The Mother of All Chapel Talks

Brad Chamberlain
Cray Inc.

CSEP 524
May 20, 2010

Lecture Structure

1. Programming Models Landscape
2. Chapel Motivating Themes
3. Chapel Language Features
4. Project Status
5. Sample Codes

Chapel:
the Programming Models Landscape

DARPA HPCS CRAY THE SUPERCOMPUTER COMPANY

Disclaimers

This lecture's contents should be considered my personal opinions (or at least one facet of them) and not necessarily those of Cray Inc. nor my funding sources.

I work in high-performance scientific computing, so my talk may reflect my biases in that regard (as compared to, say, mainstream multicore programming). That said, there are probably more similarities than differences between the two worlds (esp. as time goes on).

Terminology: Programming Models

Programming Models:

1. abstract models that permit users to reason about how their programs will execute with respect to parallelism, memory, communication, performance, etc.

   e.g., “what should/can I be thinking about when writing my programs?”

2. concrete notations used to write programs

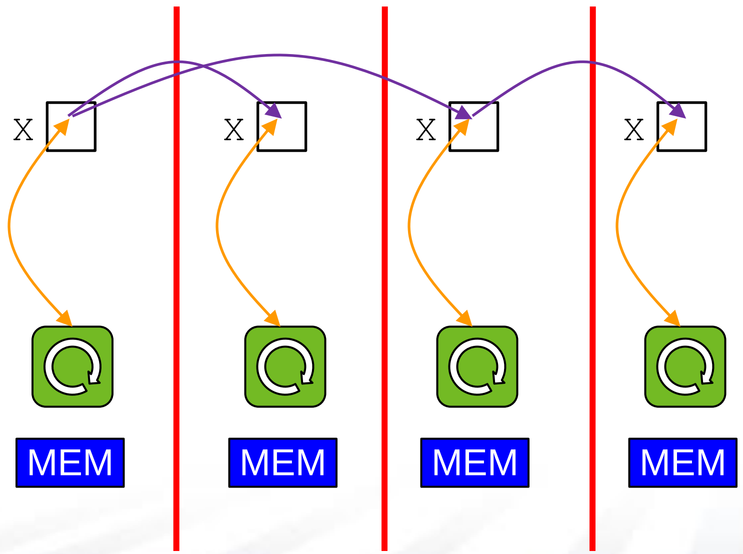
   i.e., the union of programming languages, libraries, annotations, …

HPC Programming Model Taxonomy (2010)

- **Communication Libraries**
  - **MPI**, PVM, SHMEM, ARMCI, GASNet, …
- **Shared Memory Programming Models**
  - **OpenMP**, pthreads, …
- **Hybrid Models**
  - **MPI+OpenMP**, MPI+CUDA, MPI+OpenCL, …
- **Traditional PGAS Languages**
  - **Unified Parallel C (UPC)**, **Co-Array Fortran (CAF)**, Titanium
- **HPCS Languages**
  - **Chapel**, **X10**, Fortress
- **GPU Programming Models**
  - **CUDA**, OpenCL, PGI annotations, CAPS, …
- **Others** (for which I don't have a neat unifying category)
  - Global Arrays, Charm++, ParalleX, Cilk, TBB, PPL, parallel Matlabs, Star-P, PLINQ, Map-Reduce, DPJ, Yada, …

Distributed Memory Programming

- **Characteristics:**
  - execute multiple binaries simultaneously & cooperatively
  - each binary has its own local namespace
  - binaries transfer data via communication calls
- **Examples: MPI**, PVM, SHMEM, …



MPI (Message Passing Interface) Evaluation

MPI strengths

- \+ users can get real work done with it
- \+ it is extremely general
- \+ it runs on most parallel platforms
- \+ it is relatively easy to implement (or, that's the conventional wisdom)
- \+ for many architectures, it can result in near-optimal performance
- \+ it can serve as a strong foundation for higher-level technologies

MPI weaknesses

- – encodes too much about "how" data should be transferred rather than simply "what data" (and possibly "when")
  - can mismatch architectures with different data transfer capabilities
- – only supports parallelism at the "cooperating executable" level
  - applications and architectures contain parallelism at many levels
  - doesn't reflect how one abstractly thinks about parallel algorithm
- – no abstractions for distributed data structures
  - places a significant bookkeeping burden on the programmer

Panel Question: What problems are poorly served by MPI?

My reaction: What problems are *well-served* by MPI?

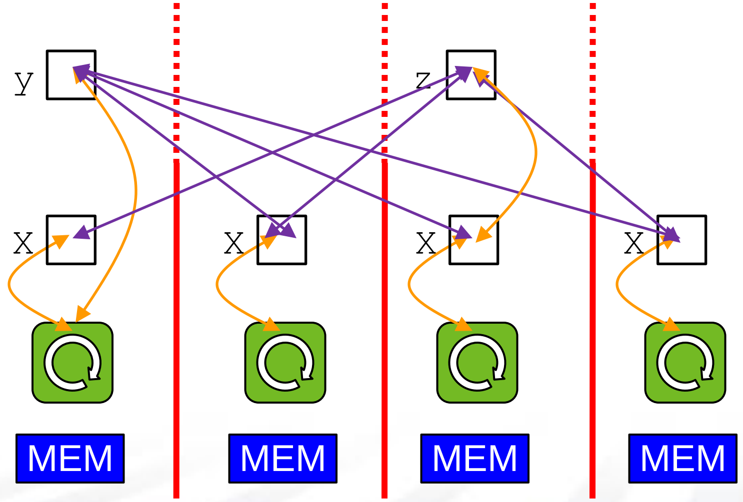
"well-served": MPI is a natural/productive way of expressing them

- **embarrassingly parallel:** arguably
- **data parallel:** not particularly, due to cooperating executable issues
  - bookkeeping details related to manual data decomposition
  - data replication, communication, synchronization
  - local vs. global indexing issues
- **task parallel:** even less so
  - *e.g.*, write a divide-and-conquer algorithm in MPI…
    …without MPI-2 dynamic process creation – yucky
    …with it, your unit of parallelism is the executable – weighty

- Its base languages have issues as well
  - **Fortran:** age leads to baggage + failure to track modern concepts
  - **C/C++:** impoverished support for arrays, pointer aliasing issues

(Traditional) PGAS Programming Models

- **Characteristics:**
  - execute an SPMD program (Single Program, Multiple Data)
  - all binaries share a namespace
    - namespace is partitioned, permitting reasoning about locality
    - binaries also have a local, private namespace
  - compiler introduces communication to satisfy remote references
- **Examples:** **UPC**, **Co-Array Fortran**, Titanium



PGAS: What's in a Name?

		memory model	*programming model*	*execution model*	*data structures*	*communication*
	MPI					
	OpenMP					
PGAS Languages	CAF	PGAS	Single Program, Multiple Data (SPMD)		co-arrays	co-array refs
	UPC				1D blk-cyc arrays/ distributed pointers	implicit
	Titanium				class-based arrays/ distributed pointers	method-based
	Chapel					

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 blk-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

PGAS Evaluation

PGAS strengths

+ Implicit expression of communication through variable names
  - decouples data transfer from synchronization
+ Ability to reason about locality/affinity supports scalable performance

Traditional PGAS language strengths

+ Elegant, reasonably minimalist extensions to established languages
+ Raises level of abstraction over MPI
+ Good support for distributed pointer-based data structures
+ Some support for distributed arrays

Traditional PGAS language weaknesses

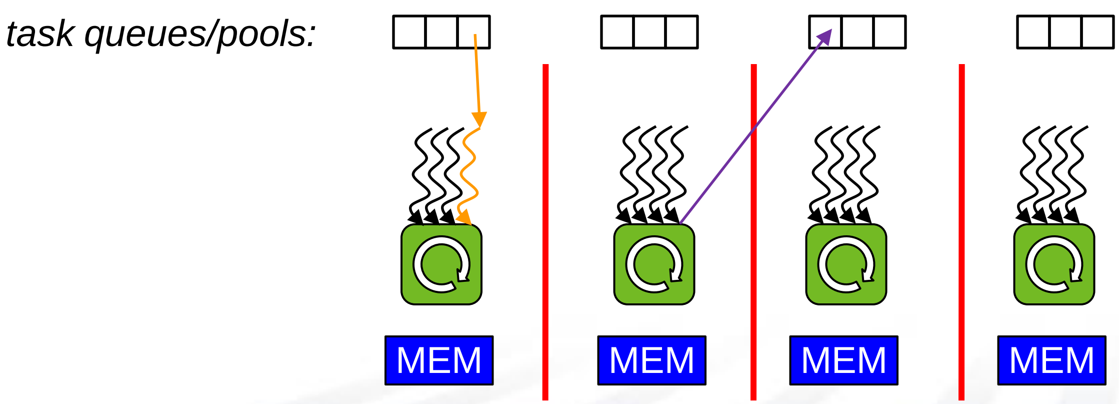
– **all:** Imposes an SPMD programming + execution model on the user

– **CAF:** Problems that don't divide evenly impose bookkeeping details
– **UPC:** Like C, 1D arrays seem impoverished for many HPC codes
– **Titanium:** Perhaps too pure an OO language for HPC
  – e.g., arrays should have value rather than reference semantics

post-SPMD/Asynchronous PGAS (APGAS)

- **Characteristics:**
  - uses the PGAS memory model
  - distinct concepts for locality vs. parallelism
  - programming/execution models are richer than SPMD
    - each unit of locality can execute multiple tasks/threads
    - nodes can create work for one another

- **Examples: Chapel**, **X10**, Fortress



ZPL

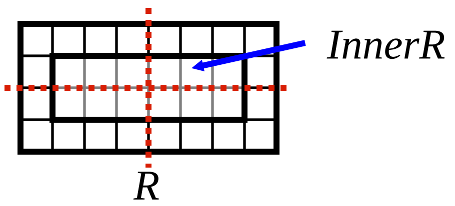
Main concepts:

- abstract machine model: CTA
- data parallel programming via global-view abstractions
  - regions: first-class index sets
- WYSIWYG performance model

ZPL Concepts: Regions

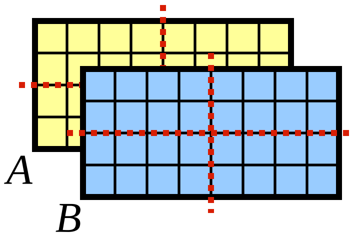
regions: distributed index sets…

```
region R      = [1..m, 1..n];
       InnerR = [2..m-1, 2..n-1];
```



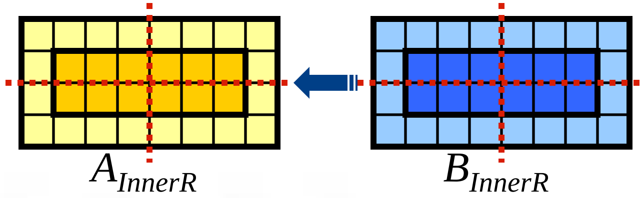
…used to declare distributed arrays…

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



…and computation over distributed arrays

```
[InnerR] A = B;
```

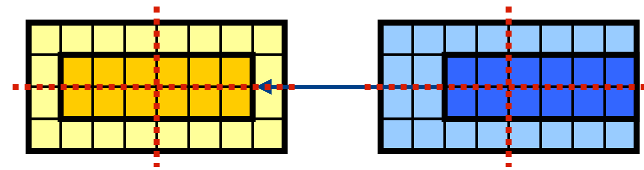


ZPL Concepts: Array Operators

array operators: describe nontrivial array indexing

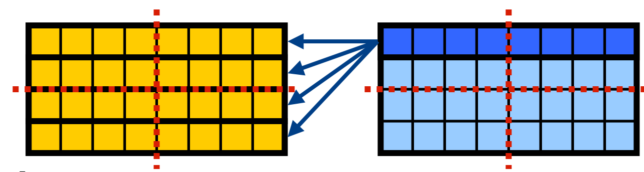
translation via *at operator* (`@`)

```
[InnerR] A = B@[0,1];
```



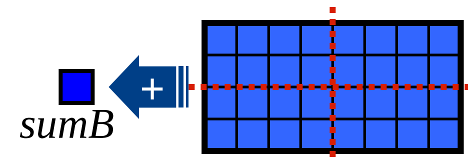
replication via *flood operator* (>>)

```
[R] A = >>[1, 1..n] B;
```



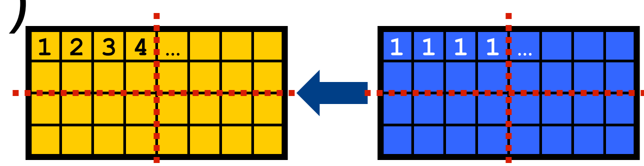
reduction via *reduction operator* (*op*<<)

```
[R] sumB = +<< B;
```



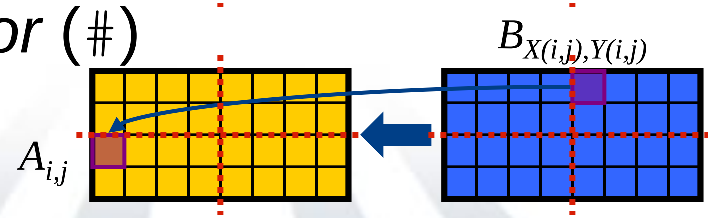
parallel prefix via *scan operator* (*op*||)

```
[R] A = +|| B;
```



arbitrary indexing via *remap operator* (#)

```
[R] A = B#[X,Y];
```



ZPL Concepts: Syntactic Performance Model

```
[InnerR] A = B;
```

No Array Operators ⇒ No Communication

```
[InnerR] A = B@[0,1];
```

At Operator ⇒ Point-to-Point Communication

```
[R] A = >>[1, 1..n] B;
```

Flood Operator ⇒ Broadcast (log-tree) Communication

```
[R] sumB = +<< B;
```

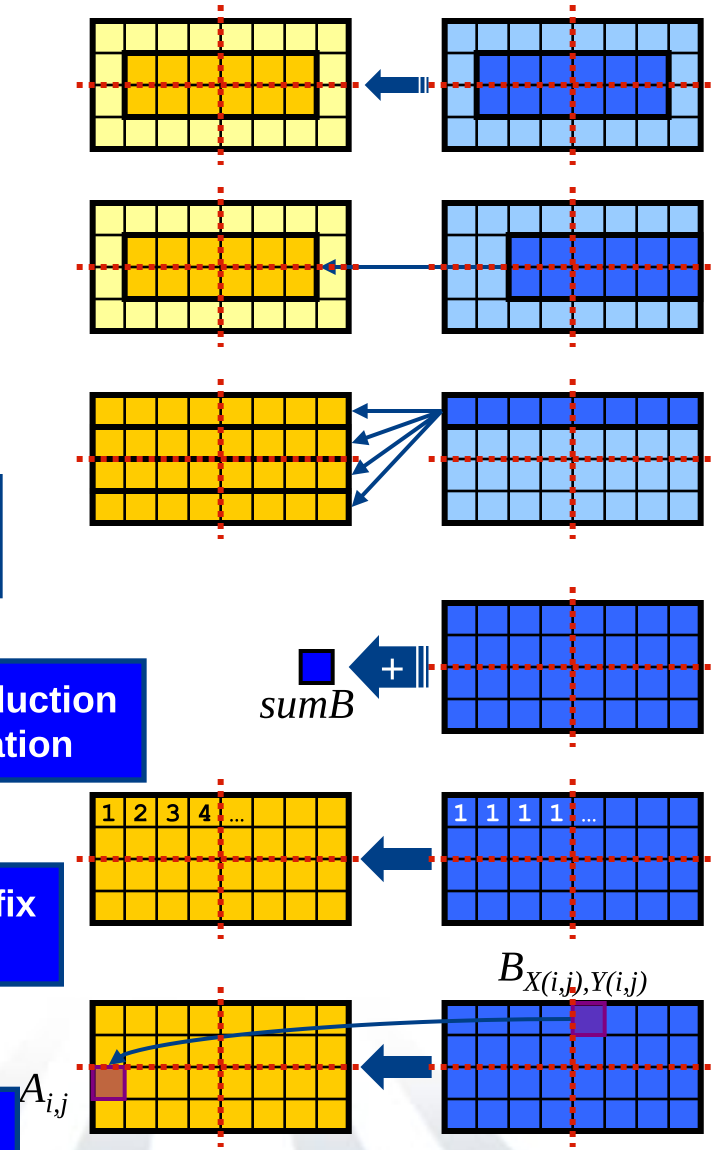
Reduce Operator ⇒ Reduction (log-tree) Communication

```
[R] A = +|| B;
```

Scan Operator ⇒ Parallel-Prefix (log-tree) Communication

```
[R] A = B#[X,Y];
```

Remap Operator ⇒ Arbitrary (all-to-all) Communication



Why Aren't We Done? (ZPL's Limitations)

- Only supports a single level of data parallelism
  - imposed by execution model: single-threaded SPMD
  - not well-suited for task parallelism, dynamic parallelism
  - no support for nested parallelism
- Distinct types & operators for distributed and local arrays
  - supports ZPL's WYSIWYG syntactic model
  - impedes code reuse (and has potential for bad cross-products)
  - annoying
- Only supports a small set of built-in distributions for arrays
  - e.g., Block, Cut (irregular block), …
  - if you need something else, you're stuck

ZPL's Successes

- First-class concept for representing index sets
  - ⇒ makes clouds of scalars in array declarations and loops concrete
  - ⇒ supports global-view of data and control; improved productivity
  - ⇒ useful abstraction for user and compiler

  ***The Design and Implementation of a Region-Based Parallel Language*.** Bradford L. Chamberlain. PhD thesis, University of Washington, November 2001

- Semantics constraining alignment of interacting arrays
  - ⇒ communication requirements visible to user and compiler in syntax

  ZPL's WYSIWYG performance model. Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. In *Proceedings of the IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments*, 1998.

- Implementation-neutral expression of communication
  - ⇒ supports implementation on each architecture using best paradigm

  A compiler abstraction for machine independent parallel communication generation. Bradford L. Chamberlain, Sung-Eun Choi, and Lawrence Snyder. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1997.

- A good start on supporting distributions, task parallelism

  Steven J. Deitz. ***High-Level Programming Language Abstractions for Advanced and Dynamic Parallel Computations*.** PhD thesis, University of Washington, February 2005.

Chapel and ZPL

- Base Chapel's data parallel features on ZPL's successes…
  - carry first-class index sets forward
    - unify with local arrays for consistency, sanity
      $\Rightarrow$ no syntactic performance model
    - generalize to support richer data aggregates: sets, graphs, maps
  - remove alignment requirement on arrays for programmability
      $\Rightarrow$ no syntactic performance model
    - yet, preserve user/compiler ability to reason about aligned arrays
  - preserve implementation-neutral expression of communication
  - support user-defined distributions for arrays

- …while expanding to several areas beyond ZPL's scope
  - task parallelism, concurrency, synchronization, nested parallelism, OOP, generic programming, modern syntax, type inference, …

A Design Principle HPC should revisit

"Support the general case, optimize for the common case"

Claim: a lot of suffering in HPC is due to programming models that focus too much on common cases:

- *e.g.*, only supporting a single mode of parallelism
- *e.g.*, exposing too much about target architecture and implementation

Impacts:

- hybrid models needed to target all modes of parallelism (HW & SW)
- challenges arise when architectures change (e.g., multicore, GPUs)
- presents challenges to adoption ("linguistic dead ends")

That said, this approach is also pragmatic

- particularly given community size, (relatively) limited resources
- and frankly, we've achieved a lot of great science when things fit

But we shouldn't stop striving for more general approaches