

Chapel: Parallel Programmability from Desktops to Supercomputers

Brad Chamberlain, Cray Inc.

University of Bergen: April 11th, 2013



Cray: The Supercomputer Company



Sustained Performance Milestones

1 GF – 1988: Cray Y-MP; 8 Processors

- Static finite element analysis



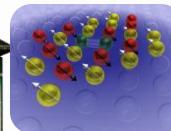
1 TF – 1998: Cray T3E; 1,024 Processors

- 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

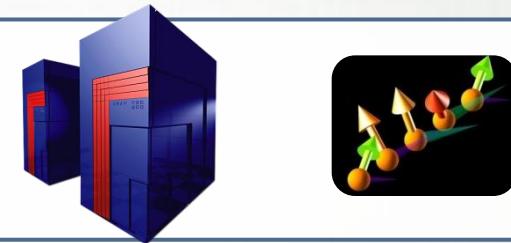
1 GF – 1988: Cray Y-MP; 8 Processors

- 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 + CUDA/OpenCL/OpenMP/OpenACC

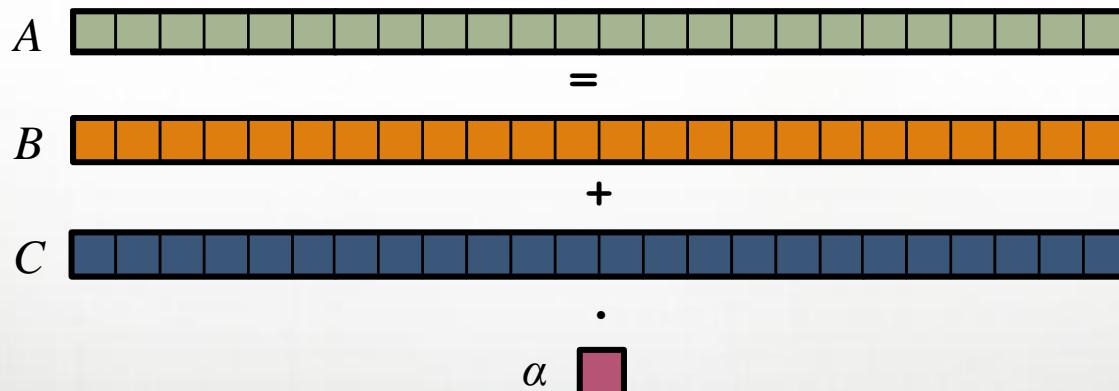
Or Perhaps Something Completely Different?

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:

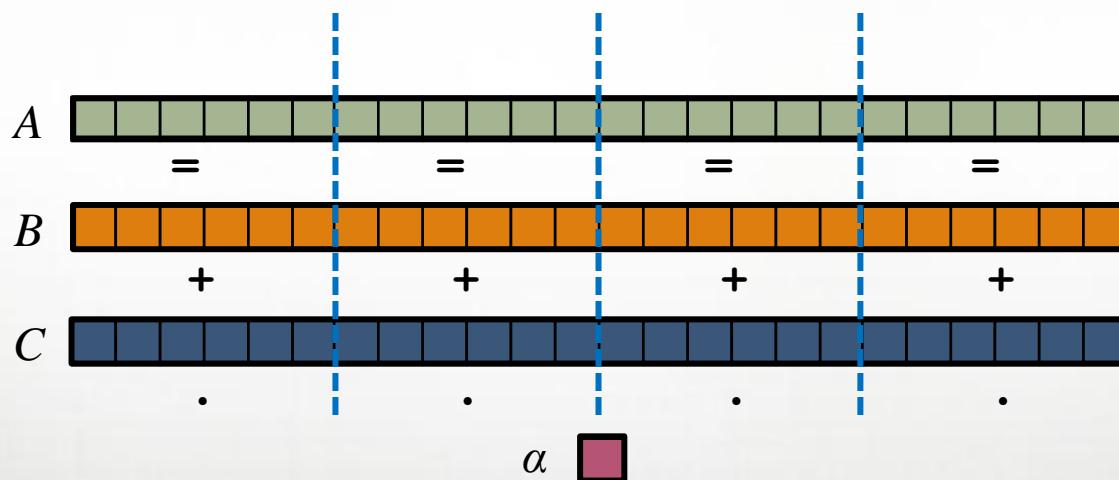


STREAM Triad: a trivial parallel computation

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:

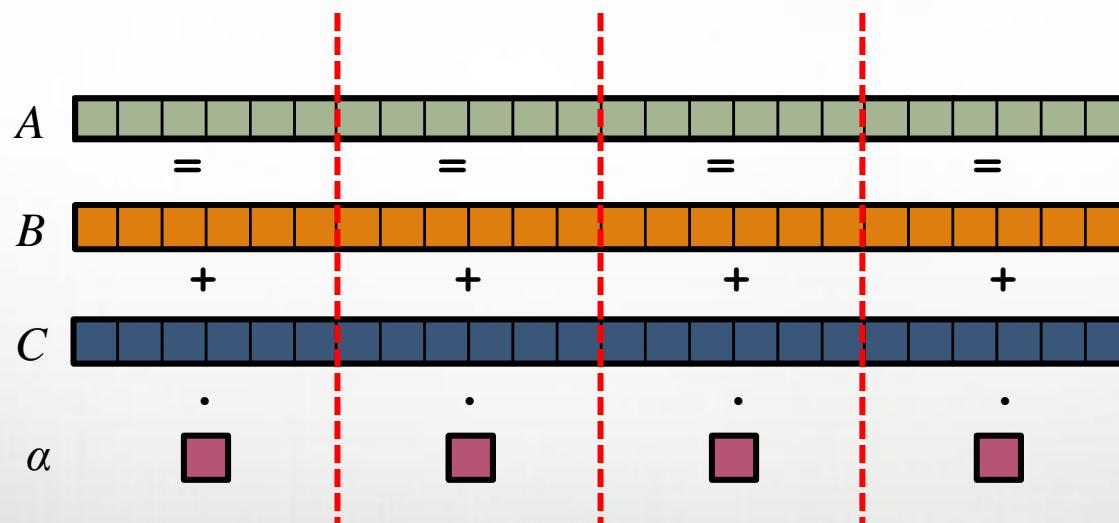


STREAM Triad: a trivial parallel computation

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):

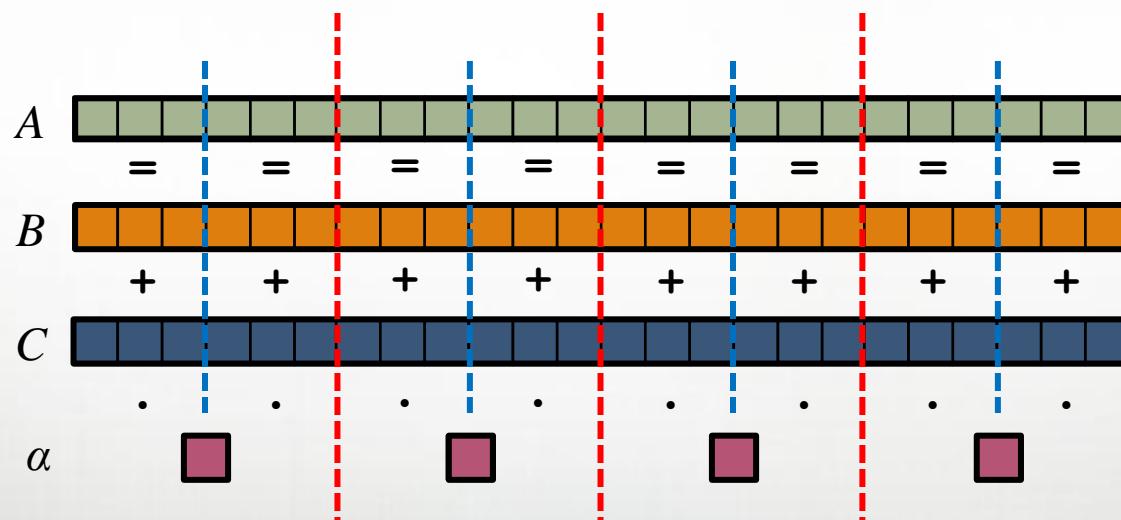


STREAM Triad: a trivial parallel computation

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

```
#include <hpcc.h>

MPI

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

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

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

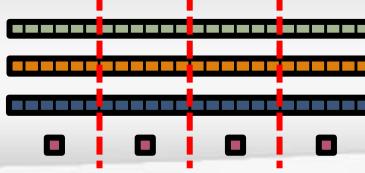
    rv = HPCC_Stream( params, 0 == myRank );
    MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM,
        0, comm );

    return errCount;
}

int HPCC_Stream(HPCC_Parms *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;
}

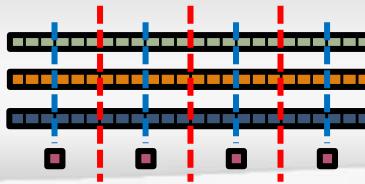
scalar = 3.0;

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

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

int HPCC_StarStream(HPCC_Parms *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_Parms *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;
        }

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

        scalar = 3.0;

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

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

        return 0;
}
```

STREAM Triad: MPI+OpenMP vs. CUDA

MPI + OpenMP

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

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

int HPCC_StarStream(HPCC_Parms *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 );
}

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

int HPCC_Stream(HPCC_Parms *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;
    }

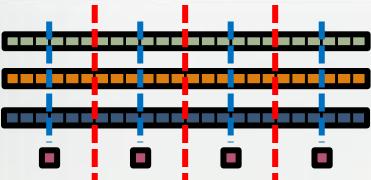
    #ifdef _OPENMP
    #pragma omp parallel for
    #endif
    for (j=0; j<VectorSize; j++) {
        b[j] = 2.0;
        c[j] = 0.0;
    }

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

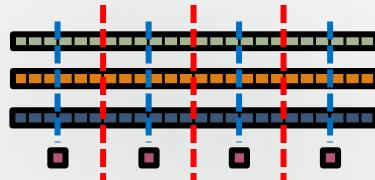
    set_array<<<dimGrid,dimBlock>>>(d_b, .5f, N);
    set_array<<<dimGrid,dimBlock>>>(d_c, .5f, N);

    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 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/OpenAcc	SIMD function/task

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

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

(“Glad I’m not an HPC Programmer!”)

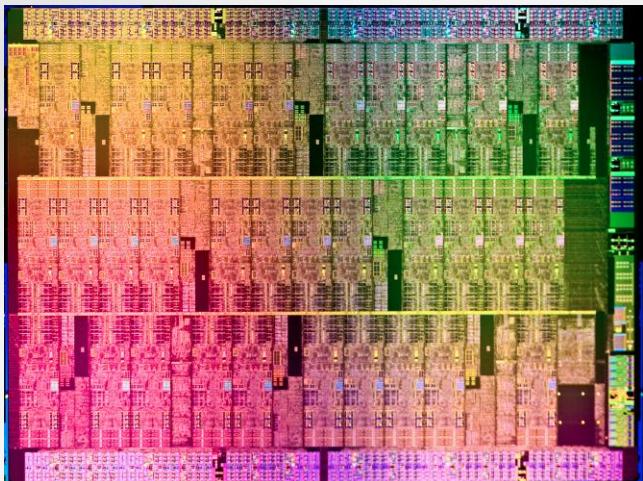
A Possible Reaction:

“This is all well and good for HPC users, but I’m a mainstream desktop programmer, so this is all academic for me.”

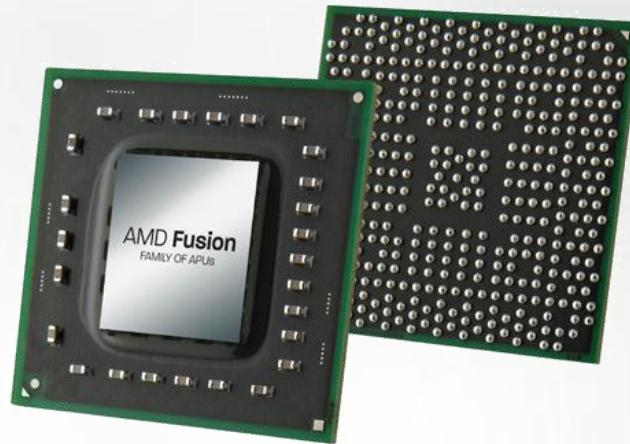
The Unfortunate Reality:

- Performance-minded mainstream programmers will increasingly deal with parallelism
- And, as chips become more complex, locality too

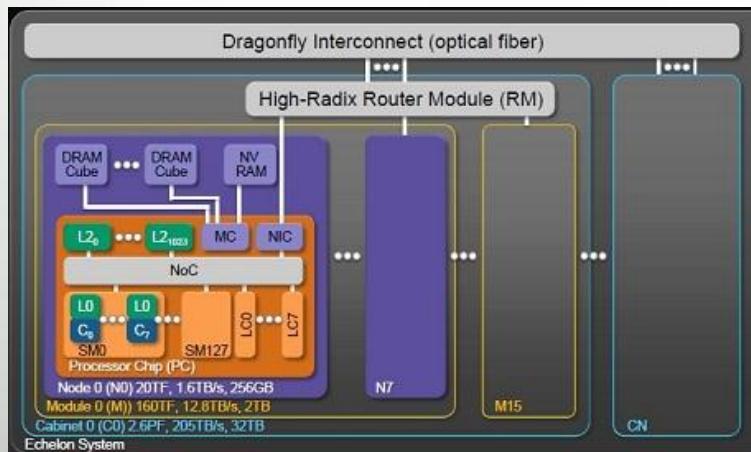
Next-generation HPC Processor Technologies



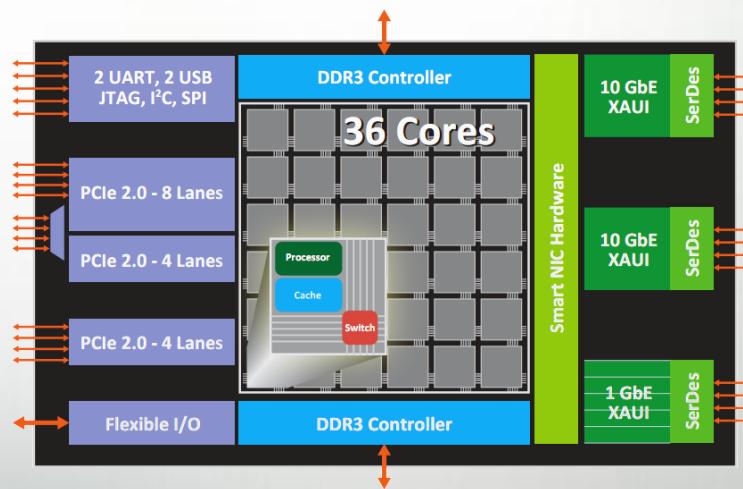
Intel MIC



AMD Fusion

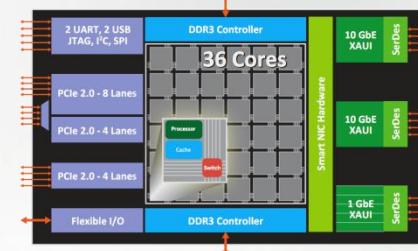
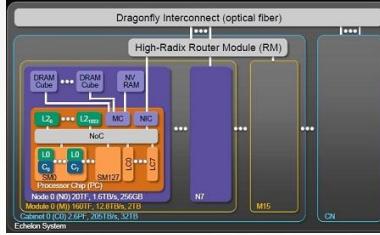
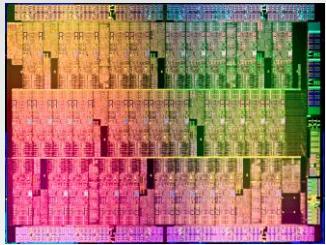


Nvidia Echelon



Tilera Tile-Gx

General Characteristics of These Architectures



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

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

Rewinding a few slides...

MPI + OpenMP

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

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

int HPCC_StarStream(HPCC_Parms *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 );
}
```

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

```
int HPCC_Stream(HPCC_Parms *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);
    if( N % dimBlock.x != 0 ) dimGrid.x+=1;

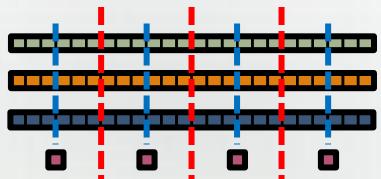
    set_array<<<dimGrid,dimBlock>>>(d_b, .5f, N);
    set_array<<<dimGrid,dimBlock>>>(d_c, .5f, N);

    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: Chapel

MPI + OpenMP

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

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

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

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

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

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

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

if (!a || !b || !c) {
if (c) HPCC_free(c);
if (b) HPCC_free(b);
if (a) HPCC_free(a);
if (doIO) {

```

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

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

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

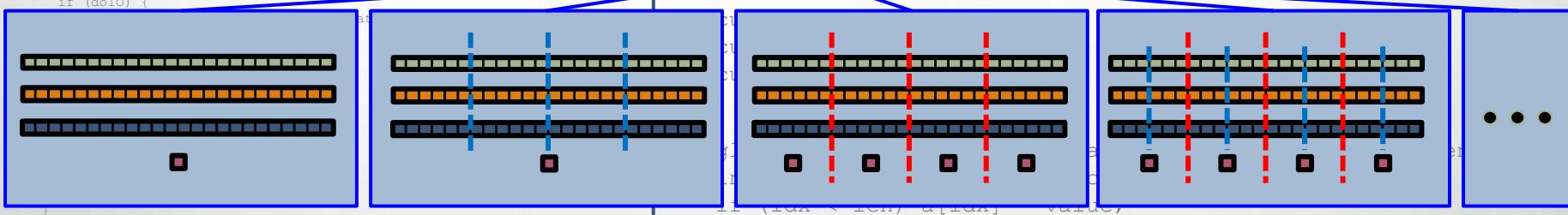
B = 2.0;
C = 3.0;

A = B + alpha * C;
```

Chapel

dmapped ...;

the special sauce



```
scalar =
#endif
#pragma omp
#endif
for (j=0;
a[j] =
HPCC_free(
HPCC_free(
HPCC_free(
return 0;
}
```

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
- Advanced Chapel Topics
- Project Status

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

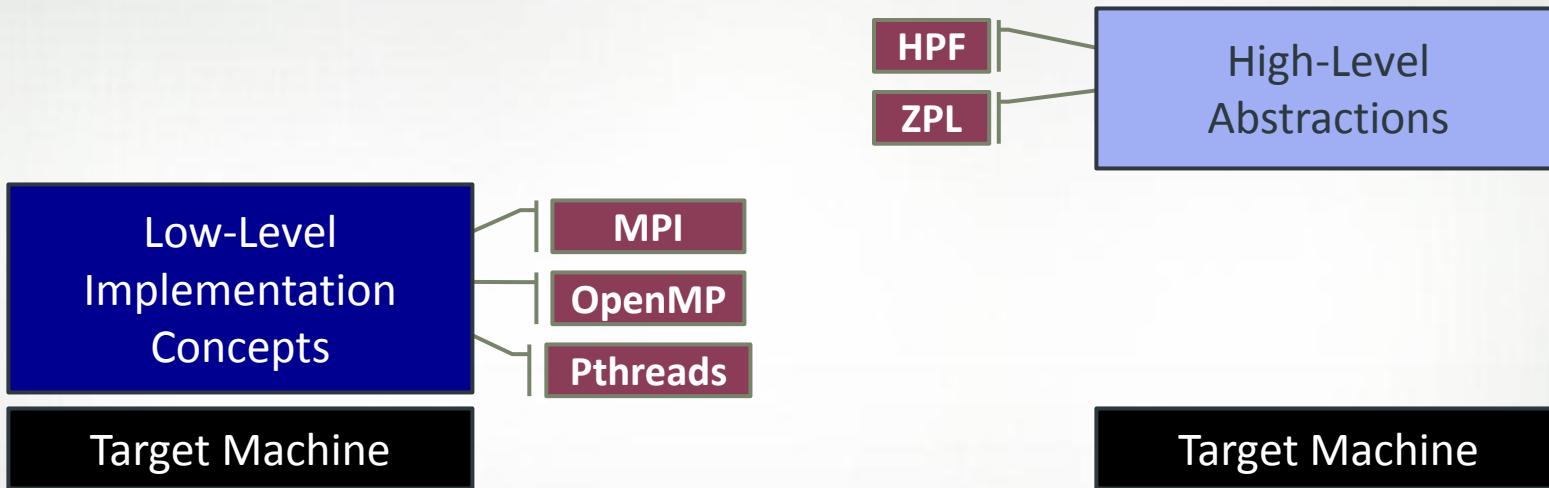
- **Styles:** data-parallel, task-parallel, concurrency, nested, ...
- **Levels:** model, function, loop, statement, expression

...target all parallelism available in the hardware

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

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

3) Multiresolution Design: Motivation



“Why is everything so tedious/difficult?”
“Why don’t my programs port trivially?”

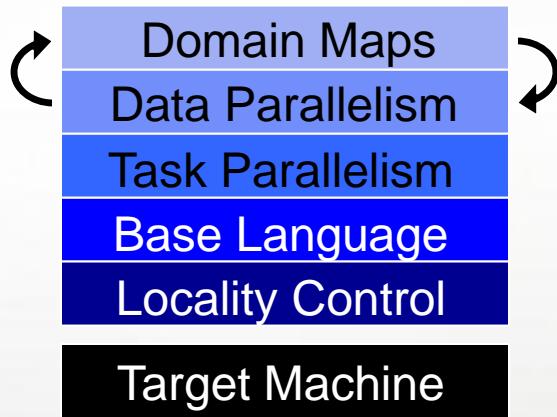
“Why don’t I have more control?”

3) Multiresolution Design

Multiresolution Design: Support multiple tiers of features

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

Chapel language concepts



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

5) Reduce HPC ↔ Mainstream Language Gap

Consider:

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

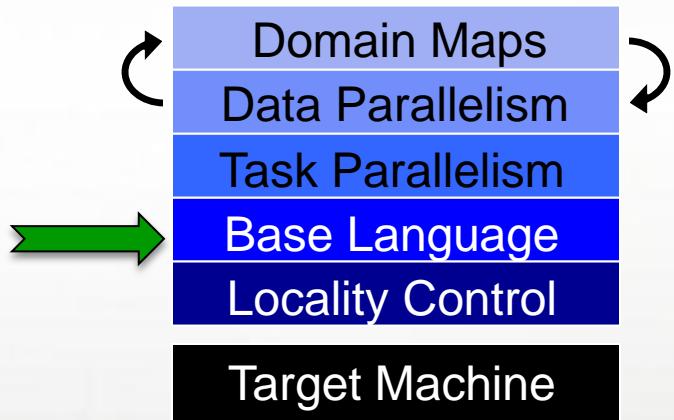
We'd like to narrow this gulf with Chapel:

- 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
- Advanced Chapel Topics
- Project Status

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)

Range Types and Algebra

```
const r = 1..10;

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

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

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

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)
```

Zippered Iteration

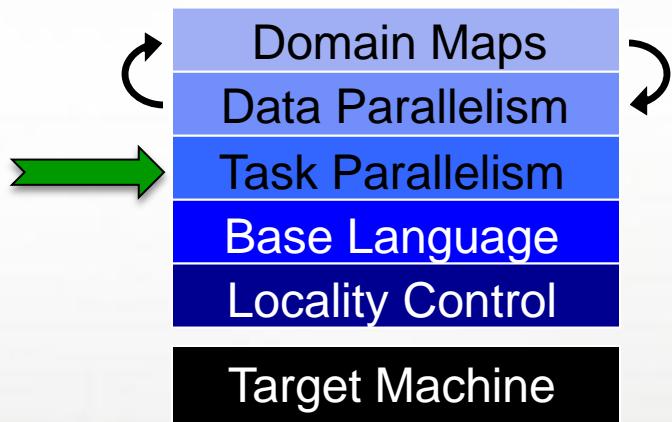
```
for (i,f) in zip(0..#n, fibonacci(n)) do
    writeln("fib #", i, " is ", f);
```

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

Other Base Language Features

- tuple types
- 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
```

Bounded Buffer Producer/Consumer Example

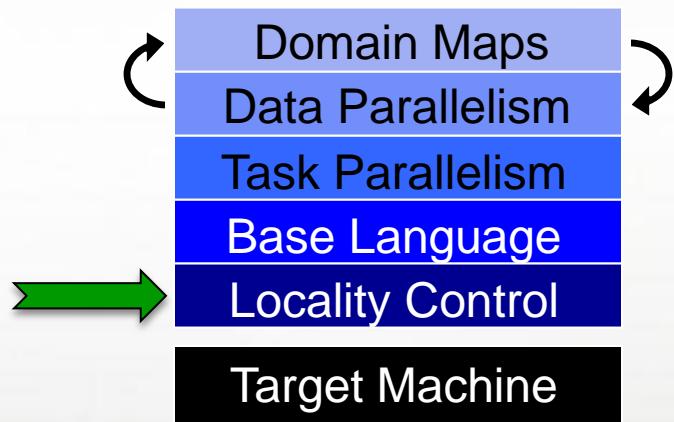
```
cobegin {
    producer();
    consumer();
}

// '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
    }
}
```

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 = ...;
```

Locales: L0 L1 L2 L3 L4 L5 L6 L7

Locale Operations

- Locale methods support queries about target system:

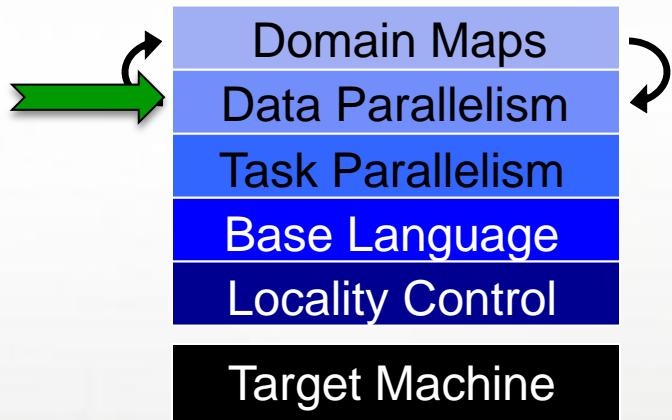
```
proc locale.physicalMemory(...) { ... }  
proc locale.numCores { ... }  
proc locale.id { ... }  
proc locale.name { ... }
```

- *On-clauses* support placement of computations:

```
writeln("on locale 0");  
on Locales[1] do  
    writeln("now on locale 1");  
  
writeln("on locale 0 again");
```

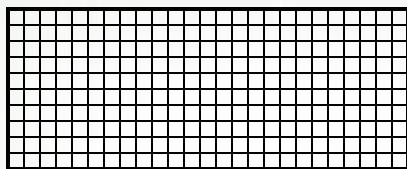
```
cobegin {  
    on A[i,j] do  
        bigComputation(A);  
  
    on node.left do  
        search(node.left);  
}
```

Data Parallel Features

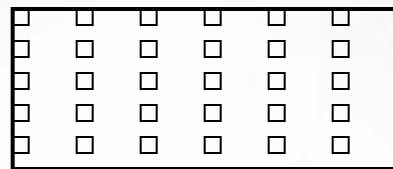


Chapel Domain Types

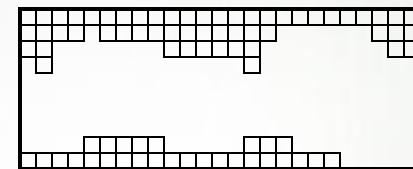
Chapel supports several types of domains (index sets) :



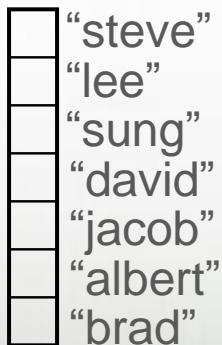
dense



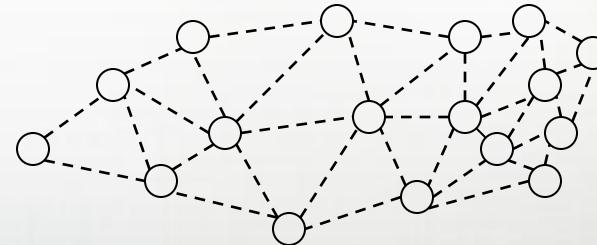
strided



sparse



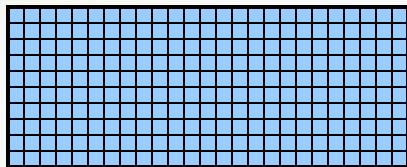
associative



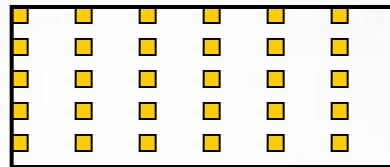
unstructured

Chapel Array Types

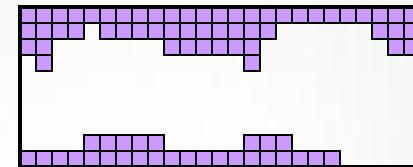
Each domain type can be used to declare arrays:



dense



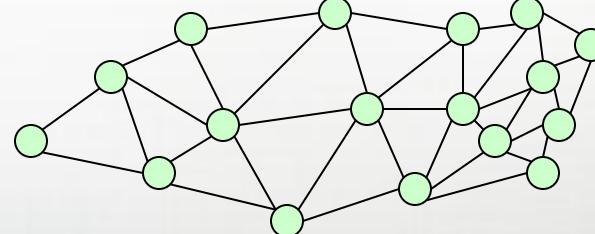
strided



sparse



associative



unstructured

Chapel Domain/Array Operations

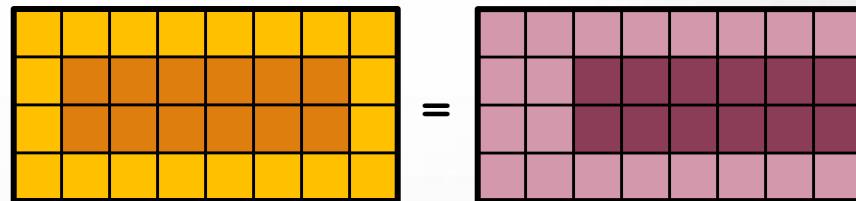
- Parallel and Serial Iteration

```
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 Operators and Functions

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

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

- And several others: indexing, reallocation, set operations, remapping, aliasing, queries, ...

Notes on Forall Loops

```
forall a in A do
```

```
    writeln("Here is an element of A: ", a);
```

Typically $1 \leq \# \text{Tasks} \ll \# \text{Iterations}$)

```
forall (a, i) in zip(A, 1..n) do
```

```
    a = i/10.0;
```

Forall-loops may be zippered, like for-loops

- Corresponding iterations will match up

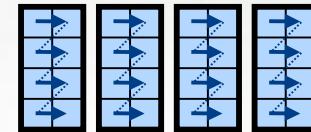
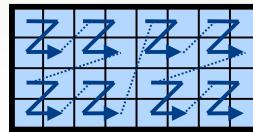
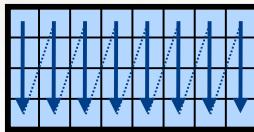
Outline

- ✓ Motivation
- ✓ Chapel Background and Themes
- ✓ Tour of Chapel Concepts
- Advanced Chapel Topics
 - Domain Maps
 - Leader-Follower Iterators
 - Project Status

Data Parallelism Implementation Qs

Q1: How are arrays laid out in memory?

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

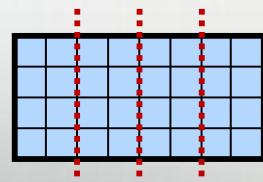
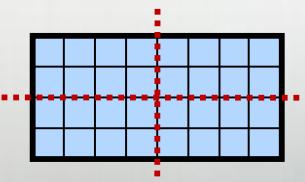
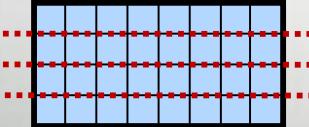


...?

- How are sparse arrays stored? (COO, CSR, CSC, block-structured, ...?)

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? ...?

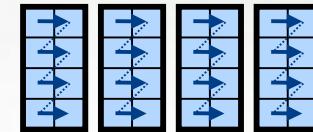
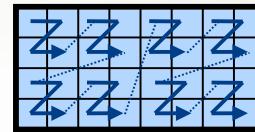
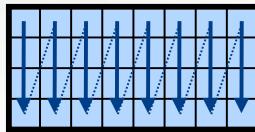
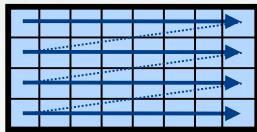


...?

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, ...?)

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

STREAM Triad: Chapel (multicore)

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



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



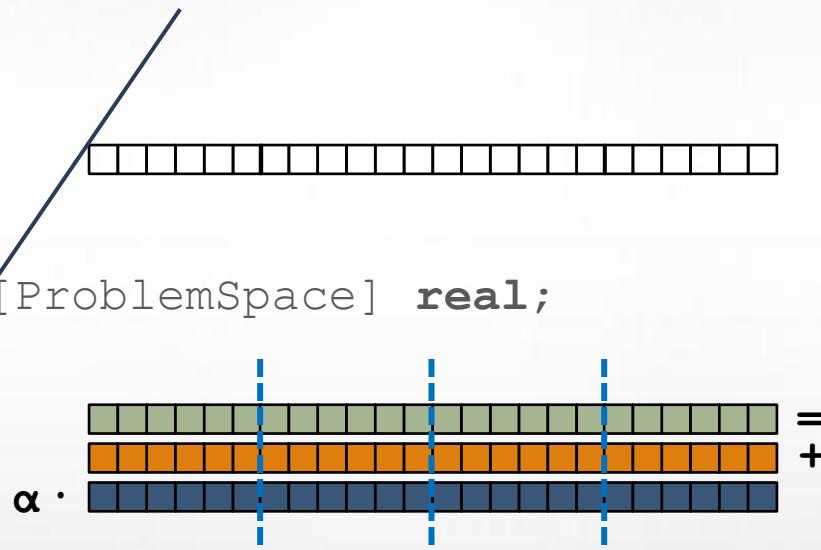
```
A = B + alpha * C;
```

STREAM Triad: Chapel (multicore)

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

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

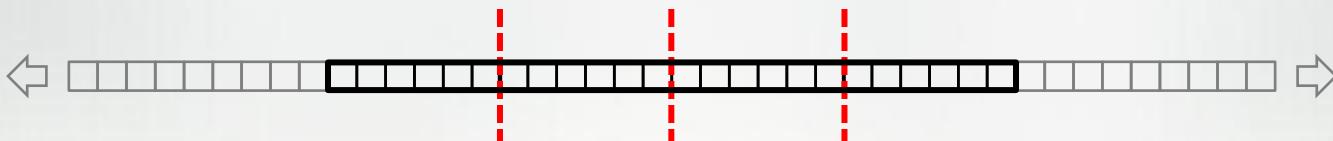
```
A = B + alpha * C;
```



No domain map specified => use default layout

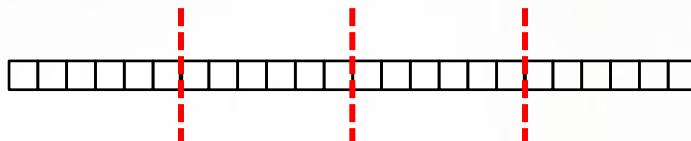
- current locale owns all indices and values
- computation will execute using local processors only

STREAM Triad: Chapel (multilocale, blocked)

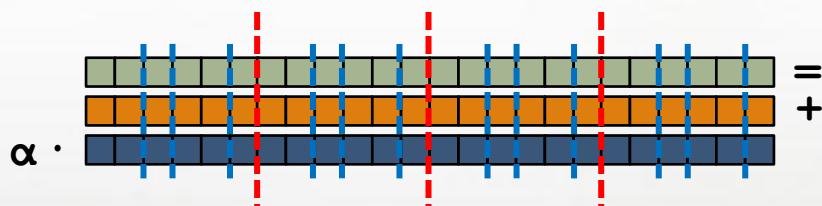


```
const ProblemSpace = {1..m}
```

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

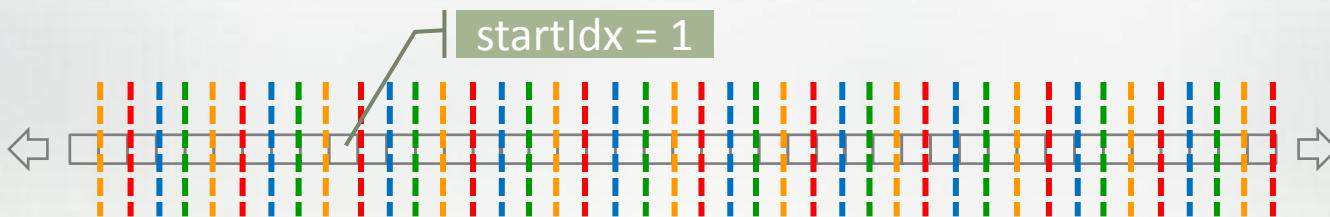


```
var A, B, C: [ProblemSpace] real;
```



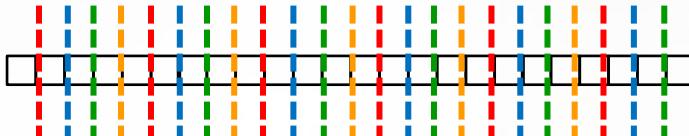
```
A = B + alpha * C;
```

STREAM Triad: Chapel (multilocale, cyclic)

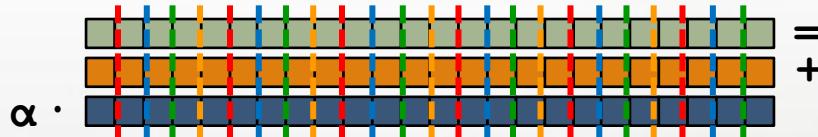


```
const ProblemSpace = {1..m}
```

```
dmapped Cyclic(startIdx=1);
```



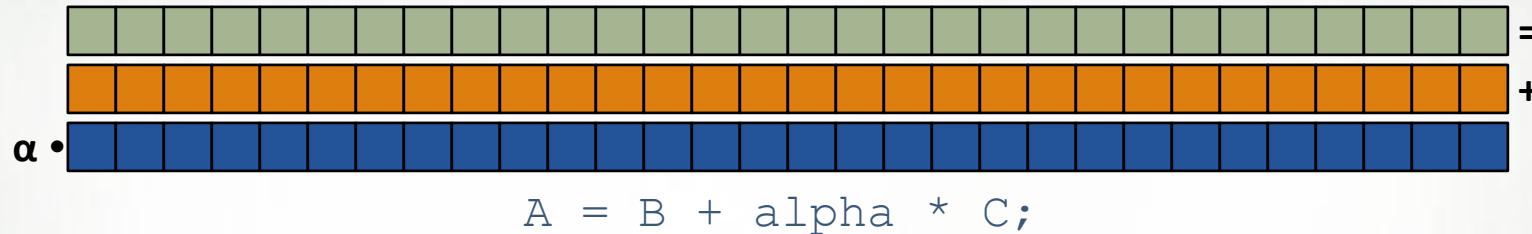
```
var A, B, C: [ProblemSpace] real;
```



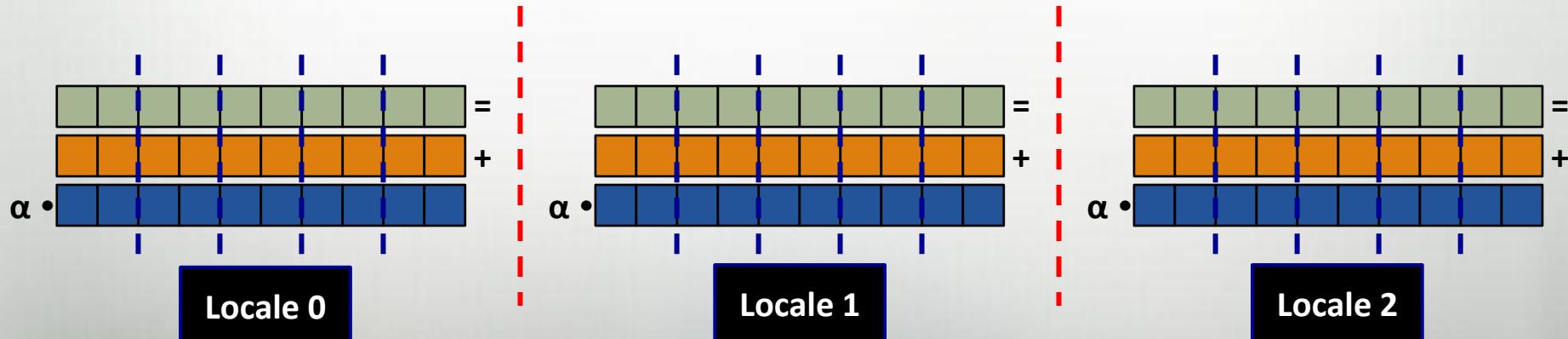
```
A = B + alpha * C;
```

Domain Maps

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

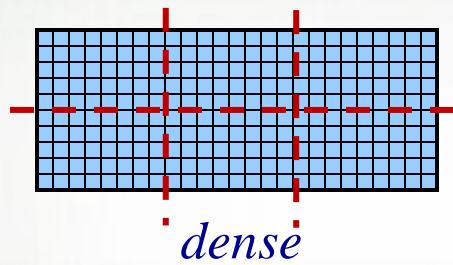


...to the target locales' memory and processors:

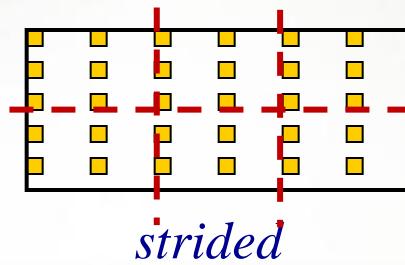


Domain Map Types

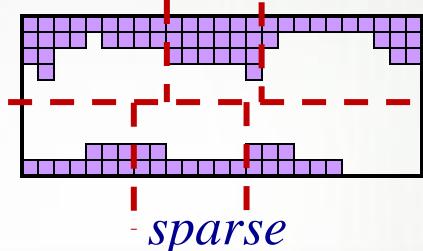
All Chapel domain types support domain maps



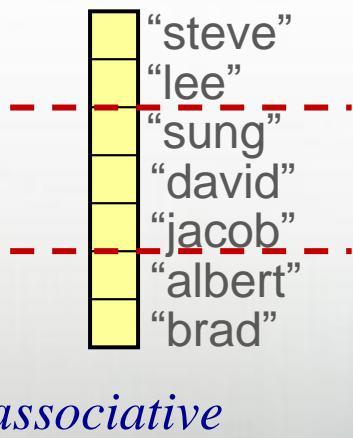
dense



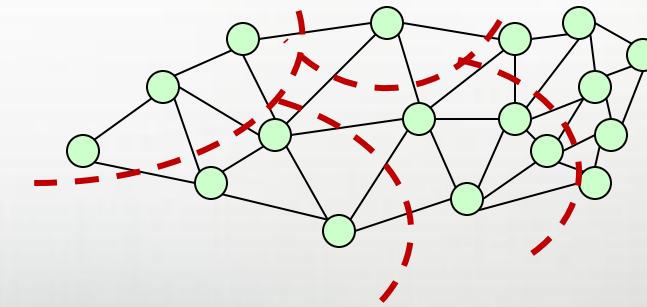
strided



sparse



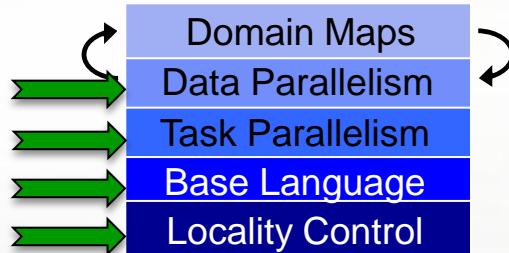
associative



unstructured

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 Map Descriptors

Domain Map

Represents: a domain map value

Generic w.r.t.: index type

State: the domain map's representation

Typical Size: $\Theta(1)$

Required Interface:

- create new domains

Domain

Represents: a domain

Generic w.r.t.: index type

State: representation of index set

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

Required Interface:

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

Array

Represents: an array

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

State: array elements

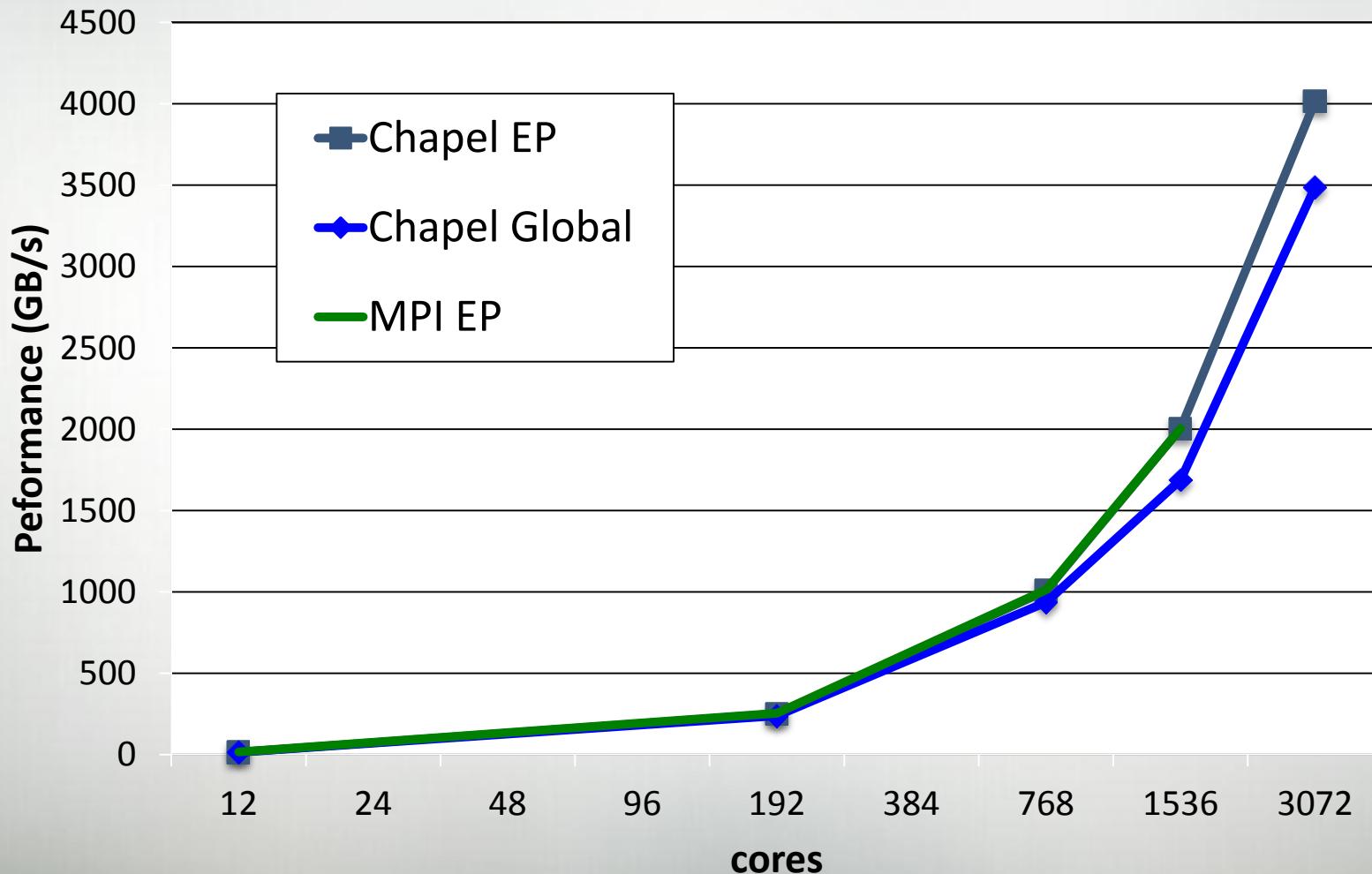
Typical Size: $\Theta(\text{numIndices})$

Required Interface:

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

HPCC Stream Performance on Jaguar (XT5)

MPI vs. Chapel STREAM Triad on Jaguar



For More Information on Domain Maps

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

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

Chapel release:

- 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`

Domain Maps: Next Steps

- More advanced uses of domain maps:
 - Dynamically load balanced domains/arrays
 - Resilient data structures
 - *in situ* interoperability with legacy codes
 - out-of-core computations
- Further compiler optimization via optional interfaces
 - particularly communication idioms (stencils, reductions, ...)

Outline

- ✓ Motivation
- ✓ Chapel Background and Themes
- ✓ Tour of Chapel Concepts
- Advanced Chapel Topics
 - ✓ Domain Maps
 - Leader-Follower Iterators
- Project Status

More Data Parallelism Implementation Qs

Q1: How are forall loops implemented?

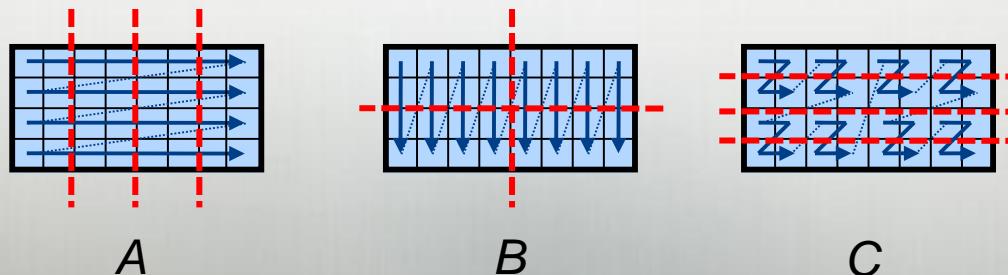
```
forall i in B.domain do B[i] = i/10.0;  
forall c in C do c = 3.0;
```

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

Q2: How are parallel zippered loops implemented?

```
forall (a,b,c) in zip(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

Q1: How are forall 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?

Q2: How are parallel zippered loops implemented?

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

into:

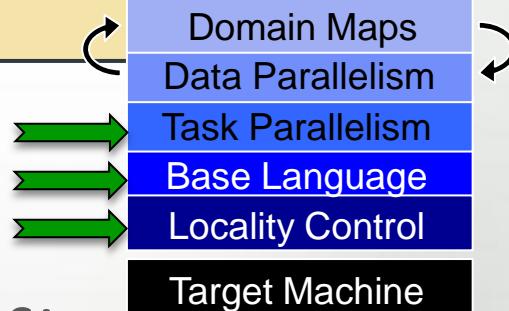
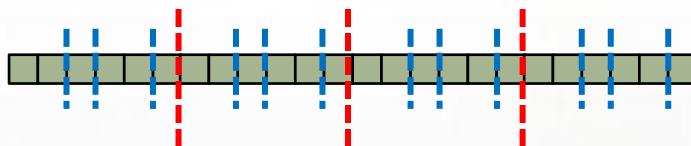
inlined **A.lead()** iterator, which yields **work**...

```
for (a,b,c) in zip(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);  
}
```

Controlling Data Parallelism

Q: “*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 zip(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 zip(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).

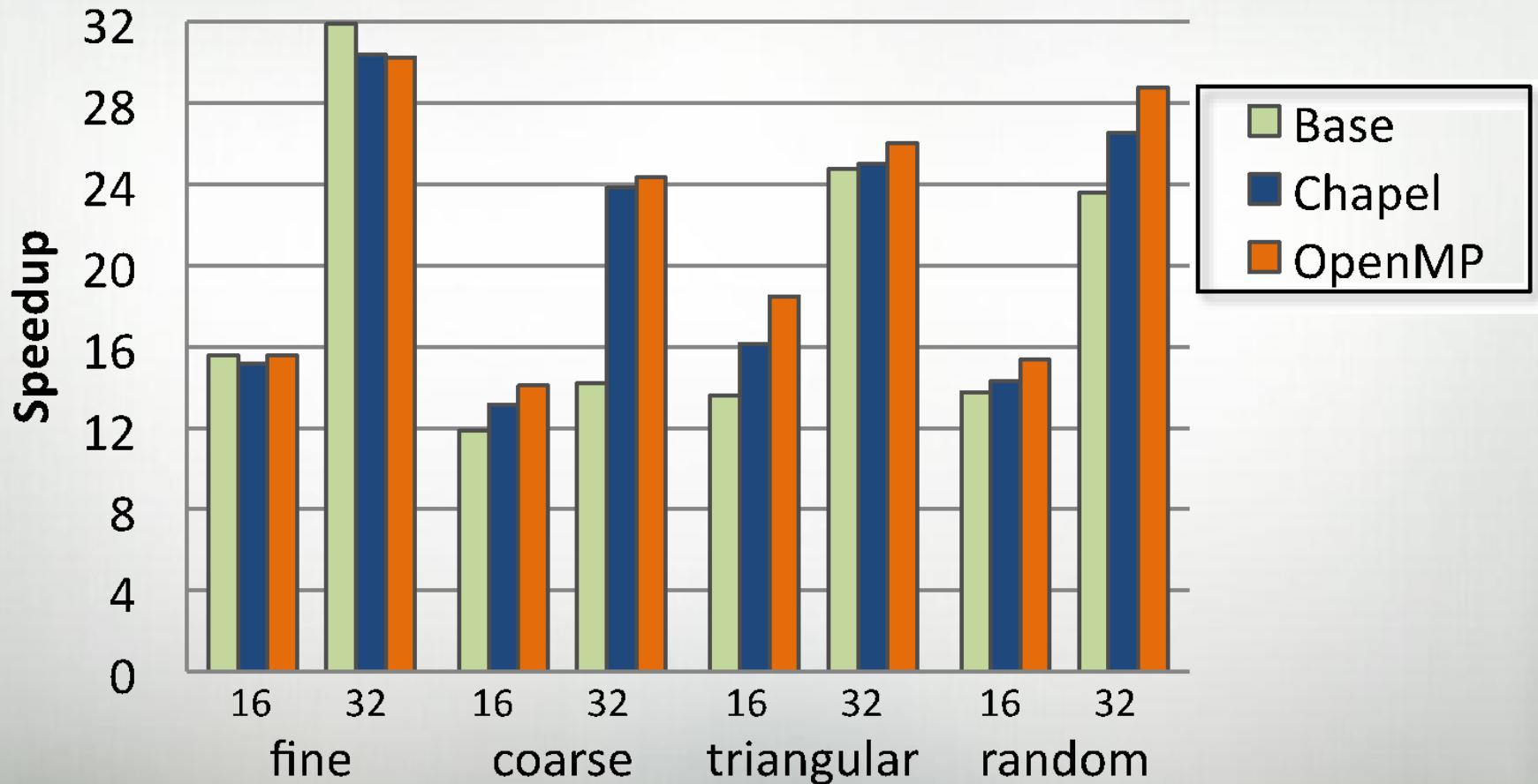
Controlling Data Parallelism

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

Invoke a standalone leader iterator explicitly
(perhaps one that you wrote yourself).

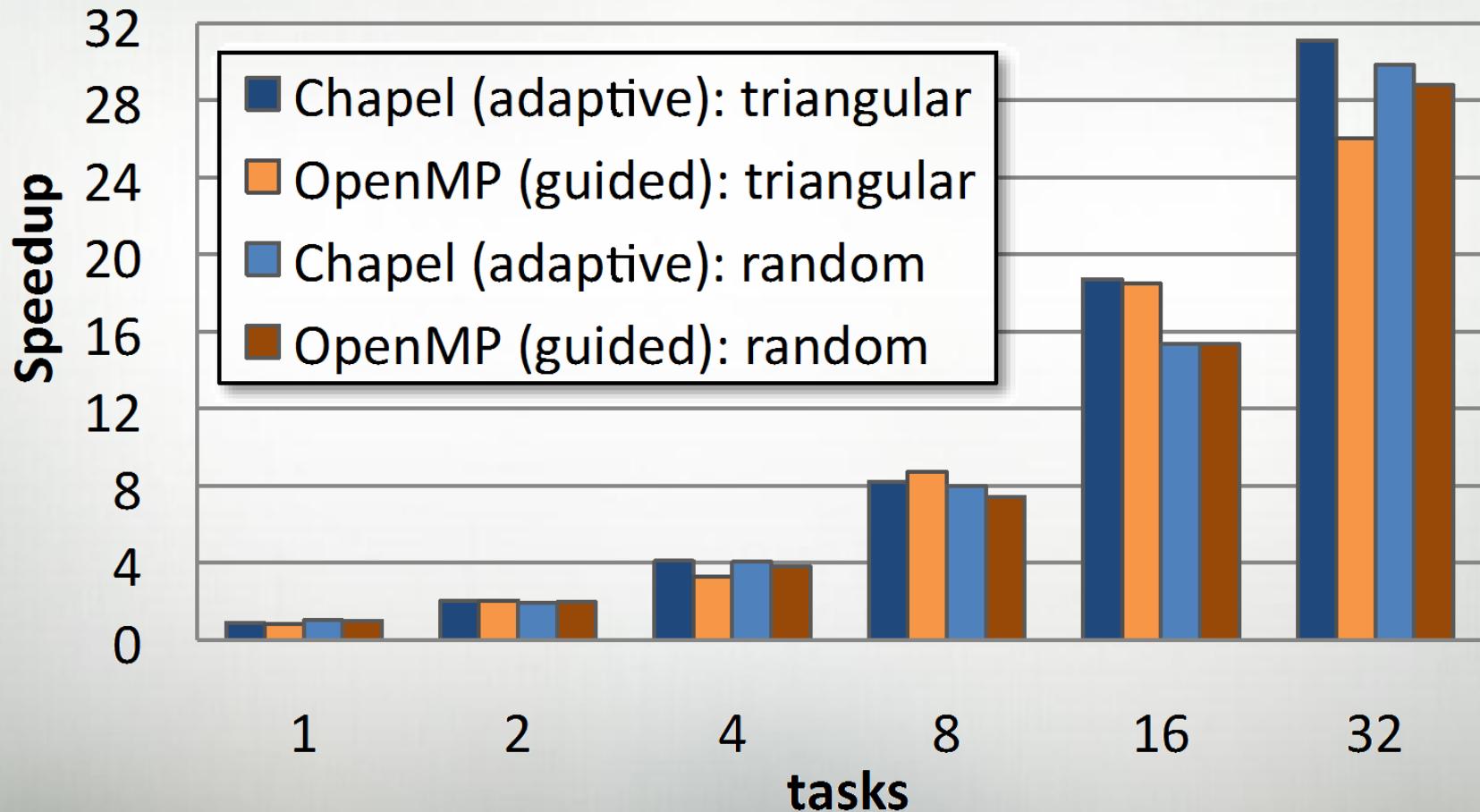
Guided Iteration: Chapel vs. OpenMP

Guided scheduling Speedups



Chapel Adaptive vs. OpenMP Guided

Adaptive Speedups



For More Information on Leader-Follower Iterators

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

Chapel release:

- Primer example introducing leader-follower iterators:
 - examples/primers/leaderfollower.chpl
- Library of dynamic leader-follower range iterators:
 - *AdvancedIter*s chapter of language specification

Summary of This Section

- 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 separates the roles of domain scientist, parallel programmer, and implementation cleanly

Outline

- ✓ Motivation
- ✓ Chapel Background and Themes
- ✓ Tour of Chapel Concepts
- ✓ Advanced Chapel Topics
- Project Status

Implementation Status -- Version 1.7.0 (Apr 2013)

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

Chapel and Education

- In teaching parallel programming, I like to cover:
 - data parallelism
 - task parallelism
 - concurrency
 - synchronization
 - locality/affinity
 - deadlock, livelock, and other pitfalls
 - performance tuning
 - ...
- I don't think there's been a good language out there...
 - for teaching *all* of these things
 - for teaching some of these things well at all
 - *until now:* We believe Chapel can potentially play a crucial role here
(see <http://chapel.cray.com/education.html> for more information and
<http://cs.washington.edu/education/courses/csep524/13wi/> for my use of Chapel in class)

The Cray Chapel Team (Summer 2012)



Chapel Community (see chapel.cray.com/collaborations.html for further details and ideas)

- **Lightweight Tasking using Qthreads:** Sandia (Kyle Wheeler, Dylan Stark, Rich Murphy)
 - [paper at CUG, May 2011](#)
- **Parallel File I/O, Bulk-Copy Opt:** U Malaga (Rafael Asenjo, Maria Angeles Navarro, et al.)
 - [papers at ParCo, Aug 2011; SBAC-PAD, Oct 2012](#)
- **I/O, LLVM back-end, etc.:** LTS (Michael Ferguson, Matthew Lentz, Joe Yan, et al.)
- **Interoperability via Babel/BRAID:** LLNL/Rice (Tom Epperly, Adrian Prantl, Shams Imam)
 - [paper at PGAS, Oct 2011](#)
- **Application Studies:** LLNL (Rob Neely, Bert Still, Jeff Keasler)
- **Interfaces/Generics/OOP:** CU Boulder (Jeremy Siek, Jonathan Turner, et al.)
- **Futures/Task-based Parallelism:** Rice (Vivek Sarkar, Shams Imam, Sagnak Tasirlar, et al.)
- **Lightweight Tasking using MassiveThreads:** U Tokyo (Kenjiro Taura, Jun Nakashima)
- **CPU-accelerator Computing:** UIUC (David Padua, Albert Sidelnik, Maria Garzarán)
 - [paper at IPDPS, May 2012](#)
- **Model Checking and Verification:** U Delaware (Stephen Siegel, T. Zirkel, T. McClory)
- **Chapel-MPI Compatibility:** Argonne (Pavan Balaji, Rajeev Thakur, Rusty Lusk, Jim Dinan)

Summary

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

- yet, without necessarily abdicating control
- Chapel does this via its multiresolution design
 - Here, we saw it in domain maps and leader-follower iterators

We believe Chapel can greatly improve productivity

...for current and emerging HPC architectures

...and for the growing need for parallel programming in the mainstream

Some Next Steps

- Hierarchical Locales
- Performance Optimizations
- Evolve from Prototype- to Production-grade
- Establish post-HPCS Project

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, ...

Blog Series:

Myths About Scalable Programming Languages:

<https://www.ieeetcsc.org/activities/blog/>

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



CRAY
THE SUPERCOMPUTER COMPANY

<http://chapel.cray.com>

chapel_info@cray.com

<http://sourceforge.net/projects/chapel/>