Multiresolution Global-View Programming in Chapel

Brad Chamberlain, Chapel Team, Cray Inc.

Argonne Training Program on Extreme-Scale Computing

August 1st, 2013

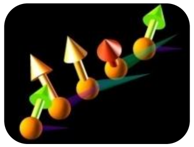
Sustained Performance Milestones

1 GF – 1988: Cray Y-MP; 8 Processors

- Static finite element analysis

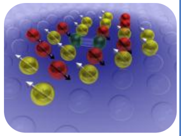
1 TF – 1998: Cray T3E; 1,024 Processors

- Modeling of metallic magnet atoms



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

- Superconductive materials



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

- TBD

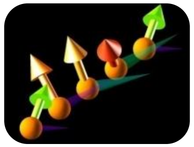
Sustained Performance Milestones

1 GF – 1988: Cray Y-MP; 8 Processors

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

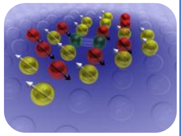
1 TF – 1998: Cray T3E; 1,024 Processors

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



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

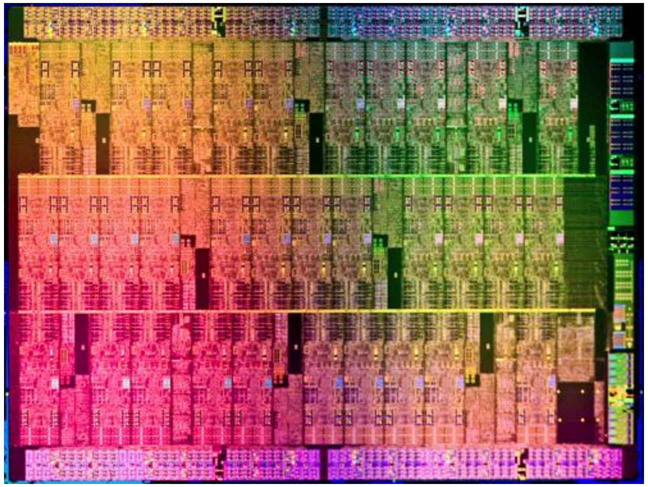
- Superconductive materials
- C++/Fortran + MPI + vectorization



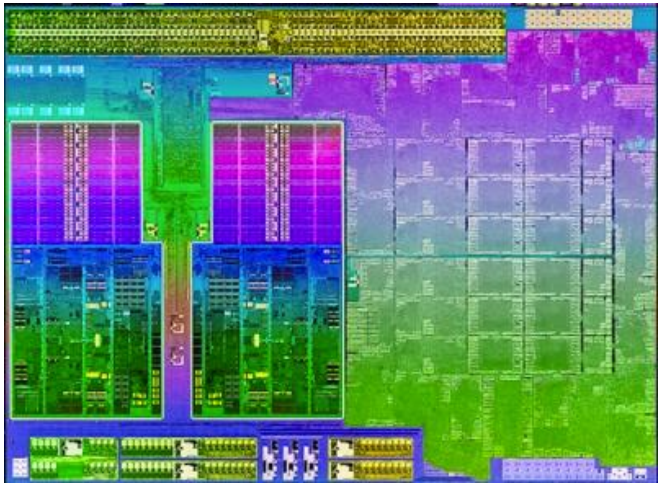
1 EF – ~2018: Cray ____; ~10,000,000 Processors

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

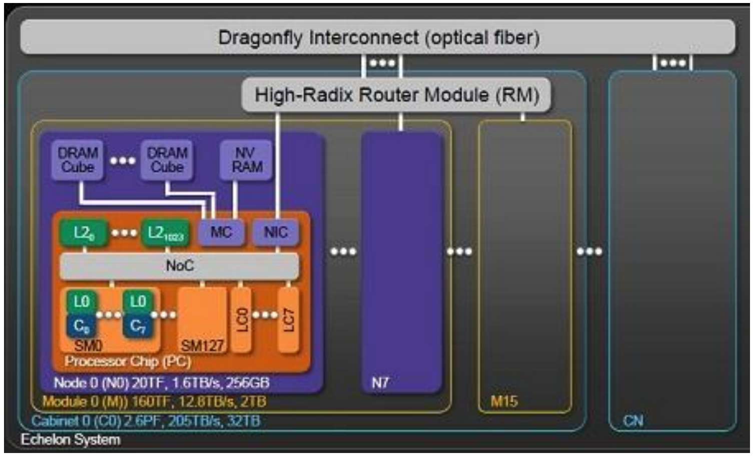
Prototypical Next-Gen Processor Technologies



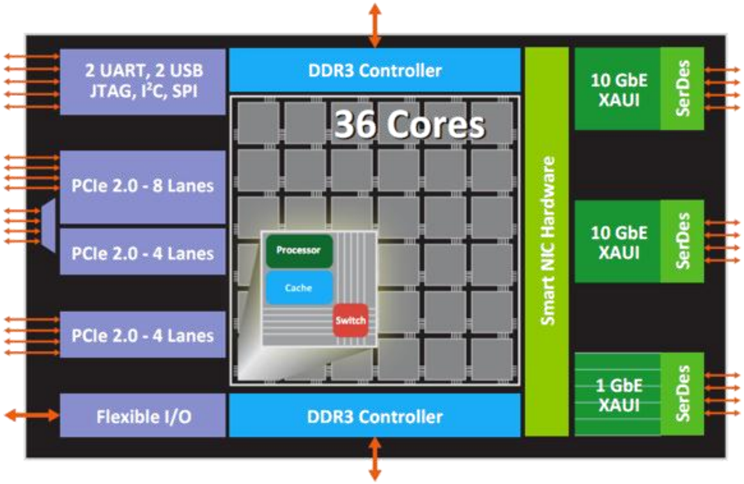
Intel MIC



AMD Trinity

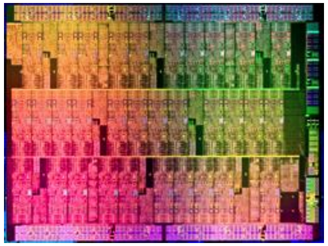


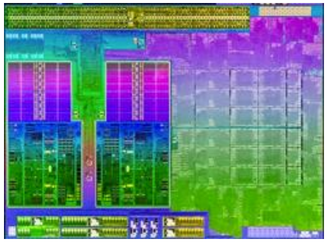
Nvidia Echelon

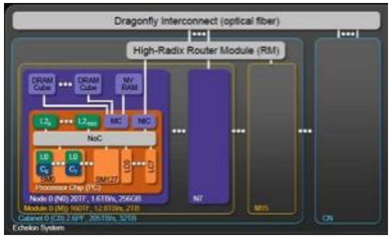


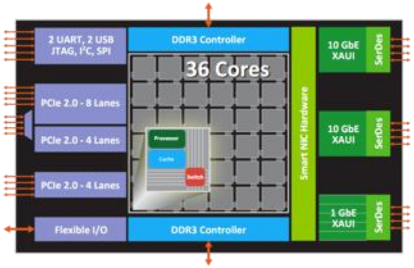
Tilera Tile-Gx

General Characteristics of These Architectures









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

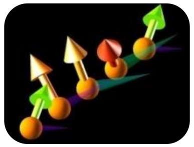
⇒ Next-gen programmers will have a lot more to think about at the node level than in the past

Sustained Performance Milestones

1 GF – 1988: Cray Y-MP; 8 Processors

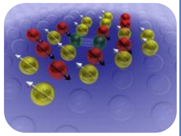
- Static finite element analysis
- Fortran77 + Cray autotasking + vectorization

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?

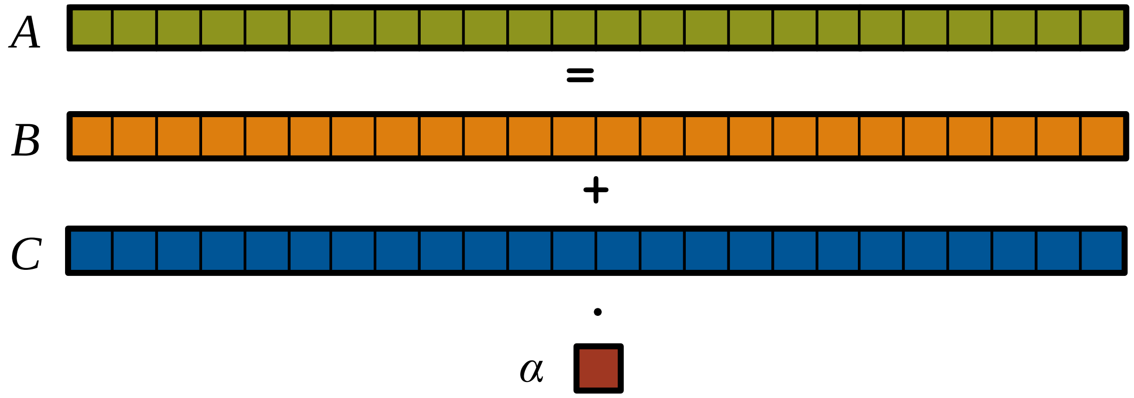
Or, perhaps something completely different?

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:

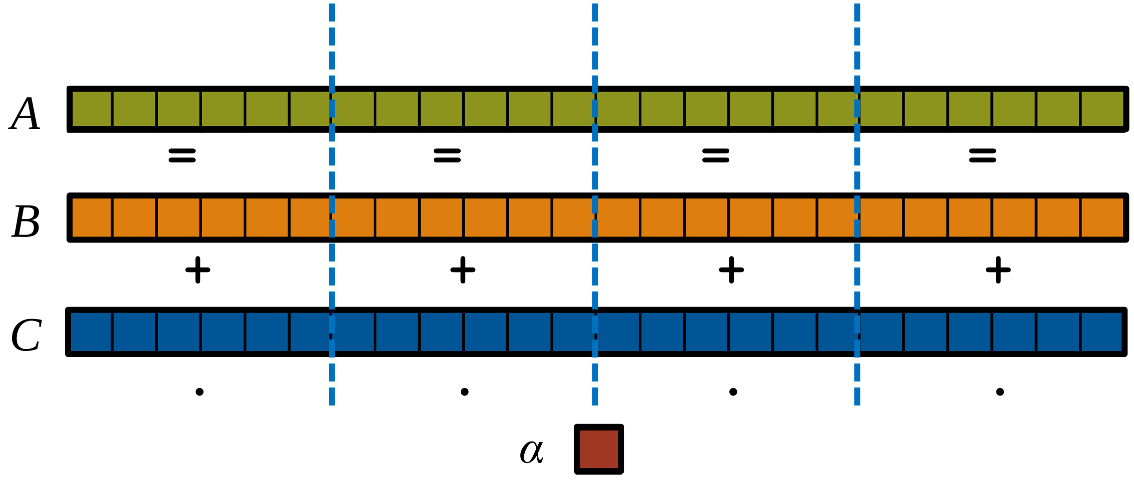


STREAM Triad: a trivial parallel computation

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:

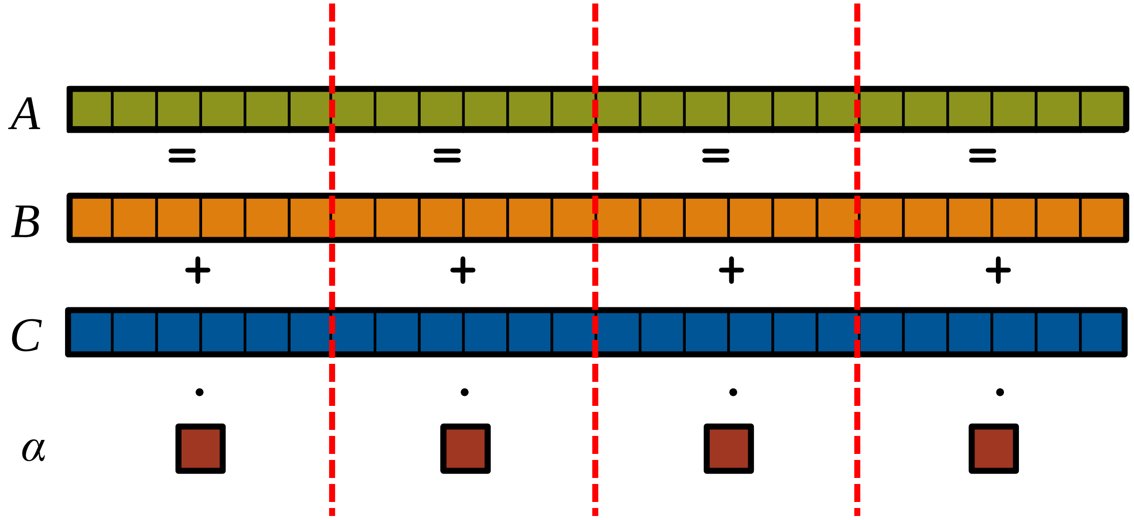


STREAM Triad: a trivial parallel computation

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):

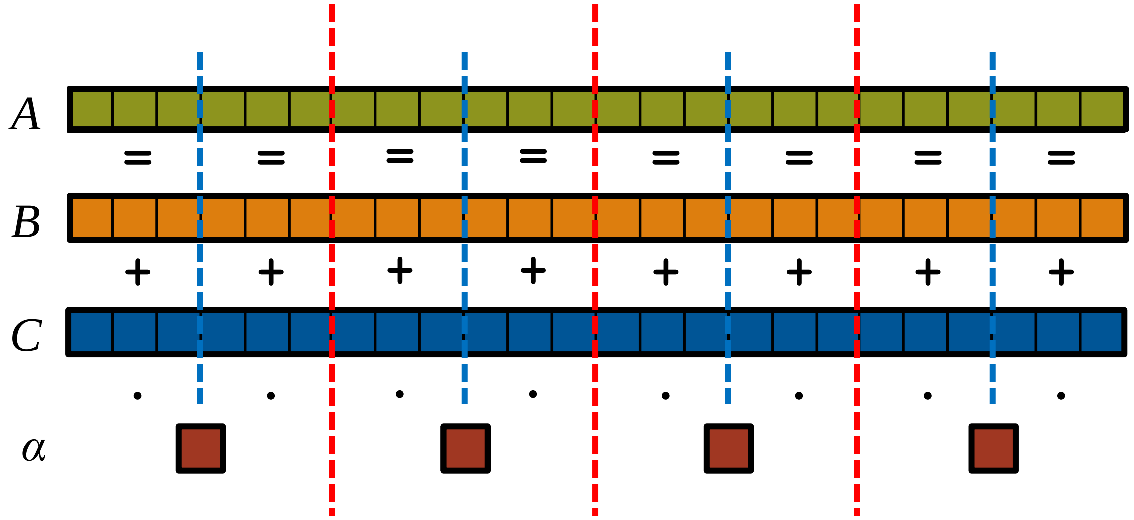


STREAM Triad: a trivial parallel computation

Given: m-element vectors A, B, C

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

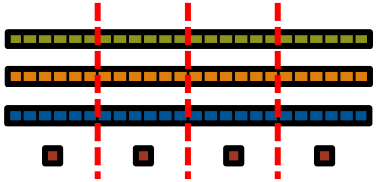
In pictures, in parallel (distributed memory multicore):



STREAM Triad: MPI

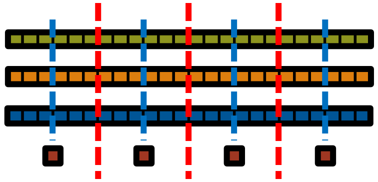
MPI

```
#include <hpcc.h>




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

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

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

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

  return errCount;
}

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

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

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



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




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

scalar = 3.0;




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

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

STREAM Triad: MPI+OpenMP



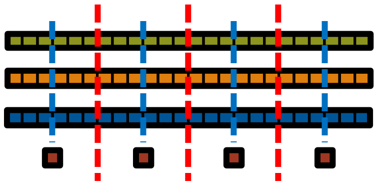
MPI + OpenMP

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

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

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

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

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

  return errCount;
}

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

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

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

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

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

  scalar = 3.0;

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

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

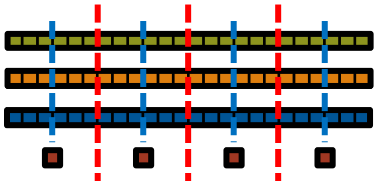
STREAM Triad: MPI+OpenMP vs. CUDA

MPI + OpenMP



```
#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

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

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

  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);


__global__ void set_array(float *a,  float value, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) a[idx] = value;
}

__global__ void STREAM_Triad( float *a, float *b, float *c,
                              float scalar, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) c[idx] = a[idx]+scalar*b[idx];
```

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

Why so many programming models?

HPC has traditionally given users…

…low-level, *control-centric* programming models
…ones that are closely tied to the underlying hardware
…ones that support only a single type of parallelism

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:

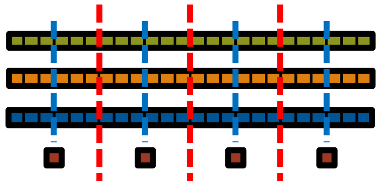
“This is all well and good for HPC users, but I’m a mainstream desktop programmer, so this is all academic for me.”

The Unfortunate Reality:

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

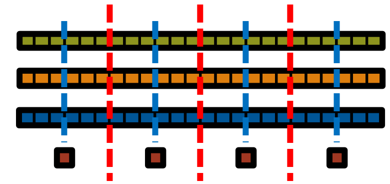
Rewinding a few slides…

MPI + OpenMP



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

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

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

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

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

  return errCount;
}

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);

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

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

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

  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);


__global__ void set_array(float *a,  float value, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) a[idx] = value;
}

__global__ void STREAM_Triad( float *a, float *b, float *c,
                              float scalar, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) c[idx] = a[idx]+scalar*b[idx];
```

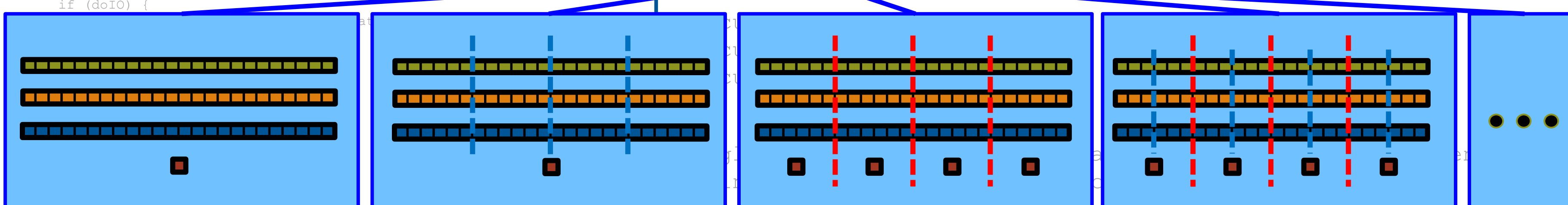
HPC suffers from too many distinct notations for expressing parallelism and locality

STREAM Triad: Chapel

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



<u>Philosophy:</u> 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 and Implementation
- Project Status and Next Steps

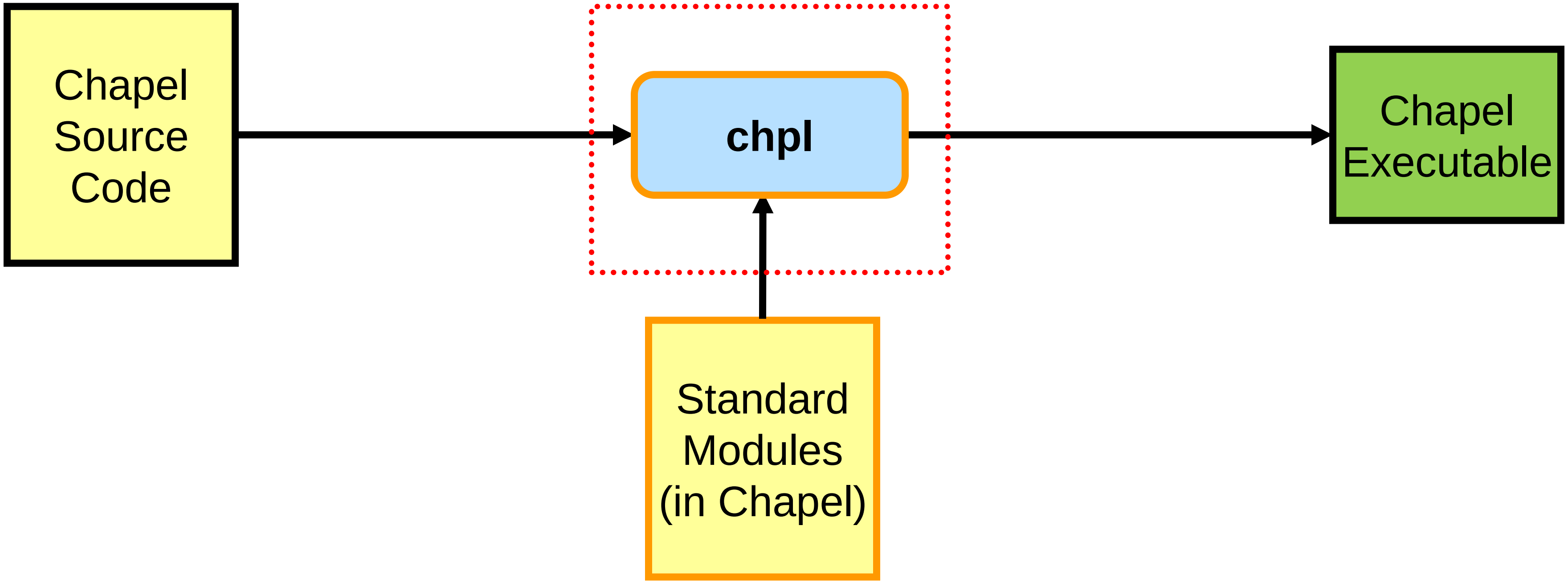
What is Chapel?

- **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**

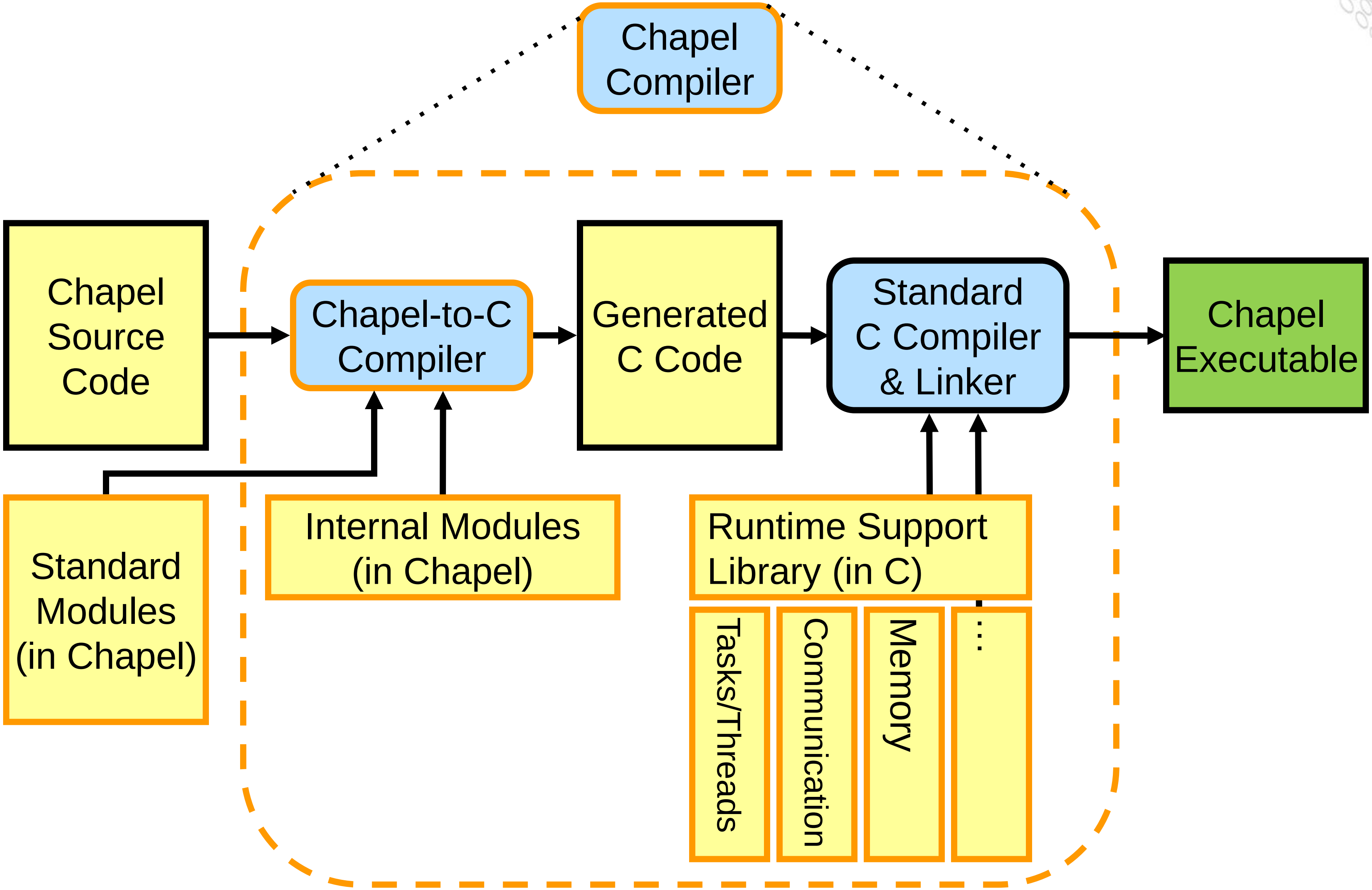
Chapel's Implementation

- **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, …

Compiling Chapel



Chapel Compiler Architecture



Motivating Chapel Themes

1) General Parallel Programming
2) Global-View Abstractions
3) Multiresolution Design
4) Control over Locality/Affinity
5) Reduce HPC $\leftrightarrow$ Mainstream Language Gap

1) General Parallel Programming

With a unified set of concepts...

...express any parallelism desired in a user's program

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

...target any parallelism available in the hardware

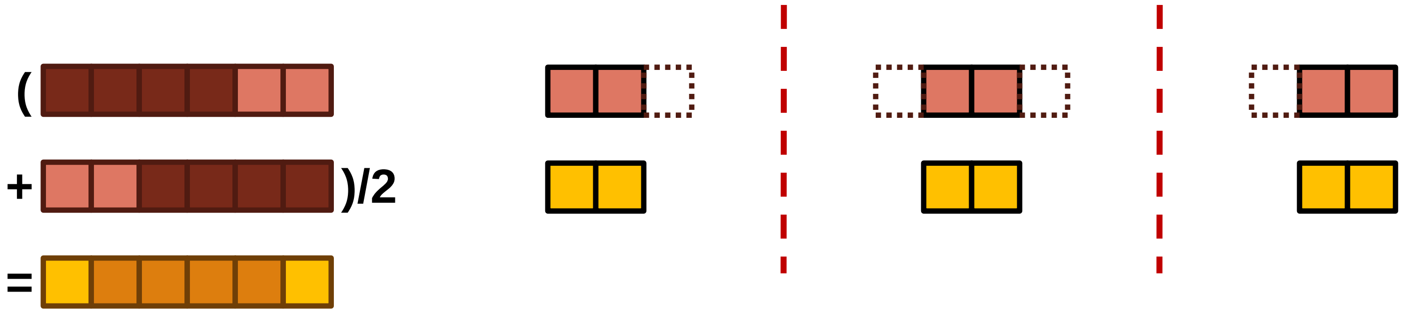
- **Types:** machines, nodes, cores, instructions

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

2) Global-View Abstractions

In pictures: "Apply a 3-Point Stencil to a vector"

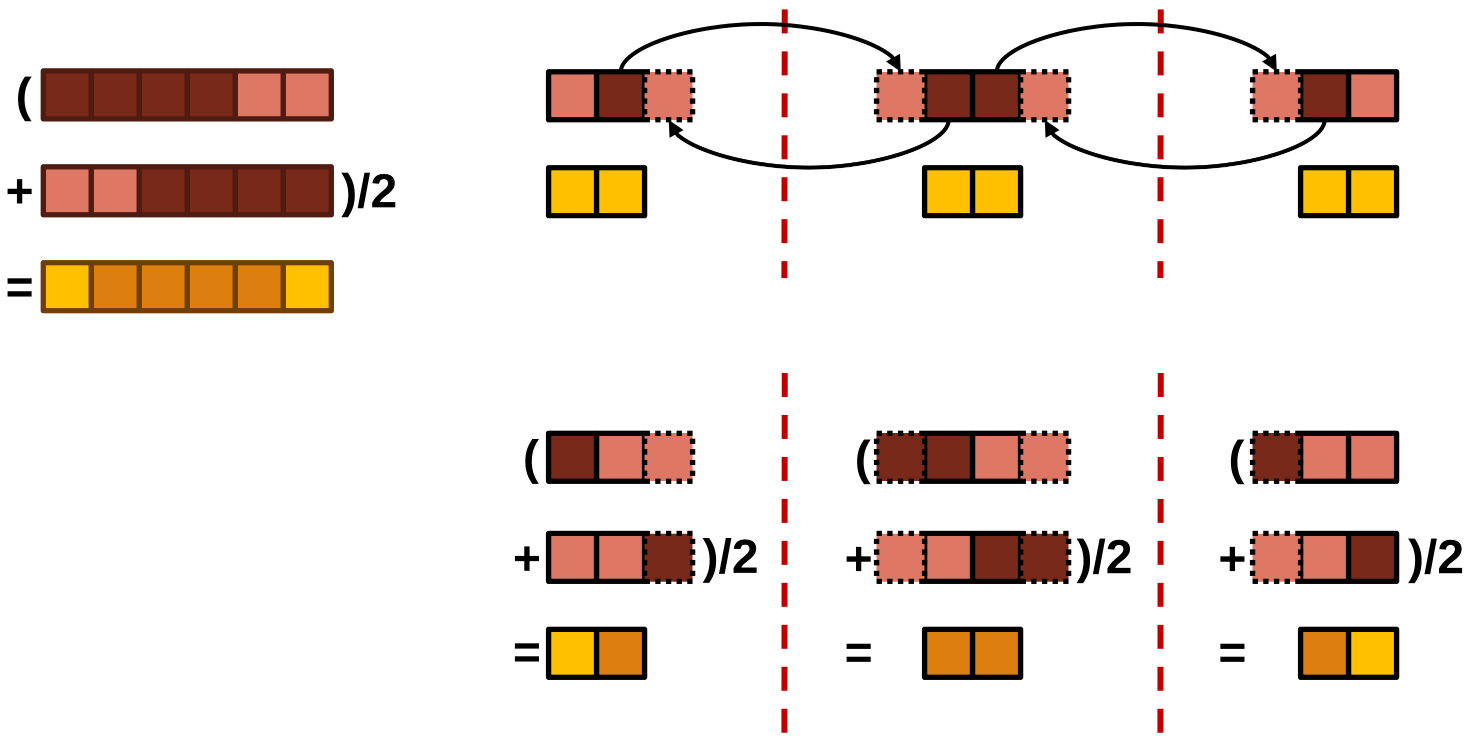
Global-View ***Local-View***



2) Global-View Abstractions

In pictures: “Apply a 3-Point Stencil to a vector”

Global-View ***Local-View***



2) Global-View Abstractions

In code: “Apply a 3-Point Stencil to a vector”

Global-View

```
proc main() {
  var n = 1000;
  var A, B: [1..n] real;

  forall i in 2..n-1 do
    B[i] = (A[i-1] + A[i+1])/2;
}
```

Local-View (SPMD)

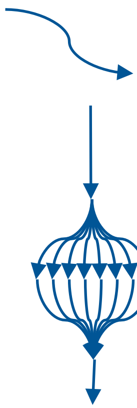
```
proc main() {
  var n = 1000;
  var p = numProcs(),
      me = myProc(),
      myN = n/p,
  var A, B: [0..myN+1] real;

  if (me < p-1) {
    send(me+1, A[myN]);
    recv(me+1, A[myN+1]);
  }
  if (me > 0) {
    send(me-1, A[1]);
    recv(me-1, A[0]);
  }
  forall i in 1..myN do
    B[i] = (A[i-1] + A[i+1])/2;
}
```

Bug: Refers to uninitialized values at ends of A

2) Global-View Abstractions

In code: "Apply a 3-Point Stencil to a vector"

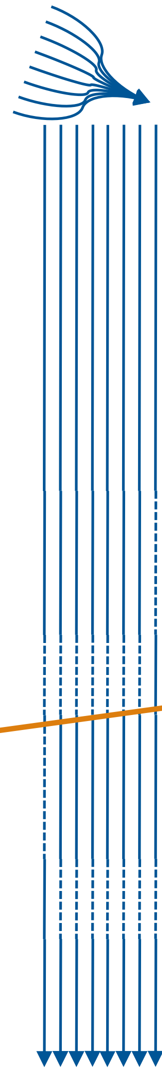
Global-View



```
proc main() {
  var n = 1000;
  var A, B: [1..n] real;

  forall i in 2..n-1 do
    B[i] = (A[i-1] + A[i+1])/2;
}
```

Communication becomes geometrically more complex for higher-dimensional arrays

Local-View (SPMD)



```
proc main() {
  var n = 1000;
  var p = numProcs(),
      me = myProc(),
      myN = n/p,
      myLo = 1,
      myHi = myN;
  var A, B: [0..myN+1] real;

  if (me < p-1) {
    send(me+1, A[myN]);
    recv(me+1, A[myN+1]);
  } else
    myHi = myN-1;
  if (me > 0) {
    send(me-1, A[1]);
    recv(me-1, A[0]);
  } else
    myLo = 2;
  forall i in myLo..myHi do
    B[i] = (A[i-1] + A[i+1])/2;
}
```

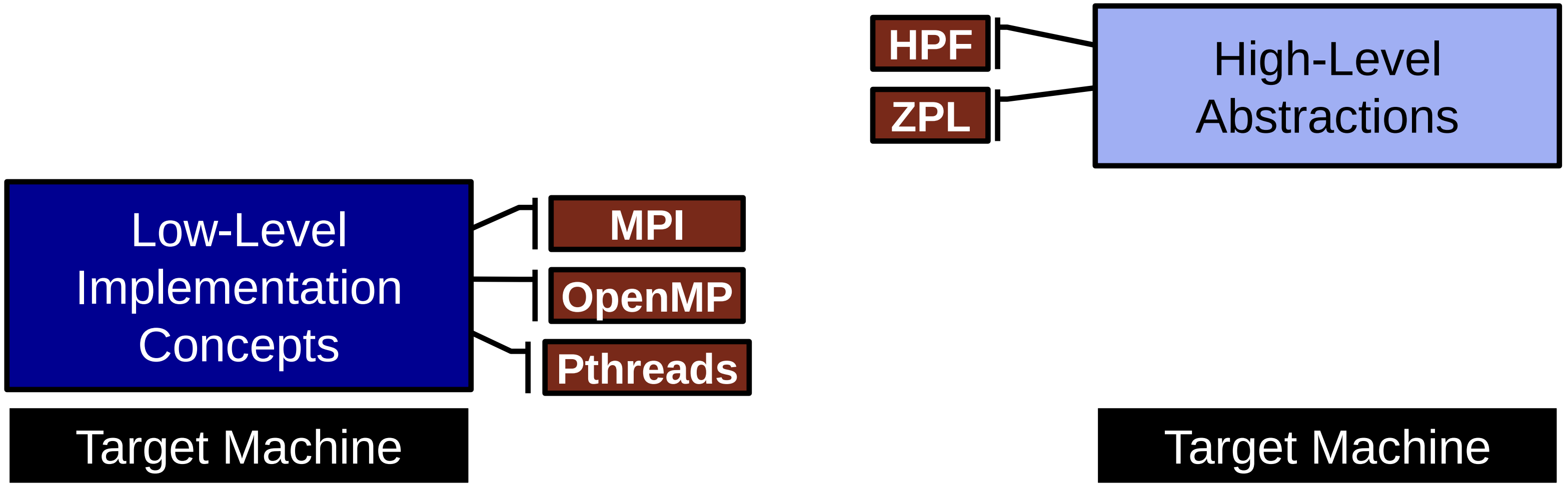
Assumes p divides n

2) Global-View Programming: A Final Note

- **A language may support both global- and local-view programming — in particular, Chapel does**

```
proc main() {
  coforall loc in Locales do
    on loc do
      MySPMDProgram(loc.id, Locales.numElements);

}

proc MySPMDProgram(myImageID, numImages) {
  ...
}
```

3) Multiresolution Design: Motivation



"Why is everything so tedious/difficult?"

"Why don't my programs port trivially?"

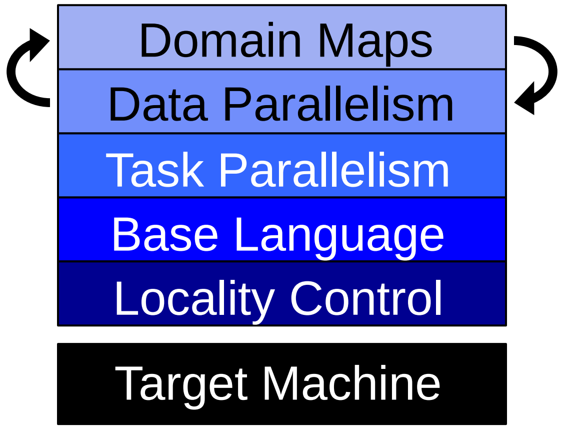
"Why don't I have more control?"

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

4) Control over Locality/Affinity

Consider:

- Scalable architectures package memory near processors
- Remote accesses take longer than local accesses

Therefore:

- Placement of data relative to tasks affects scalability
- Give programmers control of data and task placement

Note:

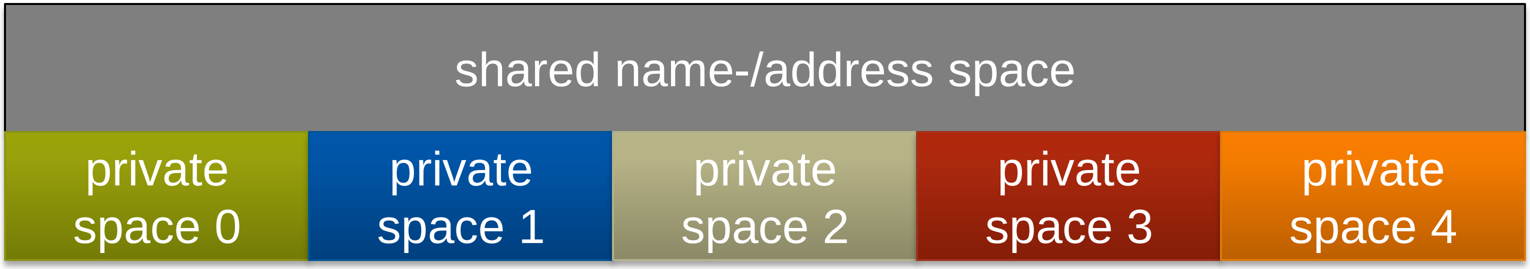
- Over time, we expect locality to matter more and more within the compute node as well

Partitioned Global Address Space Languages

(Or perhaps: partitioned global namespace languages)

abstract concept:

- support a shared namespace on distributed memory
  - permit any parallel task to access any lexically visible variable
  - doesn't matter if it's local or remote

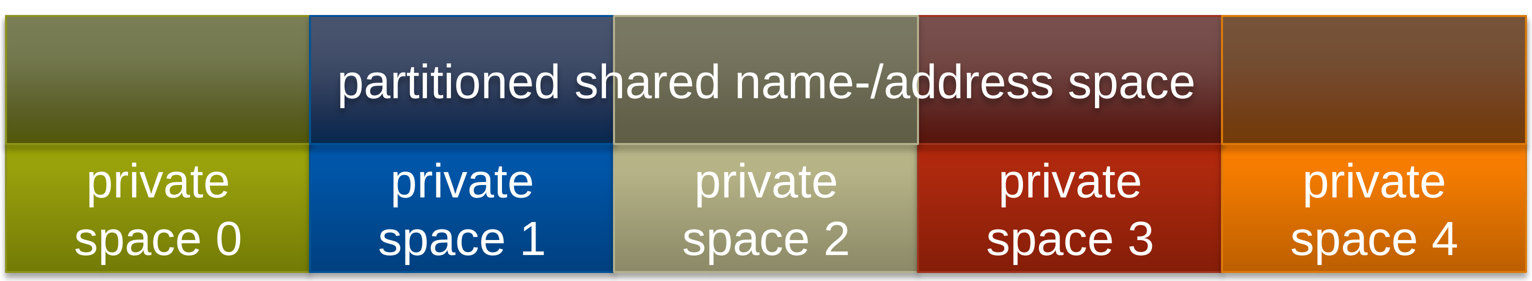


Partitioned Global Address Space Languages

(Or perhaps: partitioned global namespace languages)

abstract concept:

- support a shared namespace on distributed memory
  - permit any parallel task to access any lexically visible variable
  - doesn't matter if it's local or remote
- establish a strong sense of ownership
  - every variable has a well-defined location
  - local variables are cheaper to access than remote ones



Traditional PGAS Languages

PGAS founding members: Co-Array Fortran, UPC, Titanium

- extensions to Fortran, C, and Java, respectively
- details vary, but potential for:
  - arrays that are decomposed across compute nodes
  - pointers that refer to remote objects
- note that earlier languages could arguably also be considered PGAS, but the term hadn't been coined yet

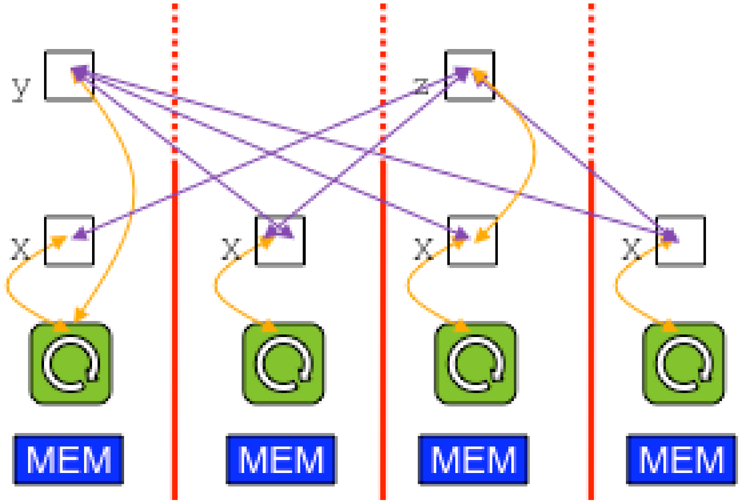
PGAS: What's in a Name?

		memory model	programming model	execution model	data structures	communication
	MPI	distributed memory	cooperating executables (often SPMD in practice)		manually fragmented	APIs
	OpenMP	shared memory	global-view parallelism	shared memory multithreaded	shared memory arrays	N/A
Trad. PGAS Languages	**CAF**	PGAS	Single Program, Multiple Data (SPMD)		co-arrays	co-array refs
	UPC				1D block-cyc arrays/ distributed pointers	implicit
	Titanium				class-based arrays/ distributed pointers	method-based
	Chapel	PGAS	global-view parallelism	distributed memory multithreaded	global-view distributed arrays	implicit

Traditional PGAS Languages

e.g., Co-Array Fortran, UPC

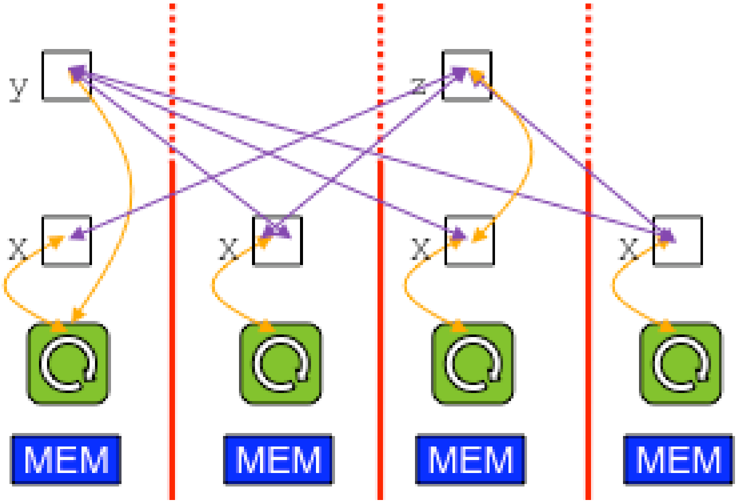
- support a shared namespace, like shared-memory
- support a strong sense of ownership and locality
  - each variable is stored in a particular memory segment
  - tasks can access any visible variable, local or remote
  - local variables are cheaper to access than remote ones
- implicit communication eases user burden; permits compiler to use best mechanisms available



Traditional PGAS Languages

e.g., Co-Array Fortran, UPC

- restricted to SPMD programming and execution models
- data structures not as flexible/rich as one might like
- retain many of the downsides of shared-memory
  - error cases, memory consistency models



5) Reduce HPC ↔ Mainstream Language Gap

Consider:

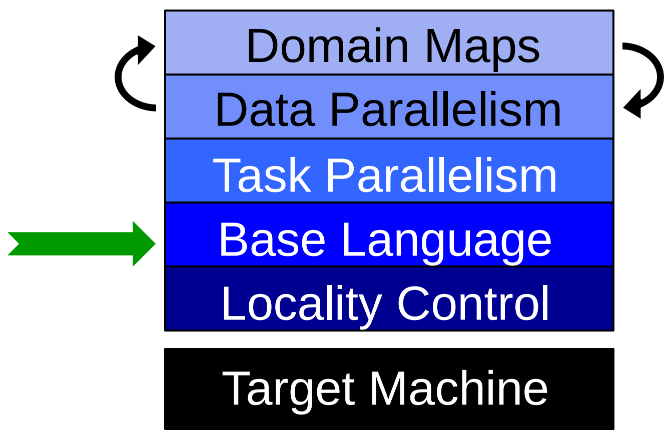
- Students graduate with training in Java, Matlab, Perl, Python
- Yet HPC programming is dominated by Fortran, C/C++, MPI

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 and Implementation



- Project Status and Next Steps

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 by 2);
printVals(r by -2);
printVals(r by 2 # 3);
printVals(r # 3 by 2);
printVals(0.. #n);

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

```
1 2 3
1 3 5 7 9
10 8 6 4 2
1 3 5
1 3
0 1 2 3 4 … n-1
```

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({1..m, 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)
```

Zippered Iteration

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

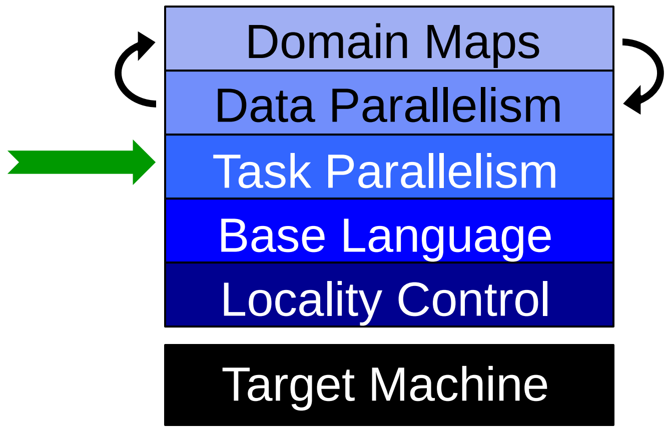
```
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
…
```

Other Base Language Features

- **tuple types and values**
- **rank-independent programming features**
- **interoperability features**
- **compile-time features for meta-programming**
  - e.g., compile-time functions to compute types, parameters
- **OOP (value- and reference-based)**
- **argument intents, default values, match-by-name**
- **overloading, where clauses**
- **modules (for namespace management)**
- **…**

Outline

- ✓ Motivation
- ✓ Chapel Background and Themes
- ➢ Tour of Chapel Concepts and Implementation



- Project Status and Next Steps

Task Parallelism: Begin Statements

```
// create a fire-and-forget task for a statement
begin writeln("hello world");
writeln("good bye");
```

Possible outputs:

```
hello world
good bye
```

```
good bye
hello world
```

Task Parallelism: Cobegin Statements

```
// create a task per child statement
cobegin {
  producer(1);
  producer(2);
  consumer(1);
}  // implicit join of the three tasks here
```

Task Parallelism: Coforall Loops

```
// create a task per iteration
coforall t in 0..#numTasks {
  writeln("Hello from task ", t, " of ", numTasks);
} // implicit join of the numTasks tasks here

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

Sample output:

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

Task Parallelism: Data-Driven Synchronization

1) ***atomic variables:*** support atomic operations (as in C++)
- e.g., compare-and-swap; atomic sum, mult, etc.

2) ***single-assignment variables:*** reads block until assigned

3) ***synchronization variables:*** store full/empty state
- by default, reads/writes block until the state is full/empty

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] = …;   // writes block until empty, leave full
} }

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

Other Task Parallel Features

Current:

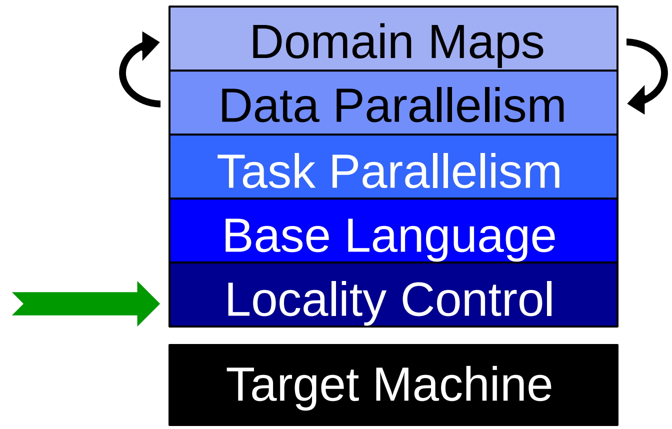
- **serial statements to conditionally squash parallelism**
- **sync statements to join dynamically generated tasks**

Planned:

- **task-private variables**
- **task teams to support**
  - collective operations (barriers, joins, reductions, etc.)
  - thread scheduling policies

Outline

- ✓ Motivation
- ✓ Chapel Background and Themes
- ➢ Tour of Chapel Concepts and Implementation



- Project Status and Next Steps

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 compute node (multicore processor or SMP)

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

- User's `main()` begins executing on locale #0

Locale Operations

- Locale methods support queries about the target system:

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

- *On-clauses* support placement of computations:

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

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

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

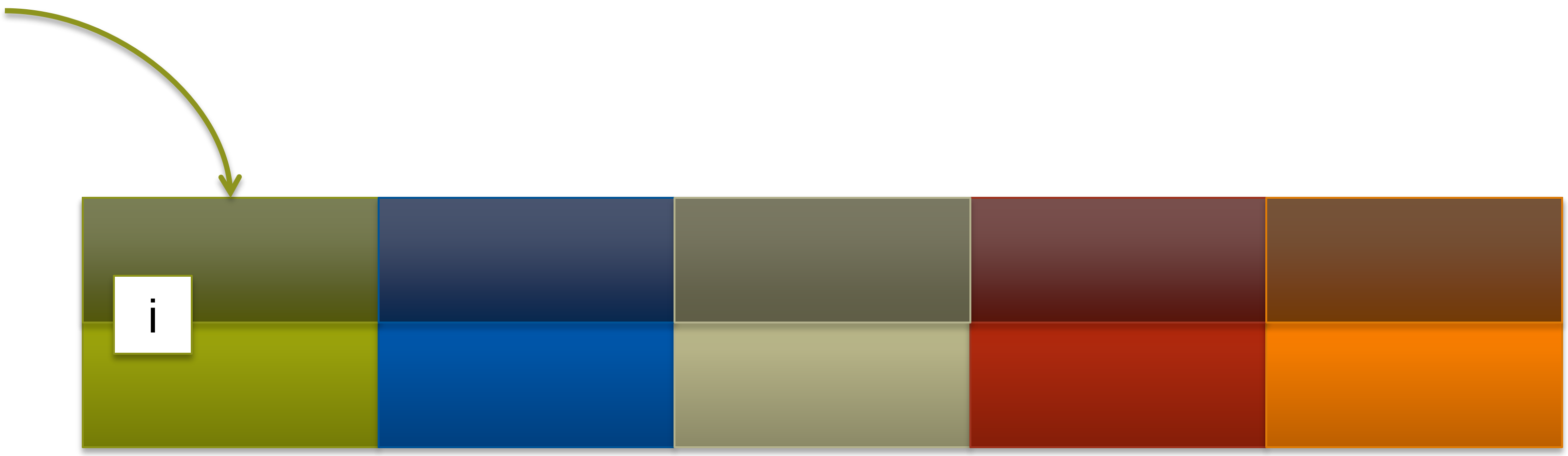
```
cobegin {
  on A[i,j] do
    bigComputation(A);

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

Chapel and PGAS

- **Chapel is PGAS, but unlike UPC/CAF, it's not SPMD**
  - ⇒ never think about "the other copies of the program"
  - ⇒ "global name-/address space" comes from lexical scoping
    - rather than: "We're all running the same program, so we must all have a variable named *x*"
    - as in traditional languages, each declaration yields one variable
    - stored on locale where task executes, not everywhere/thread 0

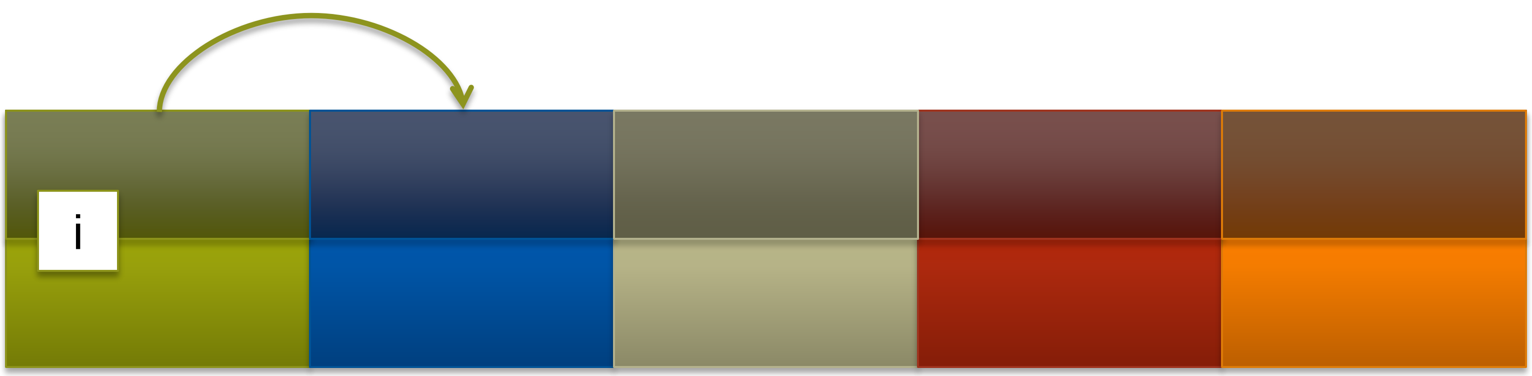
Chapel and PGAS

```
var i: int;
```



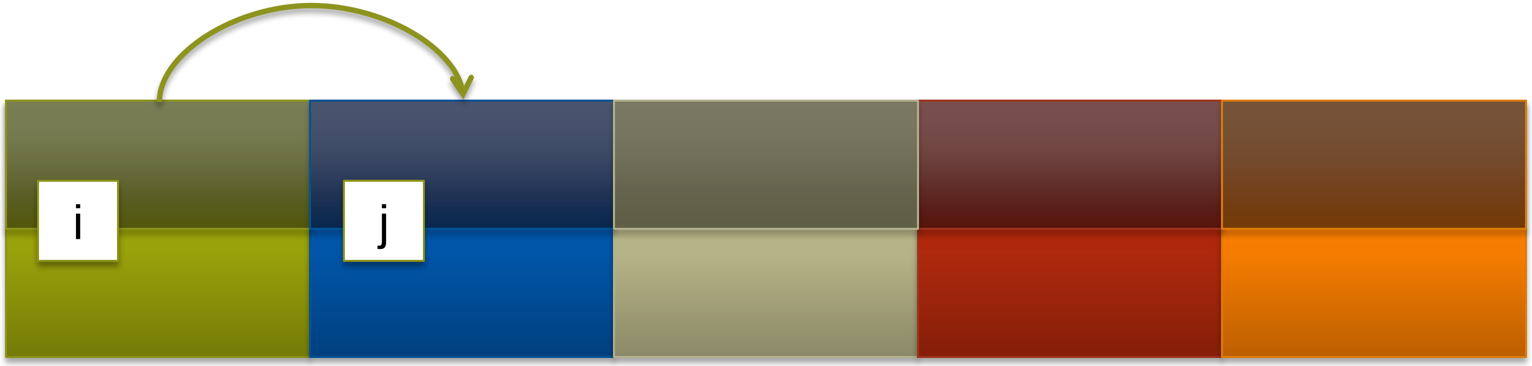
Chapel and PGAS

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



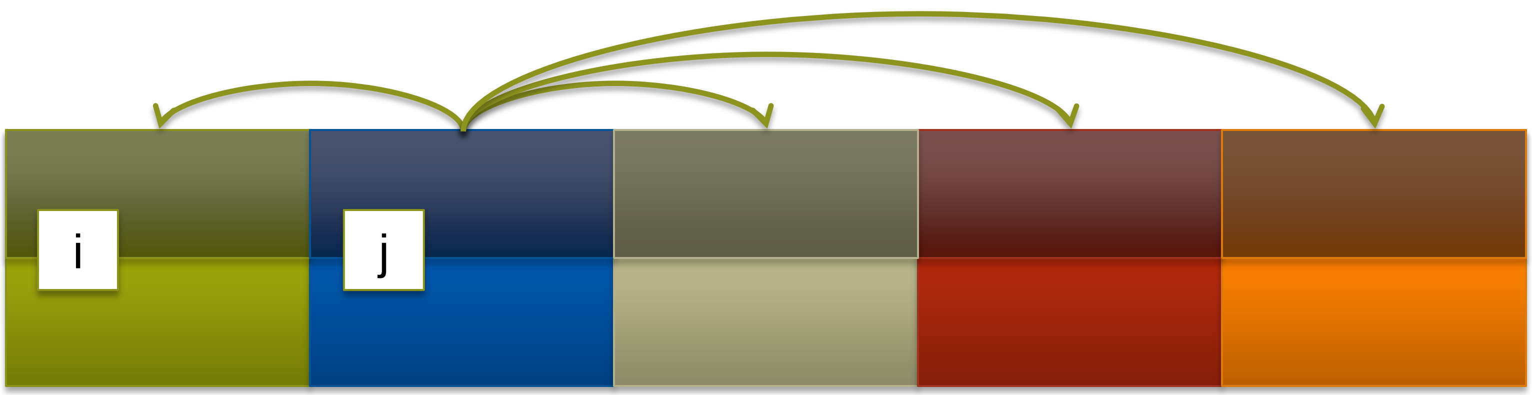
Chapel and PGAS

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



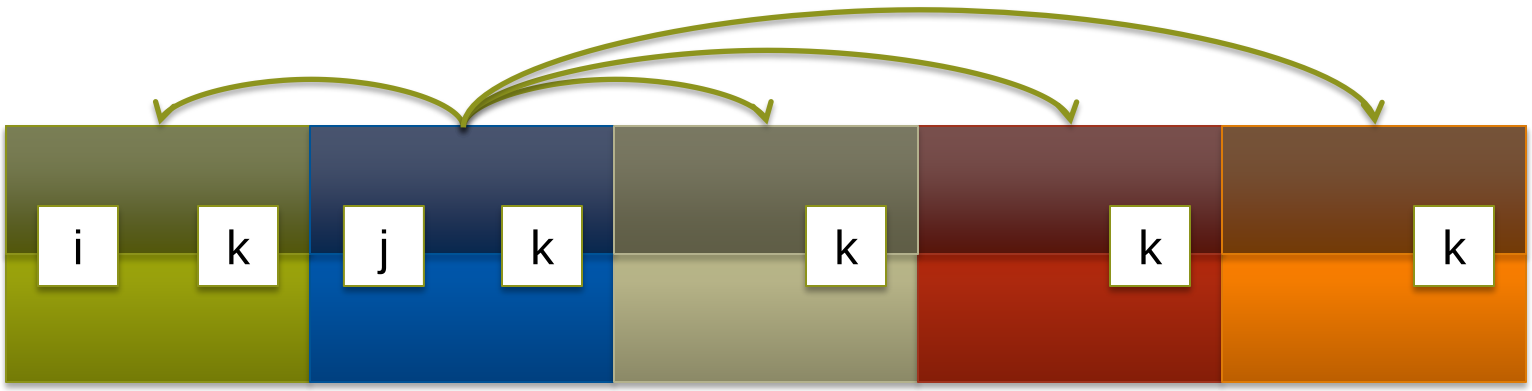
Chapel and PGAS

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



Chapel and PGAS

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

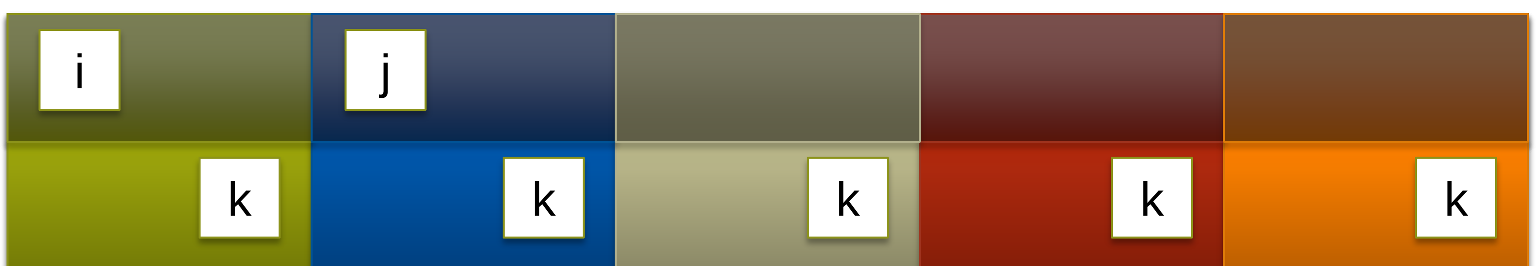


Chapel and PGAS: Public vs. Private

How public a variable is depends only on scoping

- who can see it?
- who actually bothers to refer to it non-locally?

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



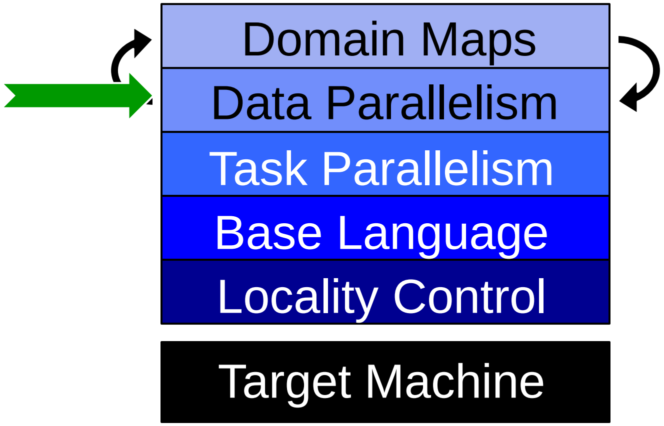
Other Locality Features

Planned:

- **Locale-private variables**
- **Hierarchical locales for reasoning about intra-node locality**
  - (more on this at the end of the talk, time permitting)

Outline

- ✓ Motivation
- ✓ Chapel Background and Themes
- ➢ Tour of Chapel Concepts and Implementation



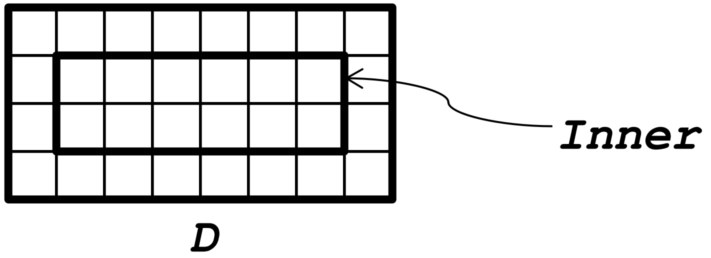
- Project Status and Next Steps

Domains

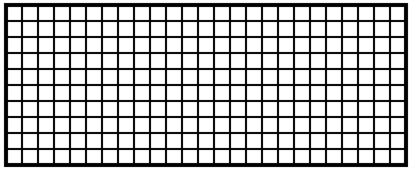
Domain:

- A first-class index set
- The fundamental Chapel concept for data parallelism

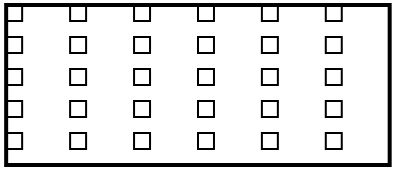
```
config const m = 4, n = 8;

var D: domain(2) = {1..m, 1..n};

var Inner: subdomain(D) = {2..m-1, 2..n-1};
```



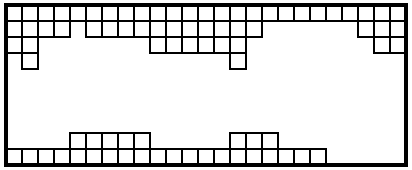
Chapel Domain Types



dense

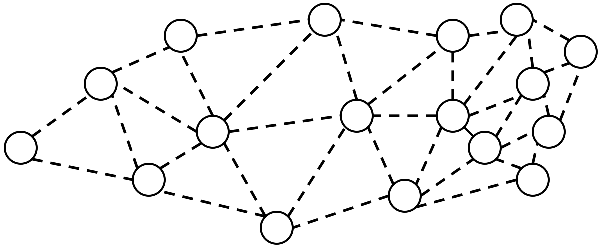


strided



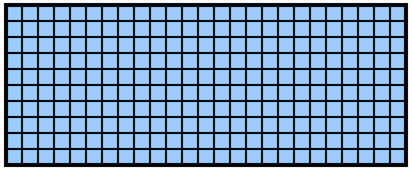
sparse

associative

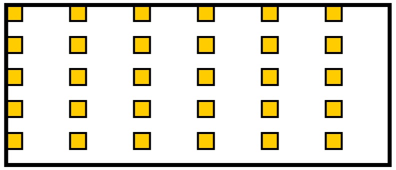


unstructured

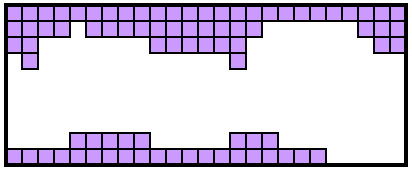
Chapel Array Types



dense

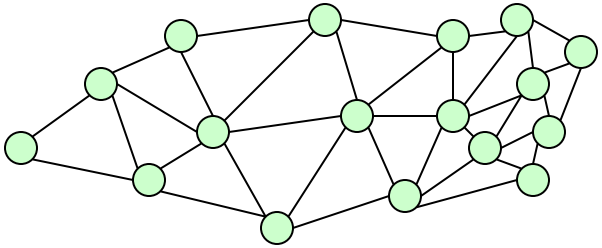


strided



sparse

associative



unstructured

Chapel Domain/Array Operations

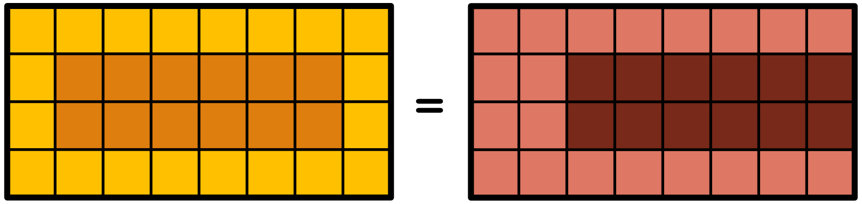
- Data Parallel Iteration (as well as serial and coforall)

```
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 many 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 ≤ #Tasks << #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

Promotion Semantics

Promoted functions/operators are defined in terms of zippered forall loops in Chapel. For example…

```
A = B;
```

…is equivalent to:

```
forall (a,b) in zip(A,B) do
  a = b;
```

Impact of Zippered Promotion Semantics

Whole-array operations are implemented element-wise…

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

⇒

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

…rather than operator-wise.

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

~~⇒~~

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

⇒ No temporary arrays required by semantics

- ⇒ No surprises in memory requirements
- ⇒ Friendlier to cache utilization

⇒ Differs from traditional array language semantics

```
A = A[0..n-1] + A[2..n+1];
```

⇒

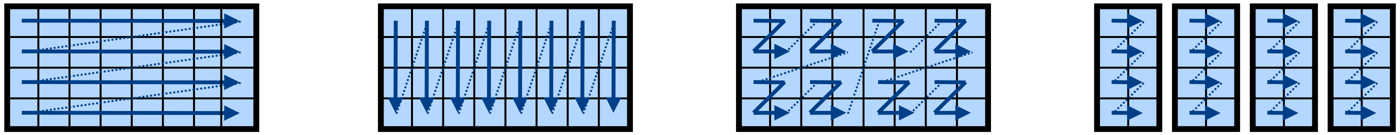
```
forall (a1, a2, a3)
  in (A, A[0..n-1], A[2..n+1]) do
    a1 = a2 + a3;
```

Read/write race!

Data Parallelism Implementation Qs

Q1: How are arrays laid out in memory?

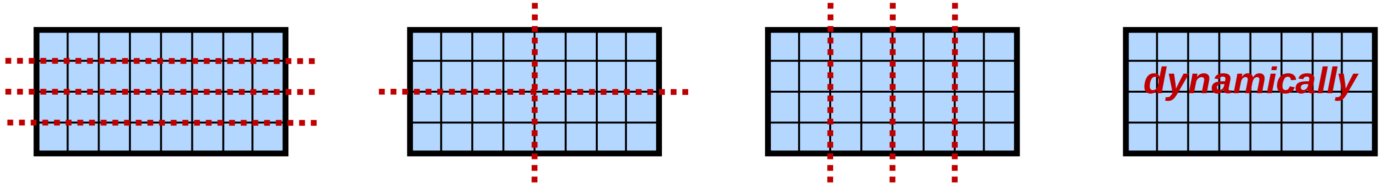
- Are regular arrays laid out in row- or column-major order? Or…?



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

Q2: How are arrays stored by the locales?

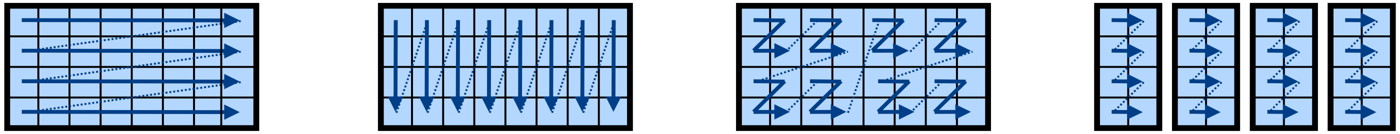
- Completely local to one locale? Or distributed?
- If distributed… In a blocked manner? cyclically? block-cyclically? recursively bisected? dynamically rebalanced? …?



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, …?)

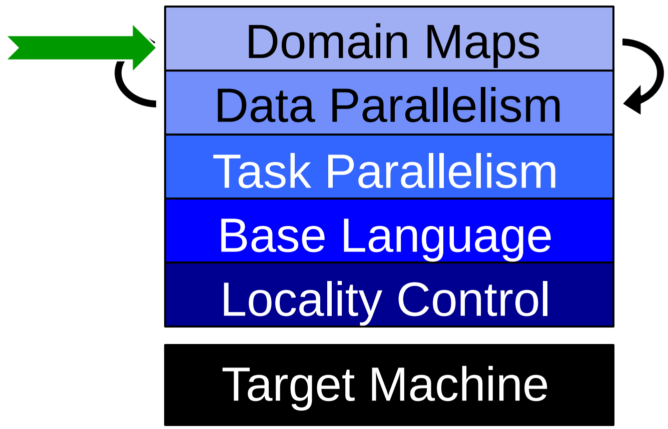
Q2: How are arrays stored by the locales?

- 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

Outline

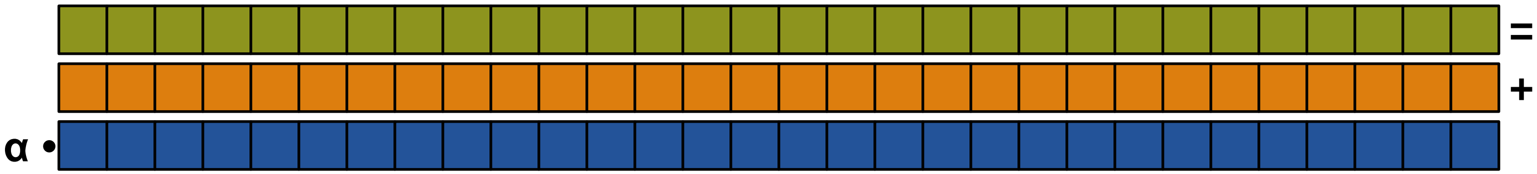
- ✓ Motivation
- ✓ Chapel Background and Themes
- ➢ Tour of Chapel Concepts and Implementation



- Project Status and Next Steps

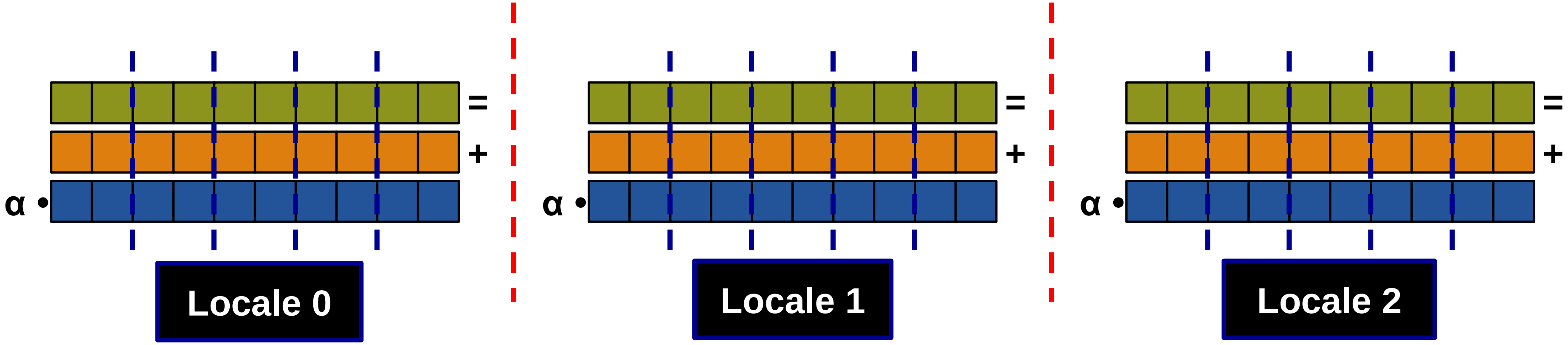
Domain Maps

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



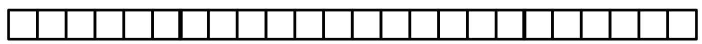
`A = B + alpha * C;`

…to the target locales’ memory and processors:

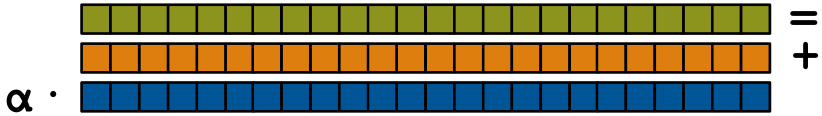


STREAM Triad: Chapel

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



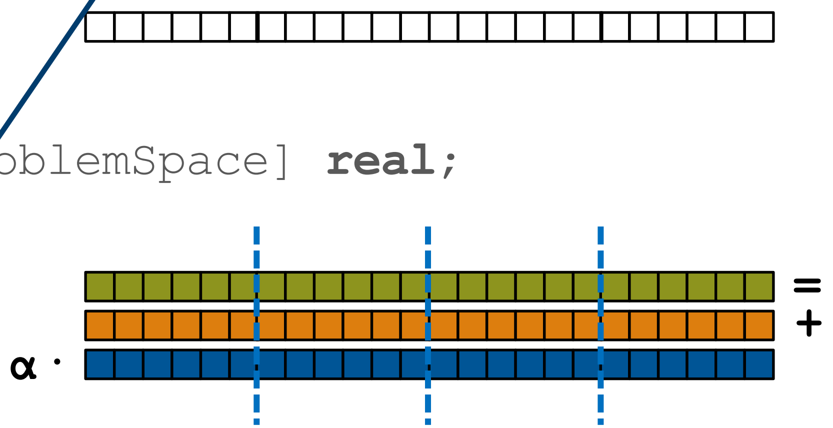
```
var A, B, C: [ProblemSpace] real;
```




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

STREAM Triad: Chapel (multicore)

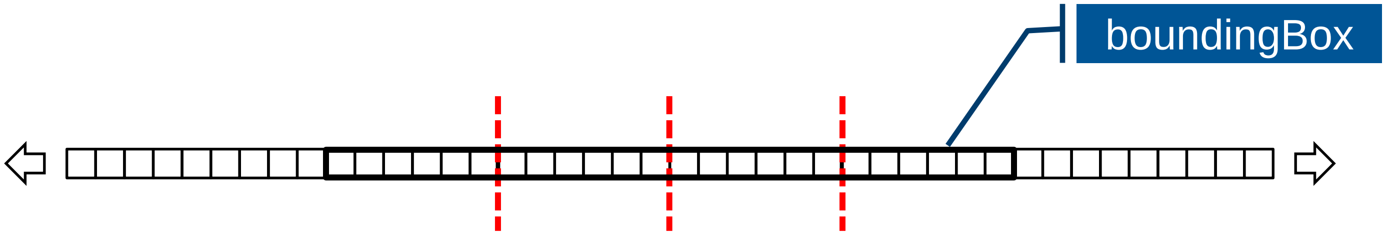
```
const ProblemSpace = {1..m};
```



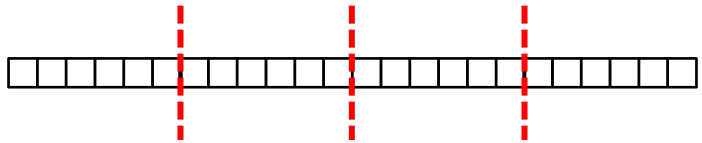

```
var A, B, C: [ProblemSpace] real;

A = B + alpha * C;
```

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

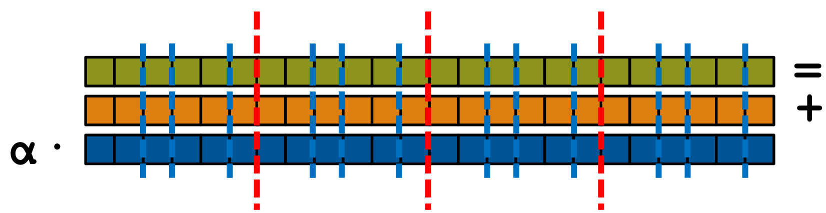
STREAM Triad: Chapel (multilocale, blocked)




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

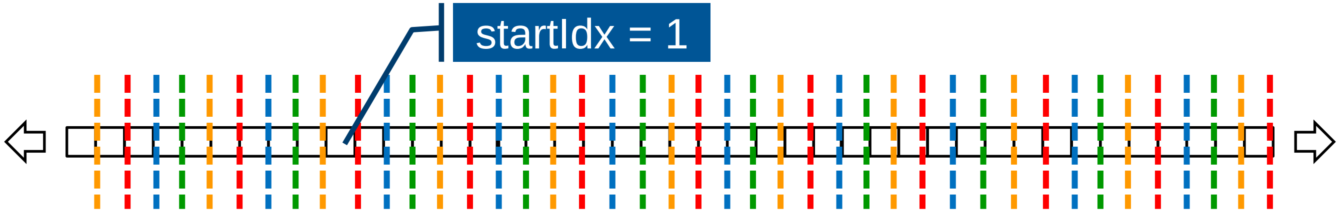


```
var A, B, C: [ProblemSpace] real;
```

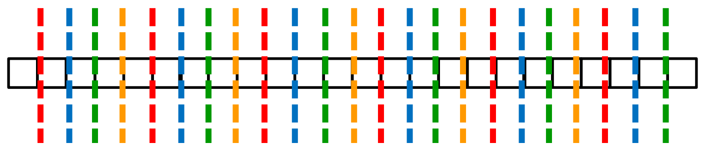



```
A = B + alpha * C;
```

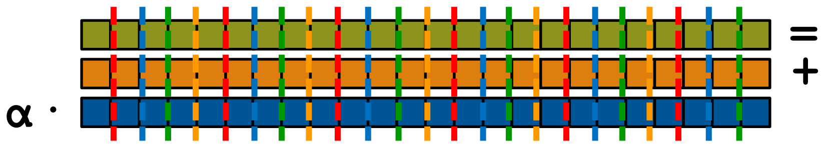
STREAM Triad: Chapel (multilocale, cyclic)




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



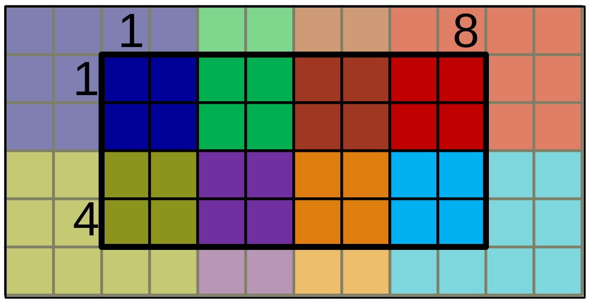
```
var A, B, C: [ProblemSpace] real;
```



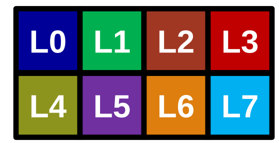

```
A = B + alpha * C;
```

Sample Distributions: Block and Cyclic

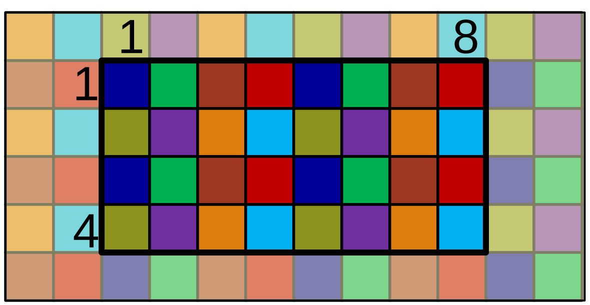
```
var Dom = {1..4, 1..8} dmapped Block( {1..4, 1..8} );
```



distributed to

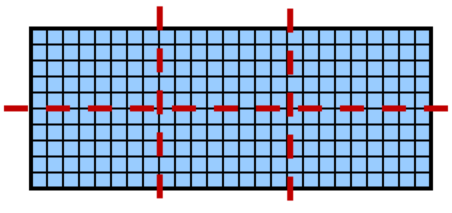



```
var Dom = {1..4, 1..8} dmapped Cyclic( startIdx=(1,1) );
```

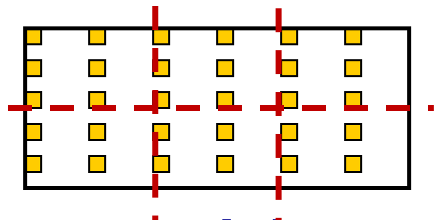


distributed to

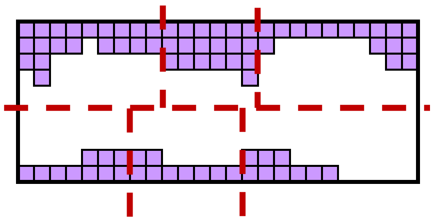
Domain Map Types



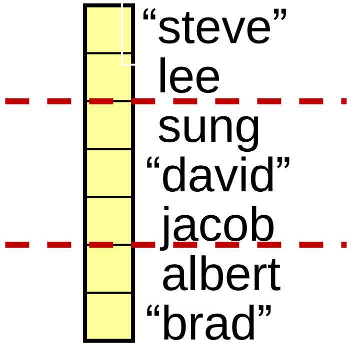
dense



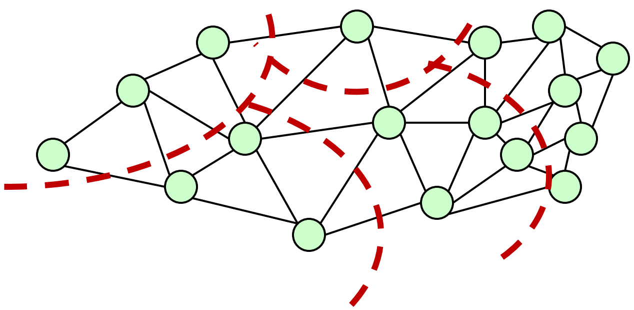
strided



sparse



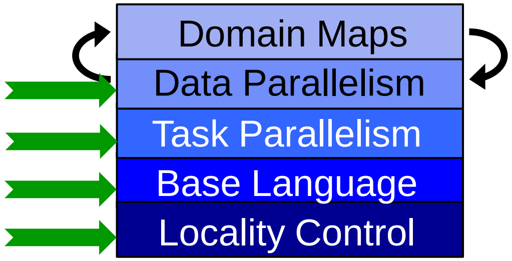
associative



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: Θ(1)

Required Interface:
- create new domains

Domain

Represents: a domain

Generic w.r.t.: index type

State: representation of index set

Typical Size: Θ(1) → Θ(*numIndices*)

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

Array

Represents: an array

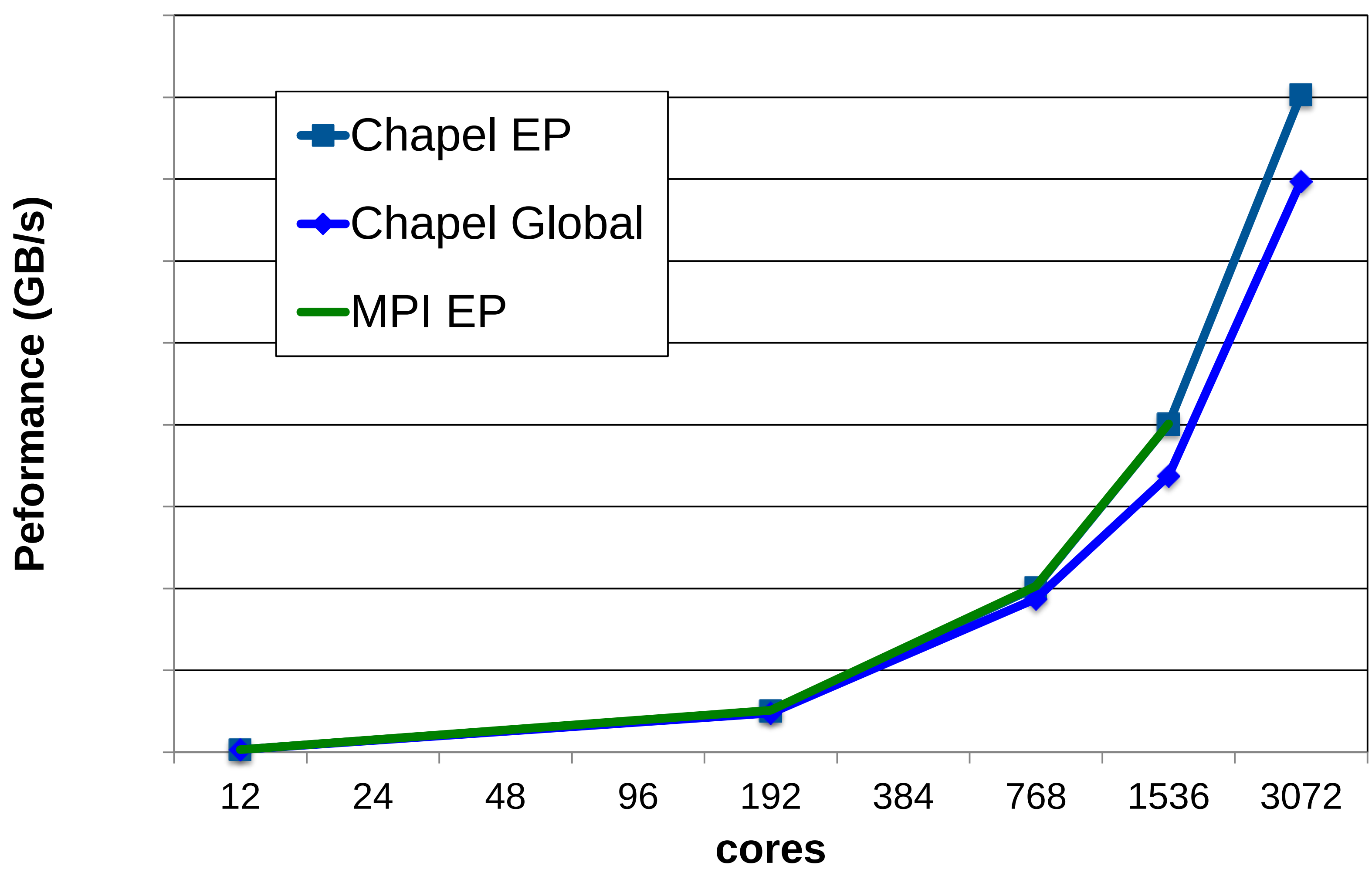
Generic w.r.t.: index type, element type

State: array elements

Typical Size: Θ(*numIndices*)

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

HPCC Stream Performance on Jaguar (XT5)



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, …)

More Data Parallelism Implementation Qs

Q1: How are forall loops implemented?

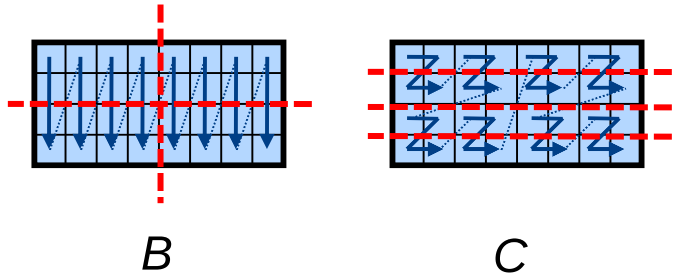
```
forall i in B.domain do B[i] = i/10.0;
```

- 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;
```

- 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;
```

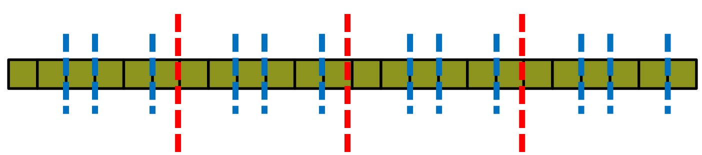
into:

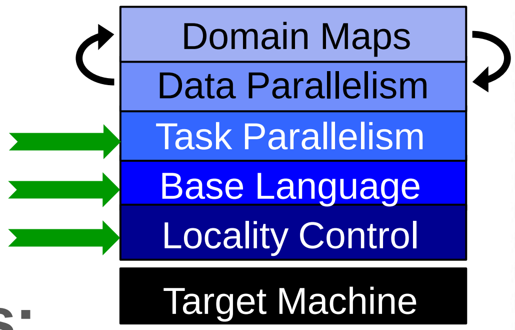
```
inlined A.lead() iterator, which creates tasks that yield work {
  for (a,b,c) in zip(A.follow(work),
                     B.follow(work)
                     C.follow(work)) do
    a = b + alpha * c;
}
```

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);
}
```





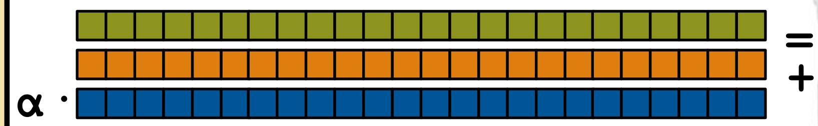
Follower iterators simply use serial features:

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

Leader-Follower Iterators: Rewriting

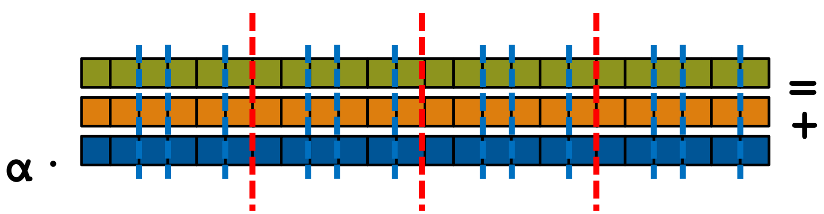
- **Putting it all together, the following loop…**

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



…would get rewritten by the Chapel compiler as:

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



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;
```

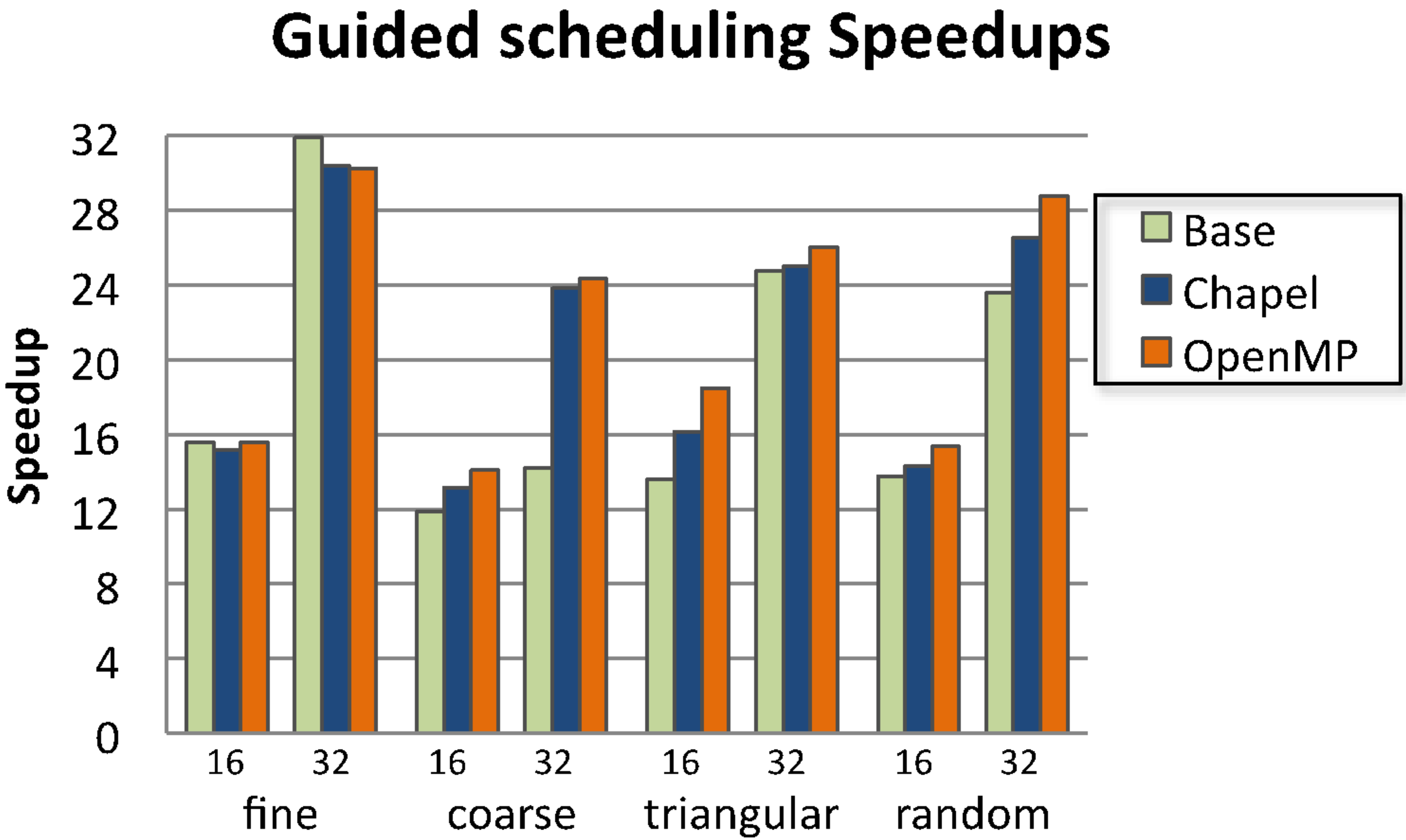
Change the array's default leader by changing its domain map (perhaps to one that you wrote yourself).

Controlling Data Parallelism

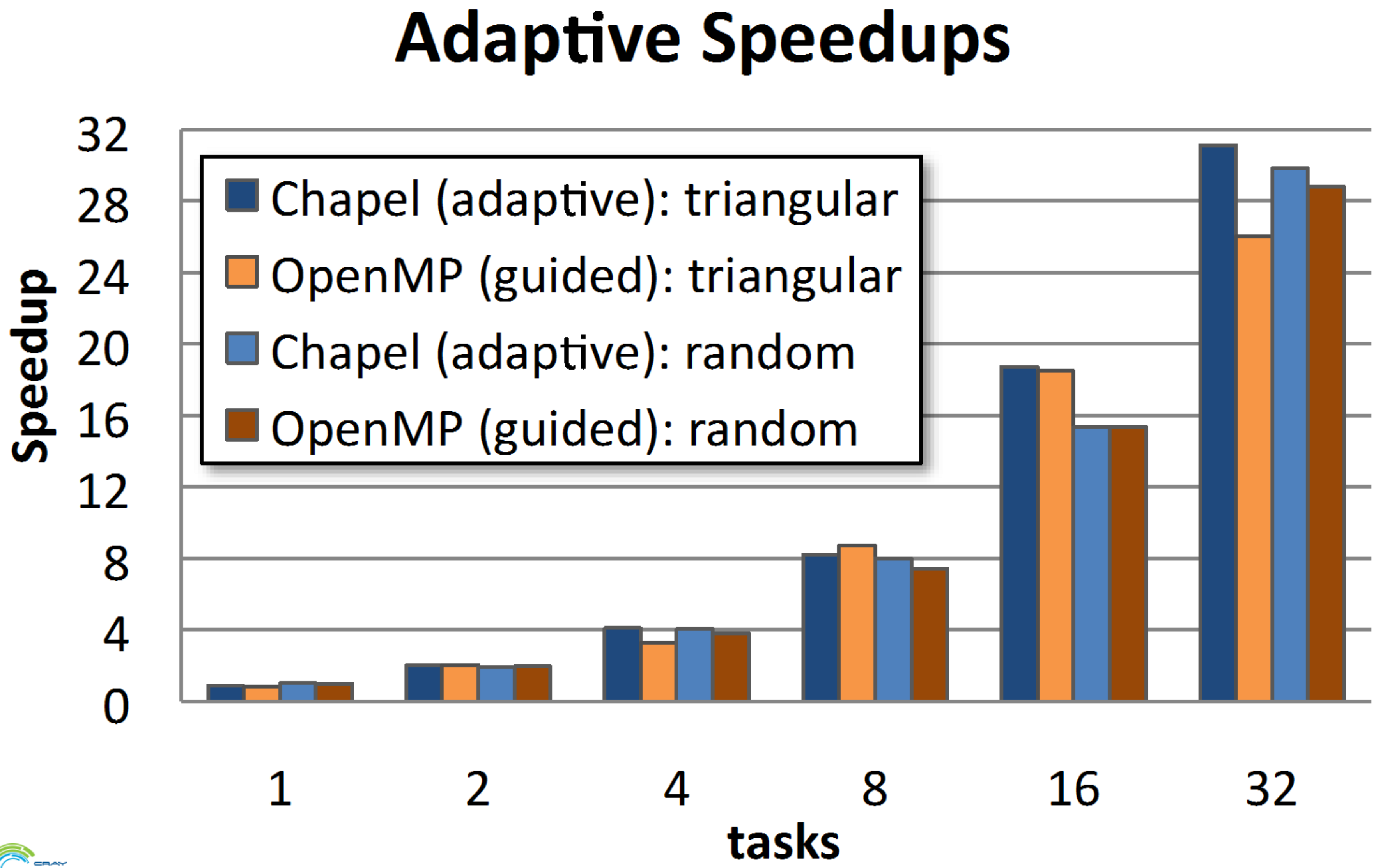
```
forall (a,b,c) in zip(dynamic(A, chunk=64), B, C) do
  a = b + alpha * c;
```

Explicitly invoke a standalone leader iterator (perhaps one that you wrote yourself).

Guided Iteration: Chapel vs. OpenMP



Chapel Adaptive vs. OpenMP Guided



Leader/Follower Experimental Takeaways

Chapel loops can be competitive with OpenMP

- OpenMP's parallel schedules are baked into the language/compiler/runtime
- Chapel's are specified in the language at the user level
  - This permits us to write more advanced iterators like work-stealing

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:
  - *AdvancedIters* section in language specification

Summary of this Domain Maps 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 and Implementation
- ➢ **Project Status and Next Steps**

Implementation Status -- Version 1.7.0 (Apr 2013)

Overall Status:

- Most features work at a functional level
  - some features need to be improved or re-implemented (e.g., OOP)
- 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

- **When 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 possible collaboration areas)

- **Lightweight Tasking using Qthreads:** Sandia (Dylan Stark, et al.)
  - **paper at CUG, May 2011**
- **Application Studies:** LLNL (Rob Neely, Bert Still, Jeff Keasler), Sandia (Richard Barrett, et al.)
- **I/O, regexp, LLVM back-end, etc.:** LTS (Michael Ferguson, et al.)
- **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**
- **Interoperability via Babel/BRAID:** LLNL/Rice (Tom Epperly, Shams Imam, et al.)
  - **paper at PGAS, Oct 2011**
- **Futures/Task-based Parallelism:** Rice (Vivek Sarkar, Shams Imam, Sagnak Tasirlar, et al.)
- **Runtime Communication Optimization:** LBNL (Costin Iancu, et al.)
- **Energy and Resilience:** ORNL (David Bernholdt, et al.)
- **Interfaces/Generics/OOP:** CU Boulder (Jeremy Siek, 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)

(and several others as well…)

Chapel: the next five years

- **Harden Prototype to Production-grade**
  - Performance Optimizations
  - Add/Improve Lacking Features
- **Target more complex/modern compute node types**
  - e.g., CPU+GPU, Intel MIC, …
- **Continue to grow the user and developer communities**
  - including nontraditional circles: desktop parallelism, “big data”
  - transition Chapel from Cray-controlled to community-governed
- **Grow the team at Cray**
  - four positions open at present (manager, SW eng, build/test/release)

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
  - These avoid locking crucial performance decisions into the language

We believe Chapel can greatly improve productivity

…for current and emerging HPC architectures

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

For More Information: Online Resources

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 Aliases:

- chapel_info@cray.com: contact the team at Cray
- chapel-users@lists.sourceforge.net: user-oriented discussion list
- chapel-developers@lists.sourceforge.net: developer discussion
- chapel-education@lists.sourceforge.net: educator discussion
- chapel-bugs@lists.sourceforge.net: public bug forum

For More Information: Suggested Reading

Overview Papers:

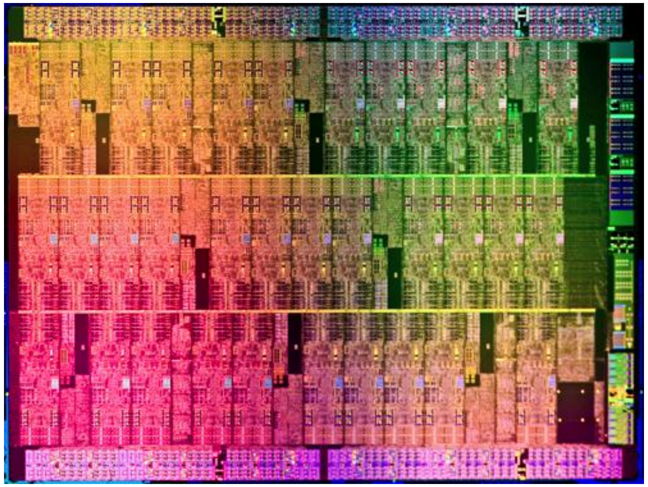
- *The State of the Chapel Union* [slides], Chamberlain, Choi, Dumler, Hildebrandt, Iten, Litvinov, Titus. CUG 2013, May 2013.
  - *a high-level overview of the project summarizing the HPCS period*

- *A Brief Overview of Chapel*, Chamberlain (pre-print of a chapter for *A Brief Overview of Parallel Programming Models*, edited by Pavan Balaji, to be published by MIT Press in 2014).
  - *a more detailed overview of Chapel's history, motivating themes, features*

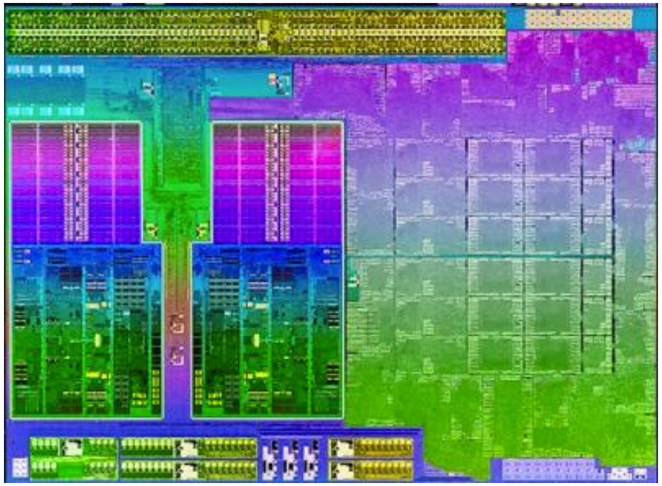
Blog Articles:

- *[Ten] Myths About Scalable Programming Languages*, Chamberlain. IEEE Technical Committee on Scalable Computing (TCSC) Blog, (https://www.ieeetcsc.org/activities/blog/), April-November 2012.
  - *a series of technical opinion pieces designed to combat standard arguments against the development of high-level parallel languages*

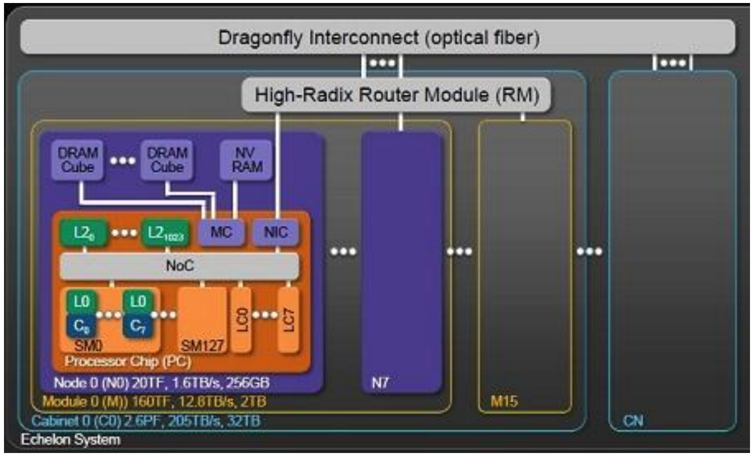
But wait, what about those next-gen processors?



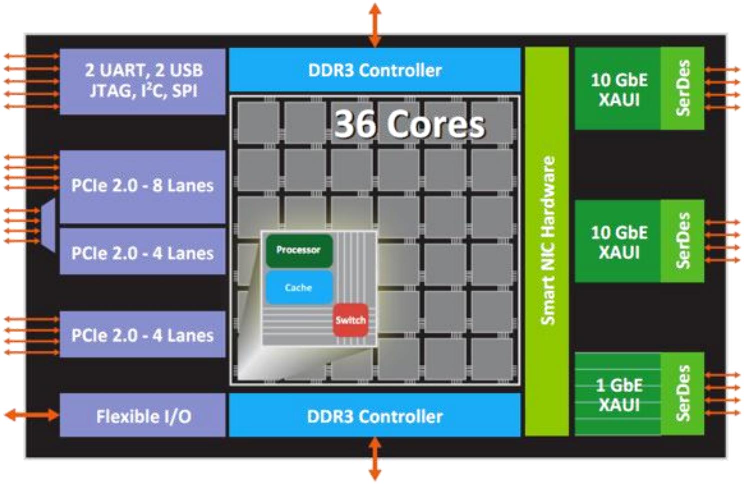
Intel MIC



AMD Trinity



Nvidia Echelon



Tilera Tile-Gx

http://download.intel.com/pressroom/images/Aubrey_Isle_die.jpg http://www.zdnet.com/amds-trinity-processors-take-on-intels-ivy-bridge-3040155225/
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

Next-Gen Scorecard for HPC Programming Models

	Fortran	C/C++	MPI	OpenMP	UPC
performance					
portability (to next-gen)					
programmability					
data parallelism					
task parallelism					
nested parallelism					
locality control					
resilience					
energy-awareness					
user-extensibility					

Next-Gen Scorecard for HPC Programming Models

	Fortran	C/C++	MPI	OpenMP	UPC
performance	✓	✓	✓	✓	~
portability (to next-gen)					
programmability					
data parallelism					
task parallelism					
nested parallelism					
locality control					
resilience					
energy-awareness					
user-extensibility					

Next-Gen Scorecard for HPC Programming Models

	Fortran	C/C++	MPI	OpenMP	UPC
performance	✓	✓	✓	✓	~
portability (to next-gen)	✓	✓			
programmability					
data parallelism					
task parallelism					
nested parallelism					
locality control					
resilience					
energy-awareness					
user-extensibility					

Next-Gen Scorecard for HPC Programming Models

	Fortran	C/C++	MPI	OpenMP	UPC
performance	✓	✓	✓	✓	~
portability (to next-gen)	✓	✓	~	~	~
programmability	X	X	X	~	X
data parallelism	~	X	X	~	~
task parallelism	X	X	X	~	X
nested parallelism	X	X	X	~	X
locality control	X	X	~	X	~
resilience	X	X	~	X	X
energy-awareness	X	X	X	X	X
user-extensibility	X	X	X	X	X

Chapel: Well-Positioned for Next-Gen

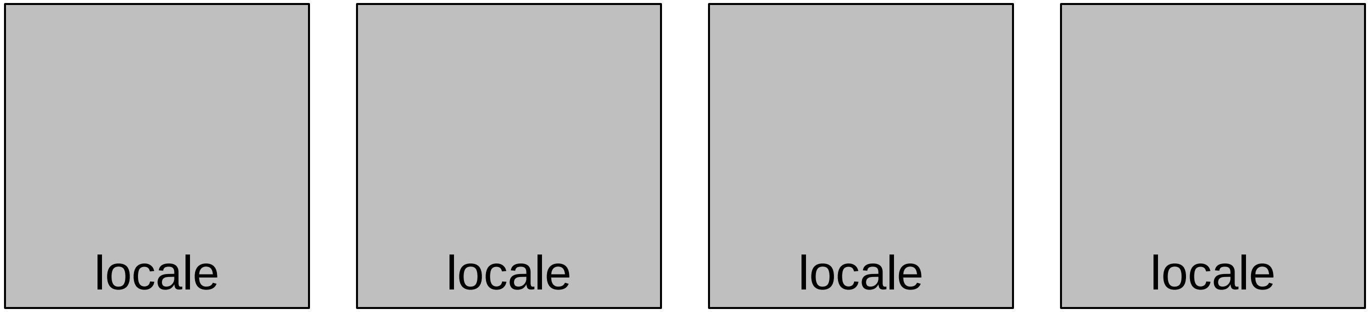
performance	~
portability (to next-gen)	~*
programmability	✓
data parallelism	✓
task parallelism	✓
nested parallelism	✓
locality control	~*
resilience	X
energy-awareness	X
user-extensibility	✓

* (The work in this section is designed to address these items)

Locales Today

Concept:

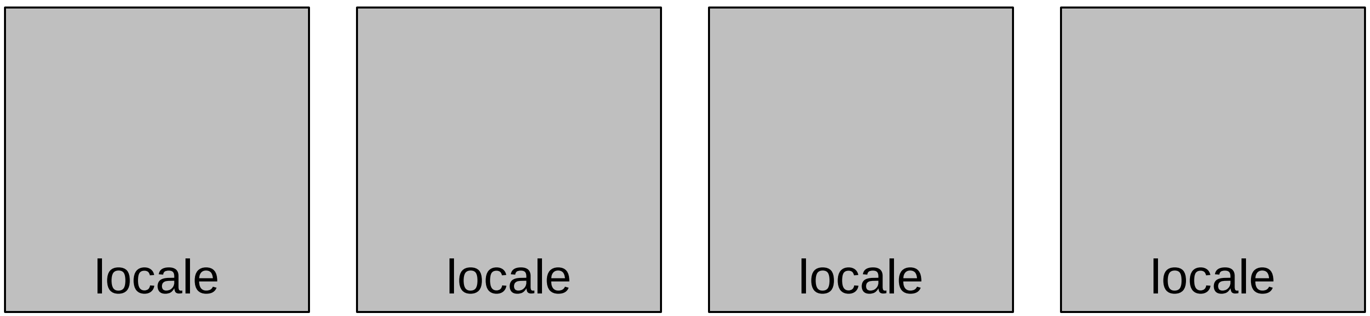
- Today, Chapel supports a 1D array of locales
  - users can reshape/slice to suit their computation's needs



Locales Today

Concept:

- Today, Chapel supports a 1D array of locales
  - users can reshape/slice to suit their computation's needs

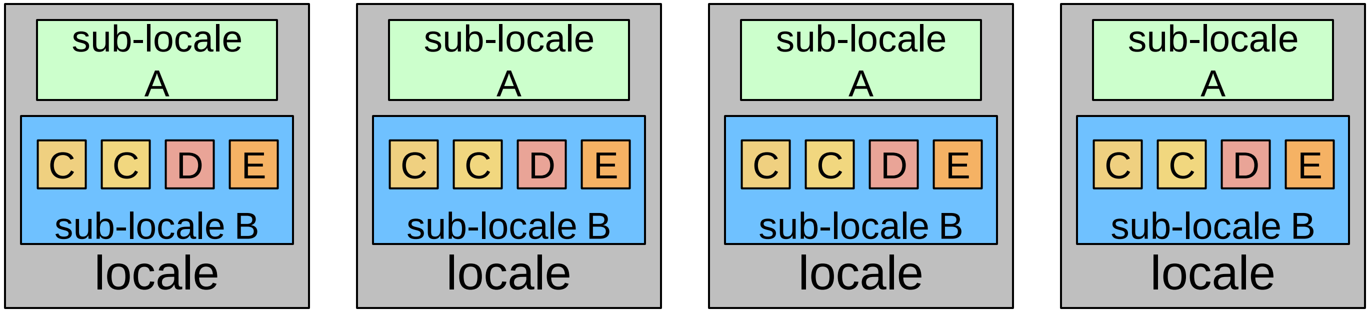


- Apart from queries, no further visibility into locales
  - no mechanism to refer to specific NUMA domains, processors, memories, …
  - assumption: compiler, runtime, OS, HW can handle intra-locale concerns
- Supports horizontal (inter-node) locality well
  - but not vertical (intra-node)

Current Work: Hierarchical Locales

Concept:

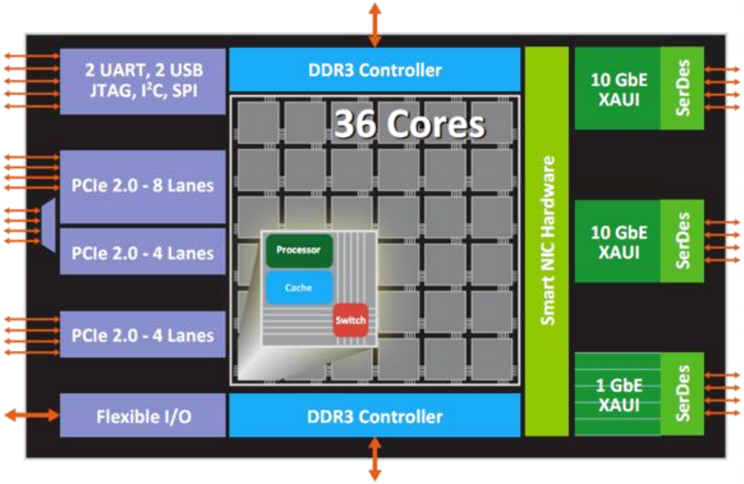
- Support locales within locales to describe architectural sub-structures within a node



- As with traditional locales, *on-clauses* and *domain maps* will be used to map tasks and variables to a sub-locale's memory and processors
- Locale structure is defined using Chapel code
  - permits architectural descriptions to be specified in-language
  - continues the multiresolution philosophy
  - introduces a new Chapel role: *architectural modeler*

Sublocales: Tiled Processor Example

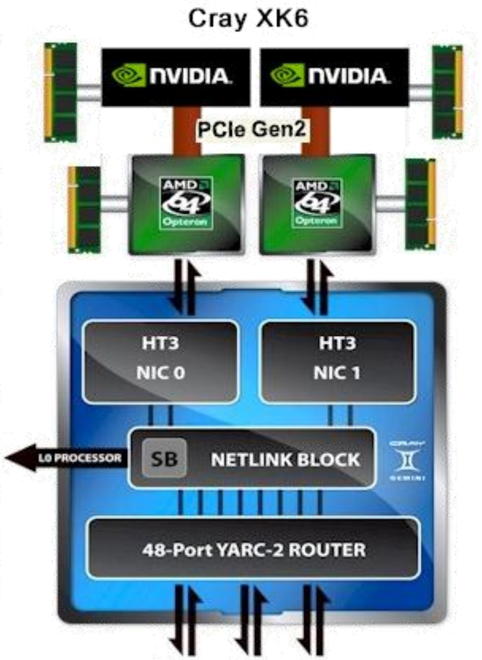
```
class locale: AbstractLocale {
  const xt = 6, yt = xTiles;
  const sublocGrid: [0..#xt, 0..#yt] tiledLoc = …;
  …memory interface…
  …tasking interface…
}

class tiledLoc: AbstractLocale {
  …memory interface…
  …tasking interface…
}
```



Tilera Tile-Gx

Sublocales: Hybrid Processor Example

```
class locale: AbstractLocale {
  const numCPUs = 2, numGPUs = 2;
  const cpus: [0..#numCPUs] cpuLoc = …;
  const gpus: [0..#numGPUs] gpuLoc = …;
  …memory interface…
  …tasking interface…
}

class cpuLoc: AbstractLocale { … }

class gpuLoc: AbstractLocale {
  …sublocales for different
     memory types, thread blocks…?
  …memory, tasking interfaces…
}
```



Sample tasking/memory interface

Memory Interface:

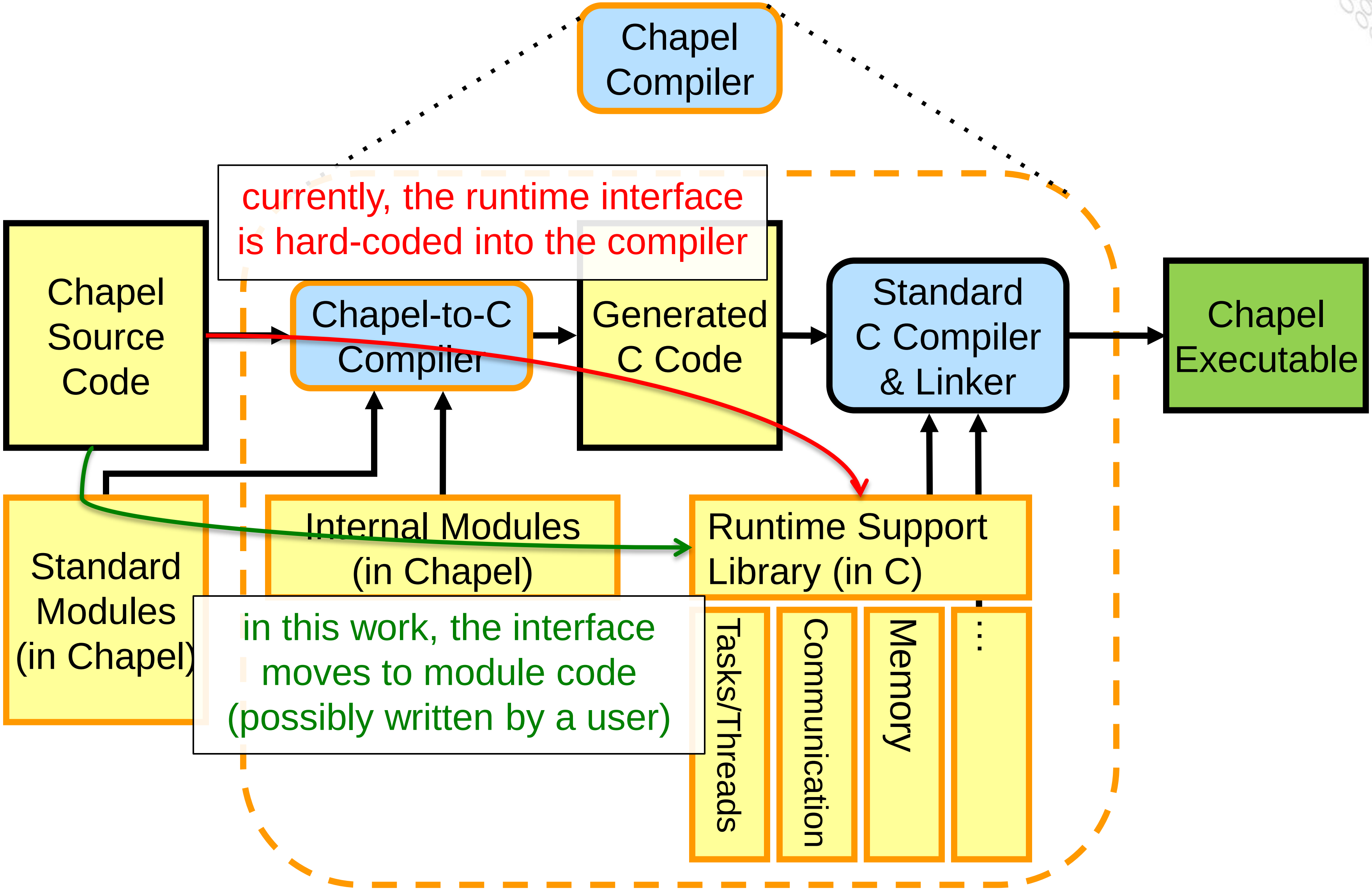
```
proc AbstractLocale.malloc(size_t size) { … }
proc AbstractLocale.realloc(size_t size) { … }
proc AbstractLocale.free(size_t size) { … }
…
```

Tasking Interface:

```
proc AbstractLocale.taskBegin(…) { … }
proc AbstractLocale.tasksCobegin(…) { … }
proc AbstractLocale.tasksCoforall(…) { … }
…
```

In practice, we expect the guts of these to typically be implemented via calls out to external C routines

Chapel Compiler Architecture



Policy Questions

Memory Policy Questions:

- If a sublocale is out of memory, what happens?
  - out-of-memory error?
  - allocate elsewhere? sibling? parent? somewhere else? (on-node v. off?)
- What happens on locales with no memory?
  - illegal? allocate on sublocale? somewhere else?

Tasking Policy Questions:

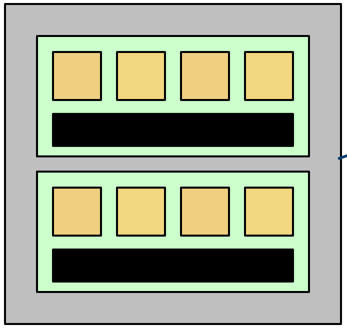
- Can a task that's placed on a specific sublocale migrate?
  - to where? sibling? parent? somewhere else?
- What happens on locales with no processors?
  - illegal? allocate on sublocale? parent locale?
  - using what heuristic? sublocale[0]? round-robin? dynamic load balance?

Goal: Any of these policies should be possible

Tasking Policy Example

Q: What happens to tasks on locales with no (direct) processors?

- e.g., a locale that serves as a container for other sublocales



on “multicore NUMA Node” do begin foo()

Tasking Policy Example

Q: What happens to tasks on locales with no (direct) processors?

e.g., a locale that serves as a container for other sublocales

A1: Run on a fixed or arbitrary sublocale?

```
proc NUMANode.taskBegin(…) {
  numaDomain[0].taskBegin(…);
}
```

Tasking Policy Example

Q: What happens to tasks on locales with no (direct) processors?

e.g., a locale that serves as a container for other sublocales

A2: Schedule round-robin?

```
proc NUMANode.taskBegin(…) {
  const subloc = (nextSubLoc.fetchAdd(1))%numSubLocs;
  numaDomain[subloc].taskBegin(…);
}

class NUMANode {
  …
  var nextSubLoc: atomic int;
  …
}
```

Tasking Policy Example

Q: What happens to tasks on locales with no (direct) processors?

e.g., a locale that serves as a container for other sublocales

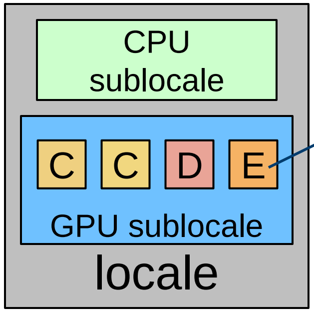
A3: Dynamically Load Balance?

```
proc NUMANode.taskBegin(…) {
  numaDomain[getBestSubLoc()].taskBegin(…);
}

proc NUMANode.getBestSubLoc() {
  const (numTasks, subloc)
          = minloc reduce (numaDomain.numTasks(),
                           0..#numSubLocs);
  return subloc;
}
```

Another Tasking Policy Example

Q: What happens to tasks on locales with no processors?

e.g., a sublocale representing a memory resource



on “Texture Memory” do begin foo()

Another Tasking Policy Example

Q: What happens to tasks on locales with no processors?

e.g., a sublocale representing a memory resource

A1: Throw an error?

```
proc TextureMemLocale.taskBegin(…) {
  halt("You can't run tasks on texture memory!");
}
```

Downside: potential user inconvenience:

```
on Locales[2].gpuLoc.texMem do var X: [1..n, 1..n] int;
on X[i,j] do begin refine(X);
```

Another Tasking Policy Example

Q: What happens to tasks on locales with no processors?

e.g., a sublocale representing a memory resource

A2: Defer to parent?

```
proc TextureMemLocale.taskBegin(…) {
  parentLocale.taskBegin(…);
}
```

Another Tasking Policy Example

Q: What happens to tasks on locales with no processors?

e.g., a sublocale representing a memory resource

A3: Or perhaps just run directly near memory?

```
proc TextureMemLocale.taskBegin(…) {
  extern proc chpl_task_create_GPU_Task(…);
  chpl_task_create_GPU_Task(…);
}
```

Contrasts with Related Work

Related work:

- Sequoia (Aiken et al., Stanford)
- Hierarchical Place Trees (Sarkar et al., Rice)

Differences:

- Hierarchy only impacts locality, not semantics as in Sequoia
  - analogous to PGAS languages vs. distributed memory
- No restrictions as to what HW must live in what node
  - e.g., no “processors must live in leaf nodes” requirement
- Does not impose a strict abstract tree structure
  - e.g., `const sublocGrid: [0..#xt, 0..#yt] tiledLoc = …;`
- User-specifiable concept
  - convenience of specifying within Chapel
  - policies for mapping to HW can be defined in-language

Hierarchical Locales: Design Challenges

Portability: Chapel code that refers to sub-locales can cause problems on systems with a different model

Mitigation Strategies

- Well-designed domain maps should buffer many typical users from these challenges
- We anticipate identifying a few broad classes of locales that characterize broad swaths of machines “well enough”
- More advanced runtime designs and compiler work could help guard most task-parallel users from this level of detail
- Not a Chapel-specific challenge, fortunately

Code Generation: Dealing with targets for which C is not the language of choice (e.g., CUDA)

Summary: Hierarchical Locales

Emerging compute nodes are presenting challenges

Chapel's support for parallelism and locality positions it better than current HPC languages

- Hierarchical locales extend it to support intra-node concerns

Hierarchical Locales have some attractive properties

- Defined in Chapel, potentially by users
- Support user-level policy decisions
- Removes hard-coding of runtime interfaces in compiler

Specification and implementation effort is underway

Longer-term Directions

Represent physical machine as a hierarchical locale and represent user's locales as a *slice* of that hierarchy

- for topology-aware programming
- for jobs with dynamically-changing resource requirements
  - due to changing job needs
  - or failing HW

Combine with containment domains (Erez, UT Austin)

- the two concepts seem well-matched for each other http://chapel.cray.com   chapel_info@cray.com   http://sourceforge.net/projects/chapel/