Programming Models and Chapel: Landscaping for Exascale Computing

Brad Chamberlain, Cray Inc.

INT Exascale Workshop

June 30th, 2011

Sustained Performance Milestones

1 GF – 1988: Cray Y-MP; 8 Processors

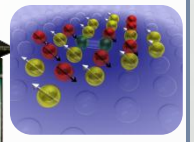
- Static finite element analysis

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

- Modeling of metallic magnet atoms

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

- Superconductive materials



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

- TBD

Sustained Performance Milestones

1 GF – 1988: Cray Y-MP; 8 Processors

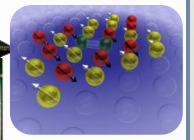
- Static finite element analysis
- Fortran77 + Cray autotasking + vectorization

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

- Modeling of metallic magnet atoms
- Fortran + MPI (?)

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

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



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

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

Why Do HPC Programming Models Change?

HPC has traditionally given users…

…low-level, *control-centric* programming models

…ones that are closely tied to the underlying hardware

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

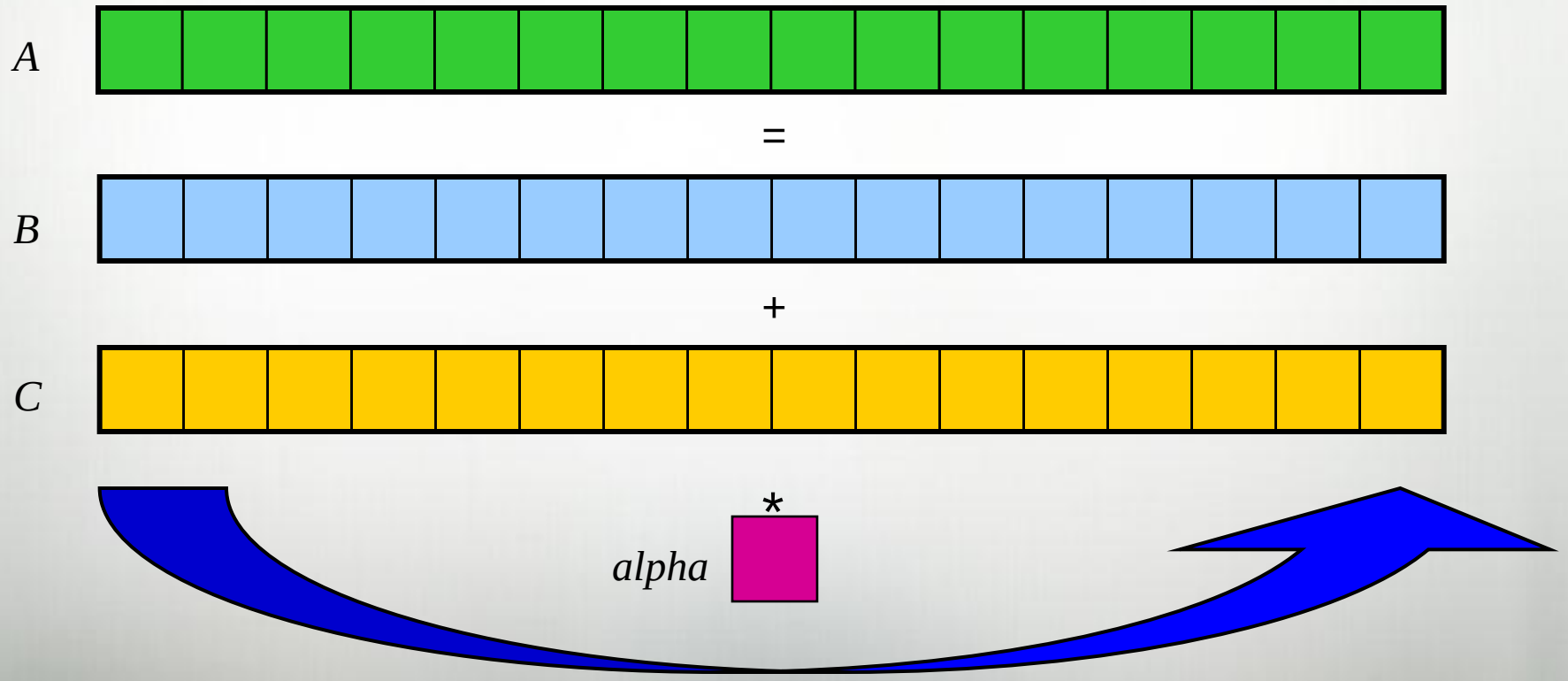
downsides: lots of user-managed detail; brittle to changes

Introduction to STREAM Triad

Given: m-element vectors A, B, C

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

Pictorially:

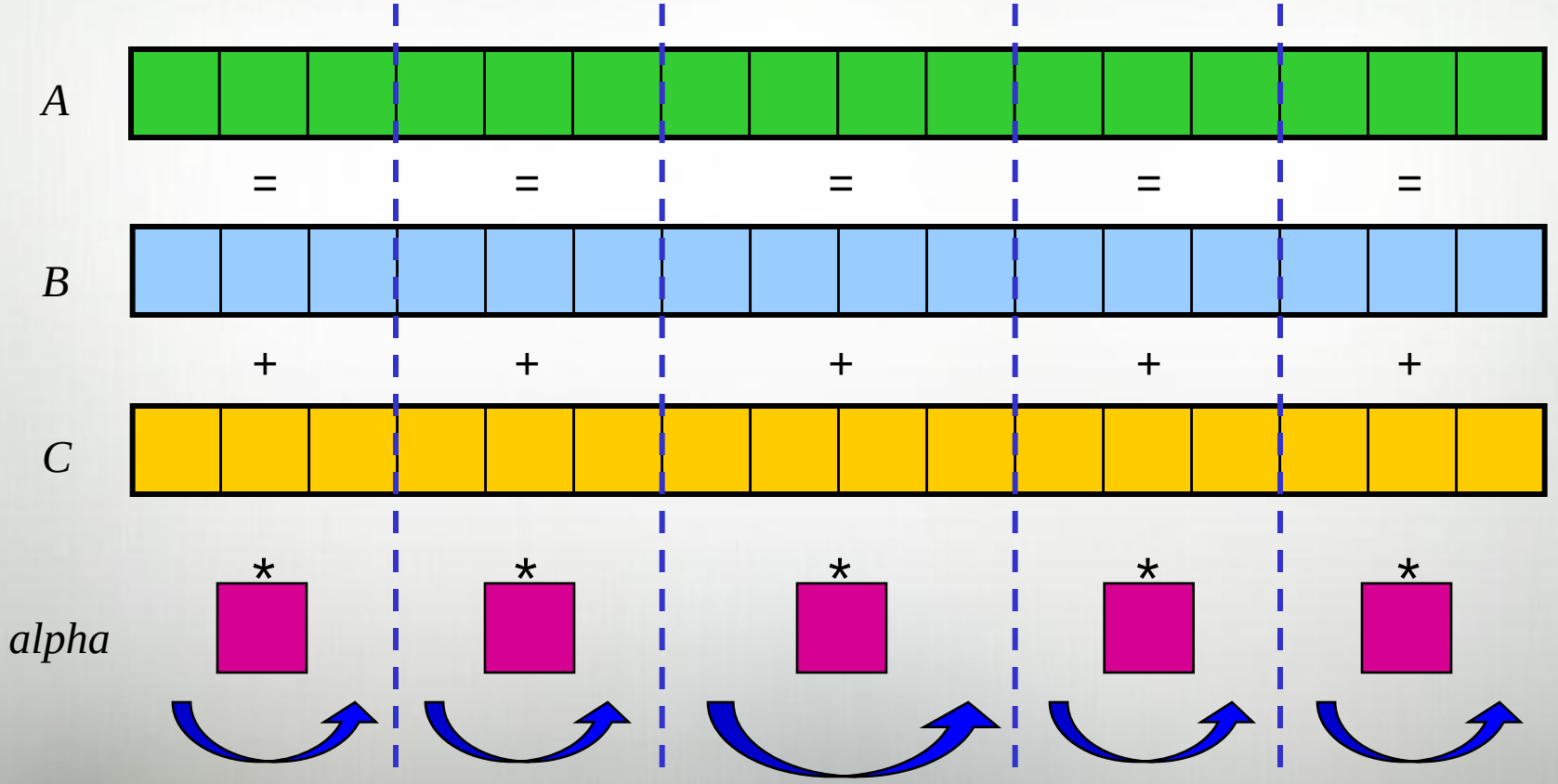


Introduction to STREAM Triad

Given: m-element vectors A, B, C

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

Pictorially (in parallel):



A Few Versions of STREAM Triad

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;
}
```

A Few Versions of STREAM Triad

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;
}
```

A Few Versions of STREAM Triad

MPI + OpenMP

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

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

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

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

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

  return errCount;
}

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

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

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

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

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

  scalar = 3.0;

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

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

  return 0;
}
```

CUDA

```
#define N       2000000

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

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

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

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

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

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

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

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

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

A Few Versions of STREAM Triad

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 == myRan
  MPI_Reduce( &rv, &errCount, 1, MPI_I

  return errCount;
}

int HPCC_Stream(HPCC_Params *params, i
  register int j;
  double  scalar;

  VectorSize = HPCC_LocalVectorSize( p

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

  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
      fprintf( outFile, "Failed to all
      fclose( outFile );
    }
    return 1;
  }

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

  scalar = 3.0;

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

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

  return 0;
}
```

CUDA

```
#define N       2000000

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

```
;
;
;
```

```
N);
N);
```

```
d_c, d_a, scalar,  N);
```

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

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

Chapel

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

const ProbSpace = [1..m] dmapped …;

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

B = …;
C = …;

A = B + alpha * C;
```

the special sauce

Why Do HPC Programming Models Change?

HPC has traditionally given users…

…low-level, *control-centric* programming models

…ones that are closely tied to the underlying hardware

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

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

one characterization of Chapel's goals:

- Raise the level of abstraction to insulate parallel algorithms from underlying hardware when possible/practical
- Yet permit control over such details using appropriate abstraction and separation of concerns

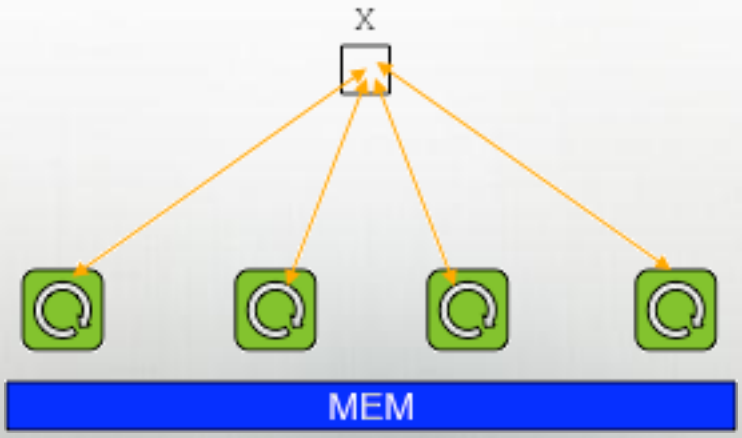
Outline

- ✓ Motivation
- ➢ Programming Model Survey
  - Current Practice
  - Prognosis for Exascale
- ❑ Chapel Overview
- ❑ Status and Future Directions
- ❑ Case Study: AMR
- ❑ Wrap-up

Shared Memory Programming Models

e.g., OpenMP, pthreads

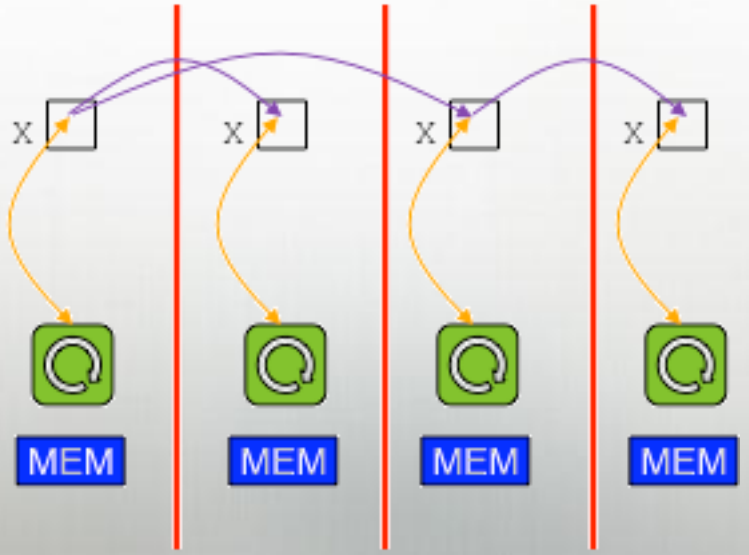
- \+ support dynamic, fine-grain parallelism
- \+ considered simpler, more like traditional programming
  - "if you want to access something, simply name it"
- – no support for expressing locality/affinity; limits scalability
- – bugs can be subtle, difficult to track down (race conditions)
- – tend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



Distributed Memory Programming Models

e.g., MPI

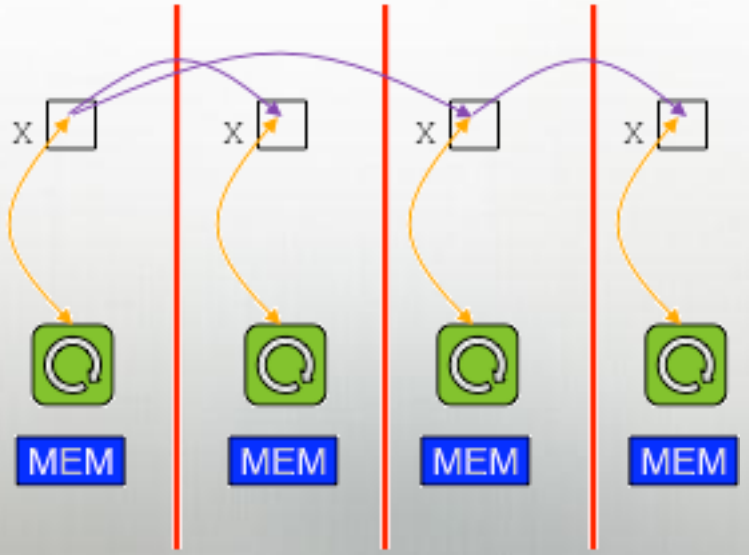
+ a more constrained model; can only access local data
+ run on most large-scale parallel platforms
  - and for many of them, can achieve near-optimal performance
+ are relatively easy to implement
+ can serve as a strong foundation for higher-level models
+ users are able to get real work done with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



Distributed Memory Programming Models

e.g., MPI

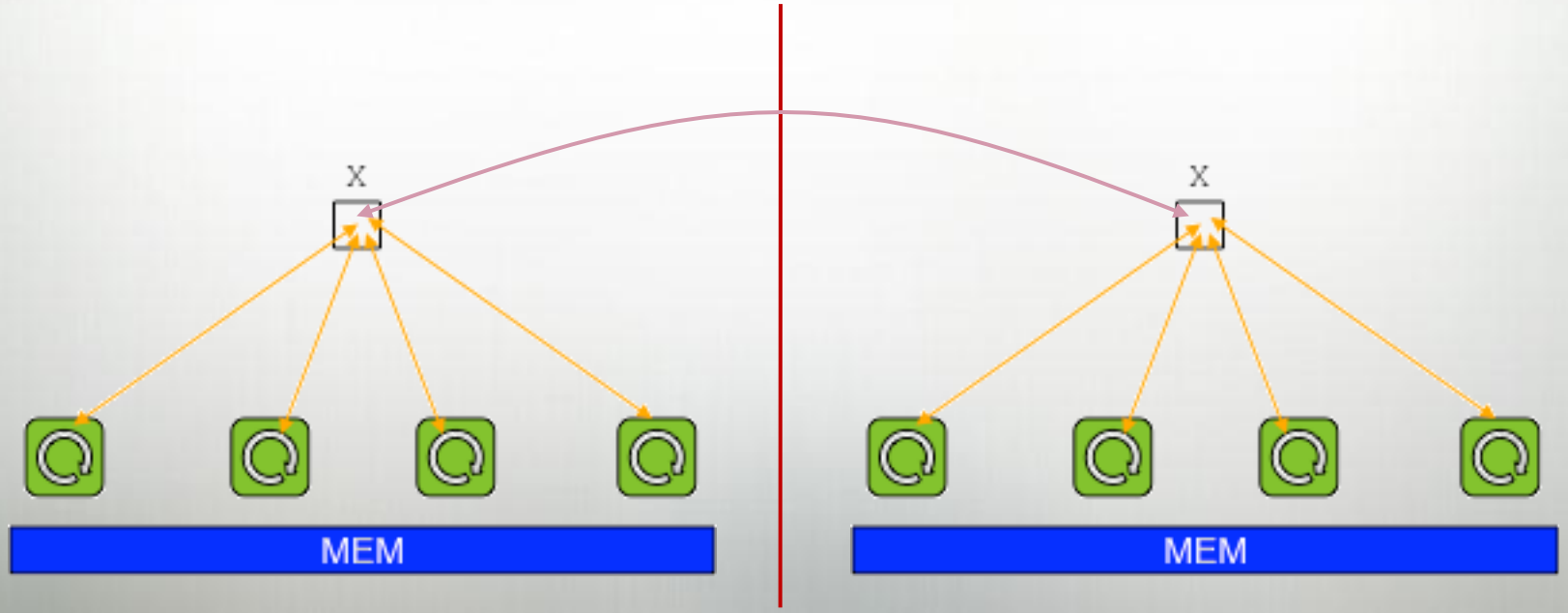
- communication must be used to get copies of remote data
  - and tends to reveal too much about *how* to transfer data, not simply *what*
- only supports "cooperating executable"-level parallelism
- couples data transfer and synchronization
- has frustrating classes of bugs of its own
  - e.g., mismatches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



Hybrid Programming Models

***e.g.*, MPI+OpenMP, MPI+pthreads, MPI+CUDA, …**

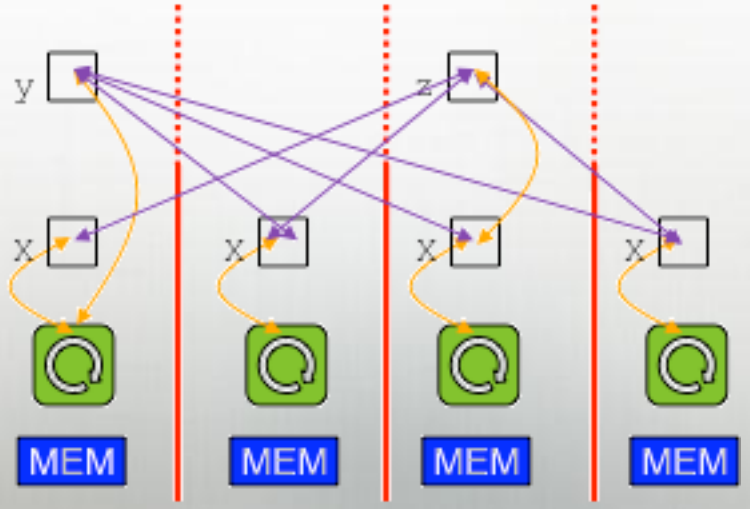
- \+ support a division of labor: each handles what it does best
- \+ permit overheads to be amortized across processor cores
- – require multiple distinct notations to express a single logical parallel algorithm, each with its own distinct semantics



PGAS (Partitioned Global Address Space) Models

e.g., Co-Array Fortran (CAF), Unified Parallel C (UPC)

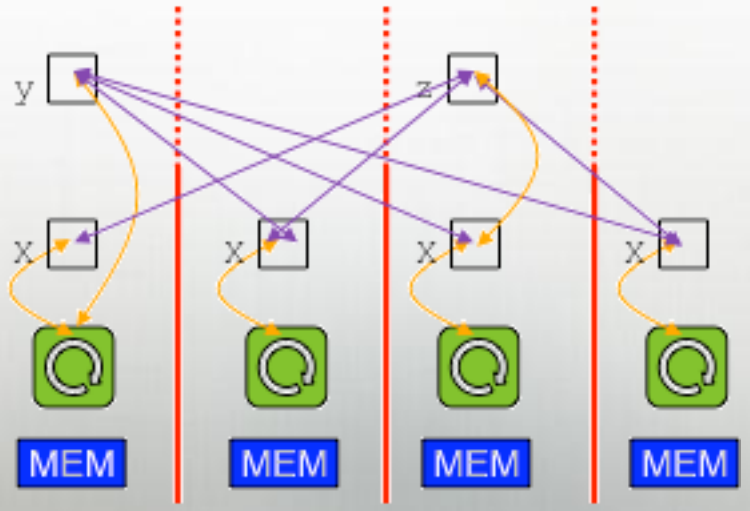
- support a shared namespace, like shared-memory
- support a strong sense of ownership and locality
  - each variable is stored in a particular memory segment
  - tasks can access any visible variable, local or remote
  - local variables are cheaper to access than remote ones
- implicit communication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



PGAS (Partitioned Global Address Space) Models

e.g., Co-Array Fortran (CAF), Unified Parallel C (UPC)

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



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

And many others…

***e.g.*, Global Arrays, Charm++, ParalleX, StarSS, Cilk, TBB, CnC, parallel Matlabs, Star-P, PLINQ, C++AMP, Map-Reduce, QLUA, DPJ, Titanium, …**

- Each interesting in its own way, but lumped together here due to lack of time and dominance/prominence in HPC
- Not trivial to categorize, but recurring themes include:
  - dynamic task parallelism
  - data-driven execution
  - advanced data structures
  - support for next-generation architectures
  - modern language features

(Chapel shares many of these as well)

Exascale Architectures

- The preceding evaluations were all w.r.t. petascale
- Exascale is expected to bring new challenges:
  - increased hierarchy within the node architecture
    - *i.e.*, locality matters within a node, not just between nodes
  - increased heterogeneity as well
    - multiple processor types
    - multiple memory types
  - limited memory bandwidth, memory::FLOP ratio

Programming Exascale Architectures

Q: Are we ready?

A: In a nutshell, no

Q: Why?

A: We've built too many assumptions about our target architectures into our programming models

- granularity and style of parallelism
- mode of communication
- single level of locality, if any at all

Programming Models' Reaction to Exascale

MPI:

- "MPI everywhere" zealots are becoming increasingly scarce
- "MPI + X" is the expected evolutionary path (solve for X)
- MPI-3 striving to support and interact with diverse models

OpenMP:

- Wrestling with role of locality, accelerators in OpenMP
- How to preserve traditional strengths while adapting?

Traditional PGAS:

- Considered by some to be well-positioned for intra-node locality concerns
- Yet, SPMD programming/execution model seems hobbling
  - so how to add dynamic execution cleanly and elegantly?

Accelerator Programming Models (X?)

CUDA:

- Far less painful than writing nuclear physics in OpenGL
- Dominating due to time-to-market, libraries, strong support
- Reasonably NVIDIA-centric
- Arguably too tied to processor architecture

OpenCL:

- Later to the game, but with broad consortium support
- Designed with portability in mind
- Not ideal for end-users; better suited as a compiler target

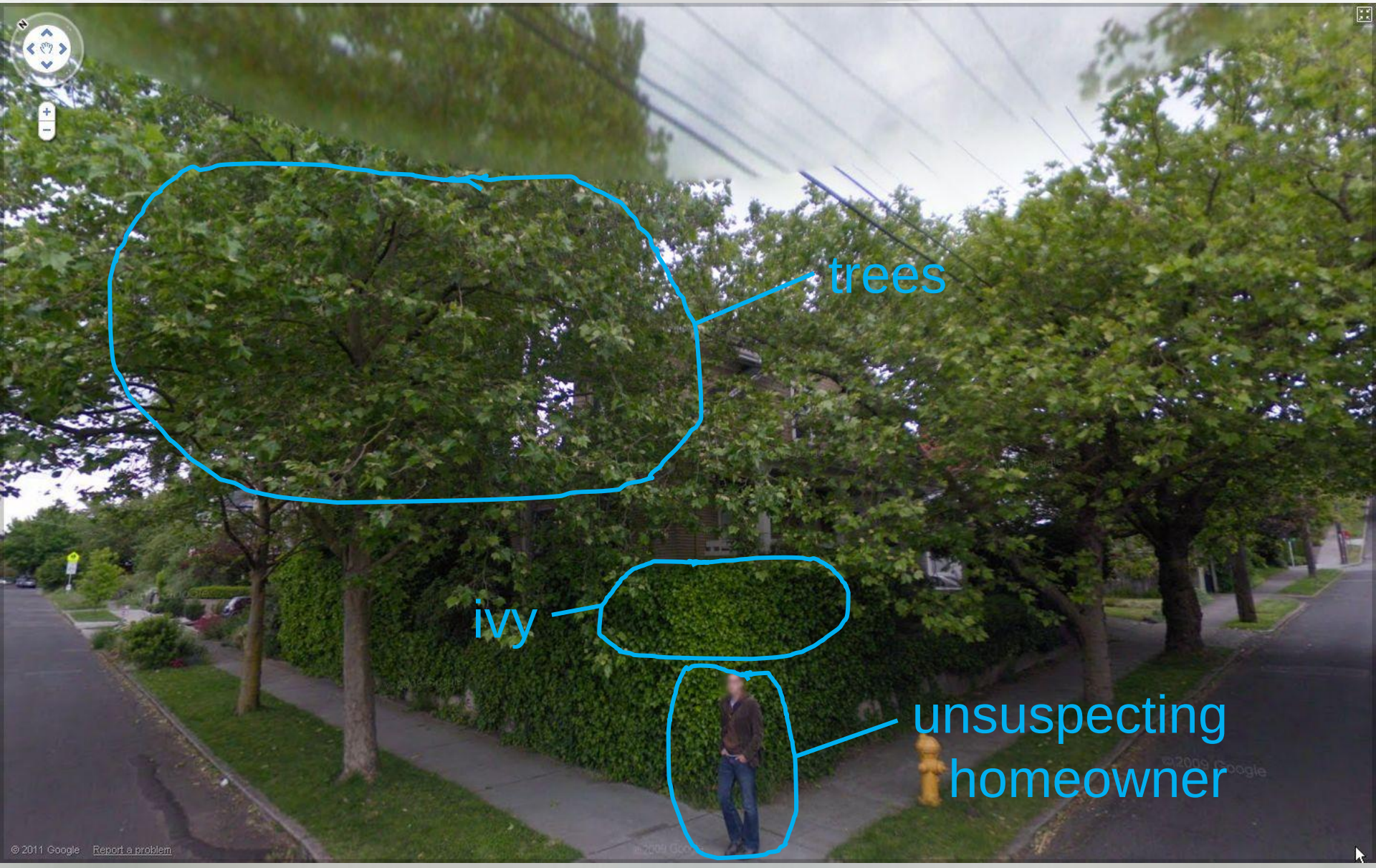
directive-based approaches (PGI, CAPS, OpenMP):

- higher-level ⇒ simpler, less control, more reliance on compiler
- traditionally harder to apply modularly
- for evolutionary approaches, I'd bet on this for X

CRAY
THE SUPERCOMPUTER COMPANY

And now, a sidebar on landscaping…

A Seattle Corner



Ivy

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

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

Trees

- higher-level
- more elegant, structured

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

Landscaping Quotes from the HPC community

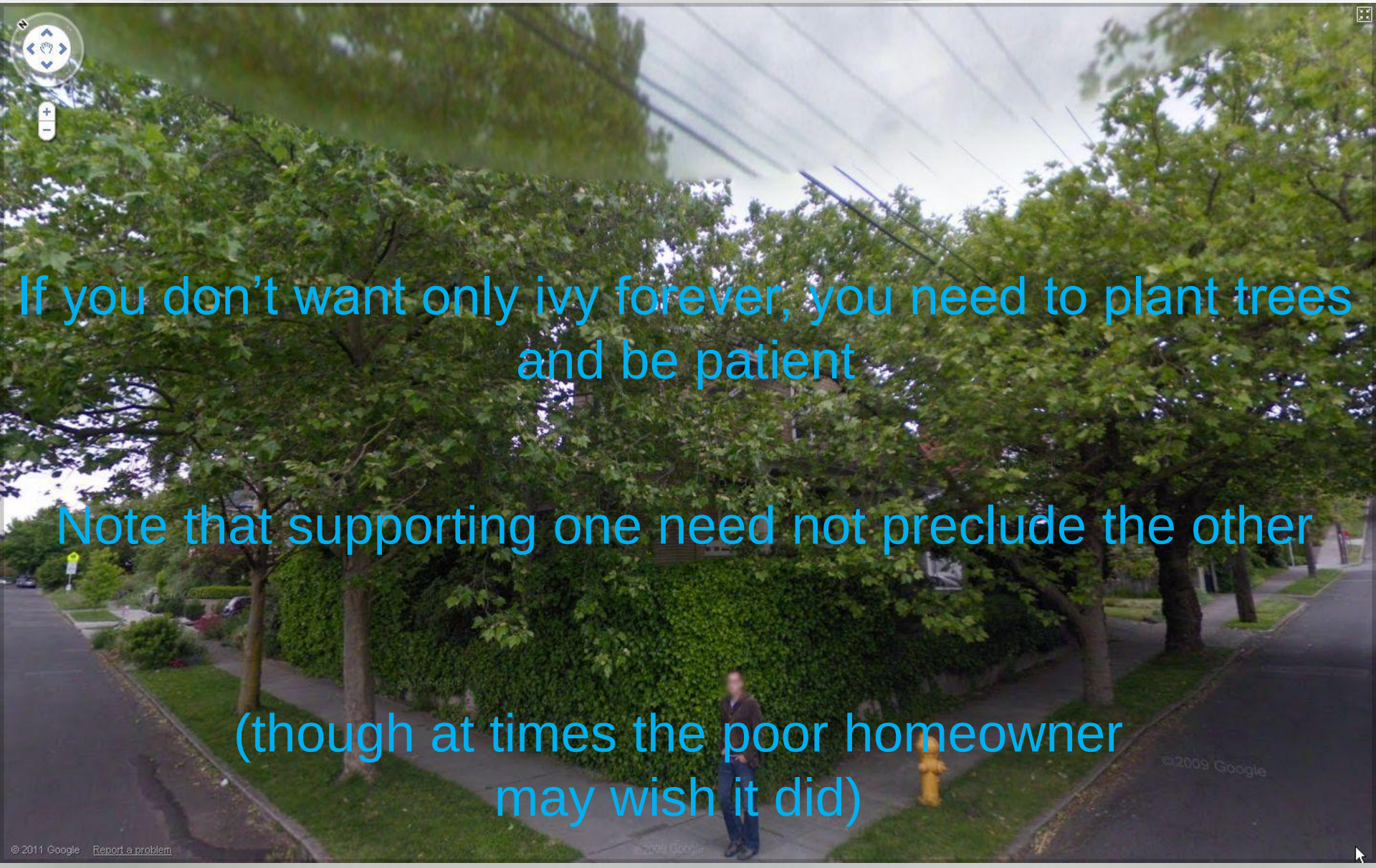
Early HPCS years:

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

More recently:

- "I really hope to see this tree fully grown someday."
- "What can I do to help the tree grow?"

A Corner in Seattle: Takeaways



Outline

- ✓ Motivation
- ✓ Programming Model Survey
- ➢ Chapel Overview
- ❑ Status and Future Directions
- ❑ Case Study: AMR
- ❑ Wrap-up

What is Chapel?

- A new parallel programming language
  - Design and development led by Cray Inc.
  - Initiated under the DARPA HPCS program
- **Overall goal:** Improve programmer productivity
  - Improve the programmability of parallel computers
  - Match or beat the performance of current programming models
  - Support better portability than current programming models
  - Improve the robustness of parallel codes
- A work-in-progress

Chapel's Implementation

- Being developed as open source at SourceForge
- Licensed as BSD software
- Target Architectures:
  - multicore desktops and laptops
  - commodity clusters
  - Cray architectures
  - systems from other vendors
  - (in-progress: next-generation node architectures)

Why a language rather than a library?

- To support compiler optimizations
- To support cleaner syntax
- Because parallel computing is lacking a good, general, modern language
- Because libraries would not help with many of the features we wanted
- Because we believe the combination of Chapel's features is greater than the sum of their parts

CRAY
THE SUPERCOMPUTER COMPANY

Q: What features did you want from a language?

CRAY
THE SUPERCOMPUTER COMPANY

A1: We wanted a *general* parallel language:
any algorithm, any hardware, any granularity

General Parallel Programming in Chapel

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

- **Systems:** multicore desktops, clusters, HPC systems, …
- **Levels:** machines, nodes, cores, instructions

In short, you should never hit a point where you say "Well, that was fun while it lasted; now back to Fortran/MPI/CUDA…"

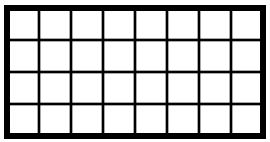
CRAY
THE SUPERCOMPUTER COMPANY

A2: We wanted excellent support for arrays: multidimensional, sparse, associative, unstructured

Domains

domain: a first-class index set

```
var m = 4, n = 8;

var D: domain(2) = [1..m, 1..n];
```

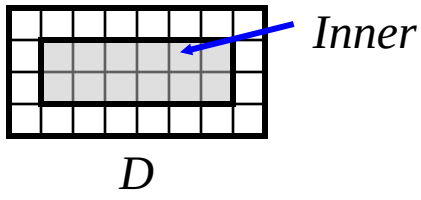


D

Domains

domain: a first-class index set

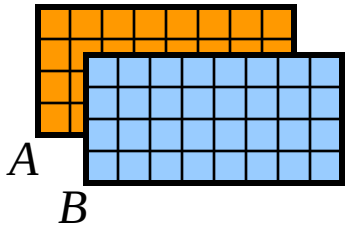
```
var m = 4, n = 8;

var D: domain(2) = [1..m, 1..n];
var Inner: subdomain(D) = [2..m-1, 2..n-1];
```



Domain Uses

- Declaring arrays:

```
var A, B: [D] real;
```



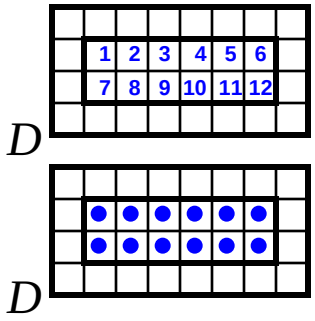
- Iteration (sequential or parallel):

```
for    ij in Inner { … }
```

or:

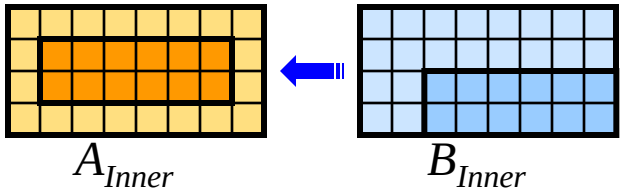
```
forall ij in Inner { … }
```

or: …



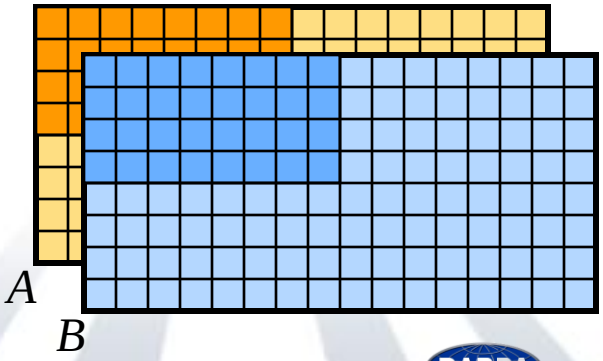
- Array Slicing:

```
A[Inner] = B[Inner+(1,1)];
```



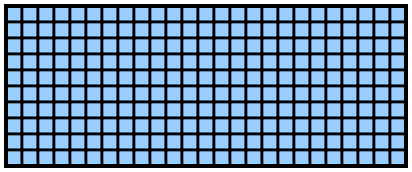
- Array reallocation:

```
D = [1..2*m, 1..2*n];
```

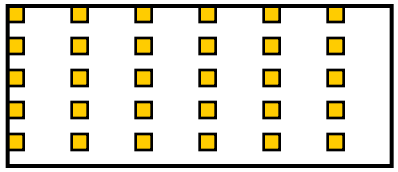


Domain/Array Types

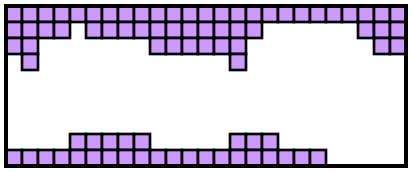
Chapel supports several types of domains and arrays…



dense

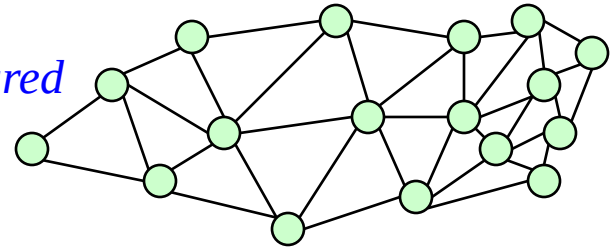


strided



sparse

unstructured



associative

…all of which support a similar set of data parallel operators:

- iteration, slicing, reallocation, promotion of scalar functions, etc.

CRAY
THE SUPERCOMPUTER COMPANY

A3: We wanted a rich task-parallel language: parallel and concurrent tasks, data-driven synchronization

Bounded Buffer Producer/Consumer Example

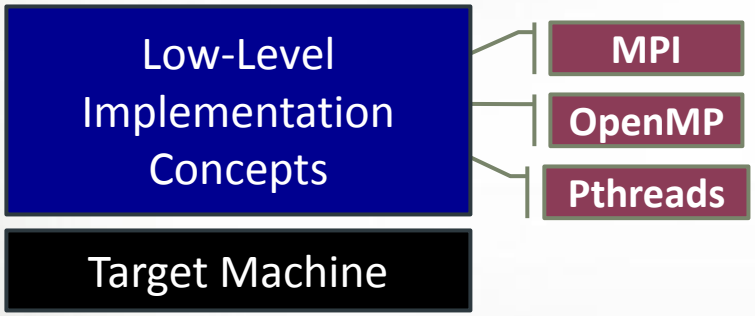
```
cobegin {
  producer();
  consumer();
}

var buff$: [0..#buffersize] sync real;

proc producer() {
  var i = 0;
  for … {
    i = (i+1) % buffersize;
    buff$(i) = …;          // data-centric synchronization
  }
}

proc consumer() {
  var i = 0;
  while … {
    i= (i+1) % buffersize;
    …buff$(i)…;            // data-centric synchronization
  }
}
```

CRAY
THE SUPERCOMPUTER COMPANY

A4: We wanted a multiresolution language

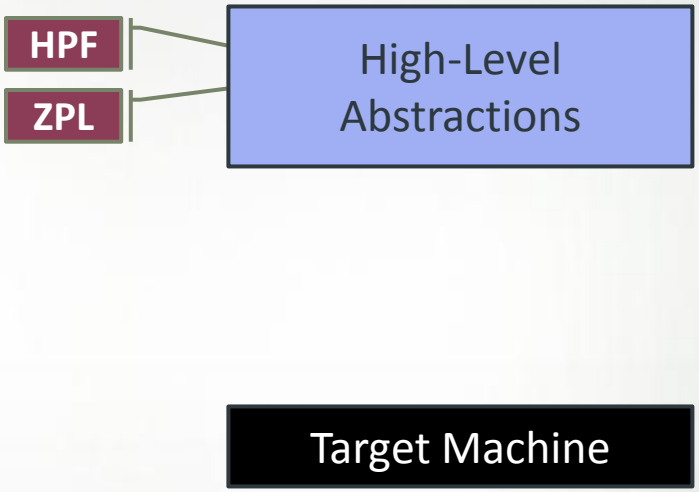
Multiresolution Language Design: Motivation

("ivy")



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

("trees")

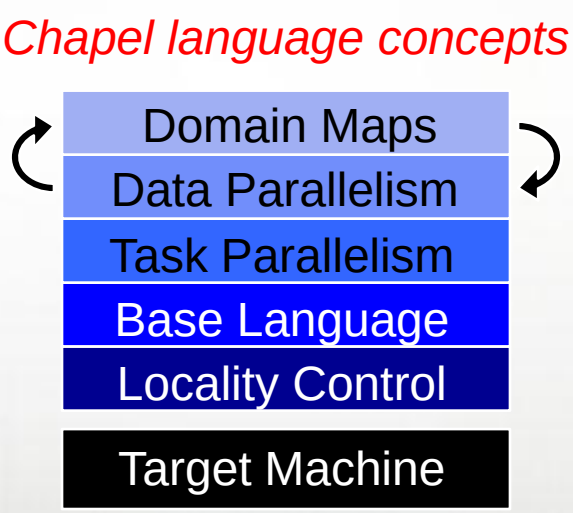


"Why don't I have more control?"

Multiresolution Language Design

Multiresolution Design: Support multiple tiers of features

- higher levels for programmability, productivity
- lower levels for performance, control
- build the higher-level concepts in terms of the lower



- separate concerns appropriately for clean design

CRAY
THE SUPERCOMPUTER COMPANY

A5: We wanted compile-time type inference and generic programming

Static Type Inference Examples

```
const pi = 3.14,           // pi is a real
      loc = 1.2 + 3.4i,    // loc is a complex
      loc2 = pi*loc,       // as is loc2
      name = "brad",       // name is a real
      verbose = false;     // verbose is boolean


proc addem(x, y) {
  return x + y;
}

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

CRAY
THE SUPERCOMPUTER COMPANY

A6: We wanted CLU-style iterators

Iterators

- ***Iterator:*** a function that generates values/variables
  - Used to drive loops
  - Like a function, but yields values back to invocation site
  - Control flow logically continues from that point
- Example

```
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
```

Iterators: Motivation

Given a program with a bunch of similar loops…

```
for (i=0; i<m; i++) {
  for (j=0; j<n; j++) {
    …A[i,j]…
  }
}

…

for (i=0; i<m; i++) {
  for (j=0; j<n; j++) {
    …A[i,j]…
  }
}

…
```

Consider the effort to convert them from RMO to CMO…

```
for (j=0; j<n; j++) {
  for (i=0; i<m; i++) {
    …A[i,j]…
  }
}

…

for (j=0; j<n; j++) {
  for (i=0; i<m; i++) {
    …A[i,j]…
  }
}

…
```

Or to tile the loops…

```
for (jj=0; jj<n; jj+=blocksize) {
  for (ii=0; ii<m; ii+=blocksize) {
    for (j=jj; j<min(m,jj+blocksize-1) {
      for (i=ii; i<min(n,ii+blocksize-1) {
        …A[i,j]…
      }
    }
  }
}

…

for (jj=0; jj<n; jj+=blocksize) {
  for (ii=0; ii<m; ii+=blocksize) {
    for (j=jj; j<min(m,jj+blocksize-1) {
      for (i=ii; i<min(n,ii+blocksize-1) {
        …A[i,j]…
      }
    }
  }
}
…
```

Iterators: Motivation

Given a program with a bunch of similar loops…

```
for (i=0; i<m; i++) {
  for (j=0; j<n; j++) {
    …A[i,j]…
  }
```

Consider the effort to convert them from RMO to CMO…

```
for (j=0; j<n; j++) {
  for (i=0; i<m; i++) {
    …A[i,j]…
  }
```

Or to tile the loops…

```
for (jj=0; jj<n; jj+=blocksize) {
  for (ii=0; ii<m; ii+=blocksize) {
    for (j=jj; j<min(m,jj+blocksize-1) {
      for (i=ii; i<min(n,ii+blocksize-1) {
        …A[i,j]…
      }
    }
```

Or to change the iteration order over the tiles…

Or to make them into fragmented loops for an MPI program…

Or to change the distribution of the work/arrays in that MPI program…

Or to label them as parallel for OpenMP or a vectorizing compiler…

Or to do *anything* that we do with loops all the time as a community…

We wouldn't program straight-line code this way, so why are we so tolerant of our lack of loop abstractions?

Iterators

- as with traditional functions…
  - …one iterator can be redefined to change the behavior of many loops
  - …a single invocation can be altered, or its arguments can be
- not necessarily any more expensive than standalone loops

CRAY
THE SUPERCOMPUTER COMPANY

A7: We wanted to control and reason about locality distinctly from parallelism

The Locale

- **Definition**
  - Abstract unit of target architecture
  - Capable of running tasks and storing variables
    - i.e., has processors and memory
  - Supports reasoning about locality
- **Properties**
  - a locale's tasks have ~uniform access to local vars
  - Other locale's vars are accessible, but at a price
- **Locale Examples**
  - A multi-core processor
  - An SMP node

Coding with Locales

- Specify # of locales when running Chapel programs

```
% a.out --numLocales=8
```

```
% a.out -nl 8
```

- Chapel provides built-in locale variables

```
const Locales: [LocaleSpace] locale;
```

- Locales support reasoning about machine resources

```
proc locale.physicalMemory(...) { ... }
```

- Locales support placement of computations:

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

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

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

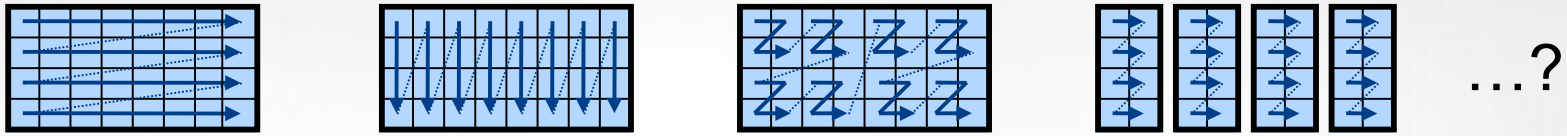
CRAY
THE SUPERCOMPUTER COMPANY

A8: We wanted to control array implementations: memory layout, distributions, parallelization strategies

Data Parallelism: Implementation Qs

Q1: How are arrays laid out in memory?

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

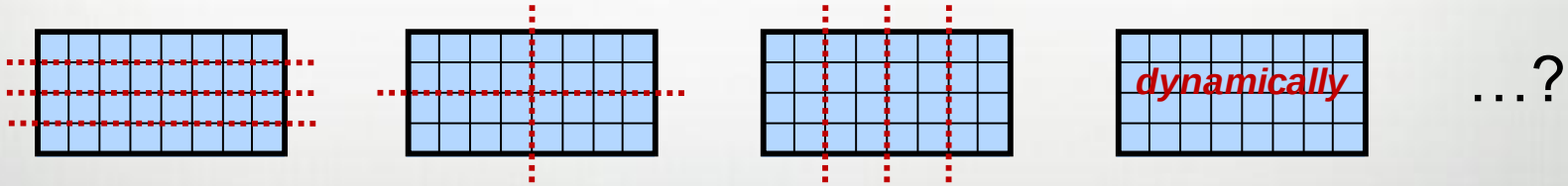


…?

- What data structure is used to store sparse arrays? (COO, CSR, …?)

Q2: How are data parallel operators implemented?

- How many tasks?
- How is the iteration space divided between the tasks?



…?

Data Parallelism: Implementation Qs

Q3: How are arrays distributed between locales?

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

Q4: What architectural features will be used?

- Can/Will the computation be executed using CPUs? GPUs? both?
- What memory type(s) is the array stored in? CPU? GPU? texture? …?

A1: In Chapel, any of these could be the correct answer

A2: Chapel's *domain maps* are designed to give the user full control over such decisions

A Few Versions of STREAM Triad

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 == myRan
  MPI_Reduce( &rv, &errCount, 1, MPI_I

  return errCount;
}

int HPCC_Stream(HPCC_Params *params, i
  register int j;
  double  scalar;

  VectorSize = HPCC_LocalVectorSize( p

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

  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
      fprintf( outFile, "Failed to all
      fclose( outFile );
    }
    return 1;
  }

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

  scalar = 3.0;

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

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

  return 0;
}
```

CUDA

```
#define N       2000000

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

```
;
;
;
```

```
N);
N);
```

```
d_c, d_a, scalar,  N);
```

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

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

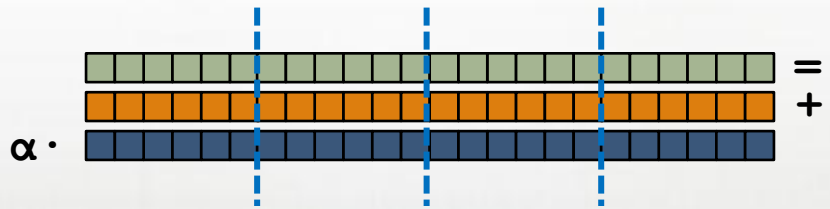
Chapel

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

const ProbSpace = [1..m] dmapped …;

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

B = …;
C = …;

A = B + alpha * C;
```

the special sauce

Global STREAM Triad in Chapel

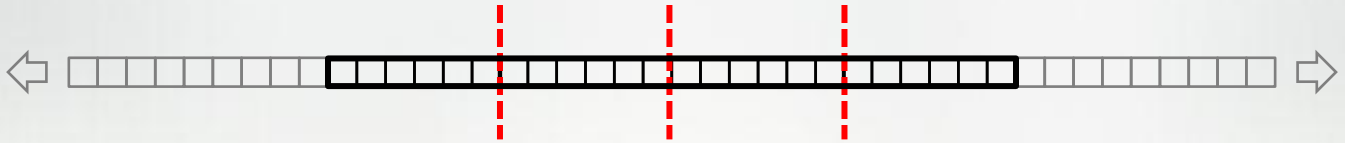
```
const ProblemSpace: domain(1, int(64))
                  = [1..m];

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

A = B + alpha * C;
```



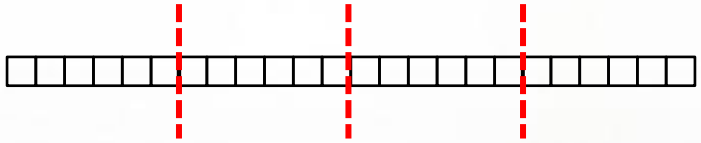
No domain map specified => use default layout

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

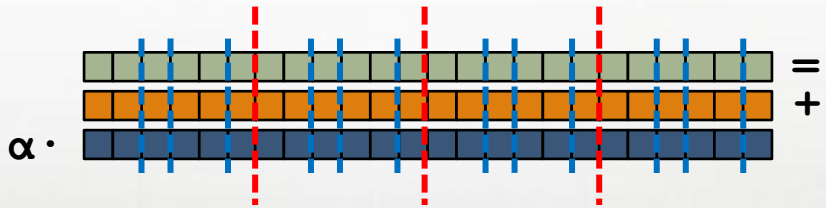
Global STREAM Triad in Chapel



```
const ProblemSpace: domain(1, int(64))
                    dmapped Block(boundingBox=[1..m])
                  = [1..m];
```

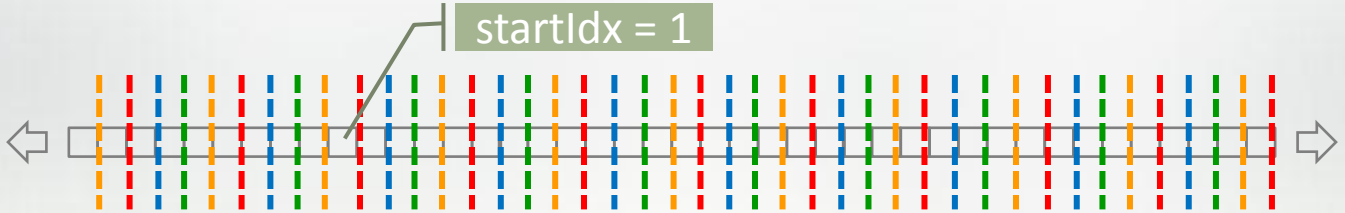


```
var A, B, C: [ProblemSpace] real;
```

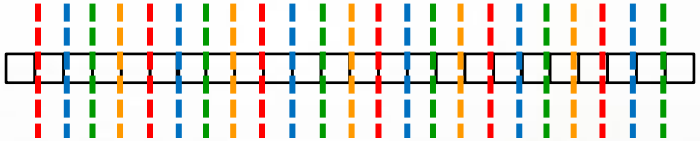



```
A = B + alpha * C;
```

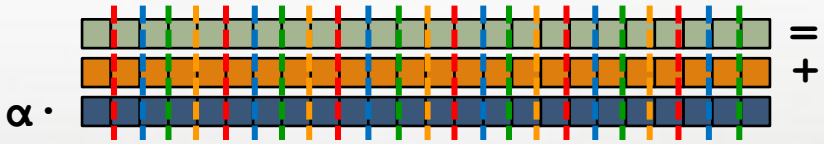
Global STREAM Triad in Chapel




```
const ProblemSpace: domain(1, int(64))
                    dmapped Cyclic(startIdx=1)
                  = [1..m];
```



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




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

Domain Maps

Domain Maps: "recipes for parallel/distributed arrays and domains (index sets)"

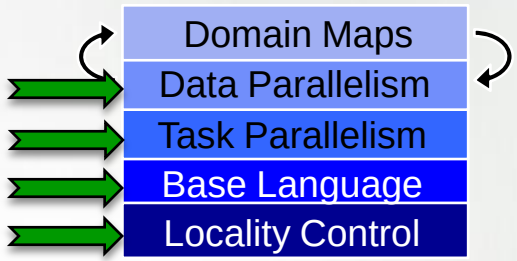
Domain maps define:

- Mapping of domain indices and array elements to locales
- Layout of arrays and index sets in memory
- Standard operations on domains and arrays
  - e.g, random access, iteration, slicing, reindexing, rank change, …

Domain Maps

Domain maps are written in Chapel using lower-level features:

- classes, iterators, type inference, generic types
- task parallelism
- locales and on-clauses
- other domains and arrays



Standard Chapel domain maps are written using the same mechanism an end-user would

Domain maps support a separation of roles:

- parallel-savvy domain scientist writes parallel code
- parallel computing expert writes and adds in domain maps

CRAY
THE SUPERCOMPUTER COMPANY

A9: We wanted a bunch of other stuff too…

- OOP, but optionally, not everywhere
- default arguments
- name-based argument passing
- namespace management
- ability to declare skyline arrays holistically
- a rich compile-time meta language
- operator/function overloading
- tuple types
- range types
- command-line settable values
- …

CRAY
THE SUPERCOMPUTER COMPANY

A10: As well as several features that remain open issues…

- exceptions/fault tolerance
- garbage collection
- parallel I/O
- visualization capabilities
- an interpreted environment
- transactional memory concepts
- interoperability

Tony's Interoperability Slide

Interoperability is crucial for any new language to succeed

- nobody can afford to start from scratch
- provides a way of bootstrapping a language
- supports user's ability to rewrite a portion of a larger application

Interoperability has not been a big part of our focus thus far

- fear of interoperability w/out performance resulting in "so what?"
- belief that we're on a path that will support interoperability well

Current Status:

- ability to declare and reference external C types, variables, functions
- some work to add a Chapel spoke to Babel by the LLNL team

Next steps (not yet scheduled/resourced):

- ability to make Chapel the callee rather than the caller (don't own main())
- MPI interoperability (in collaboration with Argonne)
- Python, Fortran interoperability (if not through Babel)

Chapel and Exascale

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

  (these were both considered “version 2.0” features)

  We are addressing these shortcomings as current/future work

Outline

- ✓ Motivation
- ✓ Programming Model Survey
- ✓ Chapel Overview
- ➢ Status and Future Directions
- ❑ Case Study: AMR
- ❑ Wrap-up

Chapel Status

The Good

- Most of the features you've heard about today are functional
- Interest in the language seems to be growing steadily
- Current doubts focus more on our ability to succeed in our current configuration rather than on the language design itself

The Bad

- Performance tends to be fairly binary: many planned improvements and optimizations remain
- Like any research software, there are bugs and dark corners

The Ugly

- HPCS funding only lasts another year

How can I help the tree grow?

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

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

Let others know about your interest in Chapel

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

Contribute to the project

Featured Collaborations

- **ORNL/Notre Dame** (Srinivas Sridharan, Jeff Vetter, Peter Kogge): Asynchronous software transactional memory over distributed memory
- **UIUC** (David Padua, Albert Sidelnik, Maria Garzarán): CPU-GPU computing
- **Sandia** (Kyle Wheeler, Rich Murphy): Chapel over Qthreads user threading
- **BSC/UPC** (Alex Duran): Chapel over Nanos++ user-level tasking
- **LTS** (Michael Ferguson): Improved I/O and strings
- **Argonne** (Rusty Lusk, Rajeev Thakur, Pavan Balaji): Chapel over MPICH
- **CU Boulder** (Jeremy Siek, Jonathan Turner): Interfaces, concepts, generics
- **U. Oregon/Paratools Inc.** (Sameer Shende): Performance analysis with Tau
- **U. Malaga** (Rafael Asenio, Maria Gonzales, Rafael Larossa): Parallel file I/O
- **PNNL/CASS-MT** (John Feo, Daniel Chavarria): Cray XMT tuning
- (your name here?)

Potential collaboration topics on Chapel webpage

Our Team

- Cray:


Brad Chamberlain


Sung-Eun Choi


Greg Titus


Vass Litvinov


Tom Hildebrandt

- External Collaborators:


Albert Sidelnik


Jonathan Turner


Srinivas Sridharan

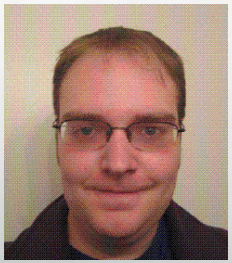
●●●

You? Your Student/Colleague?

- Interns:


Jonathan Claridge


Hannah Hemmaplardh


Andy Stone


Jim Dinan


Rob Bocchino


Mack Joyner

●●●

Outline

- ✓ Motivation
- ✓ Programming Model Survey
- ✓ Chapel Overview
- ✓ Status and Future Directions
- ➢ Case Study: AMR
- ❑ Wrap-up

AMR Framework in Chapel

Proposition: Evaluate Chapel productivity by having a grad student experienced in AMR write a framework in Chapel from scratch

- student was inexperienced in parallel programming and had never used Chapel before starting

Result: 4 months later, had a working, dimension-independent multicore-parallel AMR framework

Development overview

- Developed working, dimension-independent AMR infrastructure in just under 4 months, beginning with no Chapel experience
- Chapel made many challenges of AMR easy with little-to-no additional infrastructure required, while providing a large head start on the really hard parts
- Code size compares very favorably to existing AMR frameworks -- but keep in mind that the Chapel version is a “minimal” implementation!

Language	Parallelism	SLOC[1]	Tokens	Relative size (tokens)
C++ (D≤6) [3]	Dist. mem.	40200	261427	100%
Fortran (2D+3D) [2]	Serial	16562	151992	58%
2D		8297	71639	27%
3D		8265	80353	31%
Chapel (any D)	Shared mem.	1988	13783	**5%**

[1] source lines of code, [2] AMRClaw, [3] Chombo BoxTools+AMRTools

Slides Courtesy of Jonathan Claridge, from his SIAM CSE 2011 Minisymposium Talks

Development overview

- Developed working, dimension-independent AMR infrastructure in just under 4 months, beginning with no Chapel experience
- Chapel made many challenges of AMR easy with little-to-no additional infrastructure required, while providing a large head start on the really hard parts
- Code size compares very favorably to existing AMR frameworks -- but keep in mind that the Chapel version is a “minimal” implementation!

Language	Parallelism	SLOC[1]	Tokens	Relative size (tokens)
C++ (D≤6) [3]	Dist. mem.	40200	261427	100%
Fortran (2D+3D) [2]	Serial	16562	151992	58%
2D		8297	71639	27%
3D		8265	80353	31%
Chapel (any D)	Shared mem.	1988	13783	**5%**

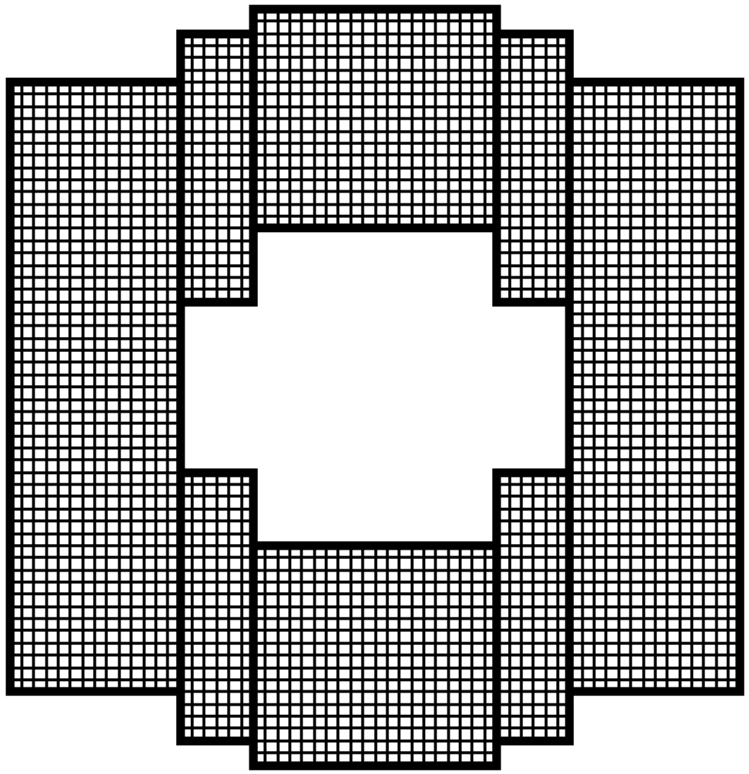
[1] source lines of code, [2] AMRClaw, [3] Chombo BoxTools+AMRTools

Reflects limitations of developer time, not Chapel itself

Slides Courtesy of Jonathan Claridge, from his SIAM CSE 2011 Minisymposium Talks

Levels

- Essentially a union of grids



```
var grids: domain(Grid);
```

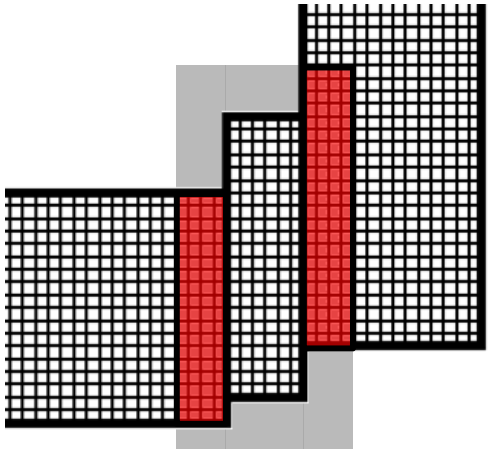
***Associative* domain**

- List of indices of *any* type
- Array and iteration syntax are **unchanged**

Slides Courtesy of Jonathan Claridge, from his SIAM CSE 2011 Minisymposium Talks

Levels: Sibling overlaps

- A grid's layer of ghost cells will, in general, overlap some of its siblings. Data will be copied into these overlapped ghost cells prior to mathematical operations.



- Calculating the **overlaps** between siblings:

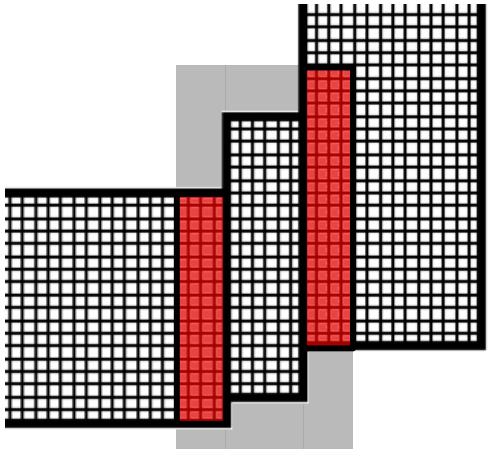
```
var neighbors: domain(Grid);
var overl                                    ension,stridable=true);

for sibl
  var overlap = extended_cells( sibling.cells );

  if overlap.numIndices > 0 && sibling != this {
    neighbors.add(sibling);
    overlaps(sibling) = overlap;
  }
}
```

Declare associative domain to store neighbors; initializes to empty.

Slides Courtesy of Jonathan Claridge, from his SIAM CSE 2011 Minisymposium Talks

Levels: Sibling overlaps

- A grid's layer of ghost cells will, in general, overlap some of its siblings. Data will be copied into these overlapped ghost cells prior to mathematical operations.



- Calculating the **overlaps** between siblings:

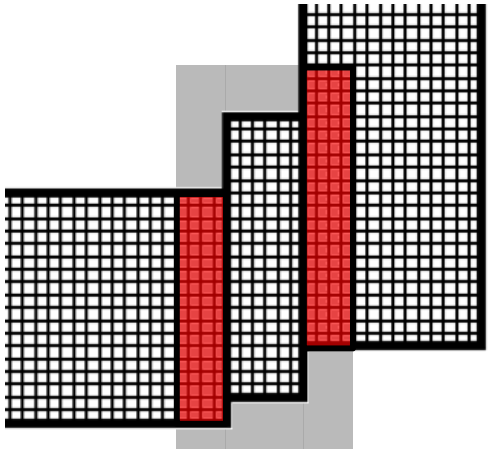
```
var neighbors: domain(Grid);
var overlaps:  [neighbors] domain(dimension,stridable=true);

for sibling
  var overl                                  g.cells );

  if overla                                  g != this {
    neighbors.add(sibling);
    overlaps(sibling) = overlap;
  }
}
```

An array of domains; stores one domain for each neighbor. New space allocated as `neighbors` grows.

Slides Courtesy of Jonathan Claridge, from his SIAM CSE 2011 Minisymposium Talks

Levels: Sibling overlaps

- A grid's layer of ghost cells will, in general, overlap some of its siblings. Data will be copied into these overlapped ghost cells prior to mathematical operations.



- Calculating the **overlaps** between siblings:

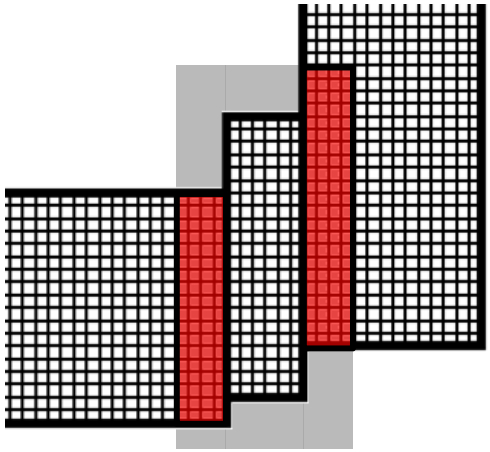
```
var neighbors: domain(Grid);
var overlaps:  [neighbors] domain(dimension,stridable=true);

for sibling in parent_level.grids {
  var over                                  ibling.cells );

  if overl                                  bling != this {
    neighb
    overlaps(sibling) = overlap;
  }
}
```

Loop over all grids on the same level, checking for neighbors.

Slides Courtesy of Jonathan Claridge, from his SIAM CSE 2011 Minisymposium Talks

Levels: Sibling overlaps

- A grid's layer of ghost cells will, in general, overlap some of its siblings. Data will be copied into these overlapped ghost cells prior to mathematical operations.



- Calculating the **overlaps** between siblings:

```
var neighbors: domain(Grid);
var overlaps:  [neighbors] domain(dimension,stridable=true);

for sibling in parent_level.grids {
  var overlap = extended_cells( sibling.cells );
}
```

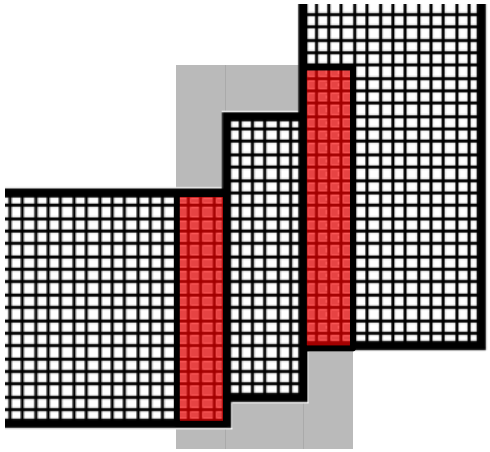
Computes intersection of the domains `extended_cells` and `sibling.cells`.

Take a moment to appreciate what this calculation would look like without domains!

Slides Courtesy of Jonathan Claridge, from his SIAM CSE 2011 Minisymposium Talks

Levels: Sibling overlaps

- A grid's layer of ghost cells will, in general, overlap some of its siblings. Data will be copied into these overlapped ghost cells prior to mathematical operations.



- Calculating the **overlaps** between siblings:

```
var neighbors: domain(Grid);
var overlaps:  [neighbors] domain(dimension,stridable=true);

for sibling in parent_level.grids {
  var overlap = extended_cells( sibling.cells );

  if overlap.numIndices > 0 && sibling != this {
    neighbors.add(sibling);
    overlaps(sibling) = overlap;
  }
}
```

If `overlap` is nonempty, and `sibling` is distinct from this grid, then update stored data.

Slides Courtesy of Jonathan Claridge, from his SIAM CSE 2011 Minisymposium Talks

Class `GridCFGhostRegion`

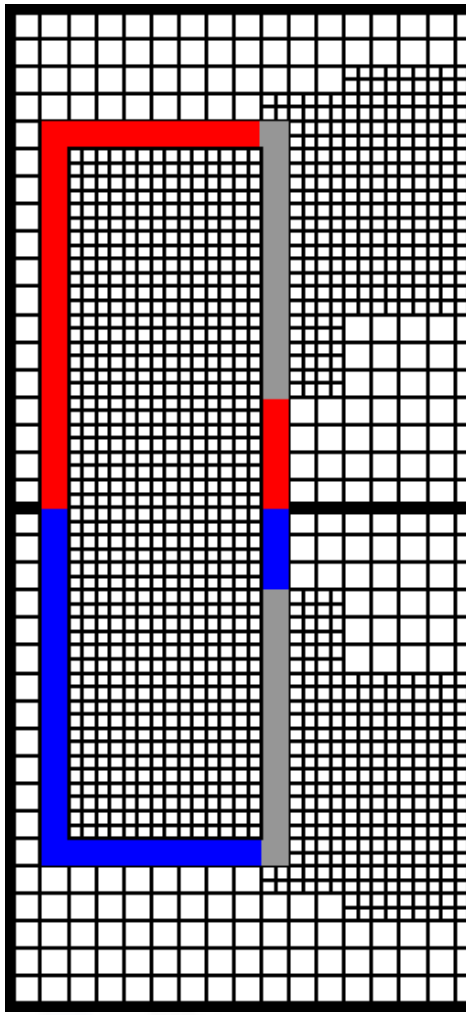
- Represents ghost cells of a fine grid that will receive data from “coarse neighbor” grids
- Fields are:

```
const grid: Grid;
```

The fine grid in question

```
const coarse_neighbors: domain(Grid);

const multidomains: [coarse_neighbors]
   MultiDomain(dimension, stridable=true);
```

- Constructor also needs to know:
  - `parent_level` of `grid`
  - `coarse_level`
  - `ref_ratio`, the refinement ratio between `coarse_level` and `parent_level`



Slides Courtesy of Jonathan Claridge, from his SIAM CSE 2011 Minisymposium Talks

Class GridCFGhostRegion

```
for coarse_grid in coarse_level.grids {
  var fine_intersection =
        grid.extended_cells( refine(coarse_grid.cells, ref_ratio) );

  if fine_intersection.numIndices > 0 {
    var boundary_multidomain = fine_intersection - grid.cells;

    for (_, region) in parent_level.sibling_ghost_regions(grid) do
      if fine_intersection(region).numIndices > 0 then
        boundary_multidomain.subtract(region);

    if boundary_multidomain.length > 0 {
       coarse_neighbors.add(coarse_grid);
       multidomains(coarse_grid) = boundary_multidomain;
    } else
      delete boundary_multidomain;
  }
}
```


Slides Courtesy of Jonathan Claridge, from his SIAM CSE 2011 Minisymposium Talks

Conclusions

What did Chapel do for us?

- Integer tuples and rectangular sets thereof are native data types
  - Drastically simplifies construction of MultiDomains
- Dimension-independence
  - After defining MultiDomains, spatial dimension only appears in variable declarations
- Clean, clear iteration syntax
  - Ability to define any object as an iterator with `these()` method

Recall Chapel's main goal:

- **Improve programmer productivity**

Slides Courtesy of Jonathan Claridge, from his SIAM CSE 2011 Minisymposium Talks

Next Steps for AMR code

- Evolve framework to support distributed memory
  - apply domain maps to distribute sets of grids across processors
  - key component: grid→locale hash function to specify mapping
  - I'd estimate this to require no more than a few hundred lines of code
- Performance measurements and optimizations
- Add more physics

Outline

- ✓ Motivation
- ✓ Programming Model Survey
- ✓ Chapel Overview
- ✓ Status and Future Directions
- ✓ Case Study: AMR
- ➢ Wrap-up

Summary

Higher-level programming models help science to be insulated from implementation

- yet, without necessarily abandoning control
- supports 90/10 rule well
- requires appropriate abstractions, separation of concerns
- Chapel does this via its multiresolution design

For exascale, programming models are likely to need:

- Various styles of parallelism: data, task, nested
- data-driven execution
- representations of hierarchical locality distinct from parallel execution model

Chapel is strong in first two; has a good start on third

What's Next?

- Improve performance
- Backfill missing features
- Target exascale node architectures
- Determine next source of funding

For More Information

- **Chapel Home Page** (papers, presentations, tutorials):
  http://chapel.cray.com
- **Chapel Project Page** (releases, mailing lists, code):
  http://sourceforge.net/projects/chapel/
- **General Questions/Info:**
  chapel_info@cray.com (or SourceForge chapel-users list)
- **AMR Framework:**
  https://chapel.svn.sourceforge.net/svnroot/chapel/trunk/test/studies/amr/
  Jonathan Claridge's dissertation (UW AMath); SIAM slides on Chapel website
- **Upcoming Tutorials:**
  SC11 (November, Seattle WA): full-day comprehensive tutorial
  + half-day broader engagement version

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