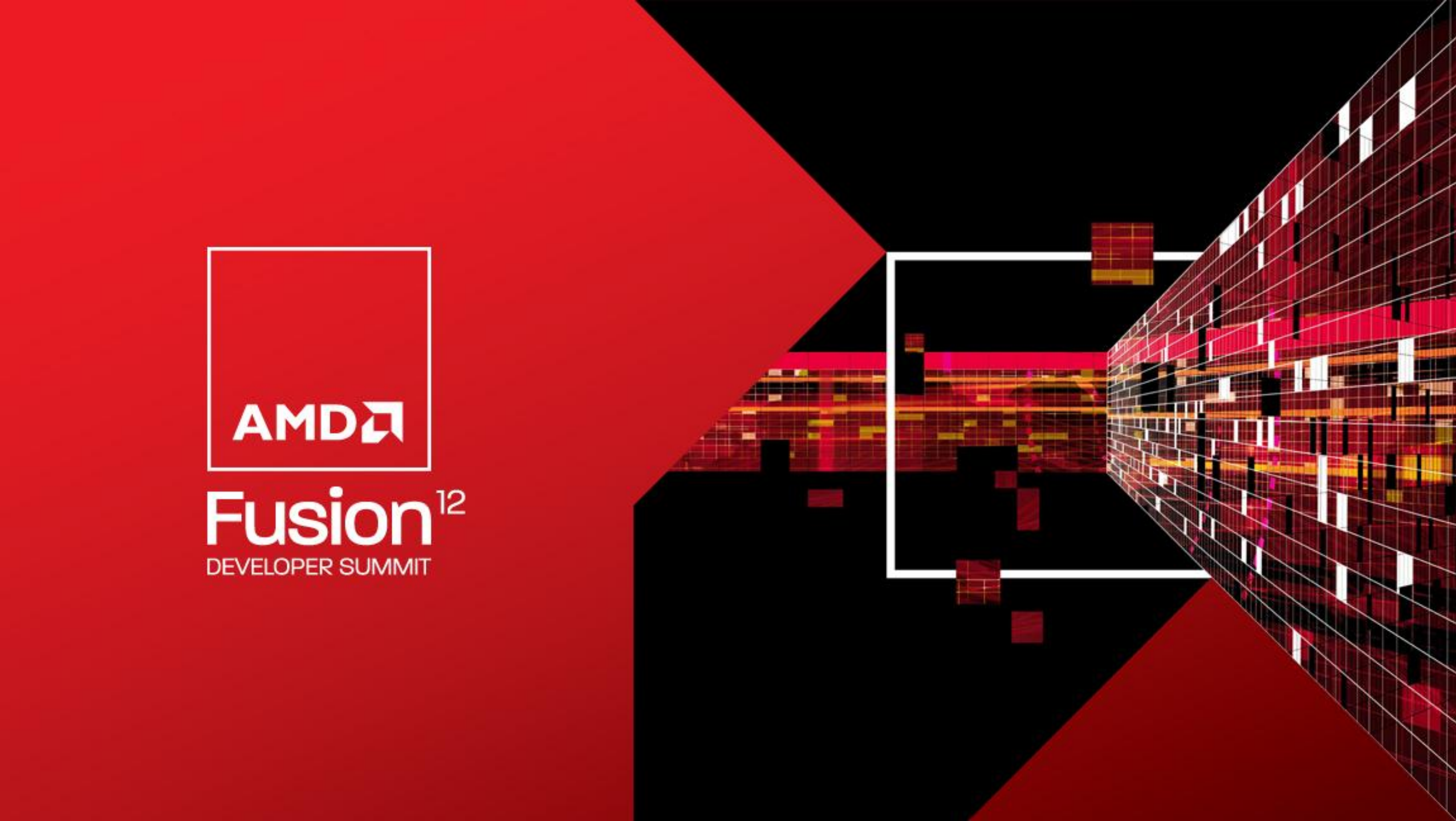
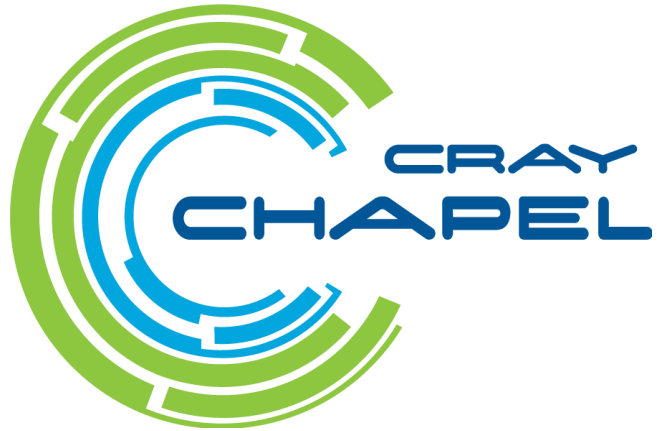

AMD
Fusion 12
DEVELOPER SUMMIT

AMD Fusion 12 DEVELOPER SUMMIT

CHAPEL

Multiresolution Parallel Programming

Brad Chamberlain
Cray Inc.
Principal Engineer



CRAY®
THE SUPERCOMPUTER COMPANY

Sustained Performance Milestones

1 GF – 1988: Cray Y-MP; 8 Processors

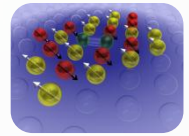
- Static finite element analysis

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 w/ Programming Models

1 GF – 1988: Cray Y-MP; 8 Processors

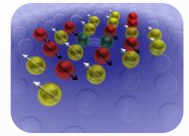
- Static finite element analysis
- Fortran77 + Cray autotasking + vectorization

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

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

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

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



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

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

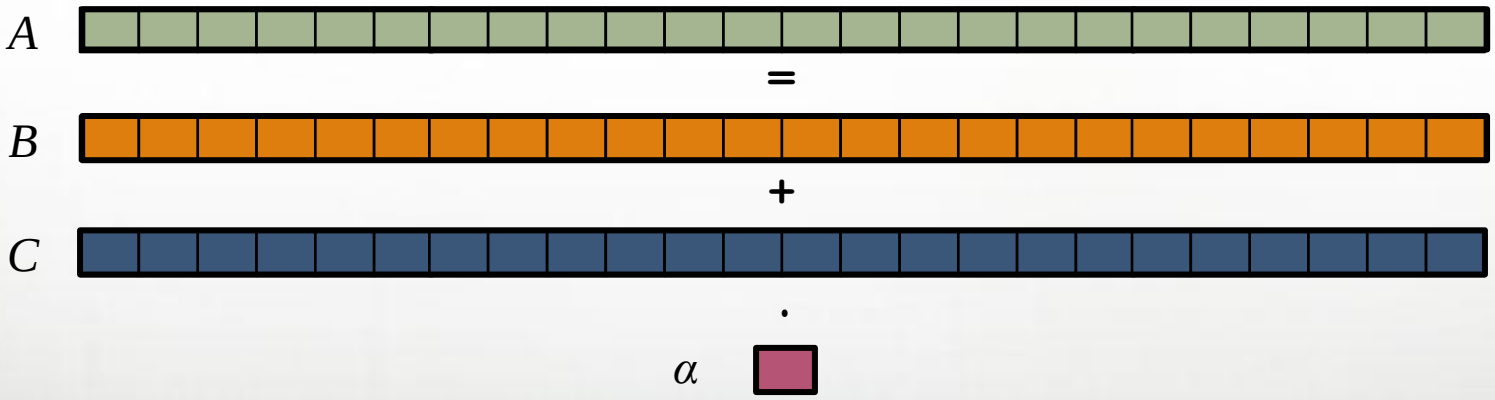
Or Perhaps Something Completely Different?

STREAM Triad: a trivial parallel computation

Given: m-element vectors A, B, C

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

In pictures:

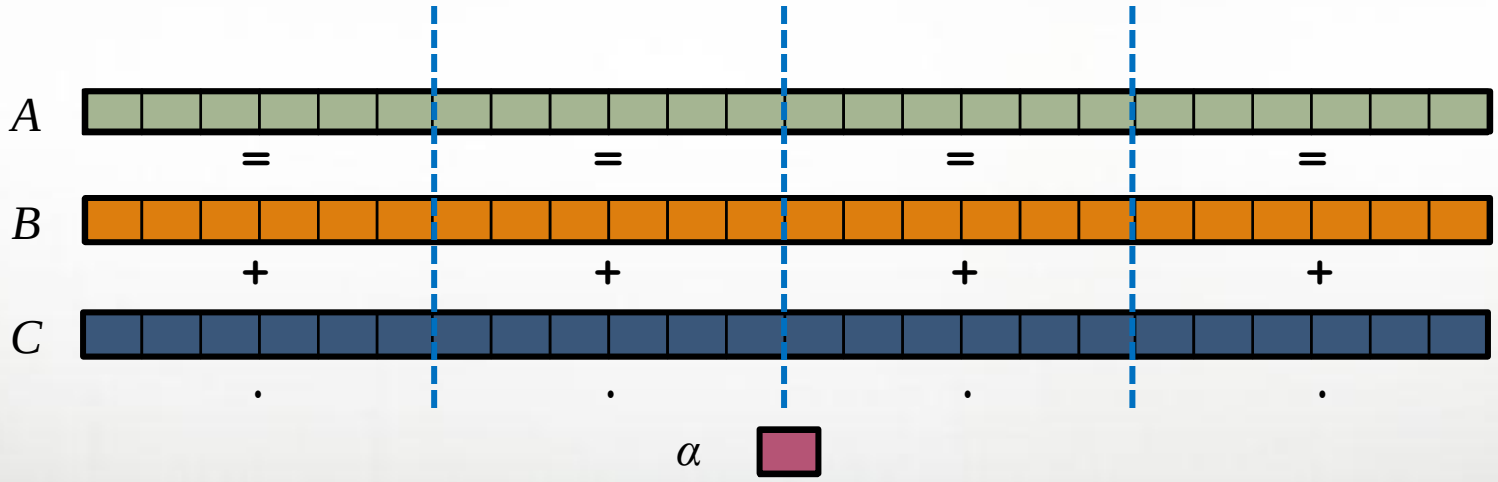


STREAM Triad: a trivial parallel computation

Given: m-element vectors A, B, C

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

In pictures, in parallel:

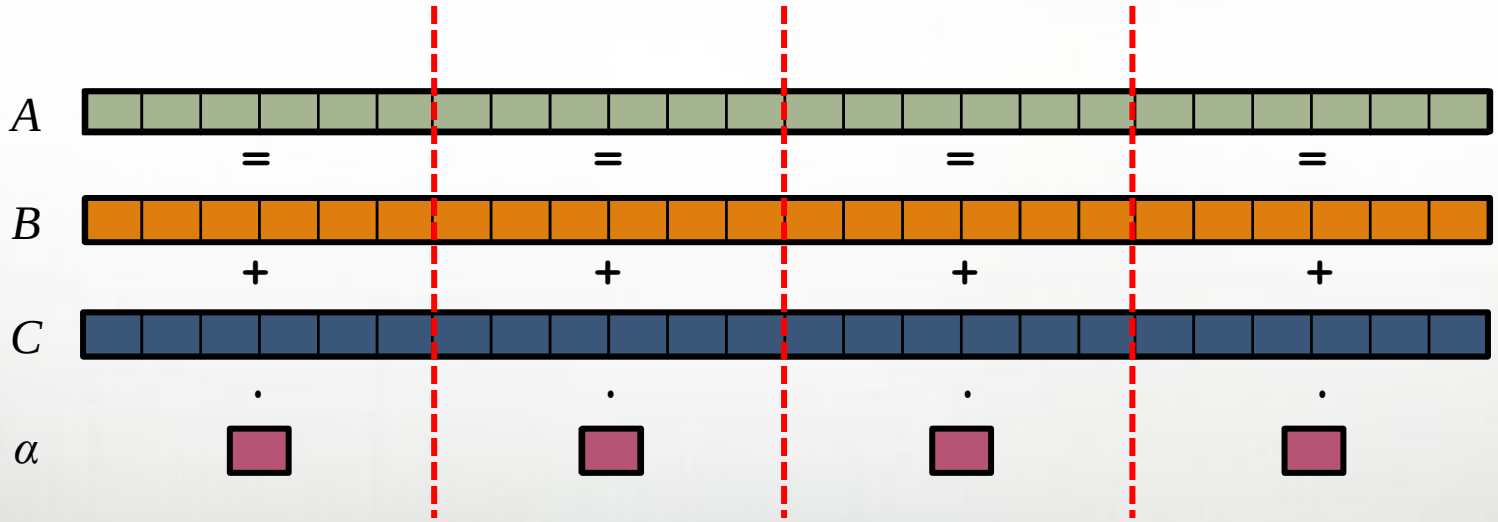


STREAM Triad: a trivial parallel computation

Given: m-element vectors A, B, C

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

In pictures, in parallel (distributed memory):

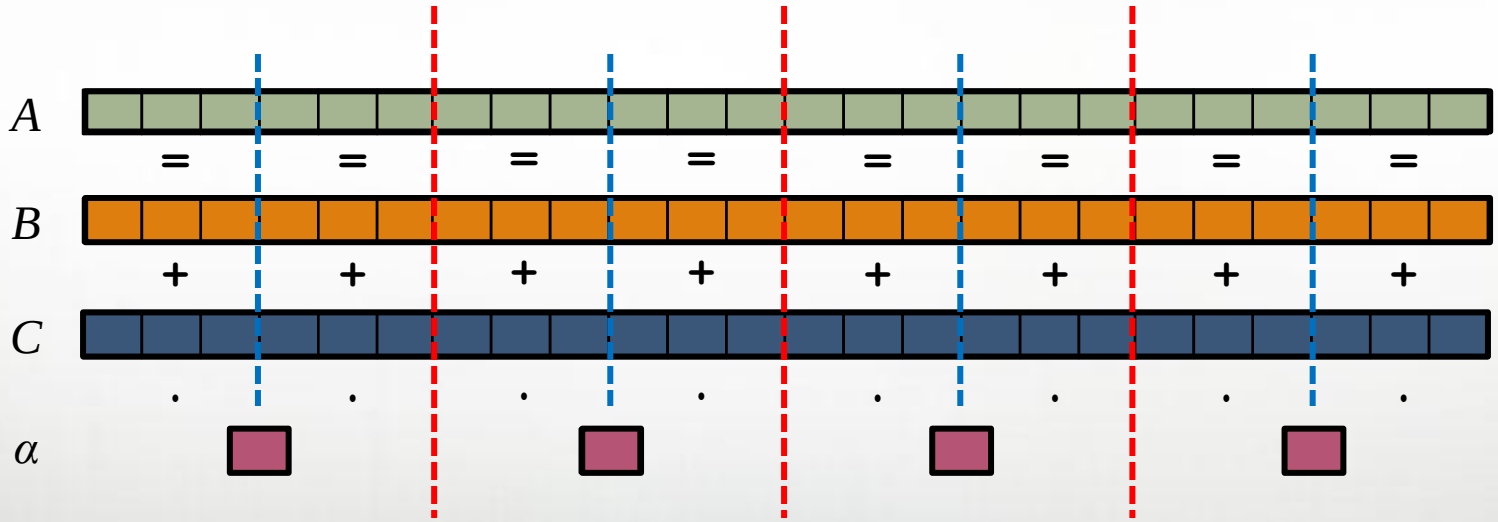


STREAM Triad: a trivial parallel computation

Given: m-element vectors A, B, C

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

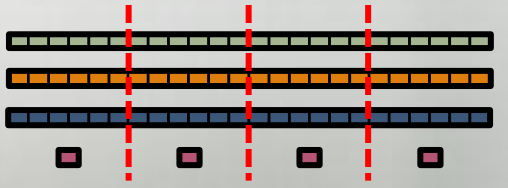
In pictures, in parallel (distributed memory multicore):



STREAM Triad: MPI

MPI

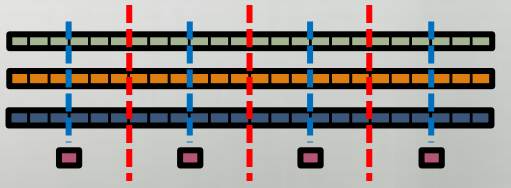
```
#include <hpcc.h>

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

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

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

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

  return errCount;
}

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

  VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(double), 0 );

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

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

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

  scalar = 3.0;

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

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

  return 0;
}
```



STREAM Triad: MPI+OpenMP

MPI + OpenMP

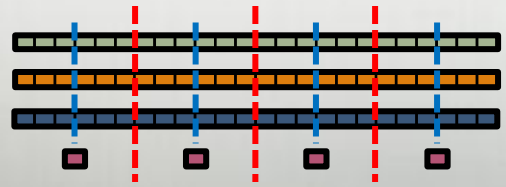
```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

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

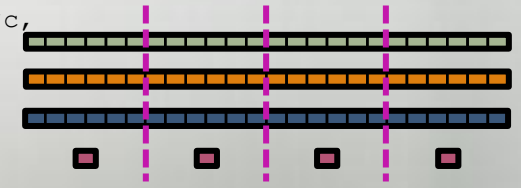
MPI + OpenMP

```
#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];
}
```



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

Examples:

Type of HW Parallelism	Programming Models	Unit of Parallelism
Inter-node	MPI/UPC/CAF	executable
Intra-node/multicore	OpenMP/pthreads	iteration/task
Instruction-level vectors/multi-threading	vendor-specific pragmas	iteration
GPU/accelerator	CUDA/OpenCL/OpenAcc	SIMD function/task

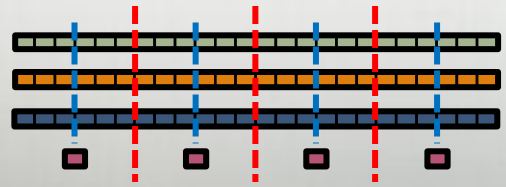
benefits: lots of control; decent generality; relatively easy to implement

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

Rewinding a few slides…

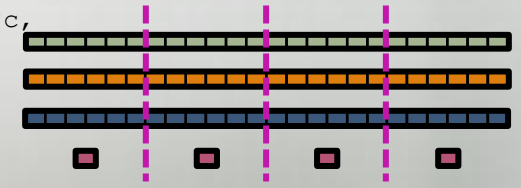
MPI + OpenMP

```
#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];
}
```



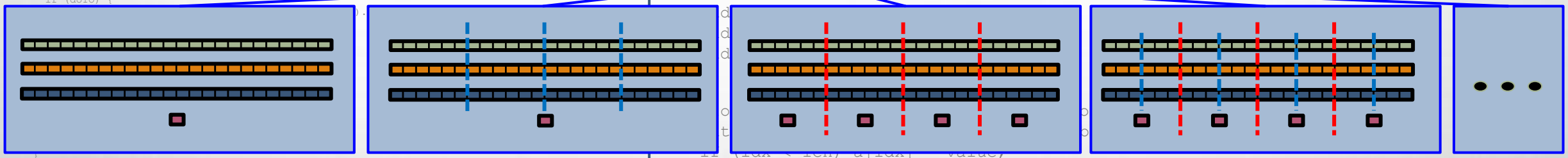
HPC suffers from too many distinct notations for expressing parallelism and locality

STREAM Triad: Chapel

Chapel

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

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

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

B = 2.0;
C = 3.0;

A = B + alpha * C;
```

the special sauce



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

Outline

- ✓ Motivation
- ➢ Chapel Background and Themes
- Tour of Chapel Concepts
- Chapel and Exascale
- Wrap-up

What is Chapel?

- An emerging parallel programming language
  - Design and development led by Cray Inc.
    - in collaboration with academia, 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:**
  - Cray architectures
  - multicore desktops and laptops
  - commodity clusters
  - systems from other vendors
  - *in-progress:* CPU+accelerator hybrids, manycore, …

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

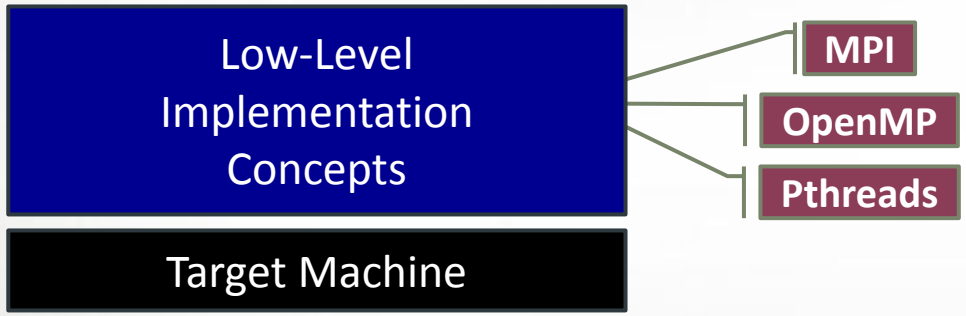
- **Styles:** data-parallel, task-parallel, concurrency, nested, …
- **Levels:** model, function, loop, statement, expression

...target all parallelism available in the hardware

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

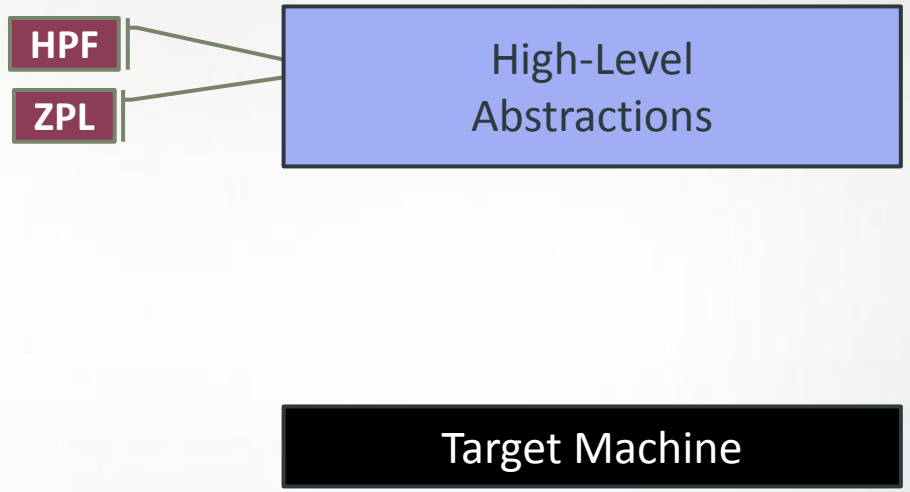
Style of HW Parallelism	Programming Models	Unit of Parallelism
Inter-node	Chapel	executable
Intra-node/multicore	Chapel	iteration/task
Instruction-level vectors/multi-threading	Chapel	iteration
GPU/accelerator	Chapel	SIMD function/task

3) Multiresolution Design: Motivation



"Why is everything so tedious/difficult?"

"Why don't my programs port trivially?"

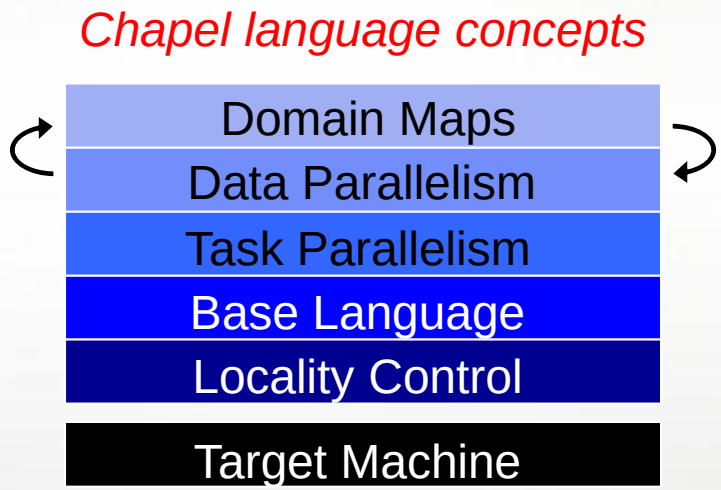


"Why don't I have more control?"

3) Multiresolution Design

Multiresolution Design: Support multiple tiers of features

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



- 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, Perl, Python
- Yet HPC programming is dominated by Fortran, C/C++, MPI

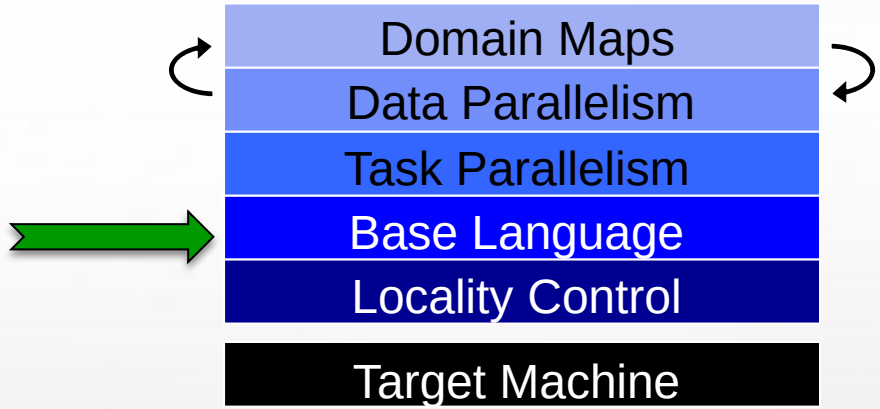
We'd like to narrow this gulf with Chapel:

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

Outline

- ✓ Motivation
- ✓ Chapel Background and Themes
- ➢ Tour of Chapel Concepts
- Chapel and Exascale
- Wrap-up

Base Language Features



Static Type Inference

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

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

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

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

```
(4.14, bradford)
```

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

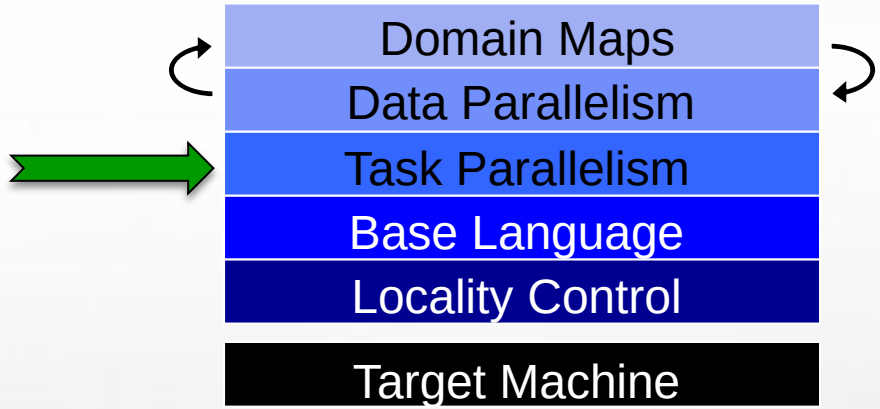
```
for ij in tiledRMO([1..n, 1..n], 2) do
  write(ij);
```

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

Other Base Language Features

- range types and algebra
- zippered iteration
- tuple 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)
- …

Task Parallel Features



Coforall Loops

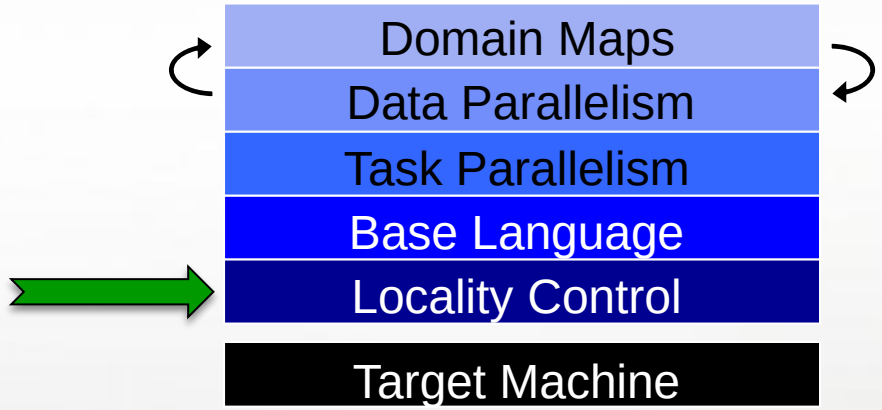
```
coforall t in 0..#numTasks do
  writeln("Hello from task ", t, " of ", numTasks);

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

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

Other Task Parallel Features

- *begin* statements for fire-and-forget tasks
- *cobegin* statements for heterogeneous tasks
- *sync* and *atomic variables* for data-centric synchronization

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

Locale Operations

- Locale methods support queries about target system:

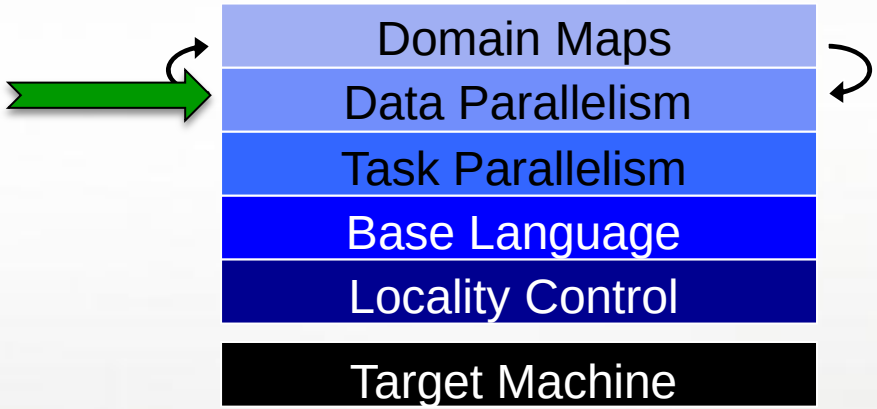
```
proc locale.physicalMemory(…) { … }
proc locale.numCores { … }
proc locale.id { … }
proc locale.name { … }
```

- *On-clauses* support placement of computations:

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

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

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

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

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

Data Parallel Features



Chapel Data Parallel Operations

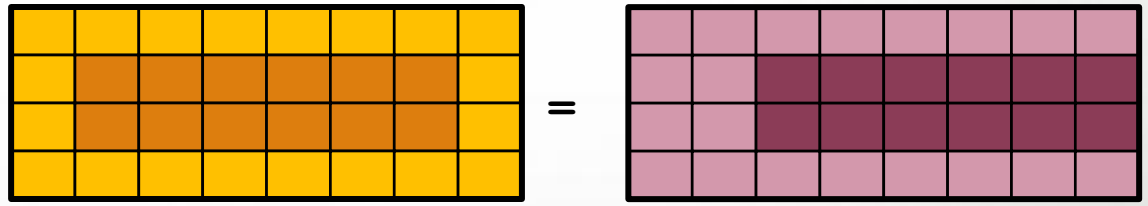
- Parallel Iteration

```
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 Operators and Functions

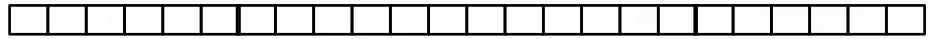
```
A = B + alpha * C;
```

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

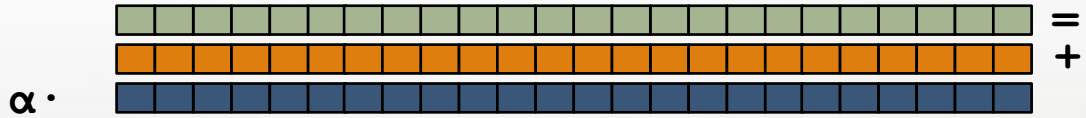
- And several others…

STREAM Triad: Chapel

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



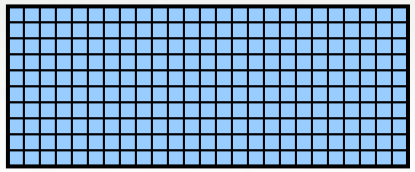
```
var A, B, C: [ProblemSpace] real;
```



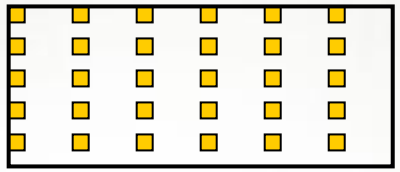

```
A = B + alpha * C;
```

Chapel Array Types

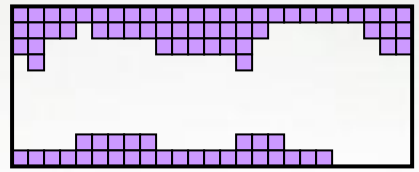
Chapel supports several types of arrays…



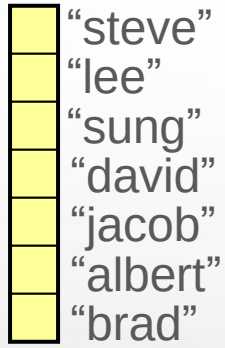
dense



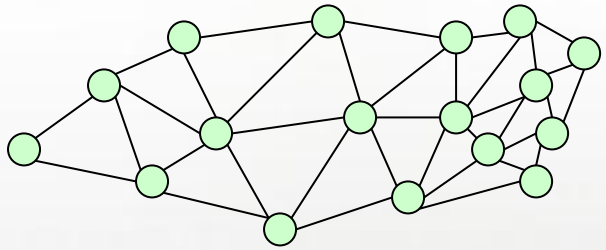
strided



sparse



associative



unstructured

…each of which supports Chapel’s data parallel operators

Data Parallelism Implementation Qs

Q1: How are arrays laid out in memory?

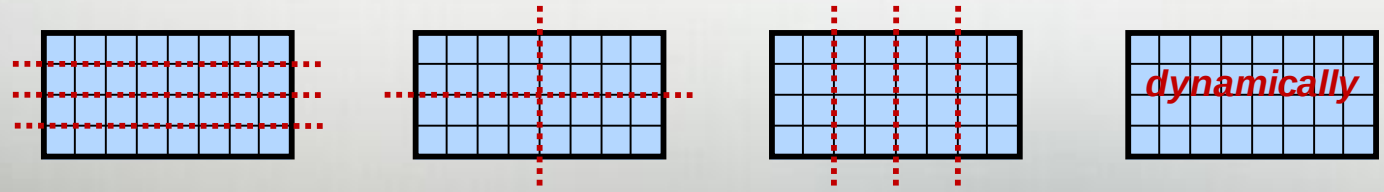
- Are regular arrays laid out in row- or column-major order? Or…?

…?

- How are sparse arrays stored? (COO, CSR, CSC, block-structured, …?)
- What about associative and unstructured?

Q2: How are arrays stored by the locales?

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

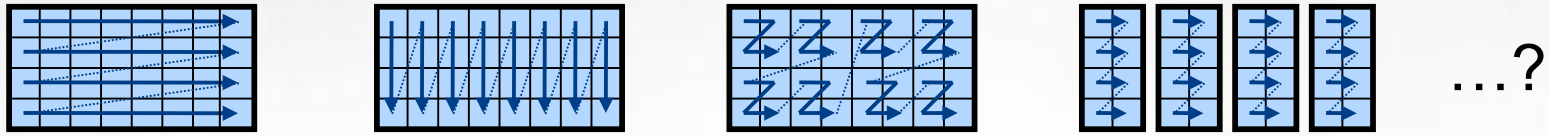


…?

Data Parallelism Implementation Qs

Q1: How are arrays laid out in memory?

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



…?

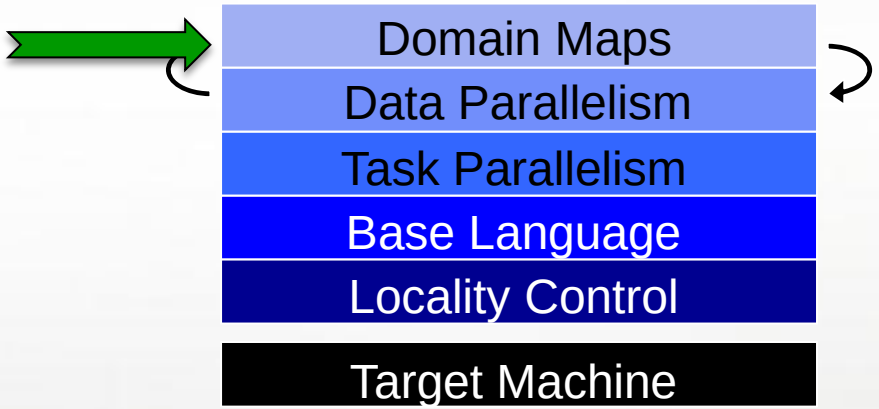
- How are sparse arrays stored? (COO, CSR, CSC, block-structured, …?)
- What about associative and unstructured?

Q2: How are arrays stored by the locales?

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

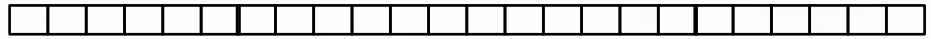
A: Chapel's *domain maps* are designed to give the user full control over such decisions

Domain Maps

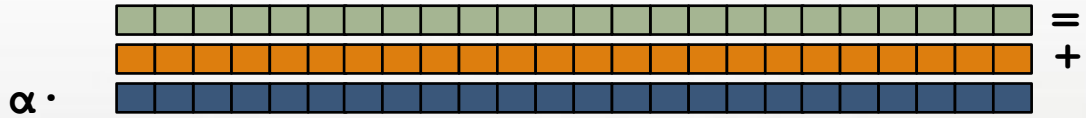

Domain Maps
Data Parallelism
Task Parallelism
Base Language
Locality Control
Target Machine

STREAM Triad: Chapel (multicore)

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



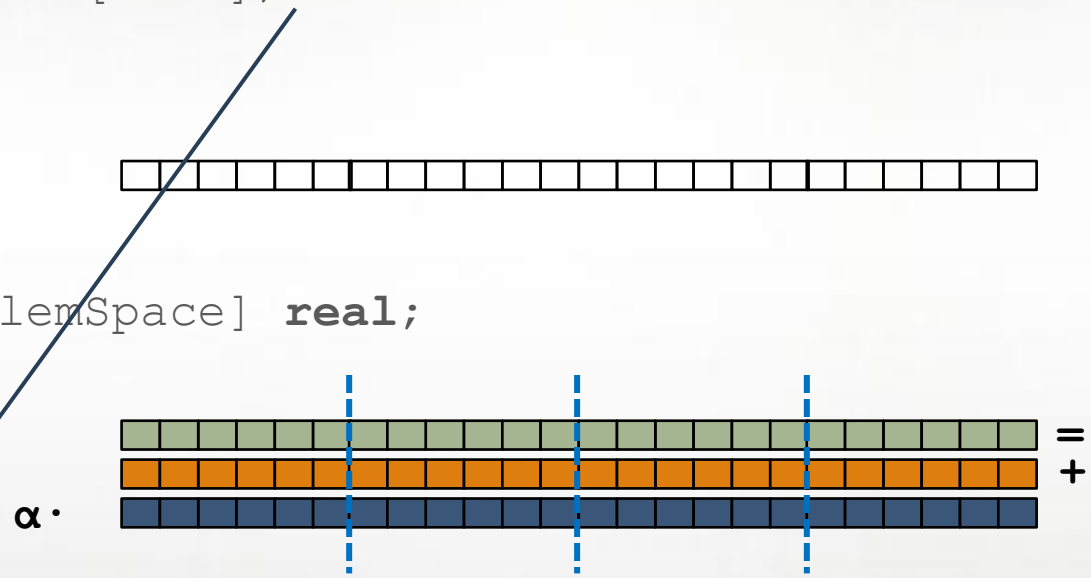
```
var A, B, C: [ProblemSpace] real;
```




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

STREAM Triad: Chapel (multicore)

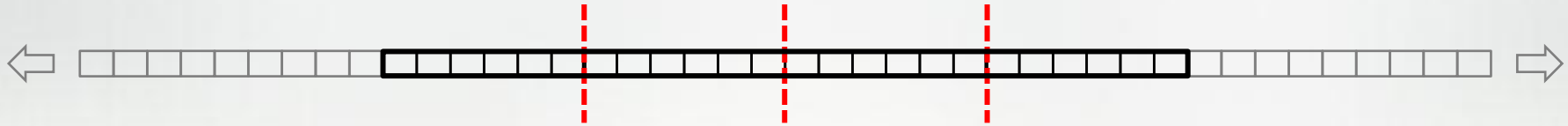
```
const ProblemSpace = [1..m];

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

A = B + alpha * C;
```



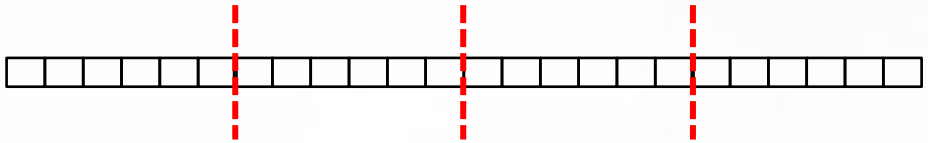
No domain map specified => use default layout

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

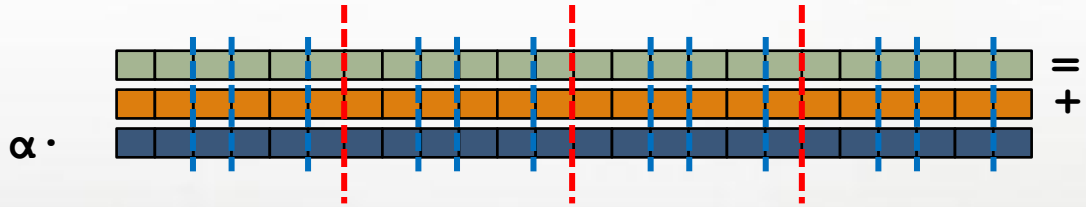
STREAM Triad: Chapel (multilocale, blocked)



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

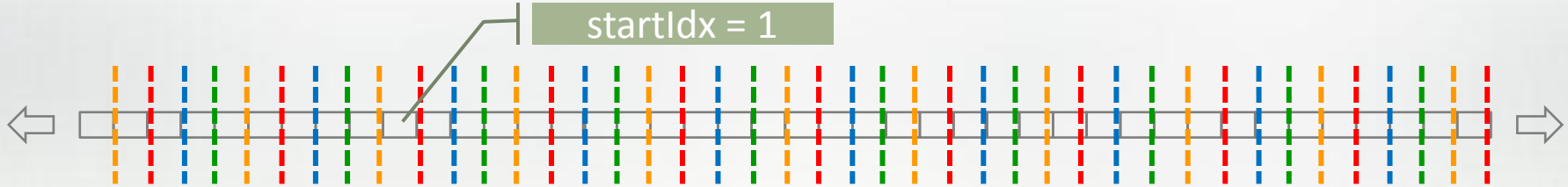


```
var A, B, C: [ProblemSpace] real;
```

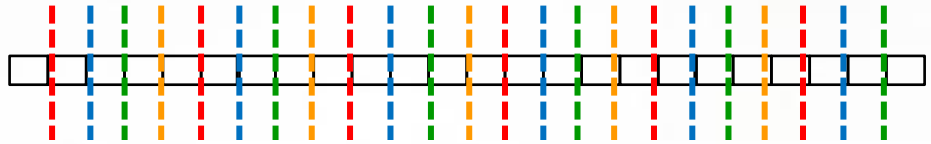



```
A = B + alpha * C;
```

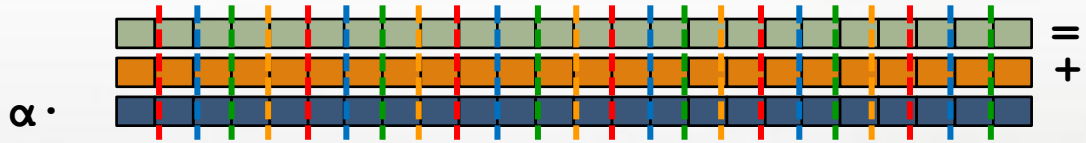
STREAM Triad: Chapel (multilocale, cyclic)




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



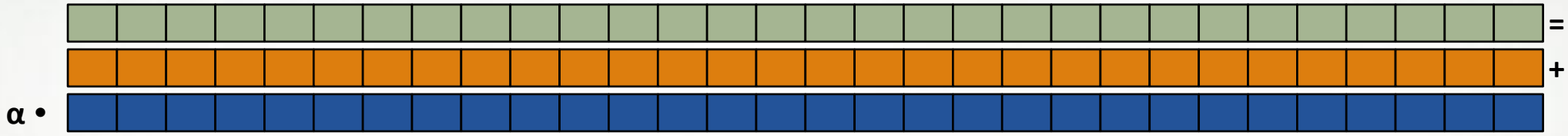
```
var A, B, C: [ProblemSpace] real;
```




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

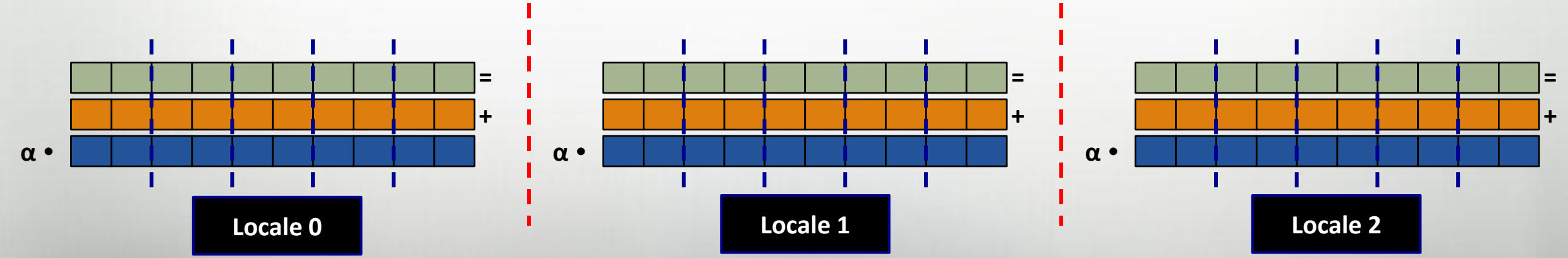
Domain Maps

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



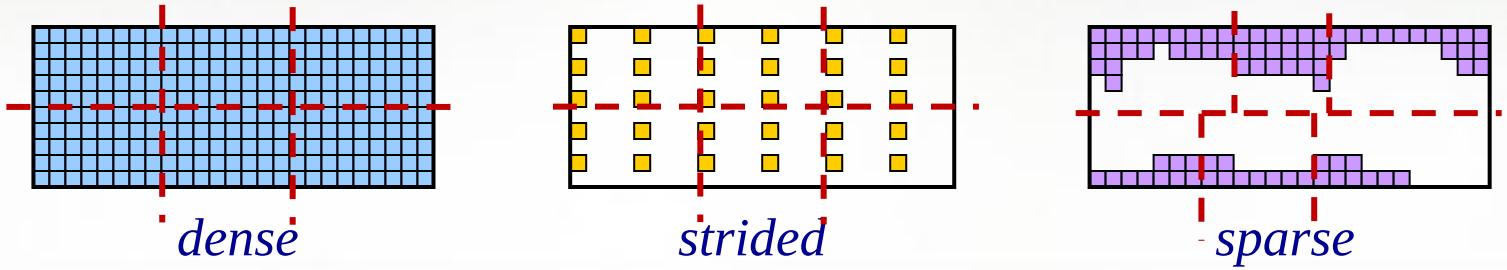

```
A = B + alpha * C;
```

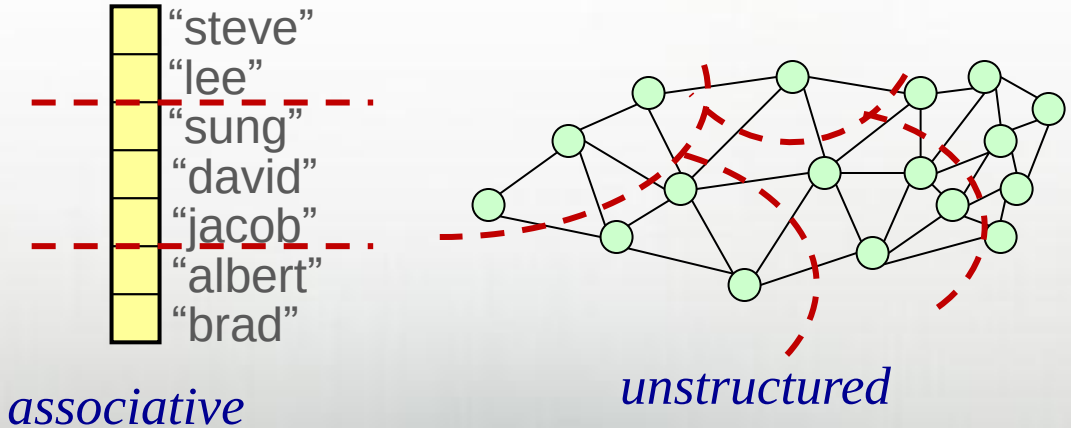
…to the target locales’ memory and processors:



Domain Map Types

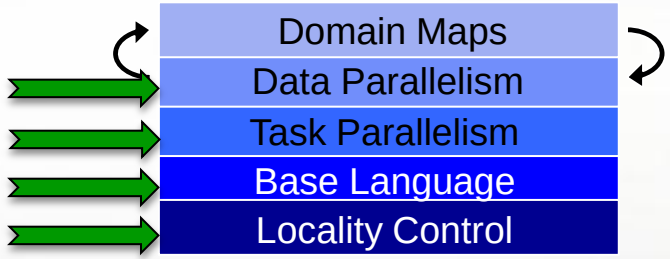
All Chapel arrays support domain maps





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 domain maps are written using the same end-user framework
   - to avoid a performance cliff between "built-in" and user-defined cases

Domain Maps: Summary

- Chapel avoids locking crucial implementation decisions into the language specification
  - local and distributed array implementations
  - parallel loop implementations
- Instead, these can be…

  …specified in the language by an advanced user

  …swapped in and out with minimal code changes
- The result cleanly separates the roles of domain scientist, parallel programmer, and implementation

For More Information on Domain Maps

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

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

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

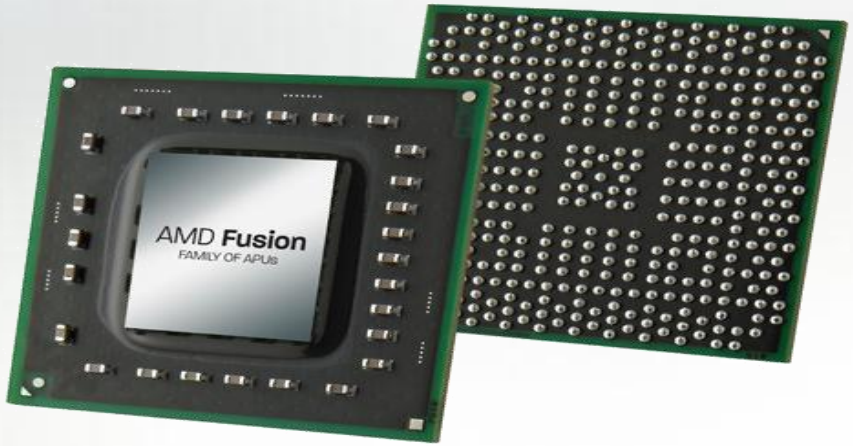
  layouts/*.chpl

  internal/Default*.chpl

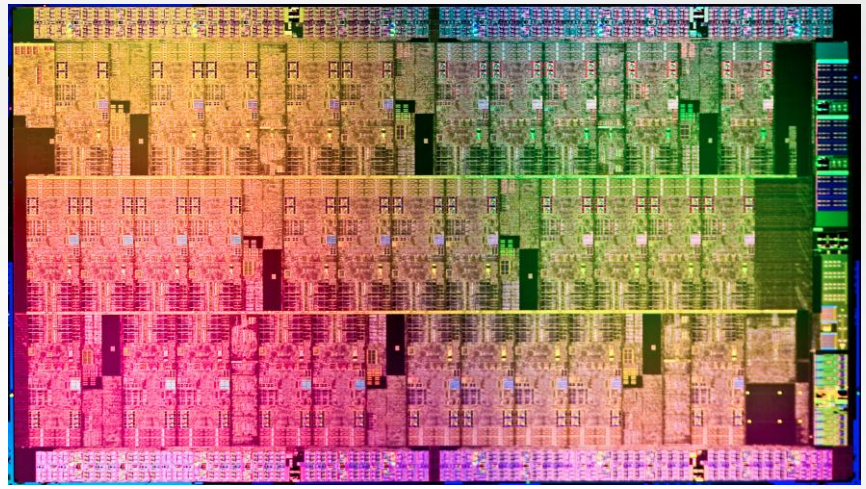
Outline

- ✓ Motivation
- ✓ Chapel Background and Themes
- ✓ Tour of Chapel Concepts
- ➢ Chapel and Exascale
- Wrap-up

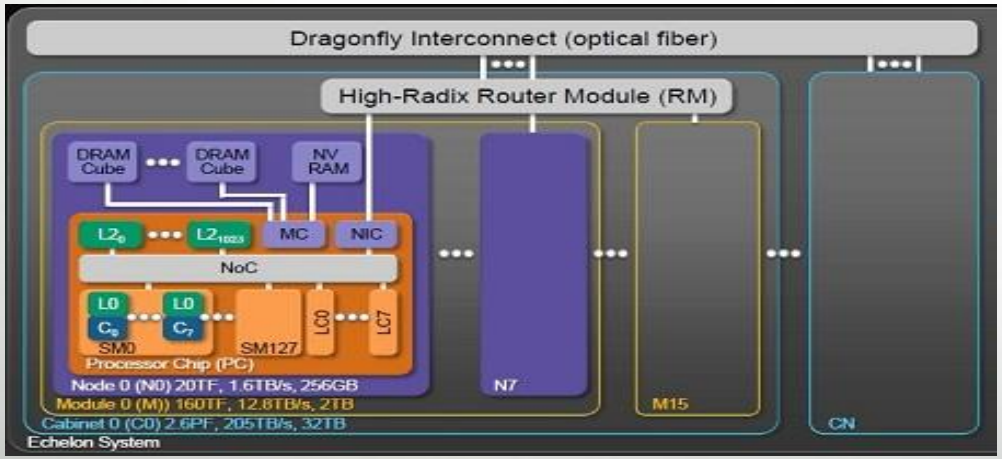
Candidate Next-Generation Processor Technologies



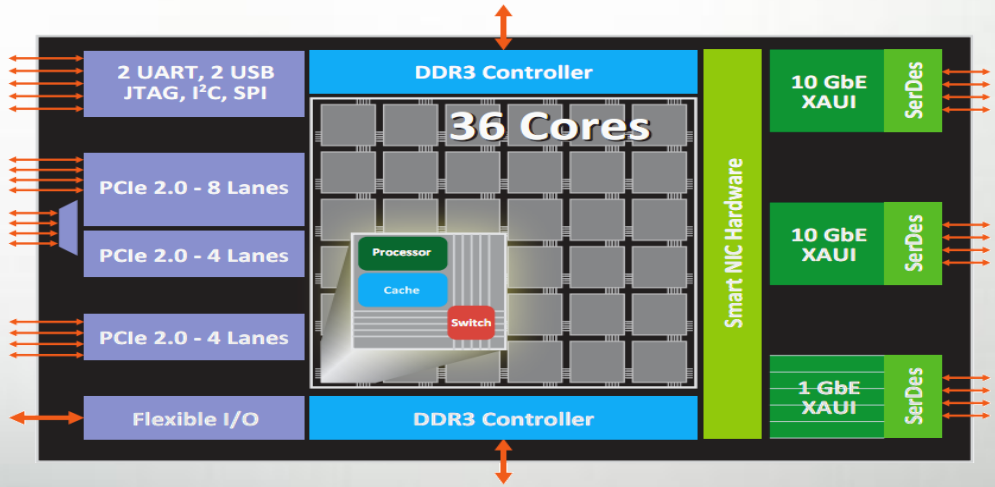
AMD Fusion



Intel MIC



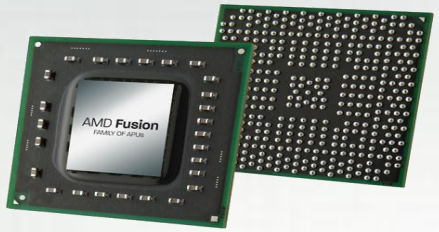
Nvidia Echelon

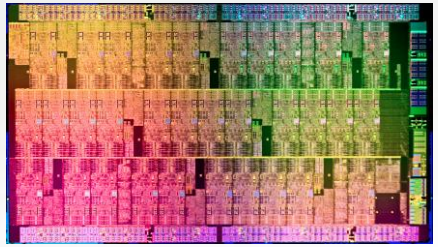


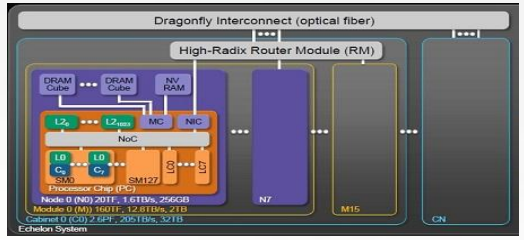
Tilera Tile-Gx

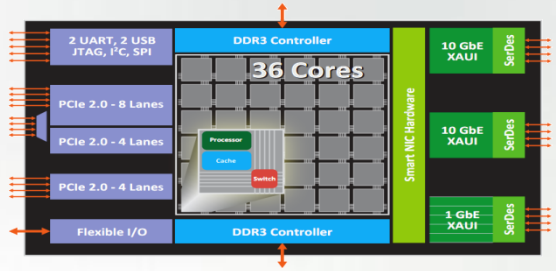
Sources: http://www.thinkcomputers.org/ces-2011-amds-fusion-apus/, http://download.intel.com/pressroom/images/Aubrey_Isle_die.jpg, http://insidehpc.com/2010/11/26/nvidia-reveals-details-of-echelon-gpu-designs-for-exascale/, http://tilera.com/sites/default/files/productbriefs/Tile-Gx%203036%20SB012-01.pdf

General Characteristics of These Architectures









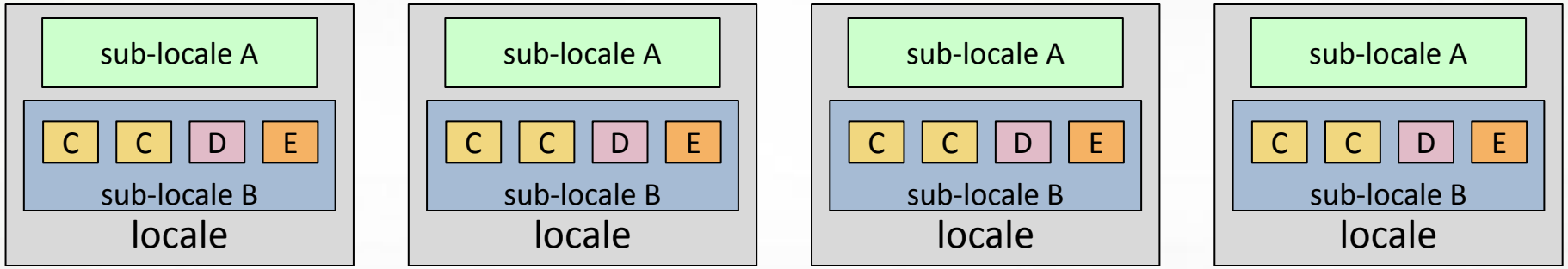
- Increased hierarchy and/or sensitivity to locality
- Heterogeneous processor and memory types

⇒ HPC (and mainstream) programmers will have a lot more to think about at the processor level

Current Work: Hierarchical Locales

Concept:

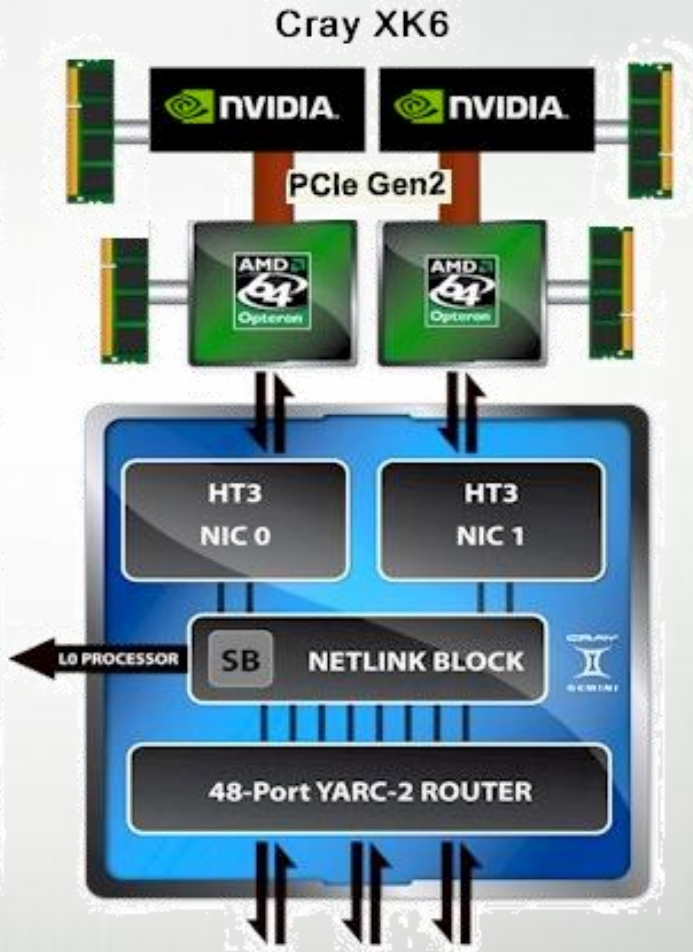
- Support sub-locales within locales to describe node architecture sub-structures



- As with current locales, on-clauses and domain maps can be used to map tasks or variables to a sub-locale's memory/processors
- Locale structures are defined as Chapel code
  - introduces a new Chapel programmer role: Architectural modeler

Sublocales: Hybrid Processor Example

```
class locale: AbstractLocale {
  const numCPUs = 2, numGPUs = 2;
  const cpu: [0..#numCPUs] cpuLoc = …;
  const gpu: [0..#numGPUs] gpuLoc = …;
  // tasking interface for node-level
  // memory interface for node-level
}
class cpuLoc: AbstractLocale {
  // sub-locales for different NUMA domains
  // tasking, memory interfaces for CPU
}
class gpuLoc: AbstractLocale {
  // sub-locales for memories, processors
  // tasking, memory interfaces for GPU
}
```



Hierarchical Locales: Challenges

Portability: Chapel code that refers to sub-locales causes problems for locales with different structure

Mitigation Strategies

- Well-designed domain maps should buffer the data parallel user from many of these challenges
- More advanced runtime designs and compiler work could help save most task parallel users from this level of detail
- Not a Chapel-specific challenge, fortunately

Communication Generation: A function of two locale types, not one

(and they may not be known at compile-time)

Outline

- ✓ Motivation
- ✓ Chapel Background and Themes
- ✓ Tour of Chapel Concepts
- ✓ Chapel and Exascale
- ➢ Wrap-up

Summary

Higher-level programming models can help insulate algorithms from implementation

- yet, without necessarily abandoning finer-grain 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

Some Next Steps

- Hierarchical Locales
- Resilience Features
- Performance Optimizations
- Lock down post-HPCS Funding
- Evolve from Prototype- to Production-grade
- Evolve from Cray- to community-language
- and much more…

Implementation Status -- Version 1.5.0 (April 19, 2012)

In a nutshell:

- Most features work at a functional level
- Many performance optimizations remain
  - particularly for distributed memory (multi-locale) execution

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
- Use Chapel for parallel programming education

Join Our Growing Community

- Cray:


Brad Chamberlain


Sung-Eun Choi


Greg Titus


Vass Litvinov


Tom Hildebrandt

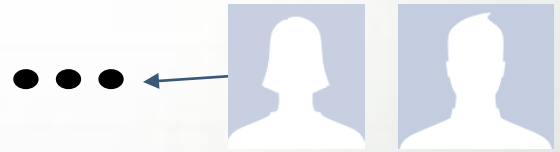
(open positions)

- External Collaborators:


Albert Sidelnik
(UIUC)


Jonathan Turner
(CU Boulder)

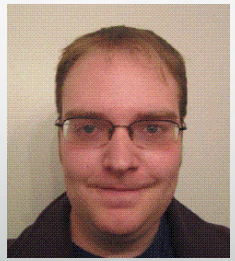

Kyle Wheeler
(Sandia)


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)

- **CPU-GPU Computing:** UIUC (David Padua, Albert Sidelnik, Maria Garzarán)
  - **paper at IPDPS, May 2012**
- **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:**
- **Parallel File I/O:**

  U Malaga (Rafael Asenjo, Maria Angeles Navarro, et al.)
  - **paper at ParCo, Aug 2011**
- **Improved I/O & Data Channels:** LTS (Michael Ferguson)
- **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.)

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

CRAY

THE SUPERCOMPUTER COMPANY

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

Disclaimer & Attribution

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. There is no obligation to update or otherwise correct or revise this information. However, we reserve the right to revise this information and to make changes from time to time to the content hereof without obligation to notify any person of such revisions or changes.

NO REPRESENTATIONS OR WARRANTIES ARE MADE WITH RESPECT TO THE CONTENTS HEREOF AND NO RESPONSIBILITY IS ASSUMED FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

ALL IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED. IN NO EVENT WILL ANY LIABILITY TO ANY PERSON BE INCURRED FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

AMD, the AMD arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. All other names used in this presentation are for informational purposes only and may be trademarks of their respective owners.

The contents of this presentation were provided by individual(s) and/or company listed on the title page. The information and opinions presented in this presentation may not represent AMD's positions, strategies or opinions. Unless explicitly stated, AMD is not responsible for the content herein and no endorsements are implied.