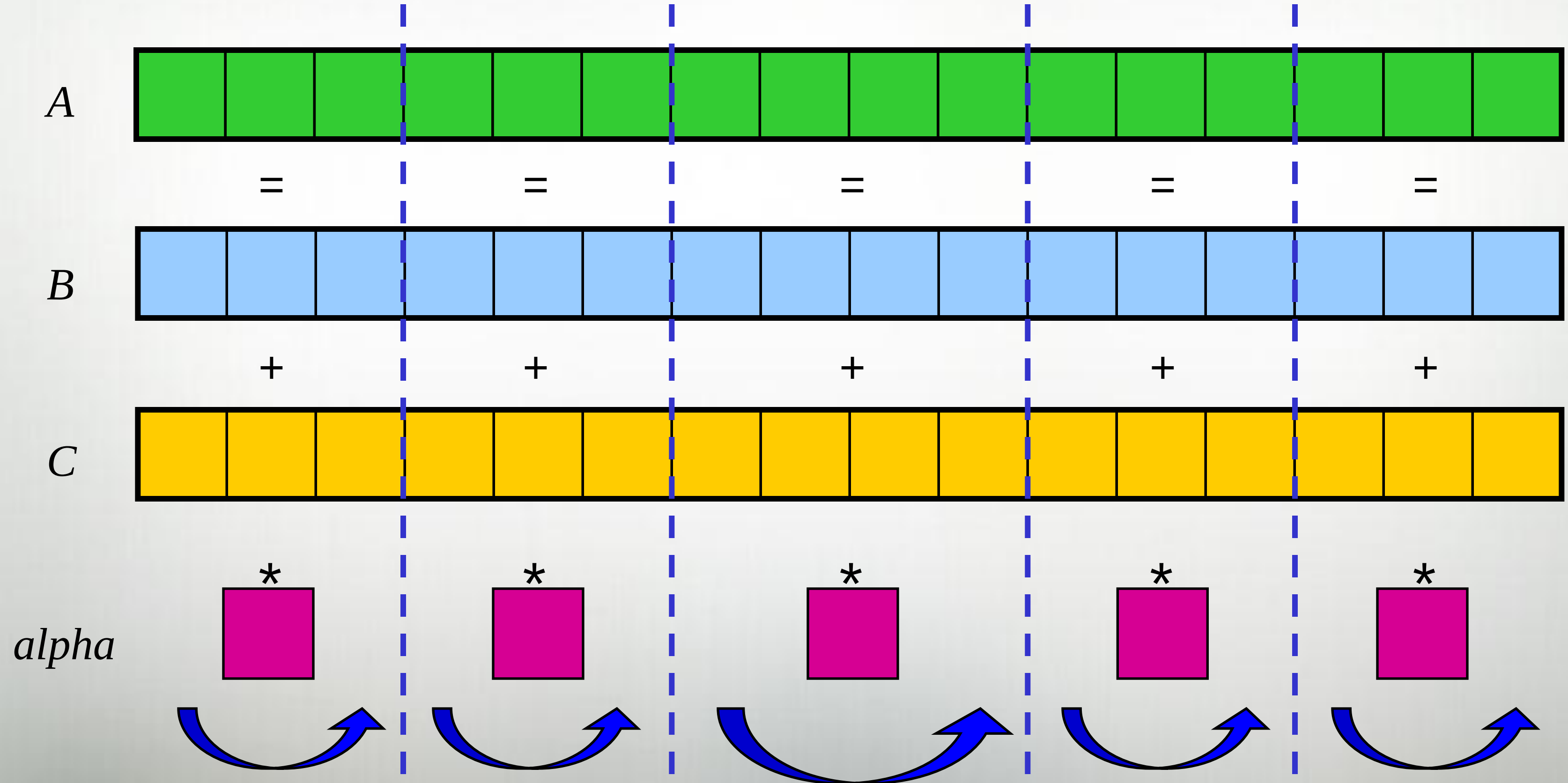


Introduction to STREAM Triad

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];
}
```

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

STREAM Triad: MPI+OpenMP vs. CUDA vs. Chapel

THE SUPERCOMPUTER COMPANY

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 == myRar
  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

Change: It is Happening Again

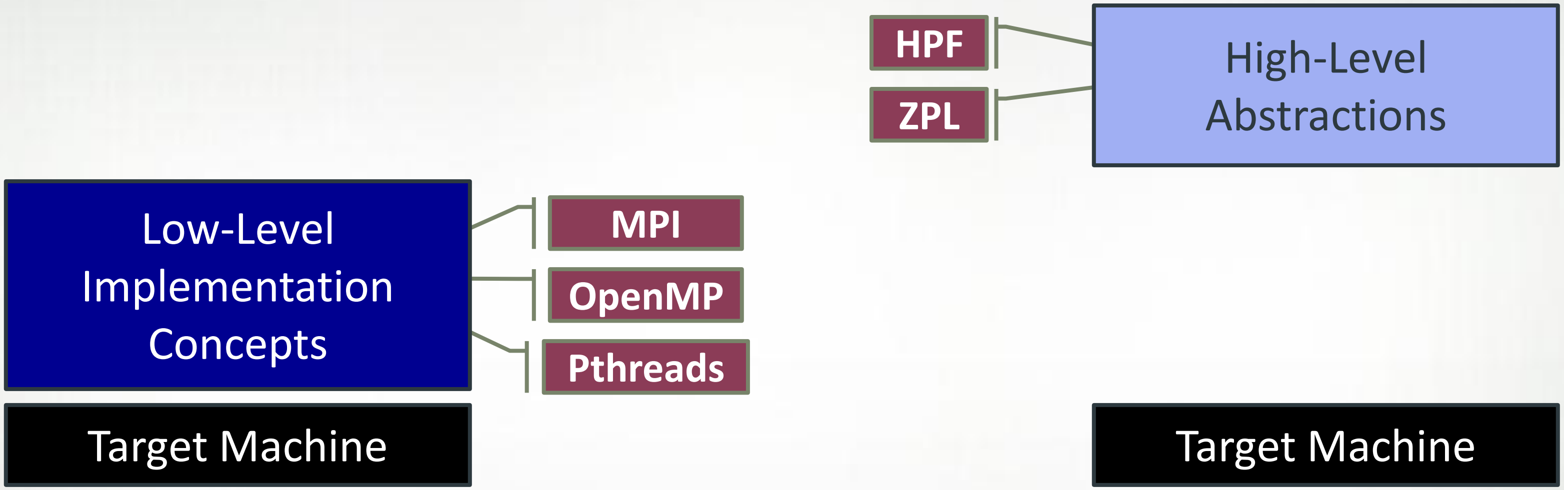
- Exascale is expected to bring new changes/challenges:
  - increased sensitivity to locality within node architectures
  - increased heterogeneity as well
    - multiple processor types
    - multiple memory types
  - limited memory bandwidth, memory::FLOP ratio
  - resiliency concerns
  - power concerns

Exascale represents an opportunity to move to a programming model that is less tied to architecture than those of the past

Outline

- ✓ Chapel Context
- ✓ Motivation
- ➢ Feature Tour
  - Base Language
  - Locality
  - Task Parallelism
  - Data Parallelism
- Advanced Features / Research Topics
- Project Status and Overview
- Chapel and Exascale

Multiresolution Language Design: Motivation



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

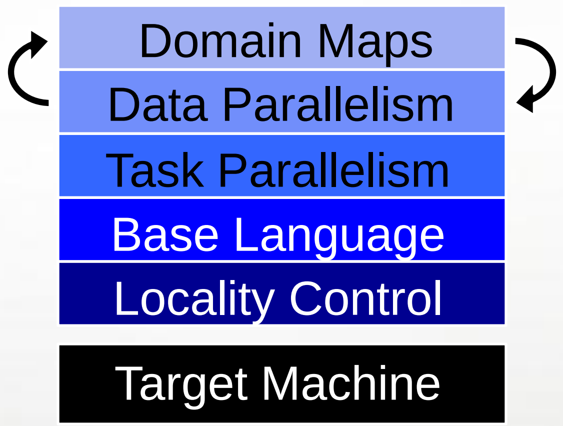
"Why don't I have more control?"

Chapel's Multiresolution Design

Multiresolution Design: Support multiple tiers of features

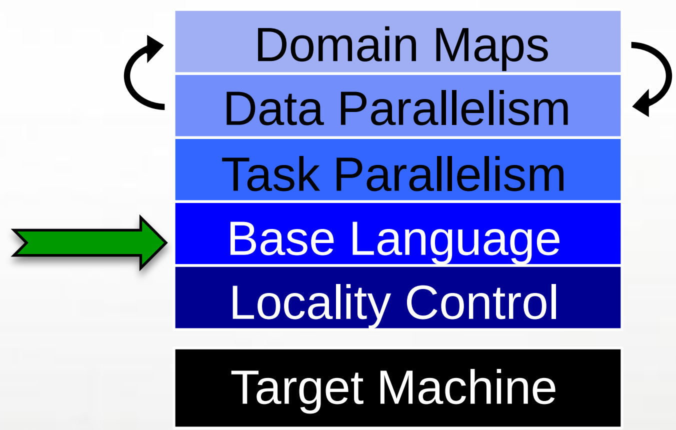
- higher levels for programmability, productivity
- lower levels for greater degrees of control

Chapel language concepts



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

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
```

Zippered Iteration

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

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

writeln(A);
```

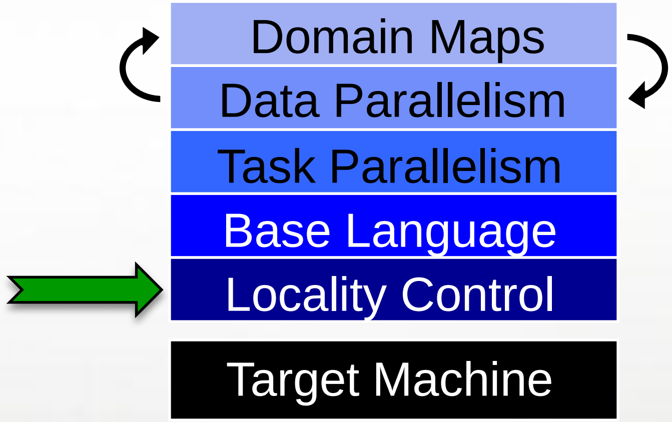
```
2.1 4.2 6.3 8.4 10.5 12.6 14.7 16.8 18.9 21.0
```

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
- argument intents, default values, match-by-name
- overloading, where clauses
- modules (for namespace management)
- …

Come to this afternoon's tutorial for a *slightly* more in-depth survey

Locality Features



The Locale Type

Definition:

- Abstract unit of target architecture
- Supports reasoning about locality
- Capable of running tasks and storing variables
  - i.e., has processors and memory

Typically: A multi-core processor or SMP node

Defining 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;
```

***Locales*:**

L0	L1	L2	L3	L4	L5	L6	L7

Locale Operations

- Locale methods support reasoning about machine resources:

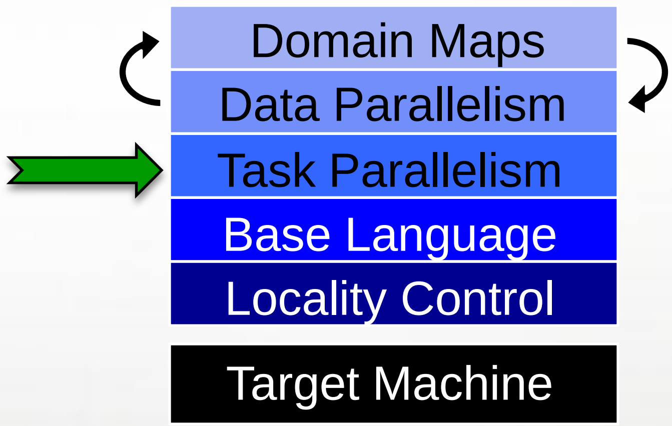
```
proc locale.physicalMemory(…) { … }
proc locale.numCores(…) { … }
proc locale.name(…) { … }
```

- *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);
```

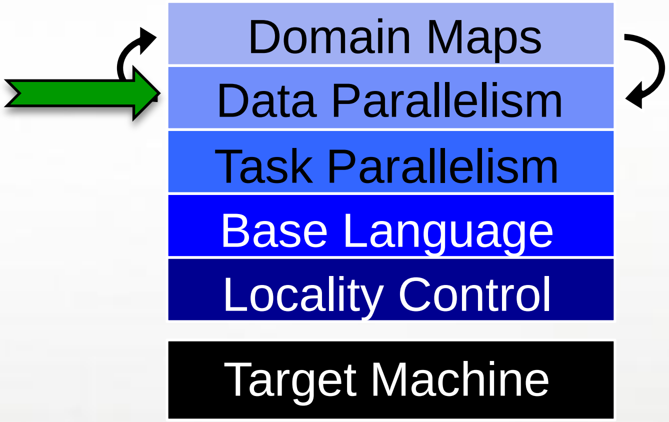
Task Parallel Features



Bounded Buffer Producer/Consumer Example

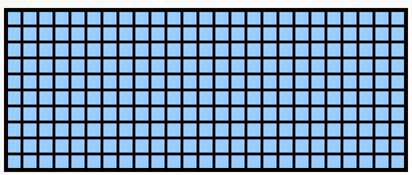
```
cobegin {
  producer();
  consumer();
}

// '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

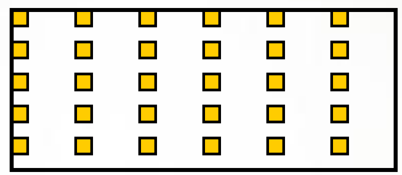


Chapel Domain/Array Types

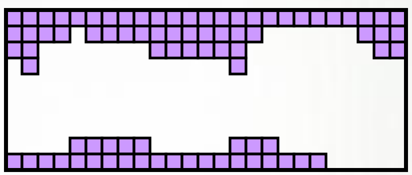
Chapel supports several types of domains and arrays:



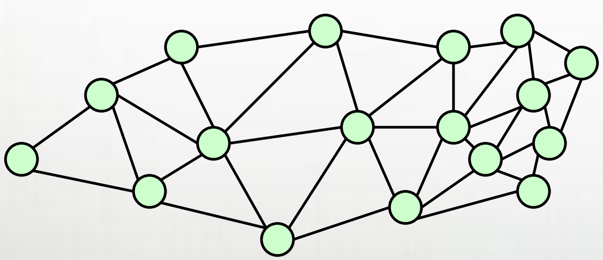
dense



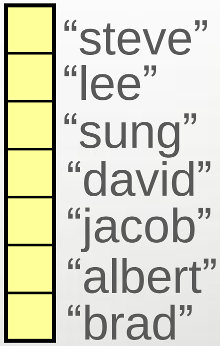
strided



sparse



unstructured



associative

Chapel Domain/Array Operations

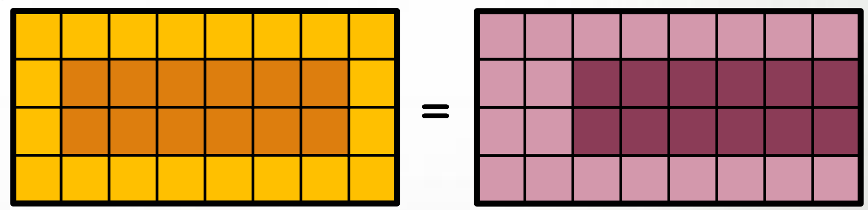
- Parallel and Serial Iteration

```
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, …

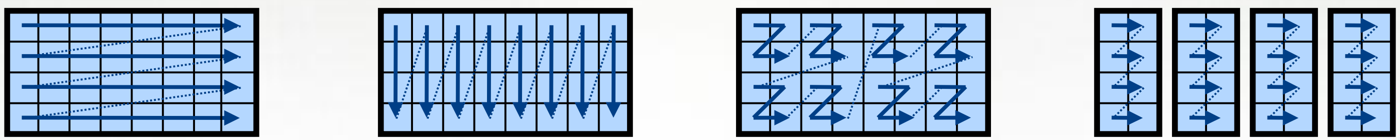
Outline

- ✓ Chapel Context
- ✓ Motivation
- ✓ Feature Tour
- ➢ Advanced Features / Research Topics
  - Domain Maps
  - Leader-Follower Iterators
- Project Status and Overview
- Chapel and Exascale

Data Parallelism Implementation Qs

Q1: How are arrays laid out in memory?

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

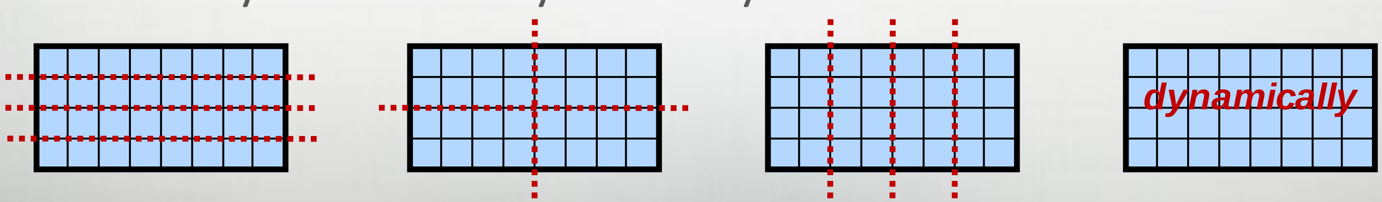


…?

- How are sparse arrays stored? (COO, CSR, CSC, block-structured, …?)
- What memories/memory types are used?

Q2: How are arrays distributed between locales/nodes?

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

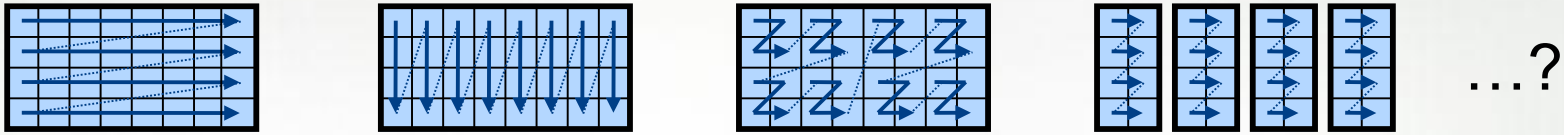


…?

Data Parallelism Implementation Qs

Q1: How are arrays laid out in memory?

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



- How are sparse arrays stored? (COO, CSR, CSC, block-structured, …?)
- What memories/memory types are used?

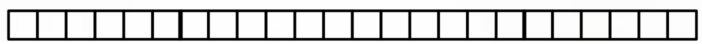
Q2: How are arrays distributed between locales/nodes?

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

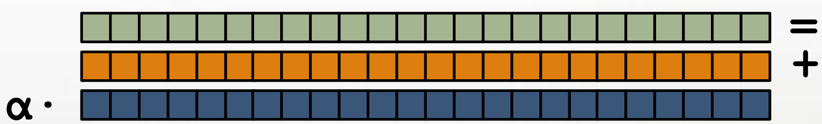
A: Chapel's *domain maps* are designed to give the user full control over such decisions

STREAM Triad: Chapel (multicore)

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



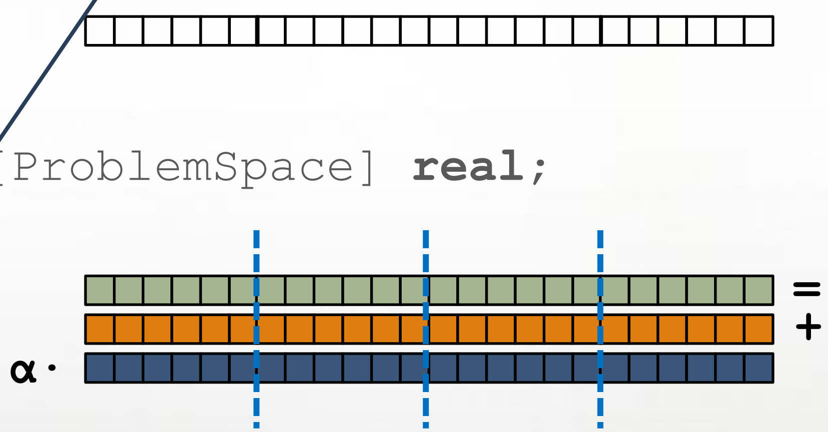
```
var A, B, C: [ProblemSpace] real;
```




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

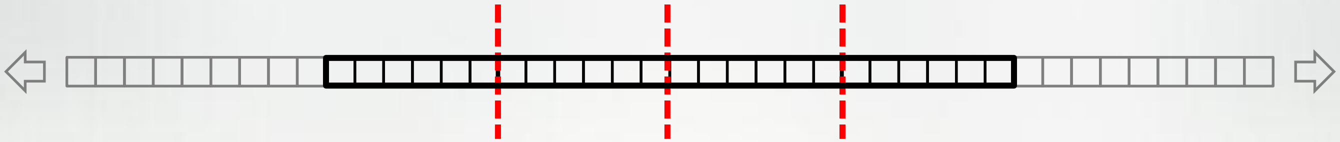
STREAM Triad: Chapel (multicore)

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

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

A = B + alpha * C;
```

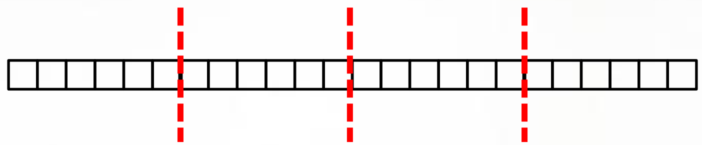


No domain map specified => use default layout
- current locale owns all indices and values
- computation will execute using local processors only

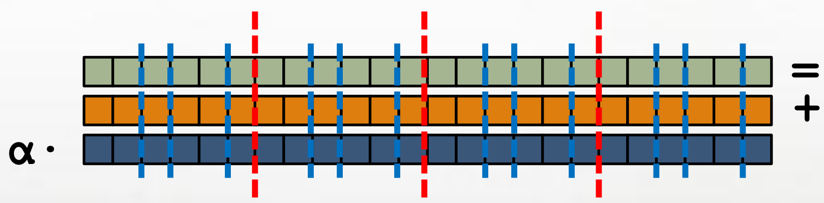
STREAM Triad: Chapel (multinode, blocked)



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

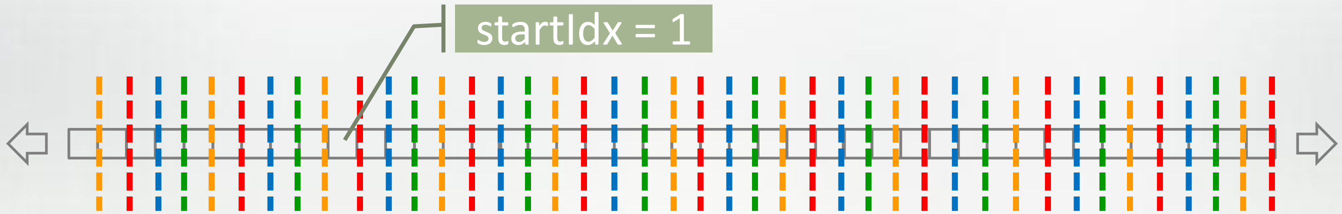


```
var A, B, C: [ProblemSpace] real;
```

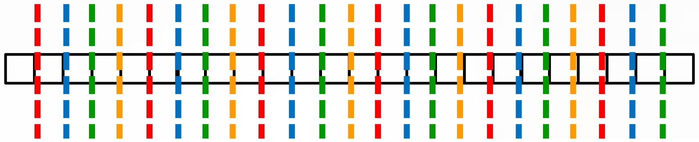



```
A = B + alpha * C;
```

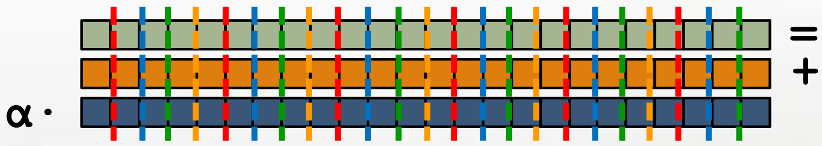
STREAM Triad: Chapel (multinode, cyclic)




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



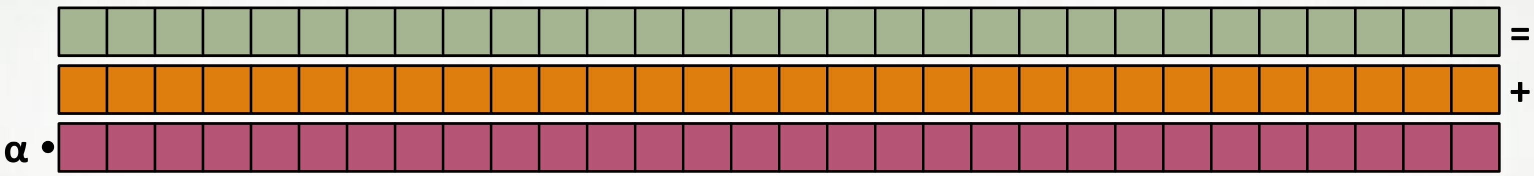
```
var A, B, C: [ProblemSpace] real;
```




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

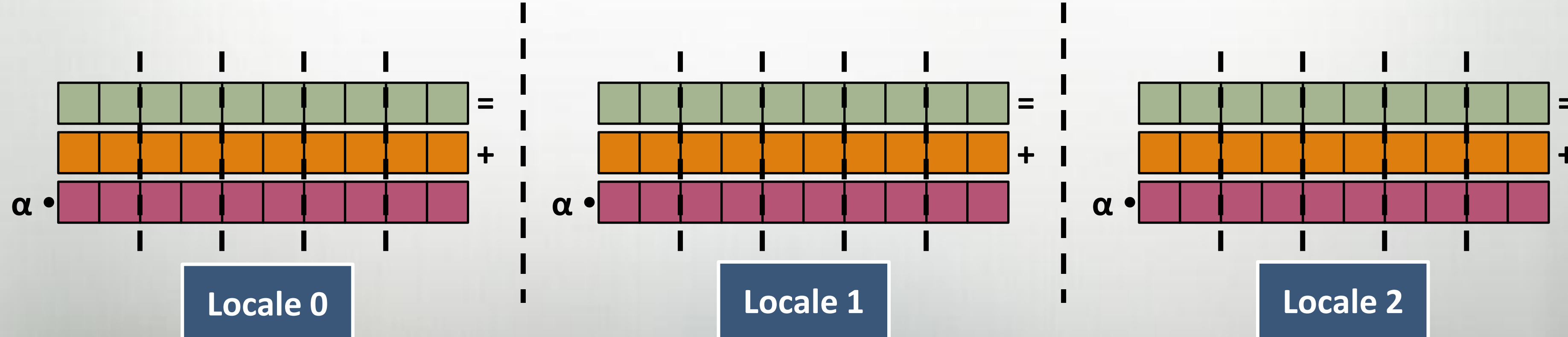
Domain Maps

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:



Domain Maps

Domain Maps: "recipes for implementing parallel/ distributed arrays and domains"

They define data storage:

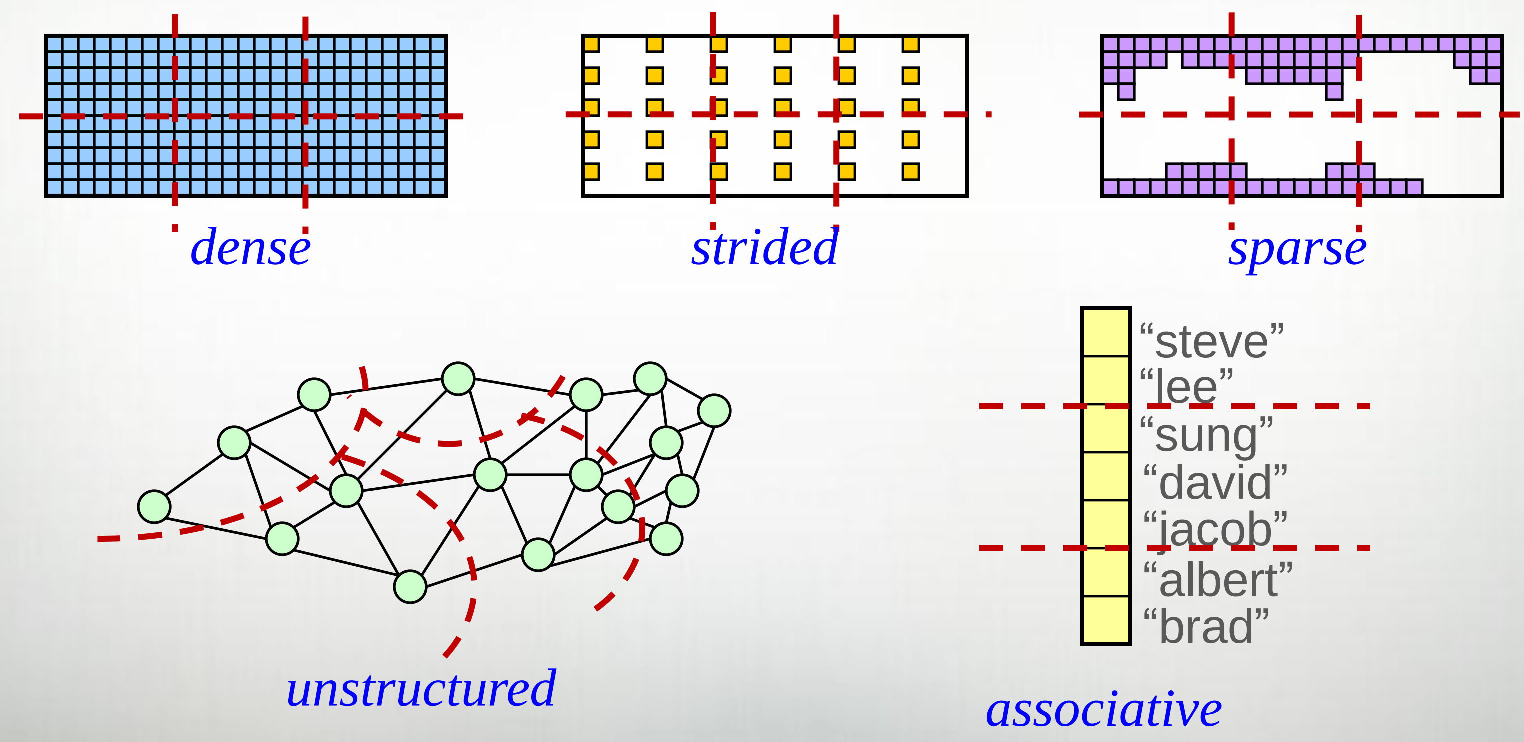
- Mapping of domain indices and array elements to locales
- Layout of arrays and index sets in each locale's memory

…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

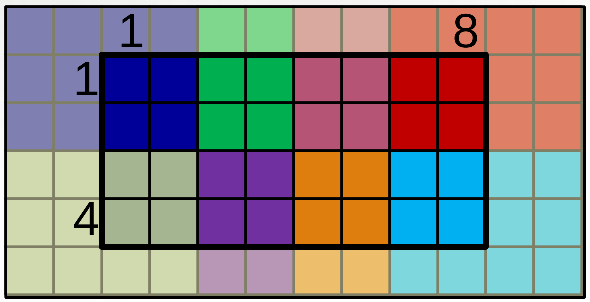
All Domain Types Support Domain Maps

All Chapel domain types support domain maps



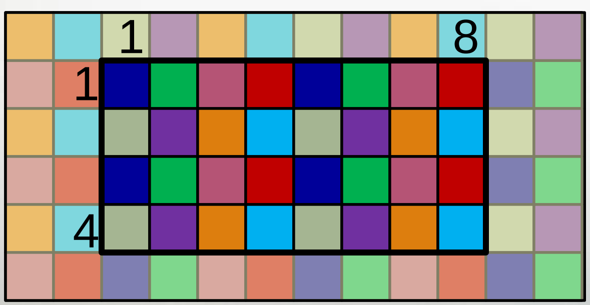
Sample Distributions: Block and Cyclic

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

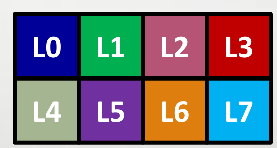


distributed to

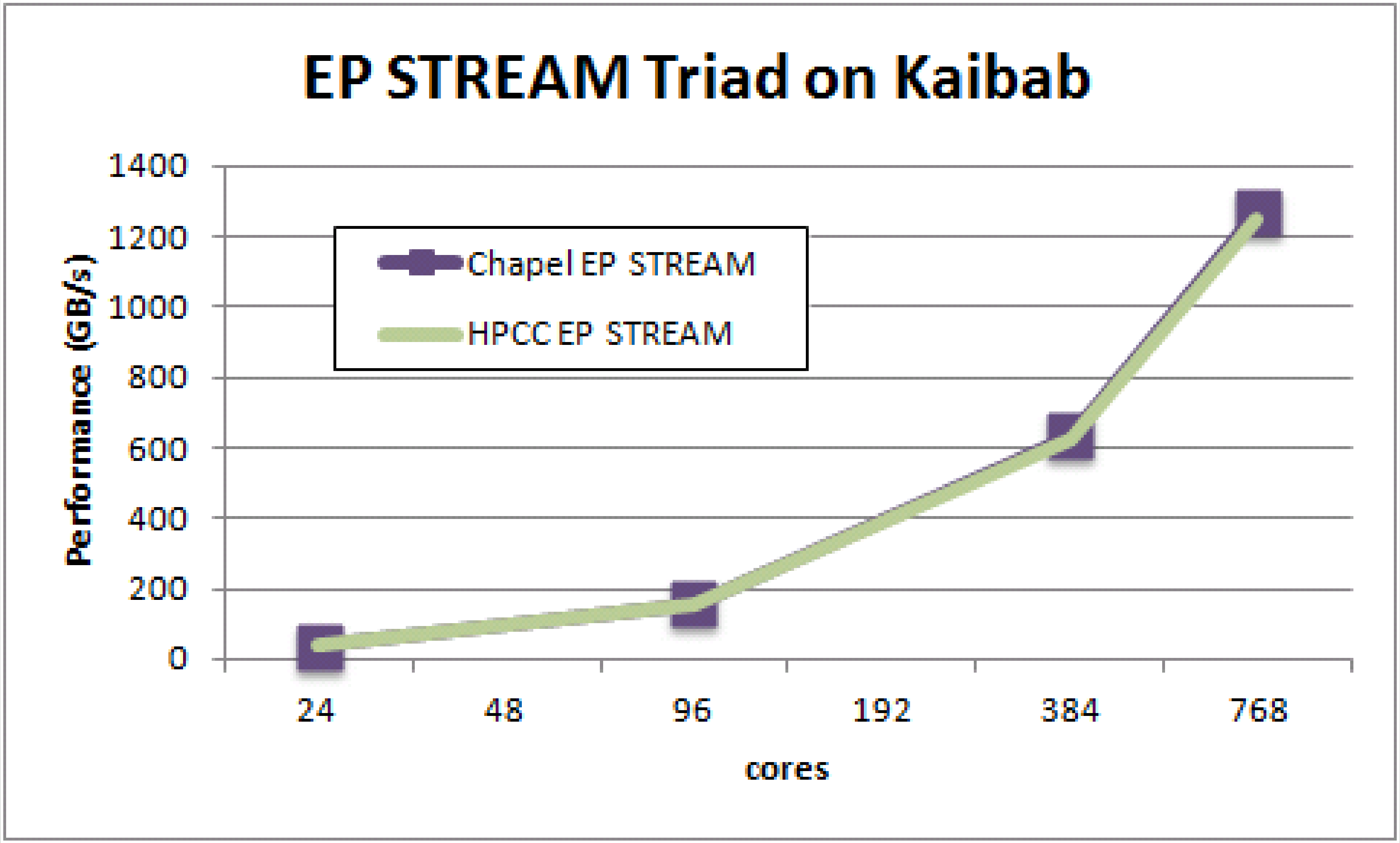

```
var Dom = [1..4, 1..8] dmapped Cyclic( startIdx=(1,1) );
```



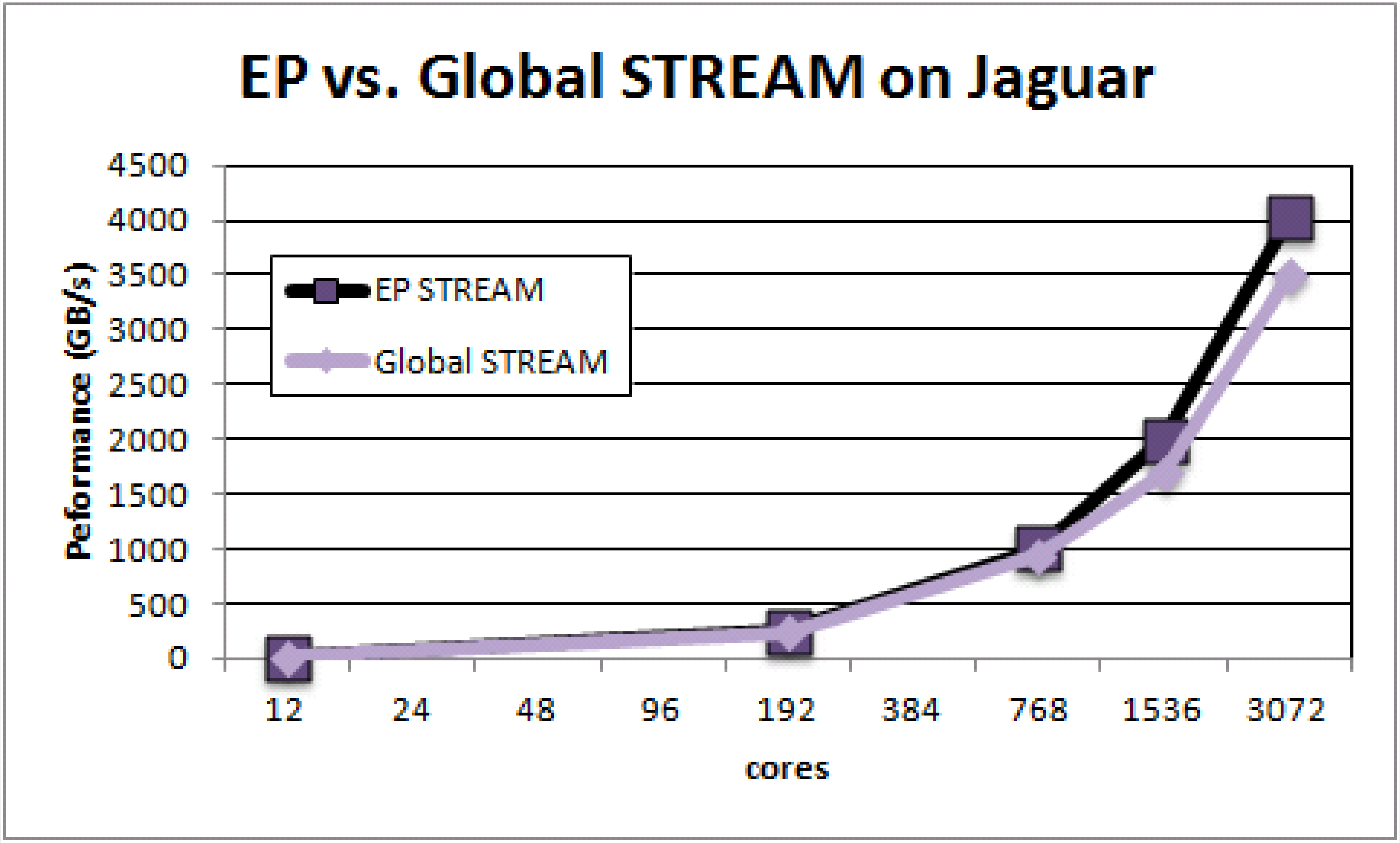
distributed to



HPCC Stream Performance on Kaibab (XE6)



HPCC Global STREAM on Jaguar (XT5)



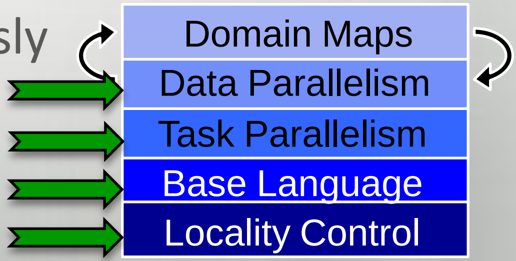
Chapel’s Domain Map Philosophy

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



For More Information on Domain Maps

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

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

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

More Data Parallelism Implementation Qs

Q3: How are data parallel loops implemented?

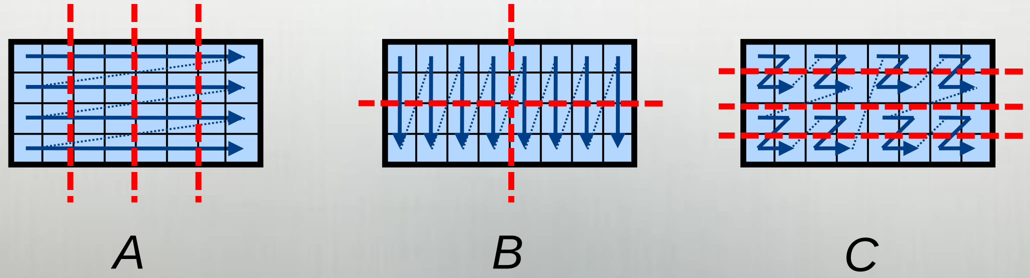
```
forall i in B.domain do B[i] = i/10.0;
forall c in C do c = 3.0;
```

- How many tasks? Where do they execute?
- How is the iteration space divided between the tasks?

Q4: How are parallel zippered loops implemented?

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

- Particularly given that the iterands might have incompatible distributions, memory layouts, and parallelization strategies



More Data Parallelism Implementation Qs

Q3: How are data parallel loops implemented?

```
forall i in B.domain do B[i] = i/10.0;
forall c in C do c = 3.0;
```

- How many tasks? Where do they execute?
- How is the iteration space divided between the tasks?

Q4: How are parallel zippered loops implemented?

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

- Particularly given that the iterands might have incompatible distributions, memory layouts, and parallelization strategies

A: Chapel's *leader-follower* iterators are designed to give users full control over such decisions

Leader-Follower Iterators: Definition

- Chapel defines all zippered forall loops in terms of leader-follower iterators:
  - *leader iterators:* create parallelism, assign iterations to tasks
  - *follower iterators:* serially execute work generated by leader

- Given…

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

…A is defined to be the *leader*

…*A*, *B*, and *C* are all defined to be *followers*

Leader-Follower Iterators: Rewriting

- *Conceptually*, the Chapel compiler translates:

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

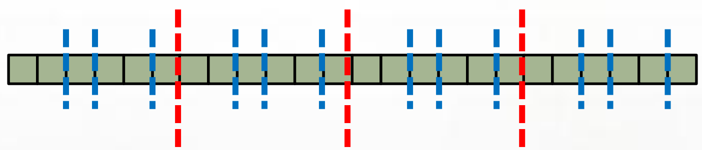
into:

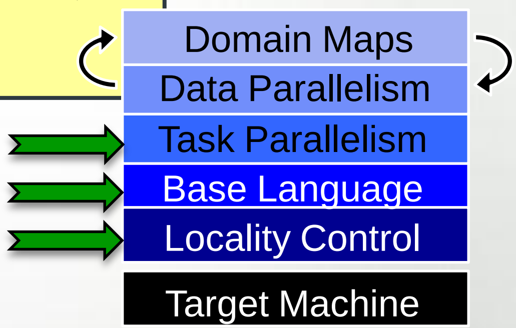
```
inlined A.lead() iterator, which yields work…
  for (a,b,c) in (A.follow(work),
                  B.follow(work)
                  C.follow(work)) do
    a = b + alpha * c;
```

Writing 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 computeMyChunk(loc.id, tid);
}
```





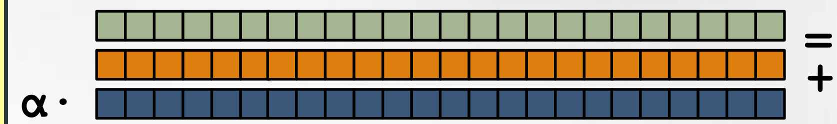
Follower iterators simply use serial features:

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

Leader-Follower Iterators: Rewriting

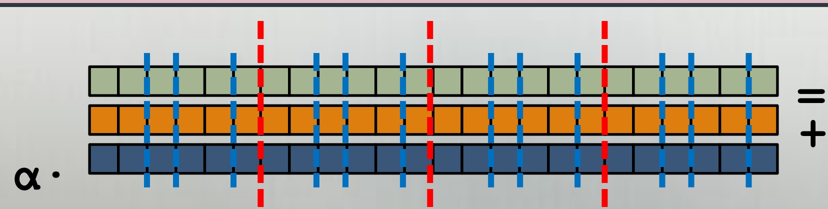
- Given the previous leader iterators…

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



…would get rewritten by the Chapel compiler as:

```
coforall loc in Locales do
  on loc do
    coforall tid in here.numCores {
      const work = computeMyChunk(loc.id, tid);
      for (a,b,c) in (A.follow(work),
                      B.follow(work)
                      C.follow(work)) do
        a = b + alpha * c;             }
```



Leader-Follower Iterators…

…permit users to write high-level parallel loops…

- …without tripping over all of the low-level details
- while still able to reason about them semantically
- and to create new loop schedules without compiler mods

…provide clear answers to our questions:

- Chapel semantics define a leader for each data parallel loop
- Leader iterators decide…
  - how many tasks to use
  - where the tasks execute
  - what work each task owns
- Followers are responsible for yielding corresponding iterations – even if they aren't local
  - gives them control over communication granularity/approach

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

```
const ProblemSize = [1..n] dmapped BlockCyclic(start=1,
                                               blocksize=64);
var A, B, C: [ProblemSize] real;

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

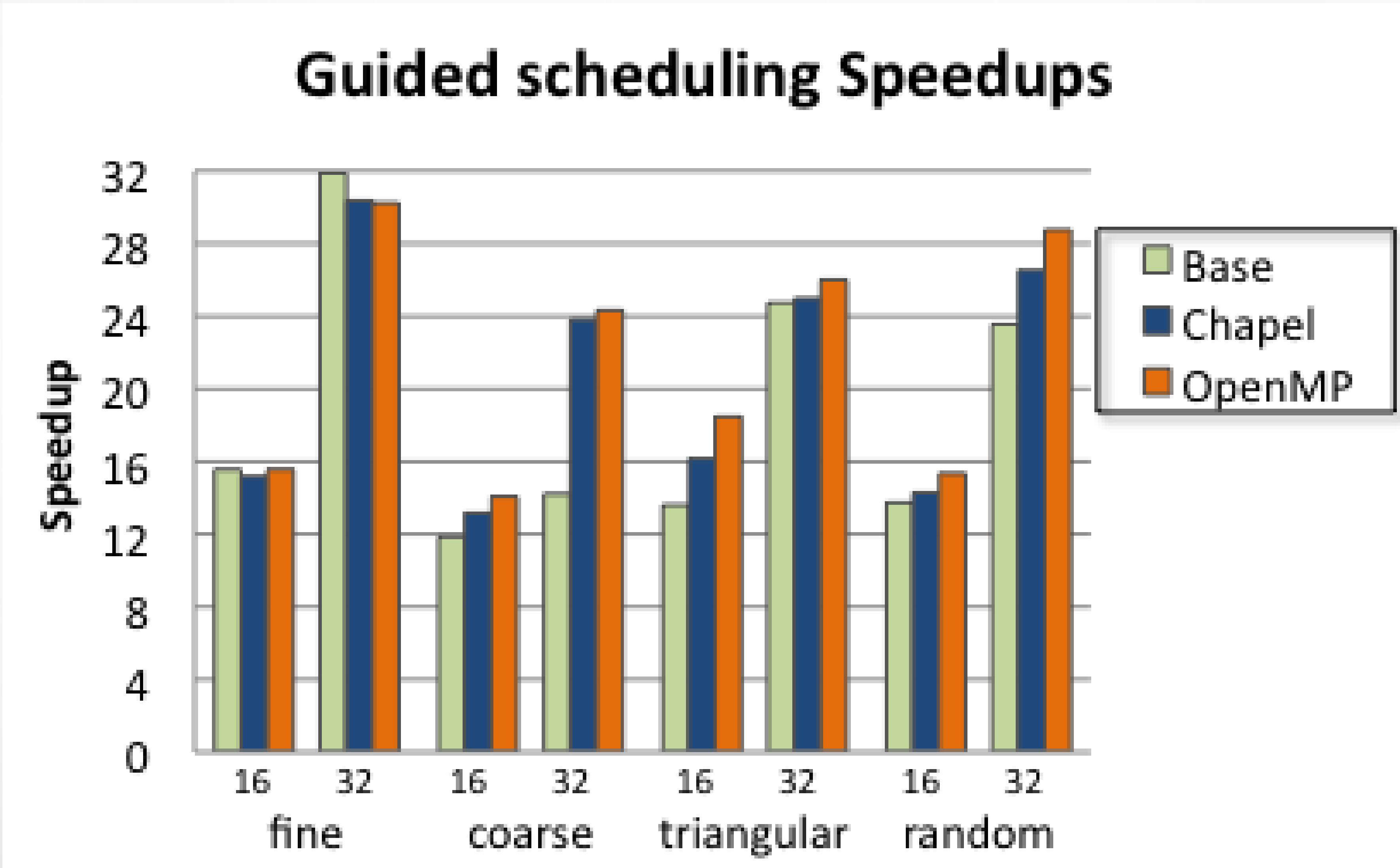
Change the array's default leader by changing its domain map (perhaps to one that you wrote yourself).

Controlling Data Parallelism

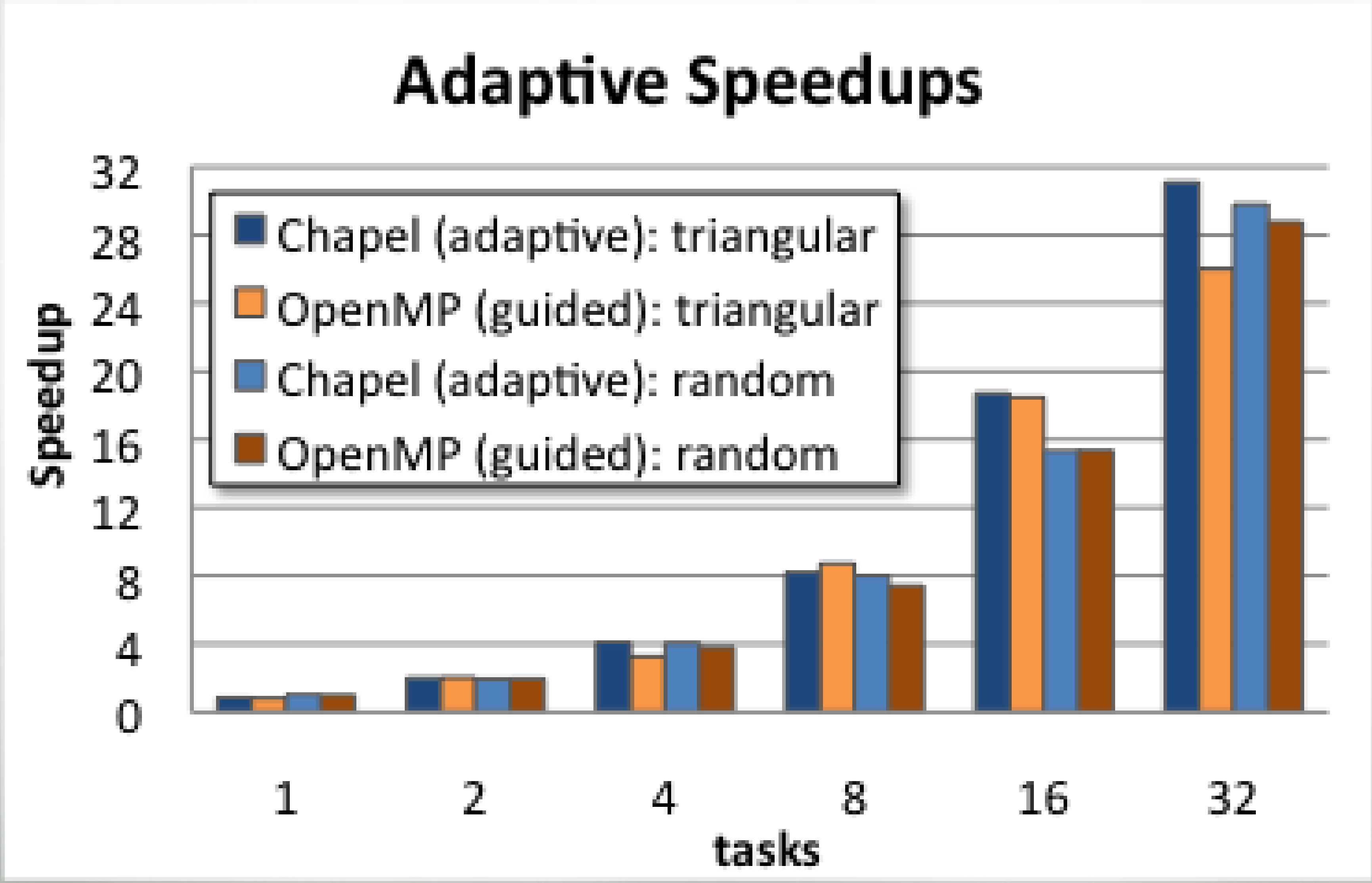
```
forall (a,b,c) in (dynamic(A, chunk=64), B, C) do
  a = b + alpha * c;
```

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

Chapel vs. OpenMP Guided



Chapel Adaptive vs. OpenMP Guided



For More Information on Leader-Follower Iterators

PGAS 2011: *User-Defined Parallel Zippered Iterators in Chapel,* Chamberlain, Choi, Deitz, Navarro; October 2011

Chapel release:

- See the *AdvancedIters* module, described in the "Standard Modules" section of the language specification for some interesting leader-follower iterators:
  - OpenMP-style dynamic schedules
  - work-stealing iterators

Summary of This Section

- Chapel avoids locking crucial implementation decisions for HPC into the language design
  - local and distributed array implementations
  - parallel loop implementations

- Instead, these can be…

  …specified in the language by an advanced user

  …switched between with minimal code changes

Outline

- ✓ Chapel Context
- ✓ Motivation
- ✓ Feature Tour
- ✓ Advanced Features / Research Topics
- ➢ Project Status and Overview
- ➢ Chapel and Exascale

Implementation Status -- Version 1.4.0 (Oct 2011)

In a nutshell:

- Most features work at a functional level
- Many performance optimizations remain

This is a good time to:

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

“I sorta like Chapel… How can I help?”

Give Chapel a try to see whether it’s on a useful path for your computational idioms

- if not, help us course correct
- pair programming with us is a good approach
- evaluate performance based on potential, not present

Let others know about your interest in Chapel

- your colleagues and management
- Cray leadership
- the broader parallel community (HPC and mainstream)

Contribute to the project

- code, collaborations, funding

Join Our Growing Community

- Cray:


Brad Chamberlain


Sung-Eun Choi


Greg Titus


Vass Litvinov


Tom Hildebrandt

???
(open reqs)

- External Collaborators:


Albert Sidelnik (UIUC)


Jonathan Turner (CU Boulder)


Kyle Wheeler (Sandia)

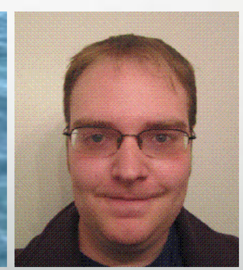
• • •

You? Your Friend/Student/ Colleague?

- Interns:


Jonathan Claridge (UW)


Hannah Hemmaplardh (UW)


Andy Stone (Colorado State)


Jim Dinan (OSU)


Rob Bocchino (UIUC)


Mackale Joyner (Rice)

• • •

Featured Collaborations (see chapel.cray.com/collaborations.html for details)

- **Tasking using Qthreads:** Sandia (Rich Murphy, Kyle Wheeler, Dylan Stark)
  - **paper at CUG, May 2011**
- **Interoperability using Babel/BRAID:** LLNL (Tom Epperly, Adrian Prantl, et al.)
  - **paper at PGAS, Oct 2011**
- **Dynamic Iterators:**
- **Bulk-Copy Opt:** U Malaga (Rafael Asenjo, Maria Angeles Navarro, et al.)
- **Parallel File I/O:**
  - **paper at ParCo, Aug 2011**
- **Improved I/O & Data Channels:** LTS (Michael Ferguson)
- **CPU-GPU Computing:** UIUC (David Padua, Albert Sidelnik, Maria Garzarán)
  - **tech report, April 2011**
- **Interfaces/Generics/OOP:** CU Boulder (Jeremy Siek, Jonathan Turner)
- **Tasking over Nanos++:** BSC/UPC (Alex Duran)
- **Tuning/Portability/Enhancements:** ORNL (Matt Baker, Jeff Kuehn, Steve Poole)
- **Chapel-MPI Compatibility:** Argonne (Rusty Lusk, Pavan Balaji, Jim Dinan, et al.)

Collaboration Ideas (see chapel.cray.com/collaborations.html for details)

- memory management policies/mechanisms
- dynamic load balancing: task throttling and stealing
- parallel I/O and checkpointing
- exceptions; resiliency
- application studies and performance optimizations
- index/subdomain semantics and optimizations
- targeting different back-ends (LLVM, MS CLR, …)
- runtime compilation
- library support
- tools: debuggers, performance analysis, IDEs, interpreters, visualizers
- database-style programming
- autotuning
- (your ideas here…)

Change: It is Happening Again

- Exascale is expected to bring new changes/challenges:
  - increased sensitivity to locality within node architectures
  - increased heterogeneity as well
    - multiple processor types
    - multiple memory types
  - limited memory bandwidth, memory::FLOP ratio
  - resiliency concerns
  - power concerns

Exascale represents an opportunity to move to a programming model that is less tied to architecture than those of the past

Chapel and Exascale

- In many respects, Chapel is well-positioned for exascale:
  - distinct concepts for parallelism and locality
  - not particularly tied to any hardware architecture
  - supports arbitrary nestings of data and task parallelism
- In others, it betrays that it was a petascale-era design
  - locales currently only support a single level of hierarchy
  - lack of fault tolerance/error handling/resilience

  (these were both considered "version 2.0" features)

We are addressing these shortcomings as current/future work

Summary

Higher-level programming models can help insulate science from implementation

- yet, without necessarily abandoning control
- Chapel does this via its multiresolution design

Exascale represents an opportunity to move to architecture-independent programming models

- ones that support general styles of parallel programming
- ones that separate issues of locality from parallelism

Next Steps

No-brainers:

- Performance Optimizations
- Feature Improvements/Bug Fixes
- Support Users and Collaborations

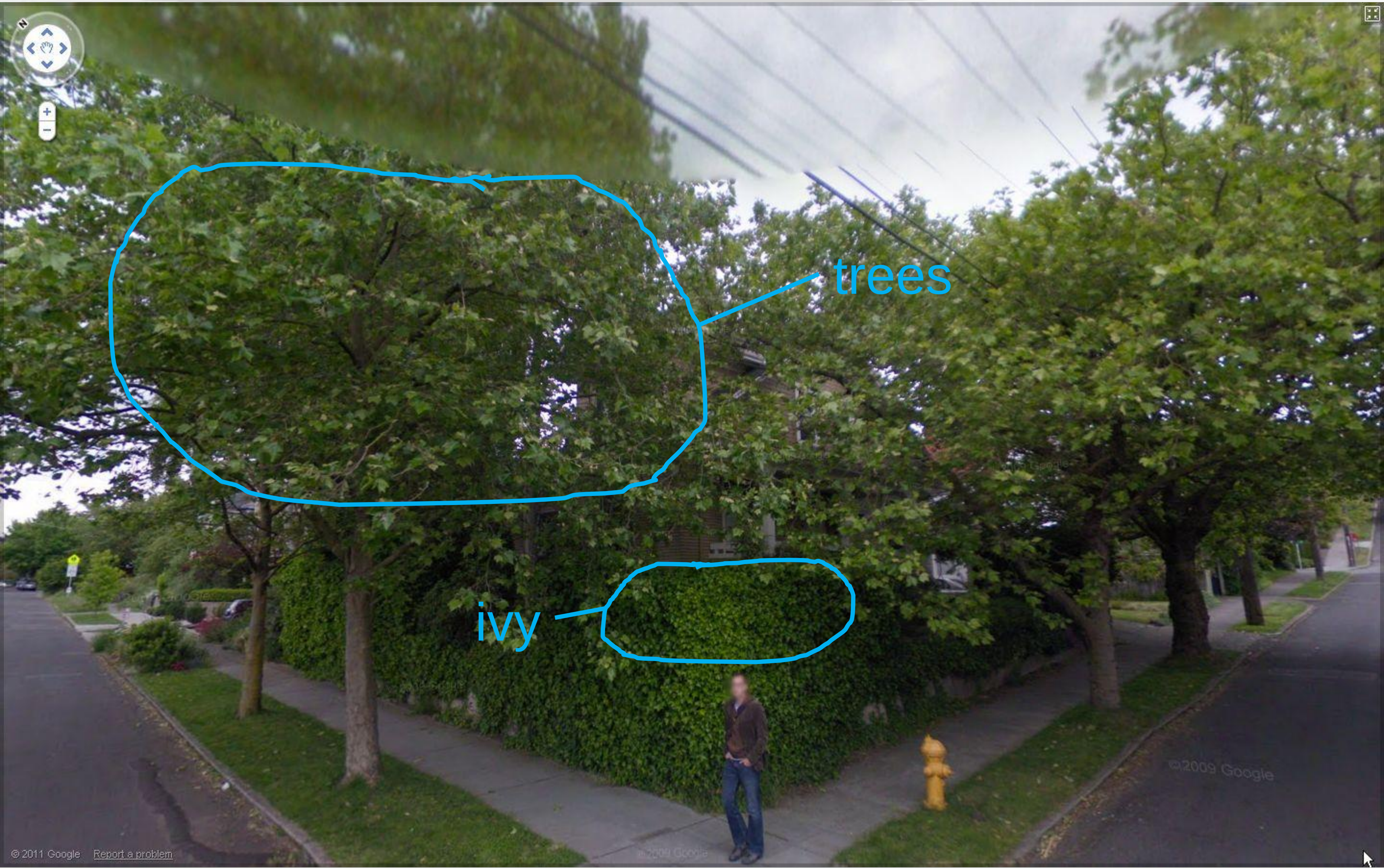
More advanced topics:

- Hierarchical Locales to target manycore/CPU+GPUs
  - additional hierarchy and heterogeneity warrants it
- Resiliency/Fault Tolerance
- Develop post-HPCS strategy/funding

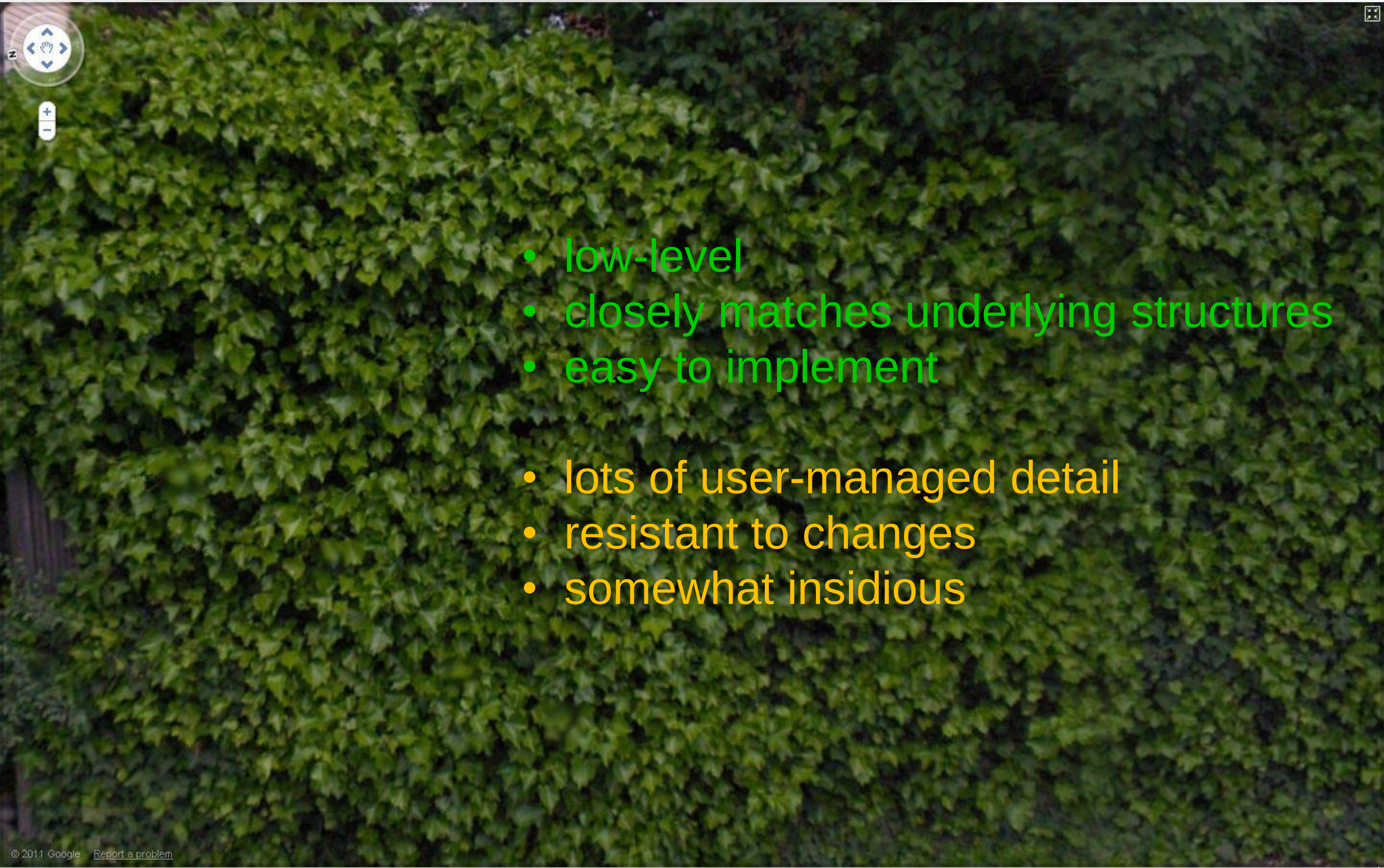
Chapel 5-year Plan: Key Components

- **Advisory Board**
  - help steer Chapel team's priorities on a regular basis
    - performance vs. features vs. a mix of both
    - which optimizations and features to prioritize
    - which benchmarks, idioms to focus on
- **Agile milestones rather than *a priori***
  - dynamically react to community's needs, R&D challenges
- **Improve openness of project, transition to community**
- **Unified Chapel reporting**
  - rather than reporting to several programs, Chapel is the program
  - reduces reporting burden, permitting team to focus more on work
  - brings those interested in Chapel to a single meeting

A Seattle Corner



Ivy



- low-level
- closely matches underlying structures
- easy to implement

- lots of user-managed detail
- resistant to changes
- somewhat insidious

Trees

- higher-level
- more elegant, structured

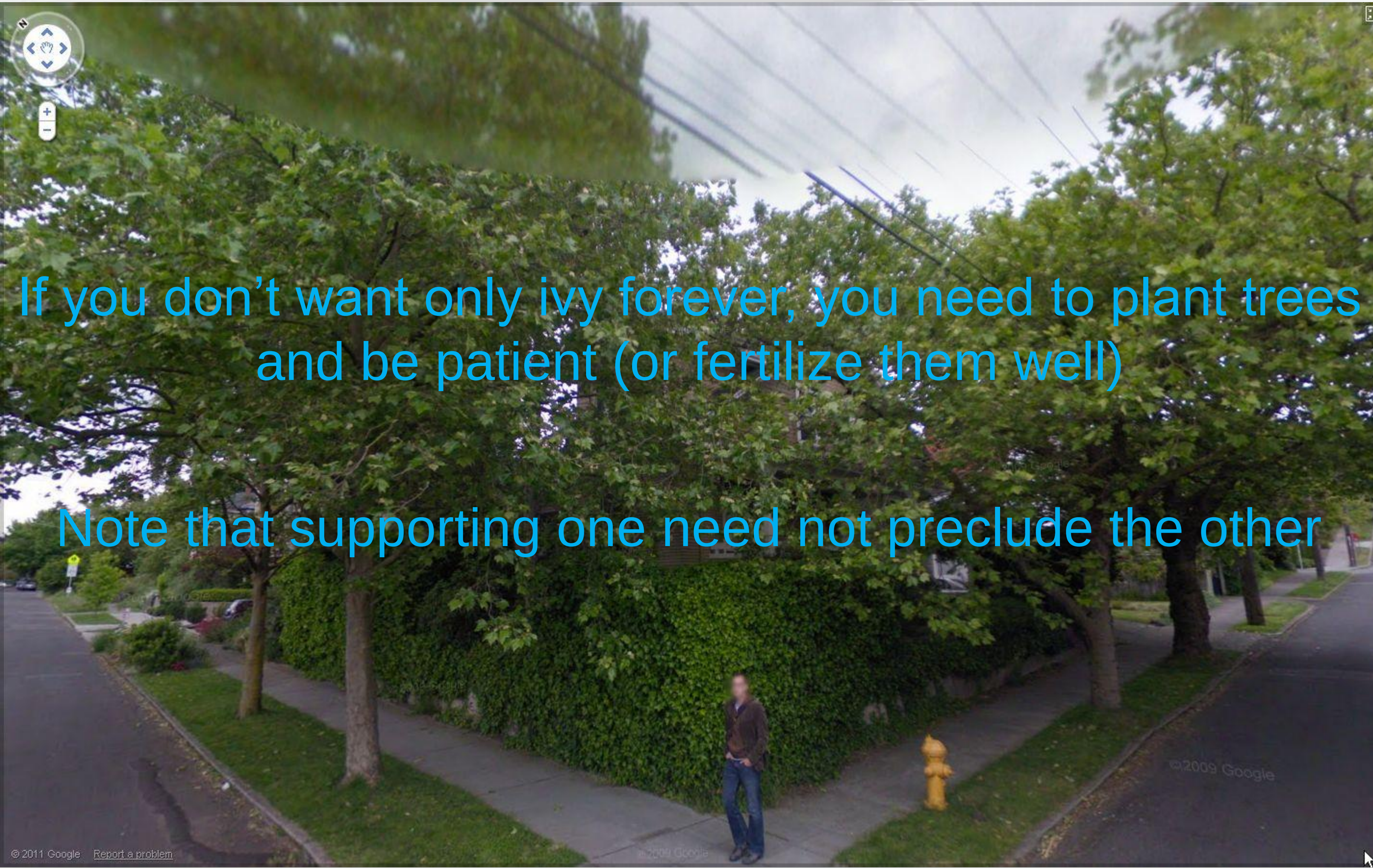
- requires a certain investment of time and force of will to establish

Landscaping Quotes from the HPC community

Early HPCS years:

- “The HPC community tried to plant a tree once. It didn’t survive. Nobody should ever bother planting one again.”
- “Why plant a tree when you can’t be assured it will grow?”
- “Why would anyone ever want anything other than ivy?”
- “We’re in the business of building treehouses that last 40 years; we can’t afford to build one in the branches of your sapling.”
- “This sapling looks promising. I’d like to climb it <u>now</u>!”

A Corner in Seattle: Takeaways



If you don’t want only ivy forever, you need to plant trees and be patient (or fertilize them well)

Note that supporting one need not preclude the other

For More Information

Chapel project page: http://chapel.cray.com

- overview, papers, presentations, language spec, …

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

- release downloads, public mailing lists, code repository, …

Mailing Lists:

- chapel_info@cray.com: contact the team
- chapel-users@lists.sourceforge.net: user-oriented discussion list
- chapel-developers@lists.sourceforge.net: dev.-oriented discussion
- chapel-education@lists.sourceforge.net: educator-oriented discussion
  chapel-bugs@lists.sourceforge.net: public bug forum
- chapel_bugs@cray.com: private bug mailing list

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