Chapel:
A Next-Generation PGAS* Language

(*Partitioned Global Address Space)

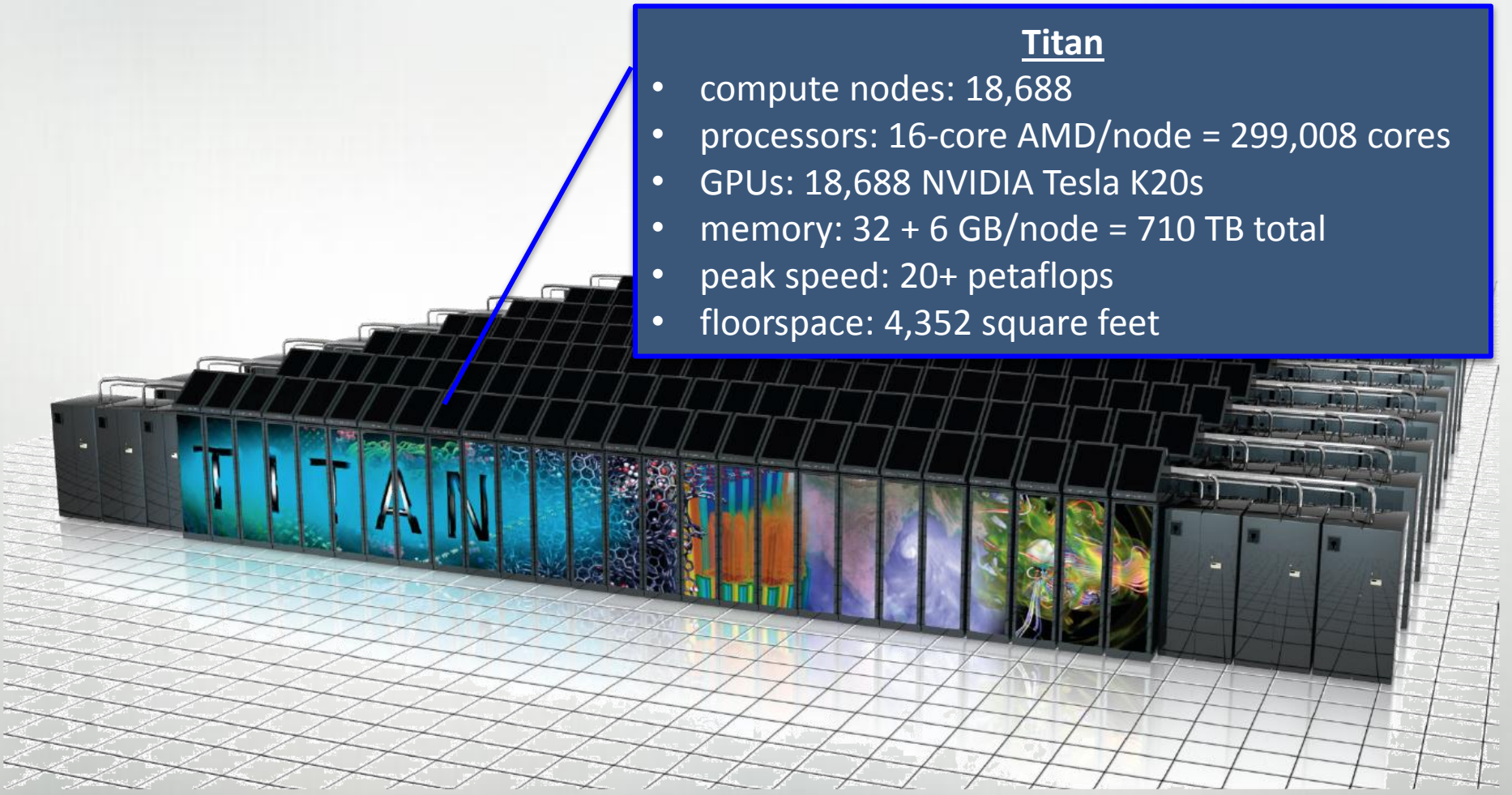
Brad Chamberlain

Applied Mathematics 483/583

May 29th, 2013

My Employer: CRAY THE SUPERCOMPUTER COMPANY

TITAN (Currently #1 on the Top 500)



Titan

- compute nodes: 18,688
- processors: 16-core AMD/node = 299,008 cores
- GPUs: 18,688 NVIDIA Tesla K20s
- memory: 32 + 6 GB/node = 710 TB total
- peak speed: 20+ petaflops
- floorspace: 4,352 square feet

For more information: http://www.olcf.ornl.gov/titan/

Blue Waters: Another large Cray

Blue Waters

- compute nodes: 25,712
- processors: 386,816 AMD cores
- GPUs: 3,072 NVIDIA Kepler GPUs
- memory: 1.476 PB total
- peak speed: 11.61 petaflops

For more information: https://bluewaters.ncsa.illinois.edu/

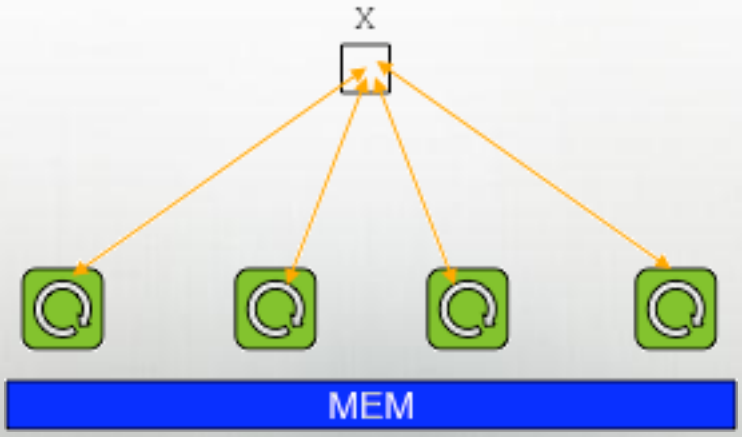
Outline

- ✓ Who is Cray?
- ➢ PGAS Languages
- Chapel and PGAS
- Chapel Motivation
- Chapel Features
- Project Status

Shared Memory Programming Models

e.g., OpenMP, Pthreads

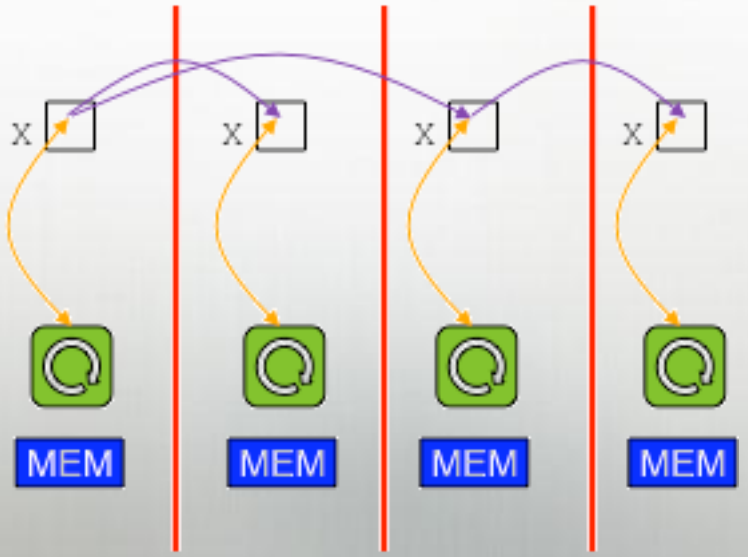
- \+ support dynamic, fine-grain parallelism
- \+ considered simpler, more like traditional programming
  - "if you want to access something, simply name it"
- – no support for expressing locality/affinity; limits scalability
- – bugs can be subtle, difficult to track down (race conditions)
- – tend to require complex memory consistency models



Message Passing Programming Models

e.g., MPI

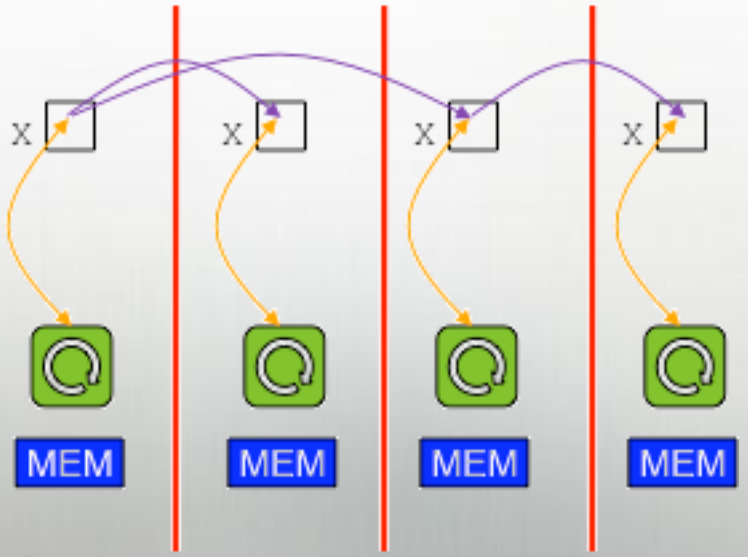
- \+ a more constrained model; can only access local data
- \+ runs on most large-scale parallel platforms
  - and for many of them, can achieve near-optimal performance
- \+ is *relatively* easy to implement
- \+ can serve as a strong foundation for higher-level models
- \+ users have been able to get real work done with it



Message Passing Programming Models

e.g., MPI

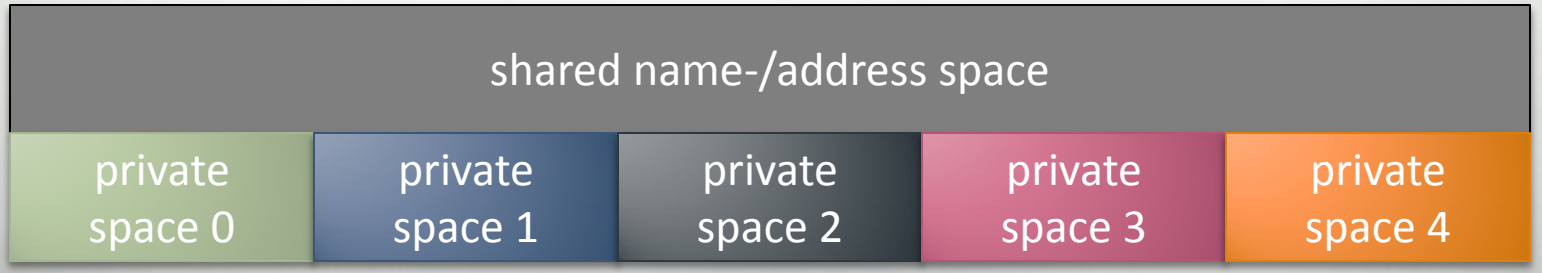
- communication must be used to get copies of remote data
  - tends to reveal too much about *how* to transfer data, not simply *what*
- only supports "cooperating executable"-level parallelism
- couples data transfer and synchronization
- has frustrating classes of bugs of its own
  - e.g., mismatches between sends/recvs, buffer overflows, etc.



Partitioned Global Address Space Languages

(Or perhaps: partitioned global namespace languages)

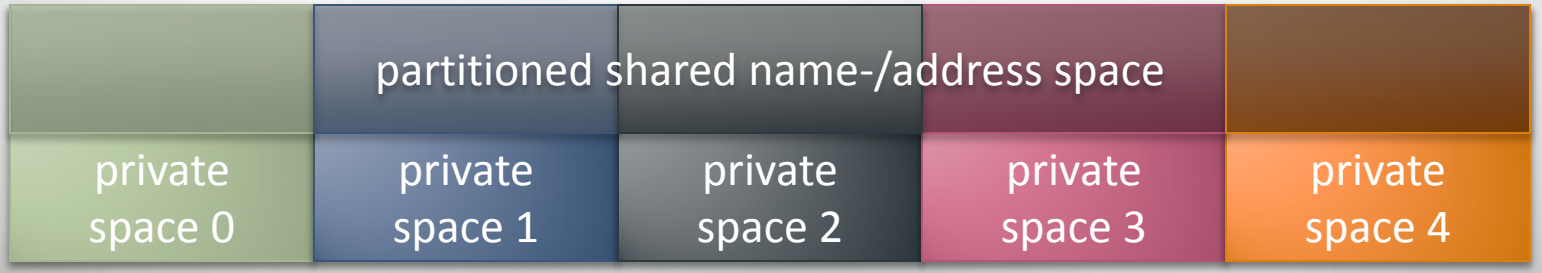
- abstract concept:
  - support a shared namespace on distributed memory
    - permit any parallel task to access any lexically visible variable
    - doesn't matter if it's local or remote



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 also be considered PGAS, but the term hadn't been coined yet

Co-Array Fortran (CAF)

CAF: The first of our "traditional" PGAS languages

- developed ~1994
- adopted into the 2008 Fortran standard

Motivating Philosophy: "What is the smallest change required to convert Fortran 95 into a robust parallel language?"

- originally referred to as F-- to emphasize "smallest change"

CAF is SPMD

- SPMD programming/execution model
  - similar to MPI* in this regard
  - program copies are referred to as 'images'

- Use intrinsic functions to query the basics:

```
integer :: p, me
p = num_images()   ! returns number of processes
me = this_image()  ! returns value in 1..num_images()
```

- Barrier sync:

```
sync_all()          ! wait for all processes/images
```

*= typical uses of it, anyway

Main CAF Concept: Co-Dimensions

Co-Dimension: an array dimension that refers to the space of CAF *images* (processes)

- defined using square brackets

  (distinguishes it syntactically from a traditional dimension)

Co-array variables:

```
integer i[*]          ! declares an integer, i, per image
real x[*]             ! declares a float, x, per image
real a(20)[*]         ! declares a 20-element array per image
real b(N,N)[*]        ! declares an N x N array per image
```

Main CAF Concept: Co-Dimensions

Co-array variables:

```
integer i[*]        ! declares an integer, i, per image
```

- Of course, traditional variables also result in a copy per image (it's SPMD after all), but *private* to that image

```
integer j           ! declares a private integer, j, per image
```

Using Co-Arrays

```
integer i[*]
real x[*]
```

- Refer to other images' values via co-array indexing:

```
if (me == 2) then
  nextX = x[me+1]     ! read neighbor's value of x
  i[1] = i            ! copy my value of 'i' into image 1's
endif
```

- Co-array indexing/square brackets ⇒ communication

CAF Summary

- Program in SPMD style
- Communicate via variables with co-dimensions
  - a copy per program image
  - refer to other images' copies via square bracket subscripts
  - take advantage of good multidimensional array support
    - multidimensional views of process grid
    - multidimensional views of local data
    - syntactic support for slicing (:)
- Other stuff too, but this gives you the main idea
- Adopted into Fortran 2008 standard
  - see also http://www.co-array.org

UPC: Unified Parallel C

UPC: Our second "traditional" PGAS language

- developed ~1999
- "unified" in the sense that it combined 3 distinct parallel C's:
  - AC, Split-C, Parallel C Preprocessor
- though a sibling to CAF, philosophically quite different

Motivating Philosophy:

- extend C concepts logically to support SPMD execution
  - 1D arrays
  - for loops
  - pointers (and pointer/array equivalence)

UPC is also SPMD

- SPMD programming/execution model
  - program copies are referred to as ‘threads’
- Built-in constants provide the basics:

```
int p, me;
p = THREADS;     // returns number of processes
me = MYTHREAD;   // returns a value in 0..THREADS-1
```

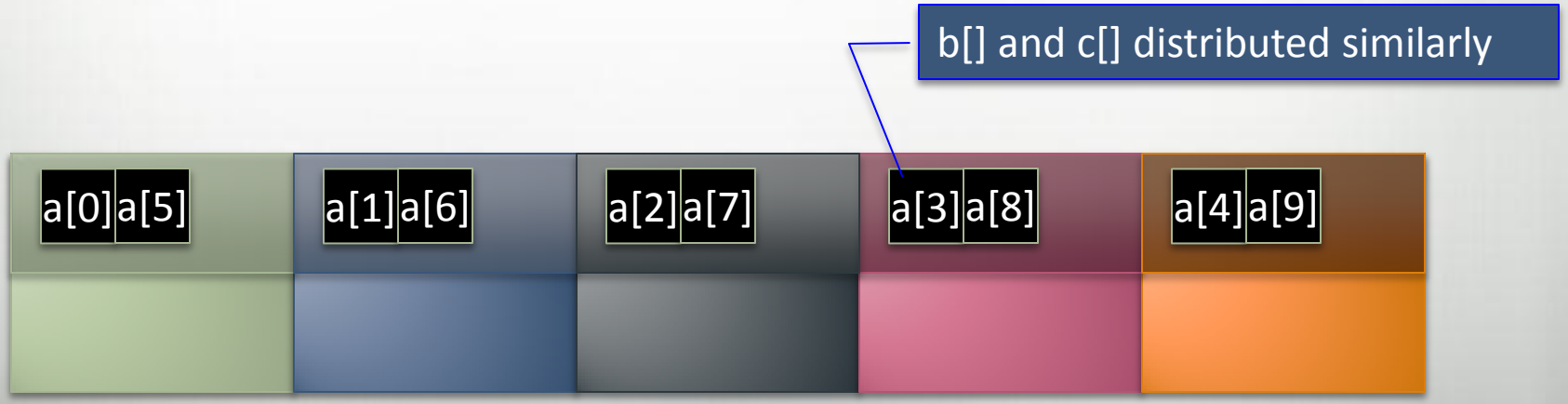
- Barrier synch statement:

```
upc_barrier;     // wait for all processes/threads
```

Distributed Arrays in UPC

- Arrays declared with the ‘shared’ keyword are distributed within the shared space
  - uses a cyclic distribution by default

```
#define N 10
shared float a[N], b[N], c[N];
```

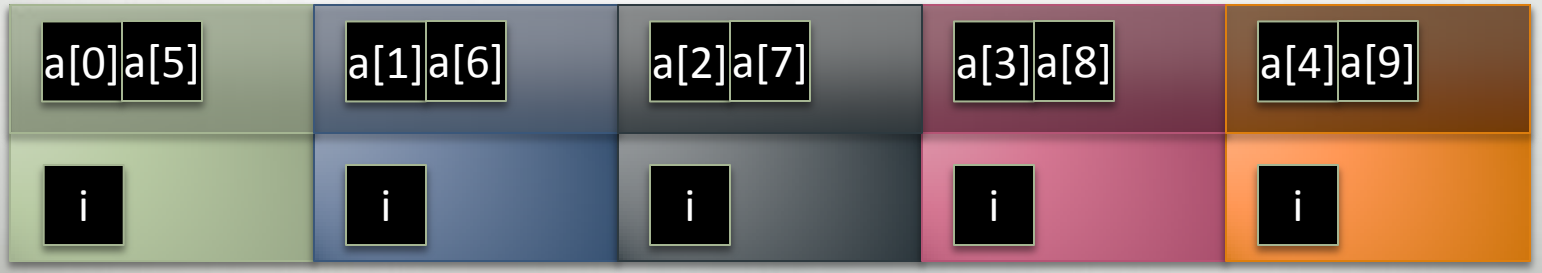


Distributed Arrays in UPC

- Arrays declared with the 'shared' keyword are distributed within the shared space
  - uses a cyclic distribution by default

```
#define N 10
shared float a[N], b[N], c[N];
upc_forall (int i=0; i<N; i++; i) {
  c[i] = a[i] + alpha * b[i];
}
```

Affinity field: Which thread should execute this iteration? (if int, %THREADS to get ID)

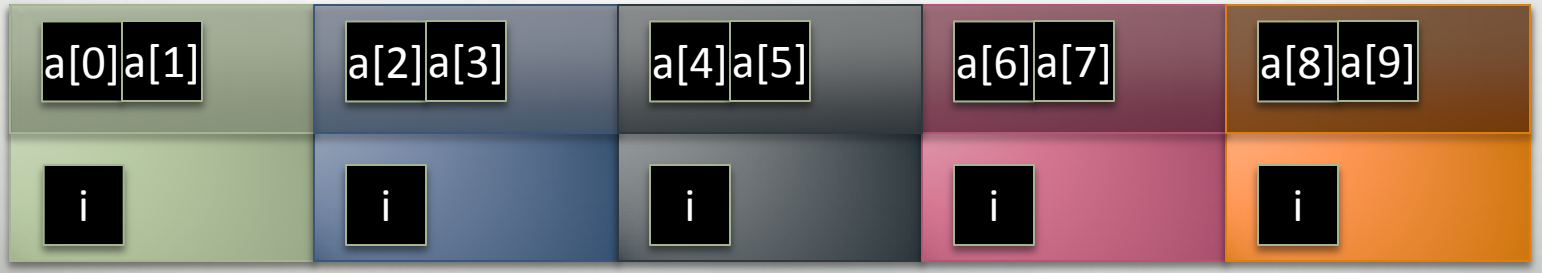


Distributed Arrays in UPC

- Arrays declared with the 'shared' keyword are distributed within the shared space
  - can specify a block-cyclic distribution as well

```
#define N 10
shared [2] float a[N], b[N], c[N];
upc_forall (int i=0; i<N; i++; &c[i]) {
  c[i] = a[i] + alpha * b[i];
}
```

Affinity field: Which thread should execute this iteration? (if ptr-to-shared, owner does)

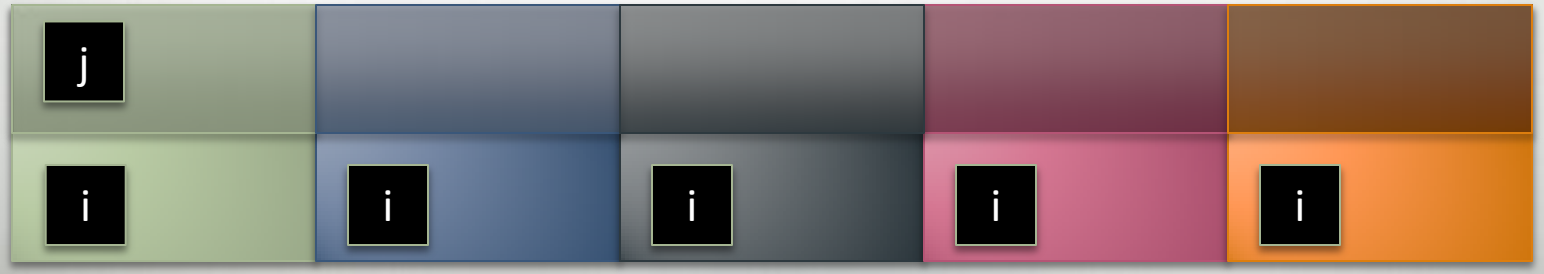


Scalars in UPC

- Somewhat confusingly (to me anyway*), shared scalars in UPC result in a single copy on thread 0

```
int i;
shared int j;
```

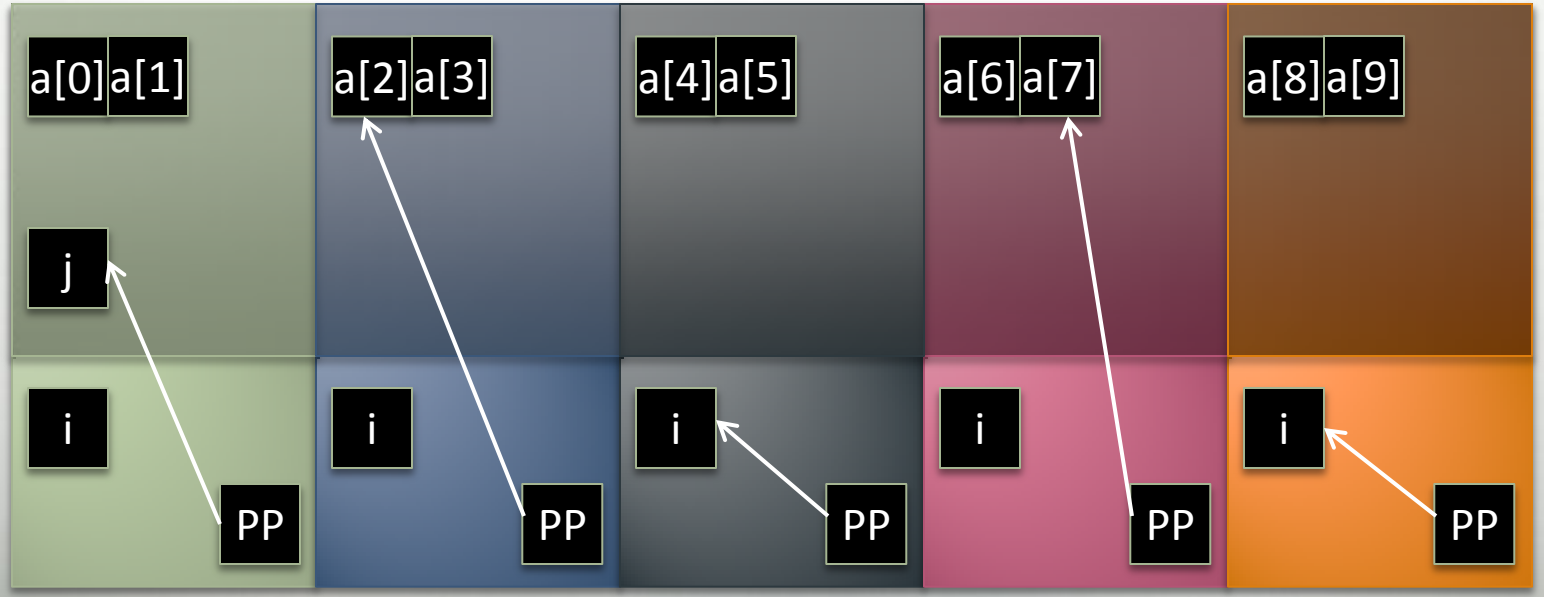
* = because it seems contrary to SPMD programming



Pointers in UPC

- UPC Pointers may be private/shared and may point to private/shared

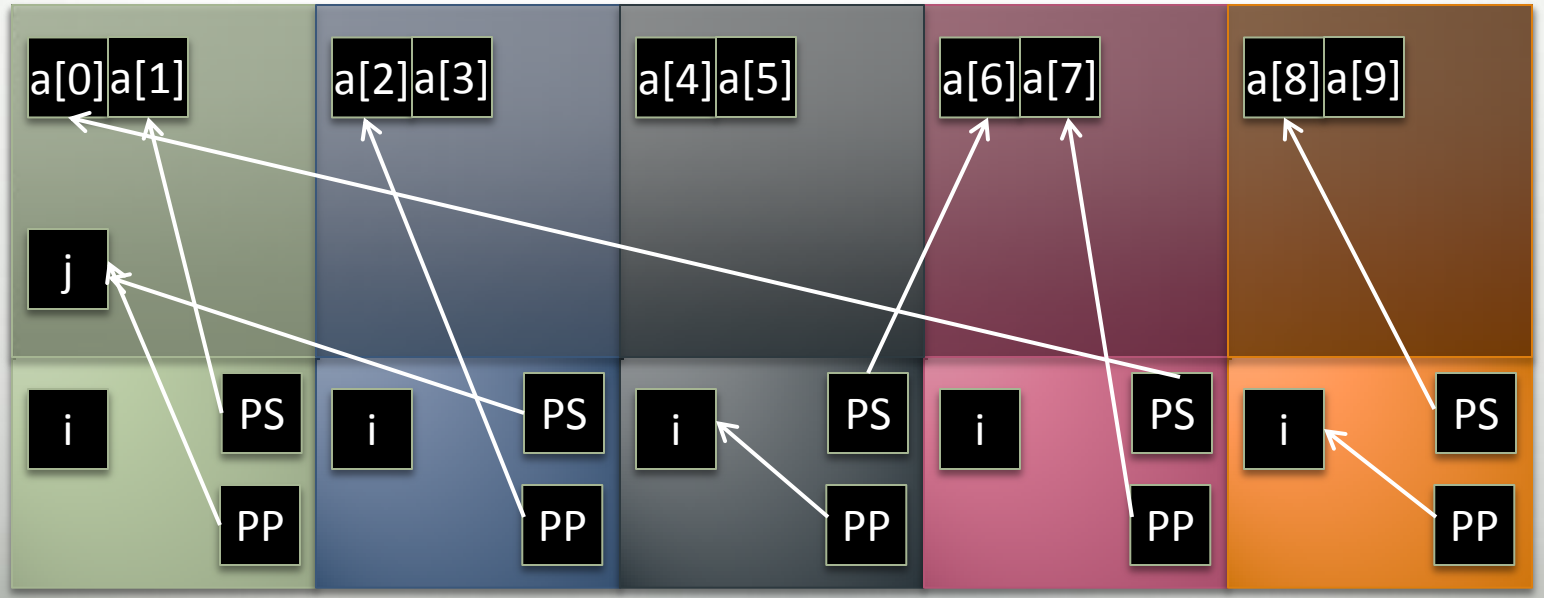
```
int* PP; // private pointer to local data
```



Pointers in UPC

- UPC Pointers may be private/shared and may point to private/shared

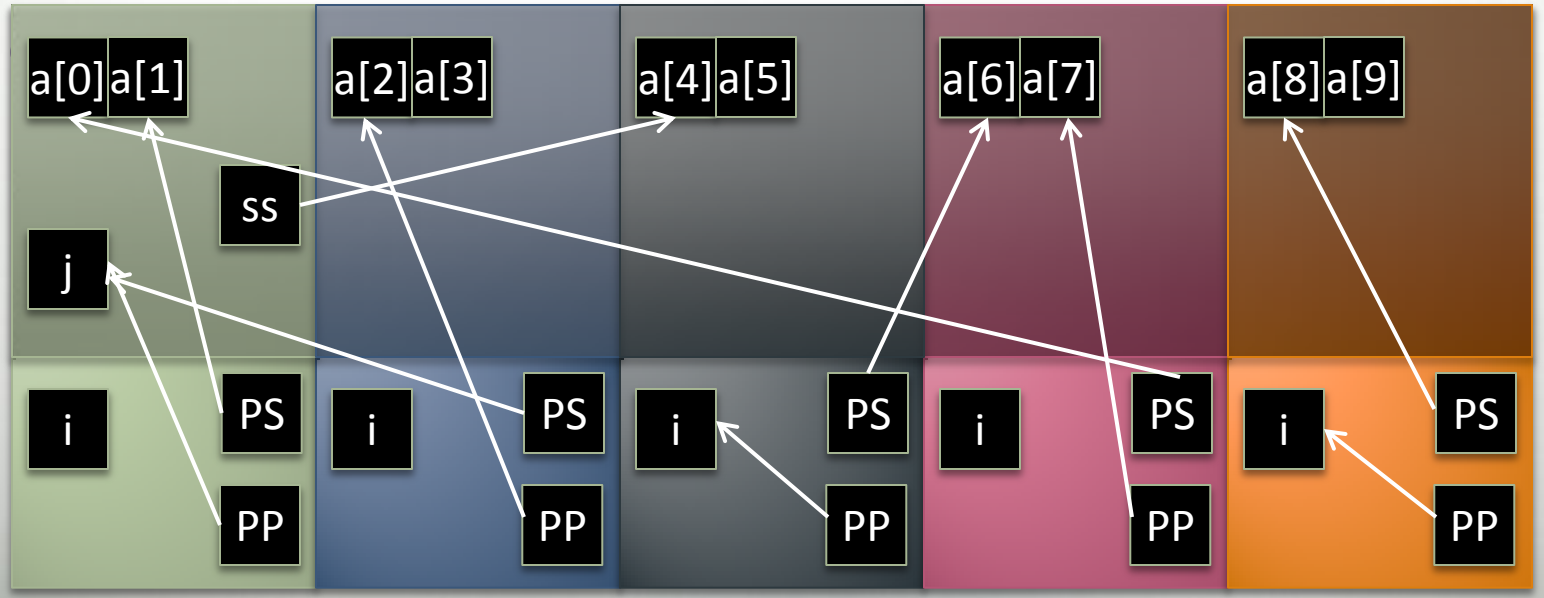
```
int* PP; // private pointer to local data
shared int* PS; // private pointer to shared data
```



Pointers in UPC

- UPC Pointers may be private/shared and may point to private/shared

```
int* PP;  // private pointer to local data
shared int* PS;  // private pointer to shared data
shared int* shared ss;  // shared pointer to shared
```



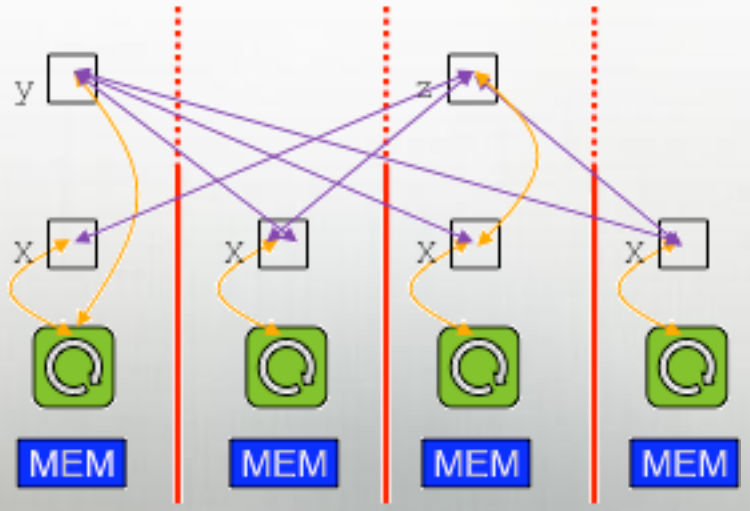
UPC Summary

- Program in SPMD style
- Communicate via shared arrays/pointers
  - cyclic and block-cyclic arrays
  - pointers to shared and private data
  - array-pointer equivalence
- Other stuff too, but this gives you the main idea
- For more information, see https://upc-lang.org/upc/

Traditional PGAS Models

e.g., Co-Array Fortran, UPC

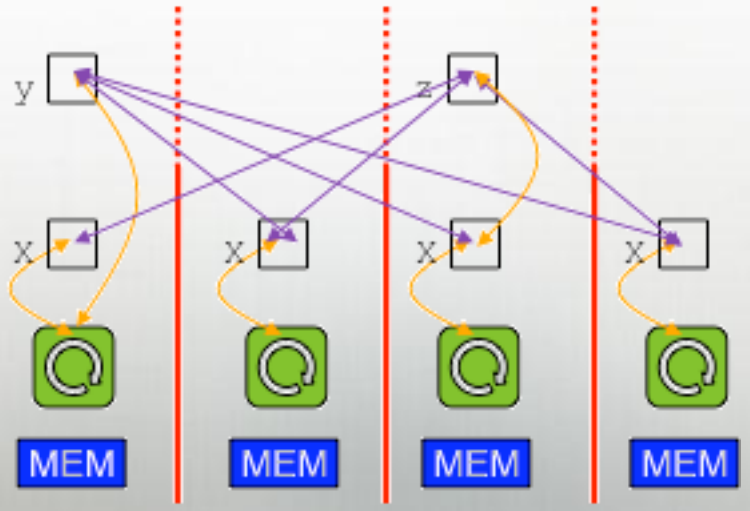
- + support a shared namespace, like shared-memory
- + support a strong sense of ownership and locality
  - each variable is stored in a particular memory segment
  - tasks can access any visible variable, local or remote
  - local variables are cheaper to access than remote ones
- + implicit communication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



Traditional PGAS Models

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



Outline

- ✓ Who is Cray?
- ✓ PGAS Languages
- ➢ Chapel and PGAS
- Chapel Motivation
- Chapel Features
- 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

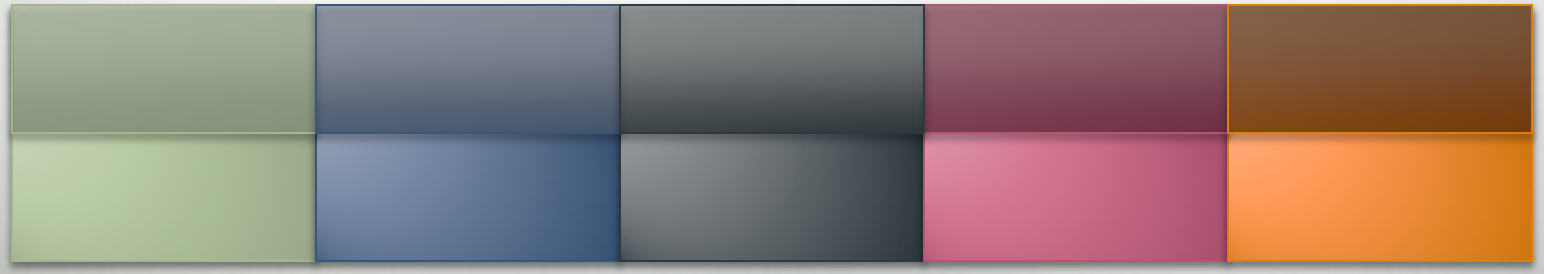
- A work-in-progress

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, …

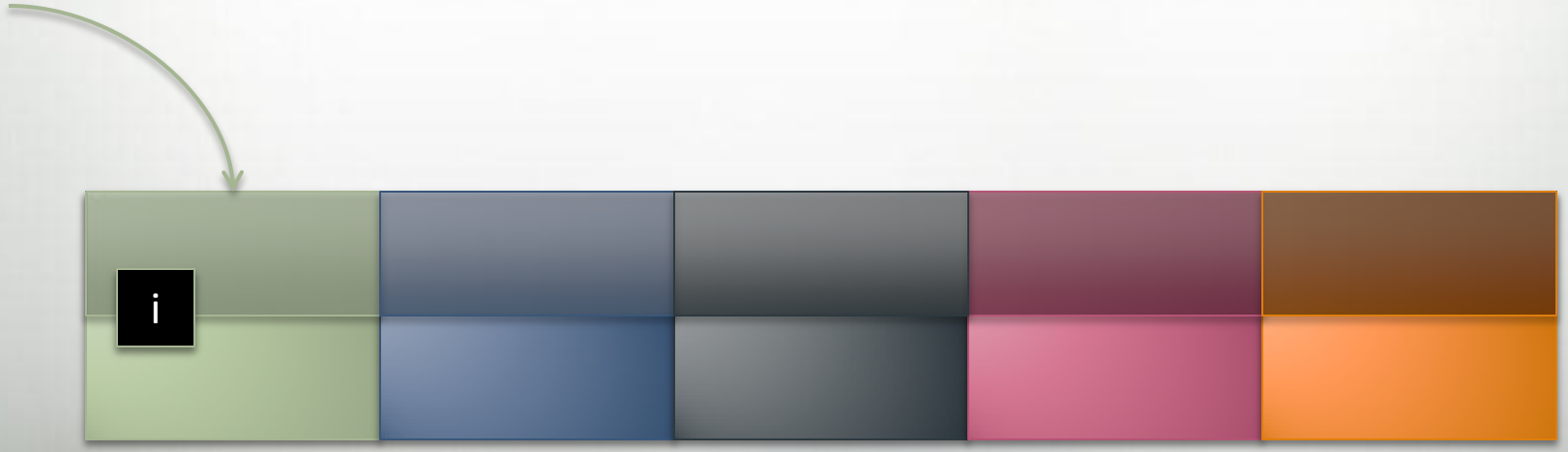
Chapel and PGAS

- Chapel differs from UPC/CAF because it's not SPMD
  - ⇒ "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
  - ⇒ user-level concept of locality is central to language
    - parallelism and locality are two distinct things
    - should never think in terms of "that other copy of the program"



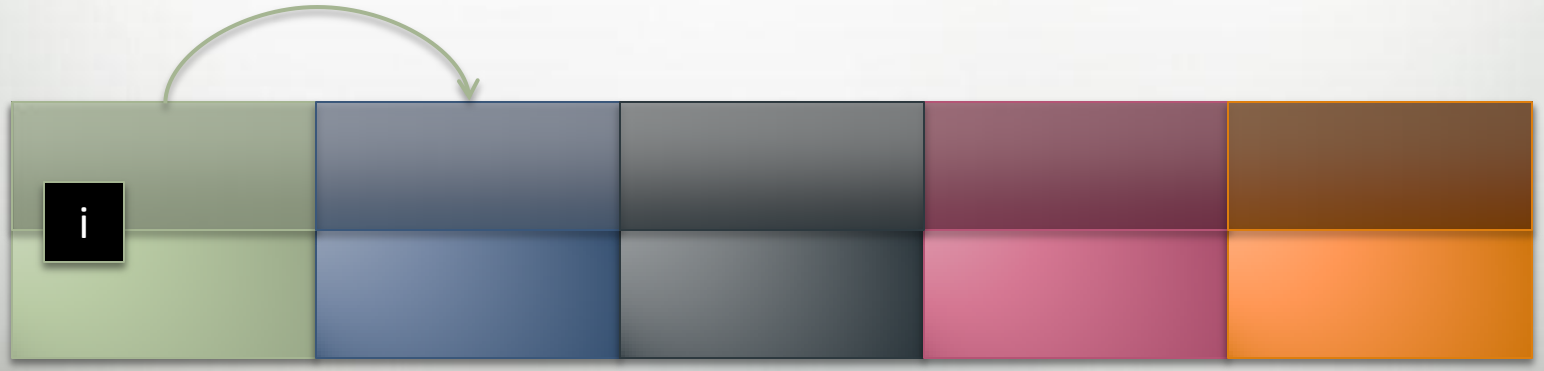
Chapel and PGAS

```
var i: int;
```



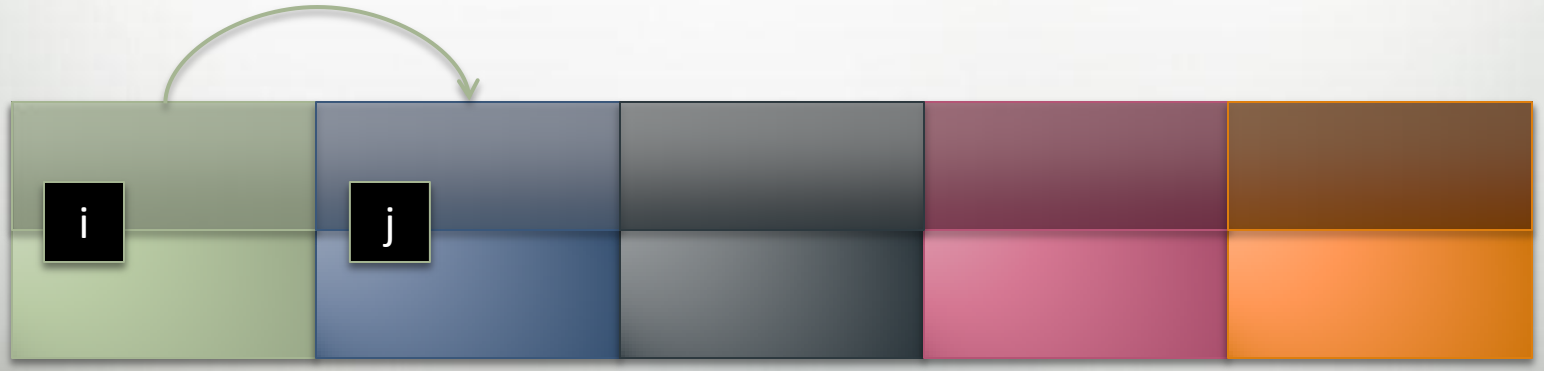
Chapel and PGAS

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



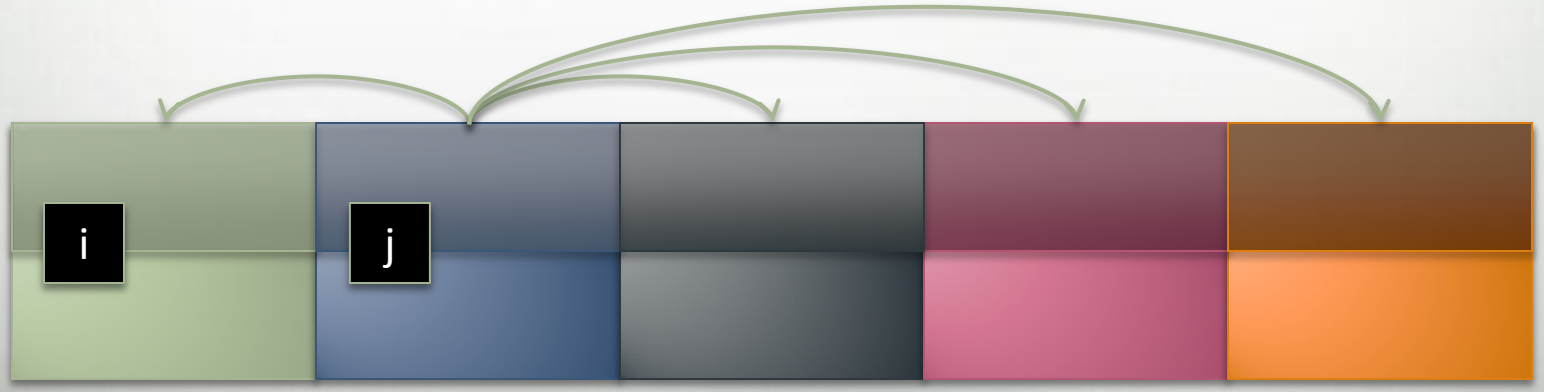
Chapel and PGAS

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



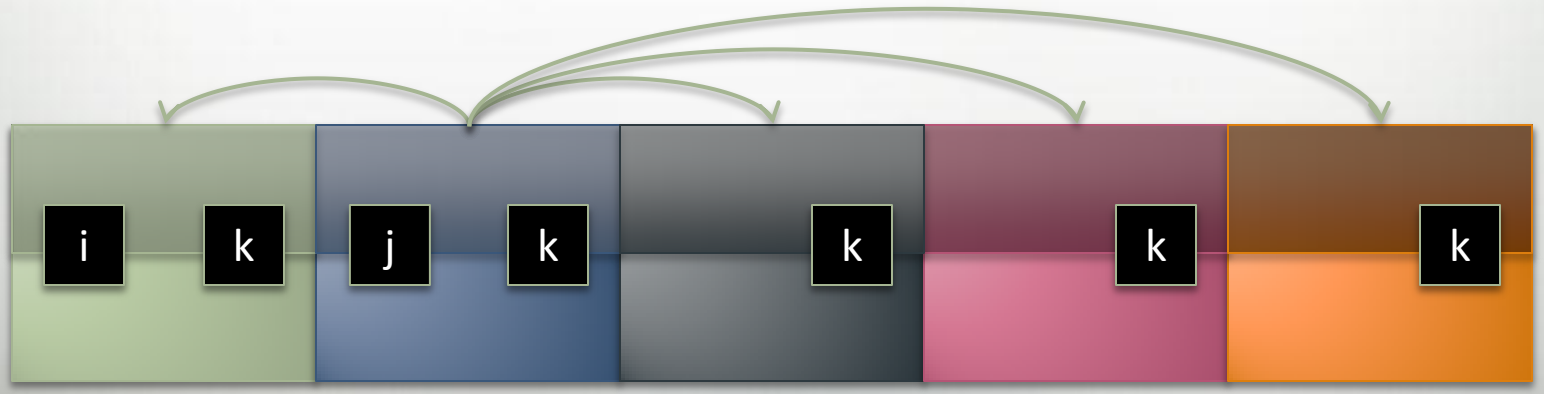
Chapel and PGAS

```
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;
    }
  }
}
```

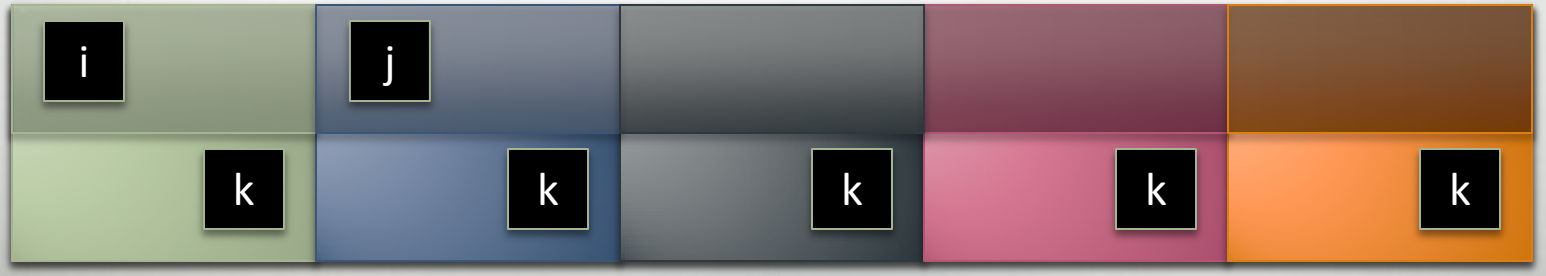


Chapel and PGAS: Public vs. Private

How public a variable is depends only on scoping

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

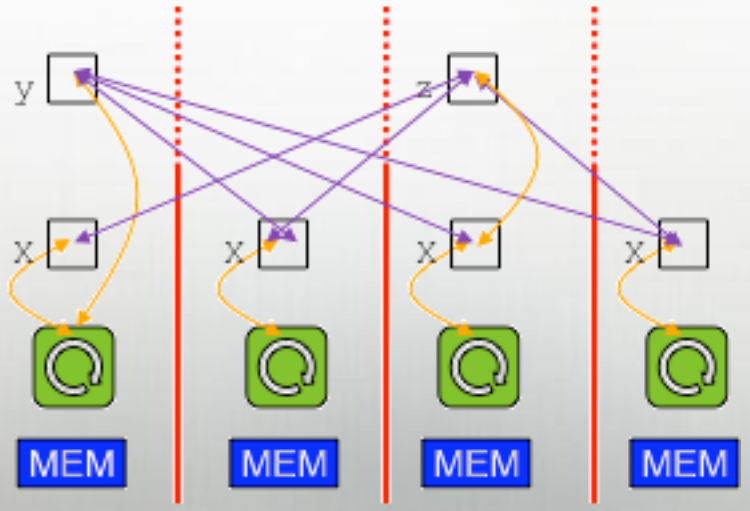
```
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      var k = i + j;
} } }
```



Next-Generation PGAS Models

e.g., Chapel (possibly X10, Fortress)

- \+ breaks out of SPMD mold via global multithreading
- \+ richer set of distributed data structures
- – retains many of the downsides of shared-memory
  - error cases, memory consistency models



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
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

Outline

- ✓ Who is Cray?
- ✓ PGAS Languages
- ✓ Chapel and PGAS
- ➢ Chapel Motivation
- Chapel Features
- Project Status

STREAM Triad: a trivial parallel computation

Given: m-element vectors A, B, C

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

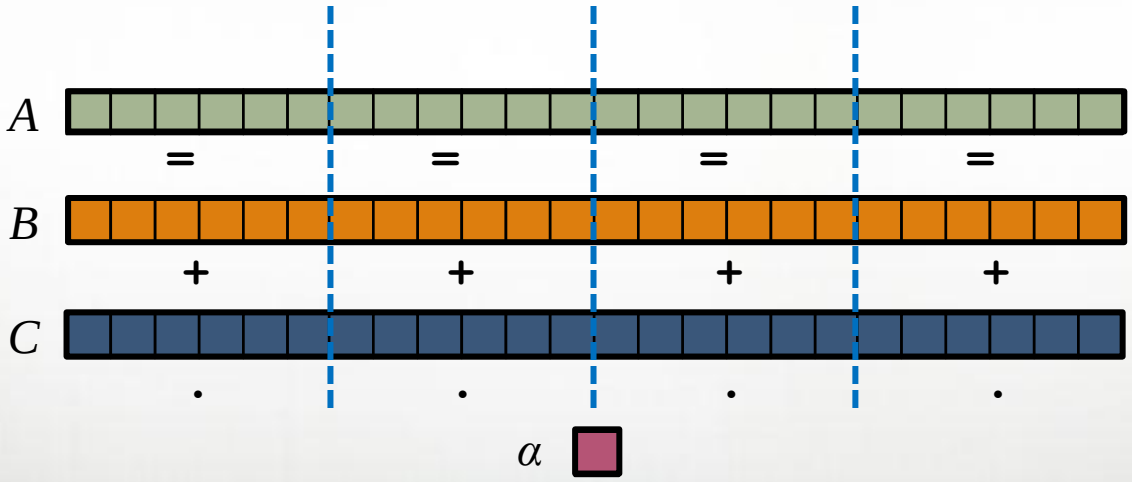
In pictures:

STREAM Triad: a trivial parallel computation

Given: m-element vectors A, B, C

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

In pictures, in parallel:

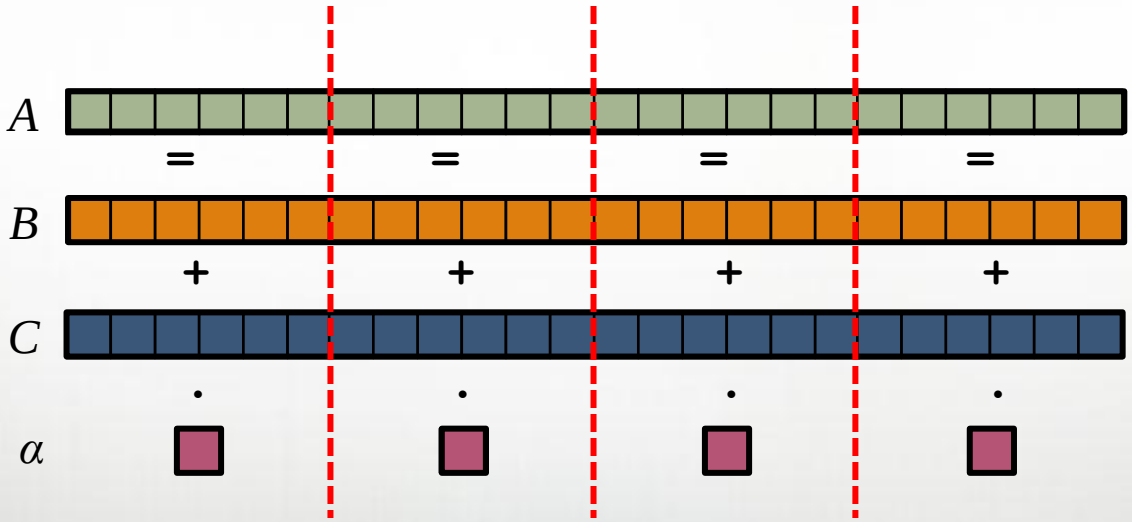


STREAM Triad: a trivial parallel computation

Given: m-element vectors A, B, C

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

In pictures, in parallel (distributed memory):

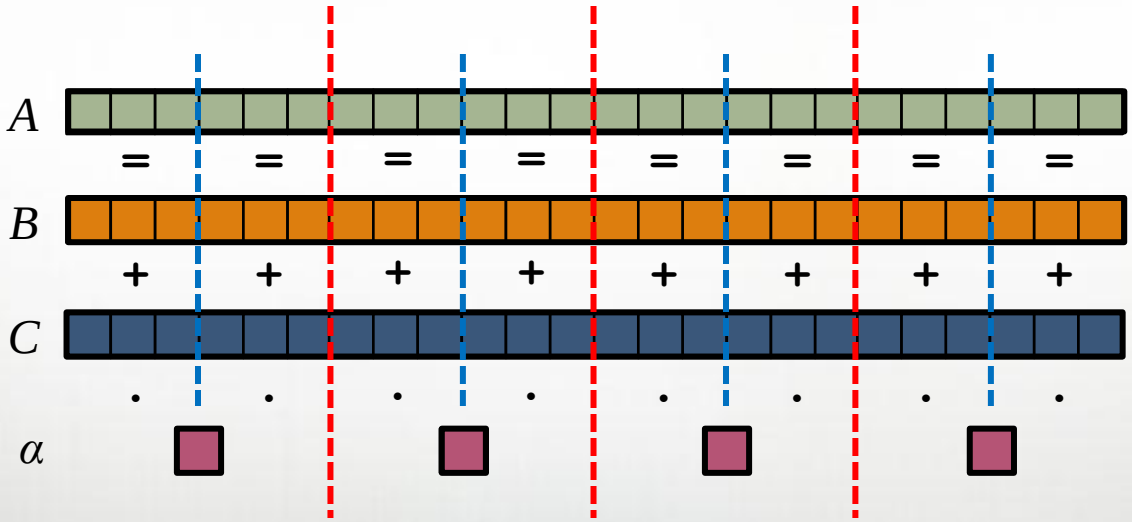


STREAM Triad: a trivial parallel computation

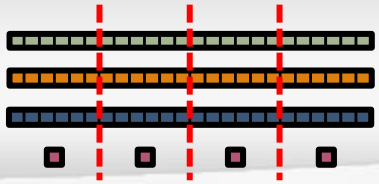
Given: m-element vectors A, B, C

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

In pictures, in parallel (distributed memory multicore):



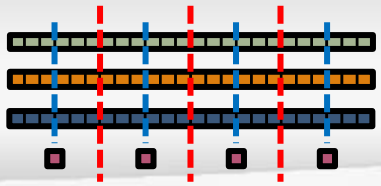
STREAM Triad: MPI



MPI

```
#include <hpcc.h>

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

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

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

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

  return errCount;
}

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

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

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

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

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

  scalar = 3.0;

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

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

  return 0;
}
```

STREAM Triad: MPI+OpenMP

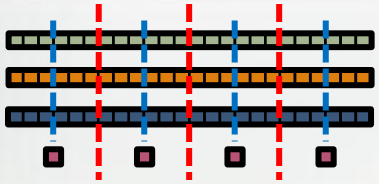


MPI + OpenMP

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

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

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

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

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

  return errCount;
}

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

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

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

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

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

  scalar = 3.0;

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

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

  return 0;
}
```

STREAM Triad: MPI+OpenMP vs. CUDA

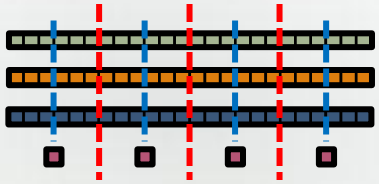
MPI + OpenMP



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

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

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

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

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

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

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

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

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

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

  scalar = 3.0;

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

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

  return 0;
}
```

CUDA



```
#define N       2000000

int main() {
  float *d_a, *d_b, *d_c;
  float scalar;

  cudaMalloc((void**)&d_a, sizeof(float)*N);
  cudaMalloc((void**)&d_b, sizeof(float)*N);
  cudaMalloc((void**)&d_c, sizeof(float)*N);

  dim3 dimBlock(128);
  dim3 dimGrid(N/dimBlock.x );
  if( N % dimBlock.x != 0 ) dimGrid.x+=1;

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

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

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

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

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

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

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:

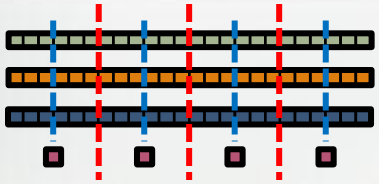
Type of HW Parallelism	Programming Model	Unit of Parallelism
Inter-node	MPI	executable
Intra-node/multicore	OpenMP/pthreads	iteration/task
Instruction-level vectors/threads	pragmas	iteration
GPU/accelerator	CUDA/OpenCL/OpenAcc	SIMD function/task

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

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

STREAM Triad: MPI+OpenMP vs. CUDA

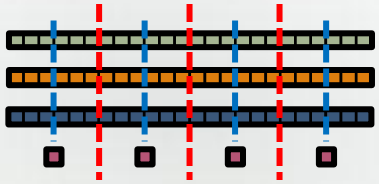
MPI + OpenMP



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

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

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

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

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

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

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

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

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

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

  scalar = 3.0;

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

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

  return 0;
}
```

CUDA



```
#define N       2000000

int main() {
  float *d_a, *d_b, *d_c;
  float scalar;

  cudaMalloc((void**)&d_a, sizeof(float)*N);
  cudaMalloc((void**)&d_b, sizeof(float)*N);
  cudaMalloc((void**)&d_c, sizeof(float)*N);

  dim3 dimBlock(128);
  dim3 dimGrid(N/dimBlock.x );
  if( N % dimBlock.x != 0 ) dimGrid.x+=1;

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

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

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

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

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

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

STREAM Triad: Chapel

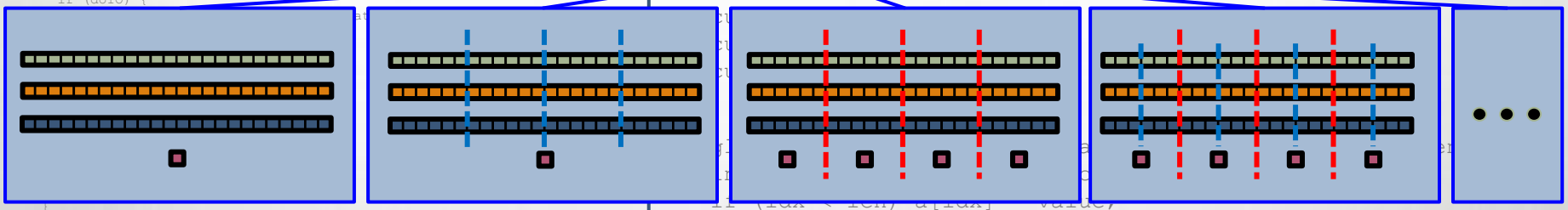
MPI + OpenMP

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

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

int HPCC_StarStream(HPCC_Params *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_Params *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) {
```

Chapel

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

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

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

B = 2.0;
C = 3.0;

A = B + alpha * C;
```

the special sauce



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

Three Motivating Chapel Themes

1) General Parallel Programming
2) Multiresolution Design
3) Reduce HPC ↔ Mainstream Language Gap

1) General Parallel Programming

With a unified set of concepts...

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

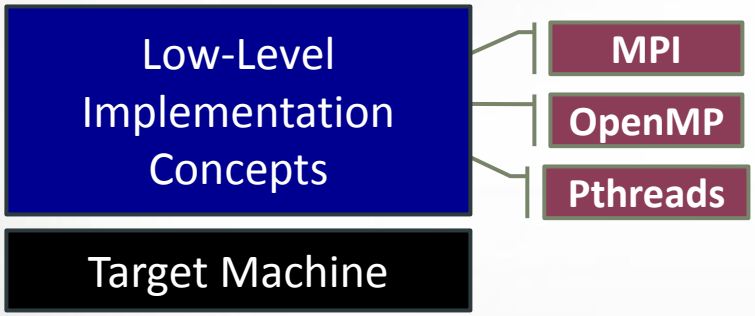
- **Styles:** data-parallel, task-parallel, concurrency, nested, …
- **Levels:** model, function, loop, statement, expression

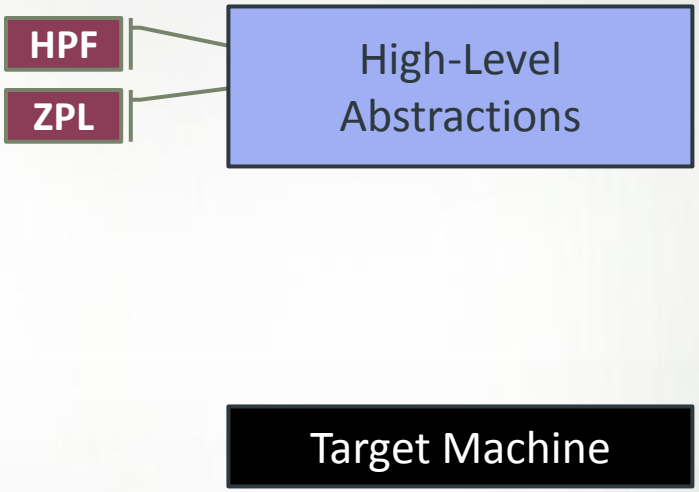
...target all parallelism available in the hardware

- **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

2) Multiresolution Design: Motivation





"Why is everything so tedious/difficult?"

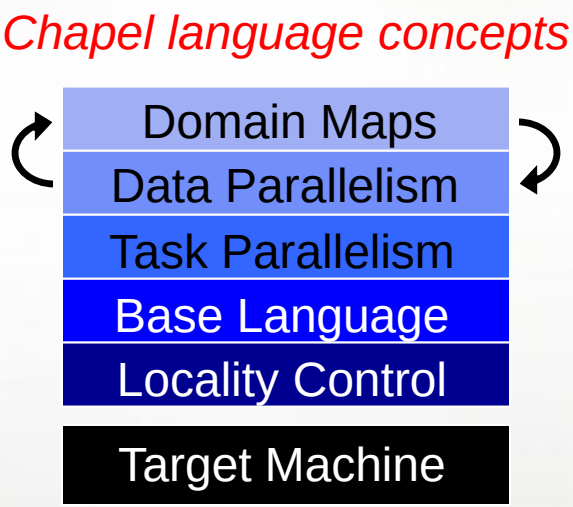
"Why don't my programs port trivially?"

"Why don't I have more control?"

2) Multiresolution Design

Multiresolution Design: Support multiple tiers of features

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



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

3) 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

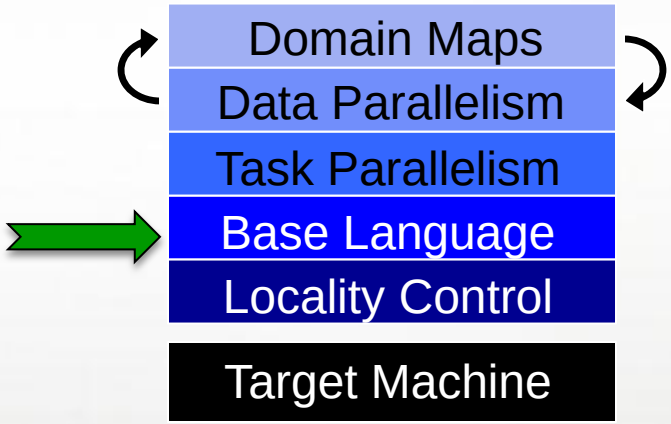
We'd like to narrow this gulf with Chapel:

- 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

- ✓ Who is Cray?
- ✓ PGAS Languages
- ✓ Chapel and PGAS
- ✓ Chapel Motivation
- ➢ Chapel Features
- • 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

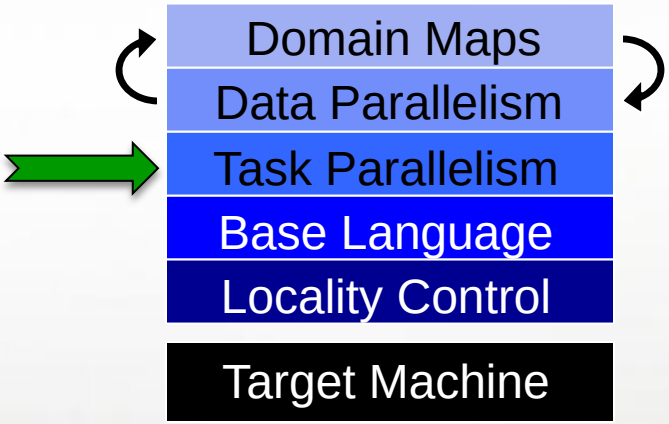
```
for (i,f) in zip(0..#n, fibonacci(n)) do
  writeln("fib #", i, " is ", f);
```

```
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
```

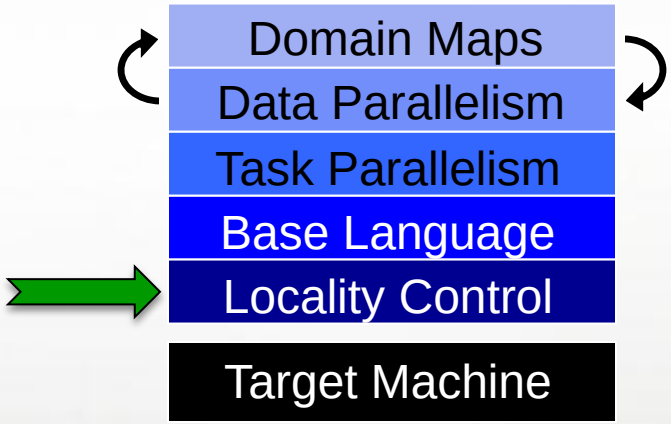
Task Parallelism: Cobegin Statements

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

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 = …;
```

Locale Operations

- Locale methods support queries about target system:

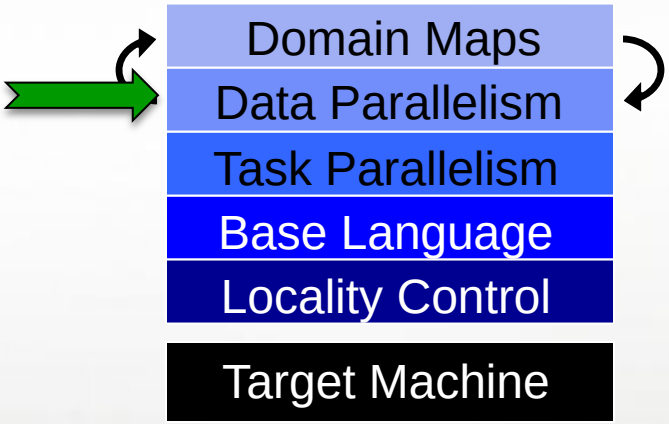
```
proc locale.physicalMemory(…) { … }
proc locale.numCores { … }
proc locale.id { … }
proc locale.name { … }
```

- *On-clauses* support placement of computations:

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

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

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

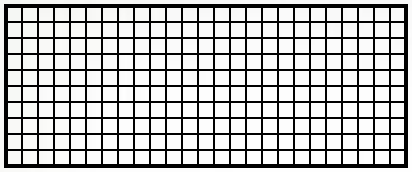
```
cobegin {
  on A[i,j] do
    bigComputation(A);

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

Data Parallel Features

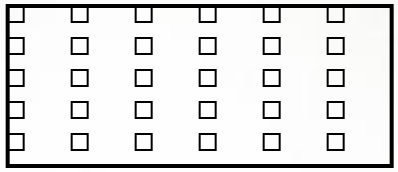


Chapel Domain Types

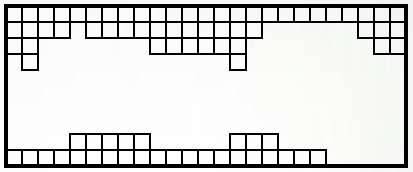
Chapel supports several types of domains (index sets) :



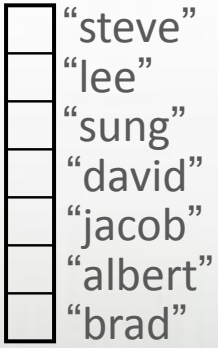
dense



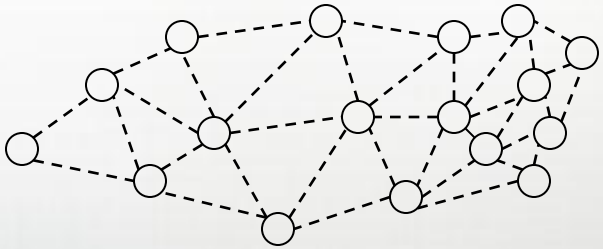
strided



sparse



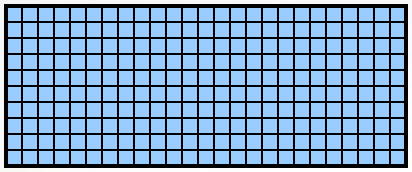
associative



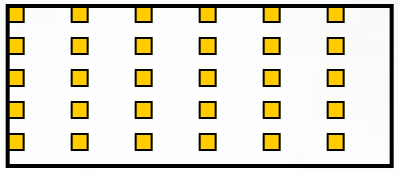
unstructured

Chapel Array Types

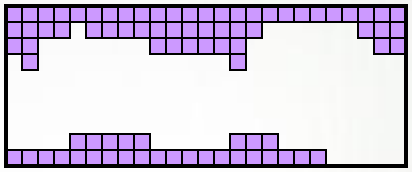
Each domain type can be used to declare arrays:



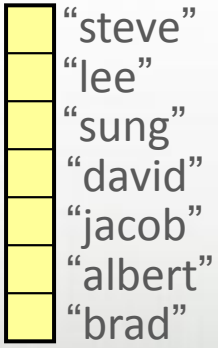
dense



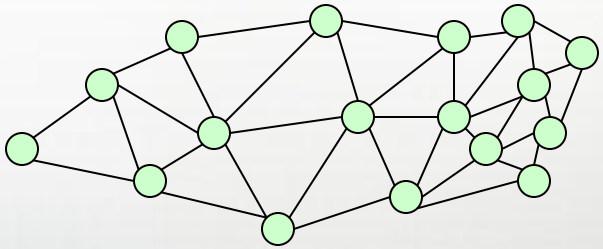
strided



sparse



associative



unstructured

Chapel Domain/Array Operations

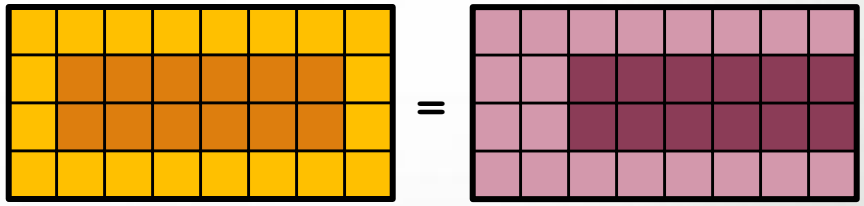
- Parallel and Serial Iteration

```
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;
```

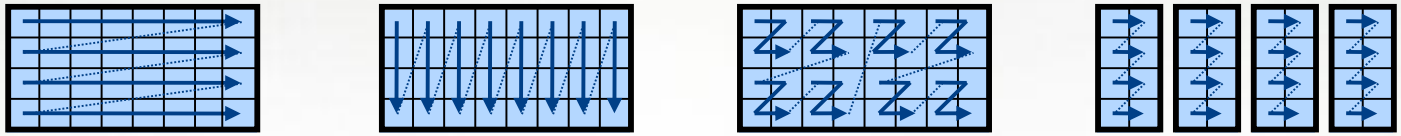
```
A = exp(B, C);
```

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

Data Parallelism Implementation Qs

Q1: How are arrays laid out in memory?

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

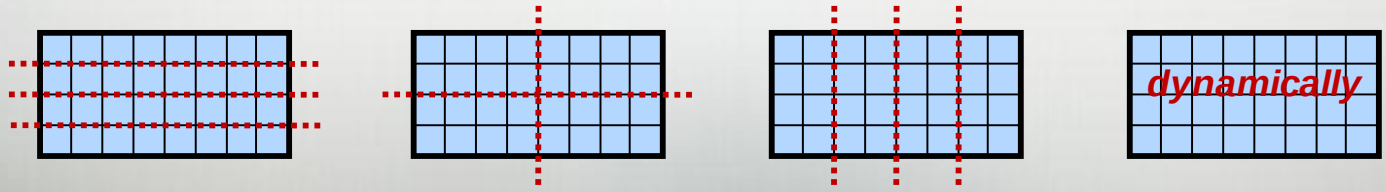


…?

- 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? …?

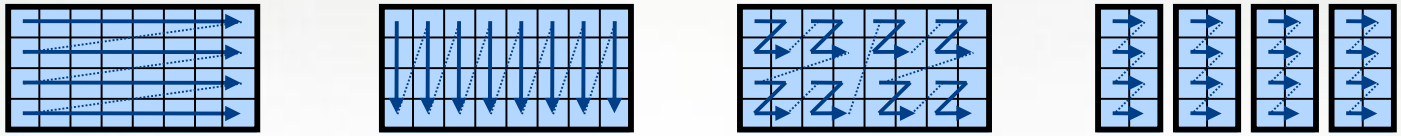


…?

Data Parallelism Implementation Qs

Q1: How are arrays laid out in memory?

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



…?

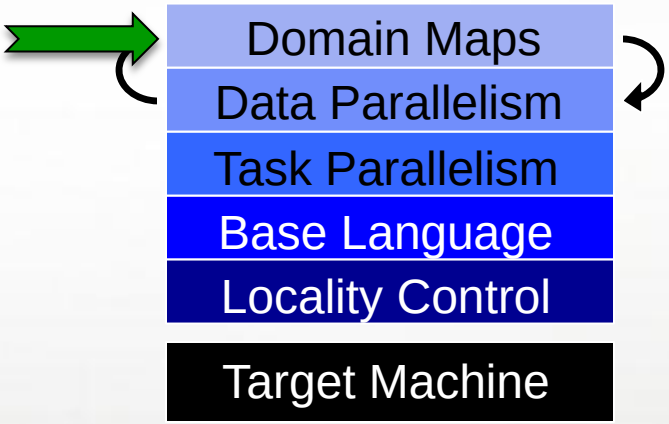
- How are sparse arrays stored? (COO, CSR, CSC, block-structured, …?)

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? …?

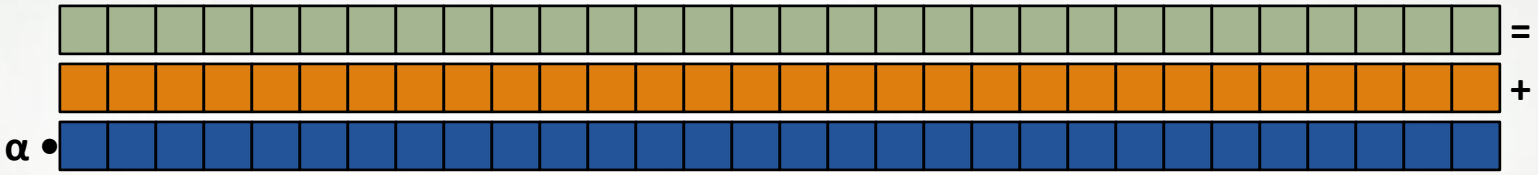
A: Chapel's *domain maps* are designed to give the user full control over such decisions

Data Parallel Features



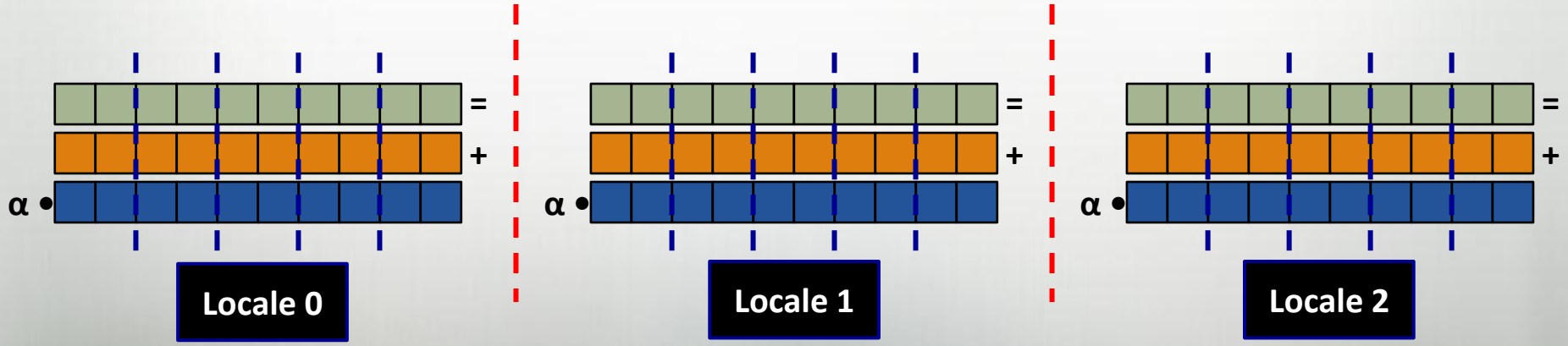
Domain Maps

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



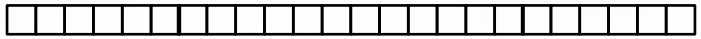

```
A = B + alpha * C;
```

…to the target locales’ memory and processors:

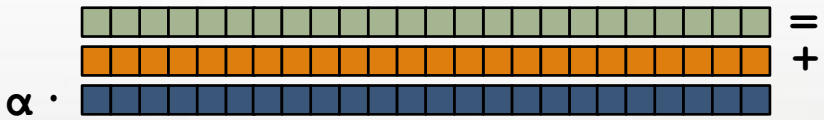


STREAM Triad: Chapel (multicore)

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



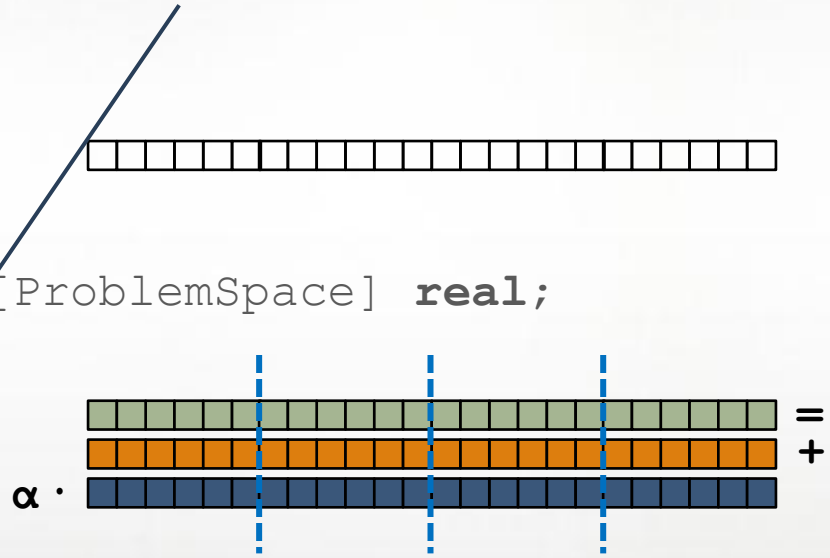
```
var A, B, C: [ProblemSpace] real;
```




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

STREAM Triad: Chapel (multicore)

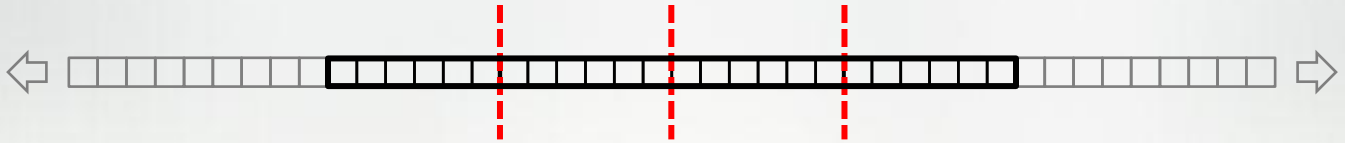
```
const ProblemSpace = {1..m};

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

A = B + alpha * C;
```



No domain map specified => use default layout

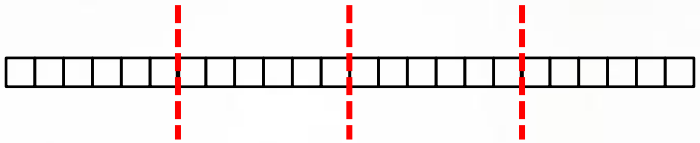
- current locale owns all indices and values
- computation will execute using local processors only

STREAM Triad: Chapel (multilocale, blocked)

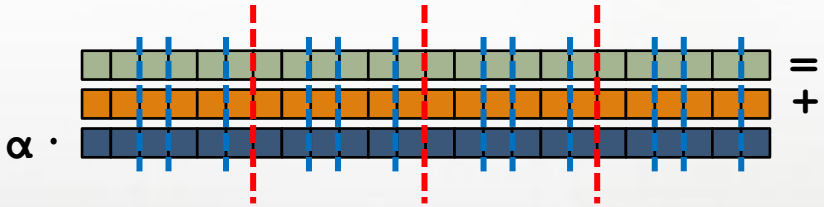


```
const ProblemSpace = {1..m}
```

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

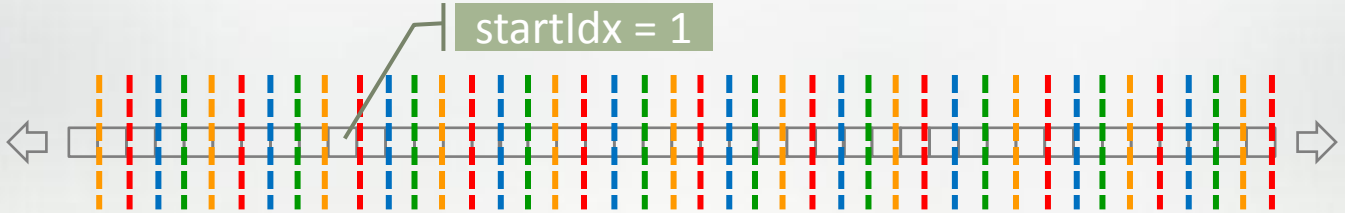


```
var A, B, C: [ProblemSpace] real;
```

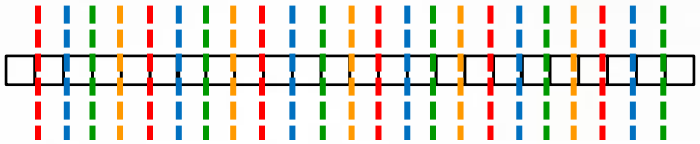



```
A = B + alpha * C;
```

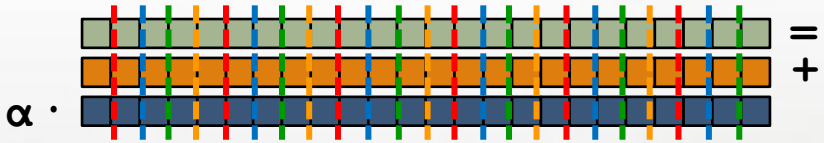
STREAM Triad: Chapel (multilocale, cyclic)




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



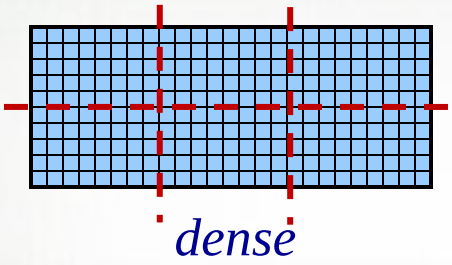
```
var A, B, C: [ProblemSpace] real;
```



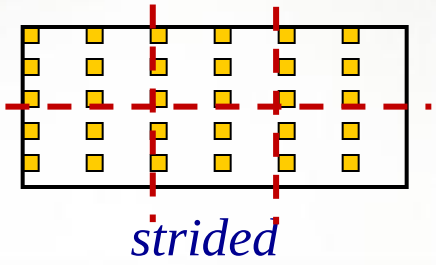

```
A = B + alpha * C;
```

Domain Map Types

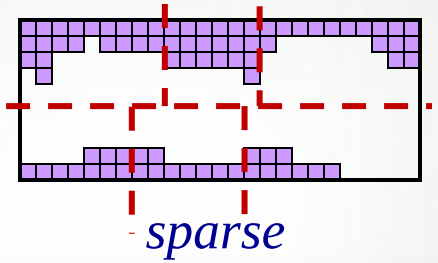
All Chapel domain types support domain maps



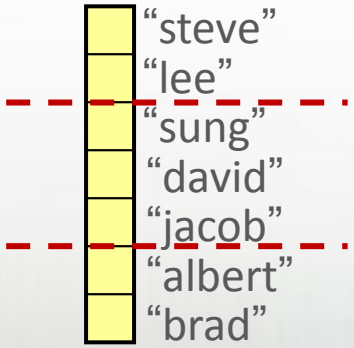
dense



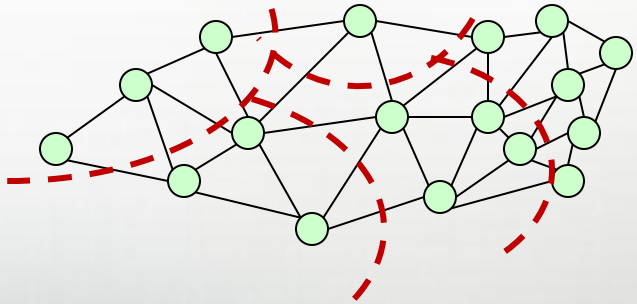
strided



sparse



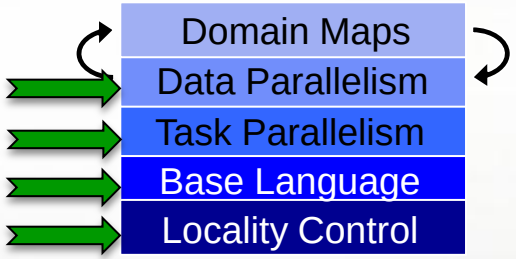
associative



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

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

Outline

- ✓ Who is Cray?
- ✓ PGAS Languages
- ✓ Chapel and PGAS
- ✓ Chapel Motivation
- ✓ Chapel Features
- ➢ 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

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

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/