Chapel's Data-Centric Approach to Parallelism and Locality

Brad Chamberlain, Cray Inc.

Future Approaches to Data-Centric Programming for Exascale

May 20th, 2011

Sustained Performance Milestones

1 GF – 1988: Cray Y-MP; 8 Processors

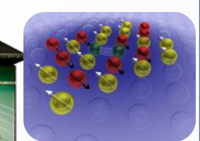
- Static finite element analysis

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

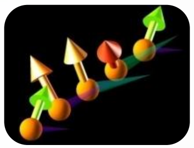
Sustained Performance Milestones

1 GF – 1988: Cray Y-MP; 8 Processors

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

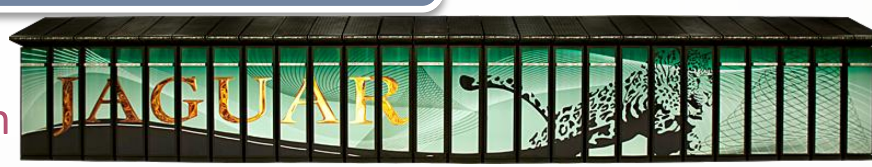
1 TF – 1998: Cray T3E; 1,024 Processors

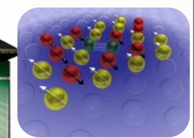
- Modeling of metallic magnet atoms
- Fortran + MPI (?)



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/??? + ???

Why Do HPC Programming Models Change?

HPC has traditionally given users…

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

…ones that are closely related to the underlying hardware

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

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

Thesis: Control-centric notations constrain implementation options more, for better or worse; for the purposes of providing optimization opportunities and porting to exascale, mostly "worse." Data-centric models could improve the situation.

C+MPI: Control-centric Sum of Squares

```
int n = computeProbSize(),
    myN = computeMyProbSize(n);
double A[myN], B[myN];
double sumOfSquares, mySumOfSquares;

for (i=0; i<myN; i++)
  mySumOfSquares += A[i]*A[i] + B[i]*B[i];

MPI_Reduce(&mySumOfSquares, &sumOfSquares,
           MPI_SUM, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Global and Local Declarations

Local Accumulation

Global Combining

C+MPI: What is Specified?

```
int n = computeProbSize(),
    myN = computeMyProbSize(n);
double A[myN], B[myN];
double sumOfSquares, mySumOfSquares;


for (i=0; i<myN; i++)
  mySumOfSquares += A[i]*A[i] + B[i]*B[i];



MPI_Reduce(&mySumOfSquares, &sumOfSquares,
           MPI_SUM, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Specified

- *Unit of Parallelism:* Cooperating Executable (via use of MPI)
- *Other Decisions:* Data Decomposition, Local Computation Style, and Synchronous Communication (via program text)

C+MPI: What is Left Unspecified?

```
int n = computeProbSize(),
    myN = computeMyProbSize(n);
double A[myN], B[myN];
double sumOfSquares, mySumOfSquares;

for (i=0; i<myN; i++)
  mySumOfSquares += A[i]*A[i] + B[i]*B[i];


MPI_Reduce(&mySumOfSquares, &sumOfSquares,
           MPI_SUM, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Unspecified

- *Communication Details:* All-to-one? Combining Tree? What arity? Who does each node send to/recv from?

(and with additional software engineering, we could arguably do more…)

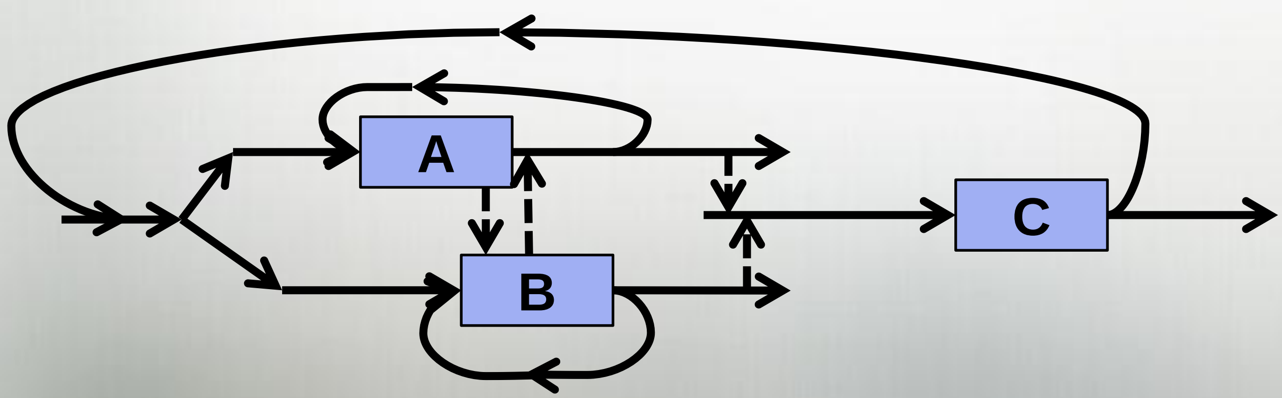
PGAS: A Step in the Right Direction

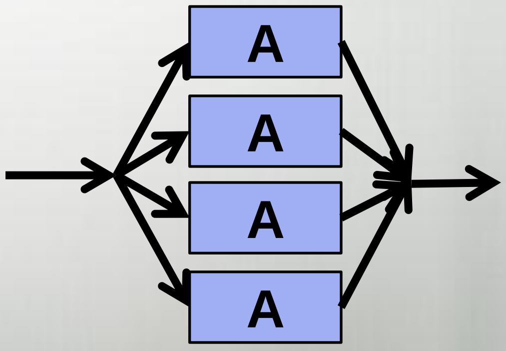
Traditional PGAS Languages: UPC, CAF, Titanium, GA

- Communication expressed as variable accesses
  - says more about *what* should be moved than *how* (or, arguably, *when*) – more data-centric
    - synchronization is decoupled from data transfer
    - arguably more amenable to compiler optimization
- Yet, control and data models are still fairly restricted
  - SPMD model of parallelism only
  - limited support for distributed arrays

Chapel: A Next-Generation PGAS Language

- A new parallel language being developed by Cray Inc. under DARPA HPCS
- a PGAS language, but non-traditional:
  - rich array types: multidimensional, sparse, associative, unstructured
  - explicit concepts for describing locality/affinity
    - e.g., *locale* type represents architectural locality
  - more general/dynamic/multithreaded parallelism





Global-View: Data-centric Sum of Squares

```
config const n = computeProblemSize();
const D = [1..n];
var A, B: [D] real;


const sumOfSquares = + reduce (A**2 + B**2);
```

Global Declarations

Global Reduction

For the purposes of this talk, global-view ≈ data-centric

Global-View: What is Specified?

```
config const n = computeProblemSize();
const D = [1..n];
var A, B: [D] real;


const sumOfSquares = + reduce (A**2 + B**2);
```

Specified

- *Intention:* “We want to compute a sum reduction”

Global-View: What is Left Unspecified?

```
config const n = computeProblemSize();
const D = [1..n];
var A, B: [D] real;


const sumOfSquares = + reduce (A**2 + B**2);
```

Unspecified

- *Unit of Parallelism:* serial? shared-memory parallel? distributed memory parallel? both? accelerators? Cray XMT?
- *Data Decomposition:* local? blocked? block-cyclic? recursive bisection? what memory types?
- *Local Computation Style:* statically partitioned? dynamically? details?
- *Communication Details:* All-to-one? Combining Tree? What arity? Who does each node send to/recv from?
    - implemented using message passing? puts/gets? active messages?

Global-View: Performance Implications

No reason to believe performance must suffer

"High-level doesn't necessarily mean slow if your abstractions are designed to map efficiently."

-Pat Hanrahan (my wording)

```
const sumOfSquares = + reduce (A**2 + B**2);
```

"Just because HPF and Java failed to revolutionize HPC doesn't mean all new high-level languages are destined to fail."

-Chamberlain corollary

Global-View: Not Sufficient For Success

```
config const n = computeProblemSize();
const D = [1..n];
var A, B: [D] real;

const sumOfSquares = + reduce (A**2 + B**2);
```

How do we implement this global-view operation in practice?

ZPL: Block-distributed arrays, serial per node, … (inflexible)

HPF: Not particularly well-defined ("trust the compiler")

Chapel: Very well-defined and flexible… stay tuned…

Outline

- ✓ Control- vs. data-centric motivation
- Up with data-centrism!
- Control-centric Chapel
- Implementing data-centric concepts
- Conclusion

Chapel's Global-View: Other Cool Stuff

```
config const n = computeProblemSize();
const D = [1..n, 1..n];
var A, B: [D] real;


const sumOfSquares = + reduce (A**2 + B**2);
```

Computation is Rank-Independent

(and with a bit more work on the user's part, the declarations could be too)

Chapel's Global-View: Other Cool Stuff

```
config const n = computeProblemSize();
const D = [1..n, 1..n];
var A, B: [D] real;


const sumOfSquares = + reduce forall ij in D do
                                (A[ij]**2 + B[ij]**2);

                    // or:  forall (a,b) in (A,B) do
                    //              (a**2 + b**2);
```

Computation also has rank-independent loop-based forms

Chapel's Global-View: Other Cool Stuff

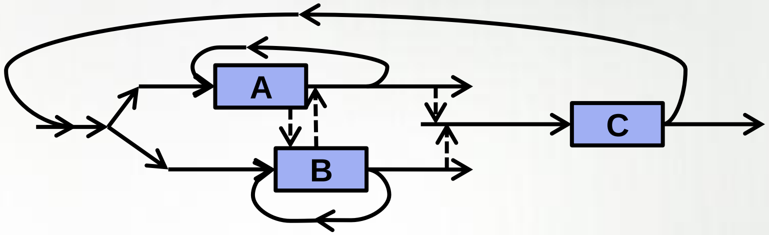
```
config const n = computeProblemSize();
const D = [1..n, 1..n];
var A, B: [D] real;

var sumOfSquares$: sync real;

begin sumOfSquares$ = + reduce forall (a,b) in (A,B)
  do
                                                  (a**2 + b**2);

doSomeOtherStuff(…);

…sqrt(sumOfSquares$)…
```



Fire off asynchronous task, storing result in sync (full/empty) variable

Read of sync variable blocks until result has been written.

Another Example: Partial Reductions

- Partial reductions can be expressed using complete reductions:

```
forall row in 1..n do
  sum[row] = + reduce [col in 1..n]
                 (A[row,col]**2 + B[row,col]**2);
```

…yet this strongly implies…

…row-centric iteration over arrays A and B

…a series of independent reduction steps across processors

Another Example: Partial Reductions

- Partial reductions can be expressed using complete reductions:

```
forall row in 1..n do
  sum[row] = + reduce [col in 1..n]
                 (A[row,col]**2 + B[row,col]**2);
```

…yet this strongly implies…

…row-centric iteration over arrays A and B

…a series of independent reduction steps across processors

- Supporting a more data-centric/global-view partial reduction conveys much more to the compiler:

```
sum = + reduce(dim=2) (A**2 + B**2);
```

…iterate over A and B in whatever order is most natural/efficient

…can easily perform single reductions over vectors of data

A Final Example: Sparse Arrays

- CSR representation ⇒ indirect indexing

```
for i in 1..n do
  for j in rowstart[i]..rowstart[i+1]-1 do
    y[i] = A[j] * x[colidx[j]];
```

- OOP representation ⇒ field/method indirection

```
for i in 1..n do
  for j in A.genCols() do
    y[i] = A.access(i,j) * x[j];
```

A Final Example: Sparse Arrays

- CSR representation ⇒ indirect indexing

```
for i in 1..n do
  for j in rowstart[i]..rowstart[i+1]-1 do
    y[i] = A[j] * x[colidx[j]];
```

- OOP representation ⇒ field/method indirection

```
for i in 1..n do
  for j in A.genCols() do
    y[i] = A.access(i,j) * x[j];
```

- Sparse support within language ⇒

  …users rejoice due to natural data-centric syntax

  …semantics exposed to the compiler for optimization

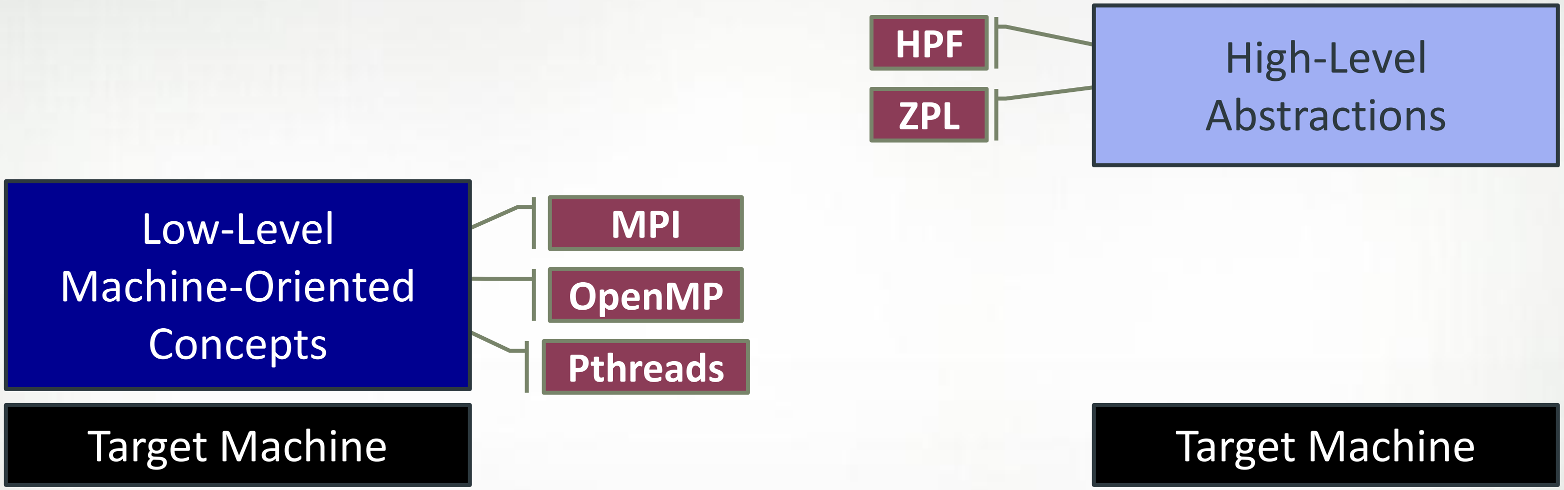
```
y = + reduce(dim=2) [(i,j) in D.domain] A[i,j]*x[j];
```

But what about the Control-Centric Programmer?

“But Brad, are you forgetting that we work in a community of control freaks?”

Chapel’s response: Multiresolution Language Design

Multiresolution Language Design: Motivation



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

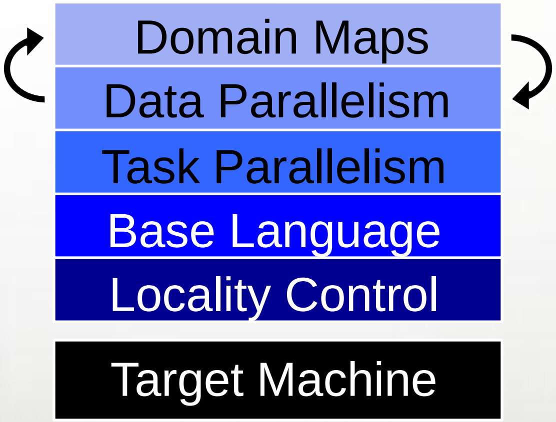
"Why don't I have more control?"

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

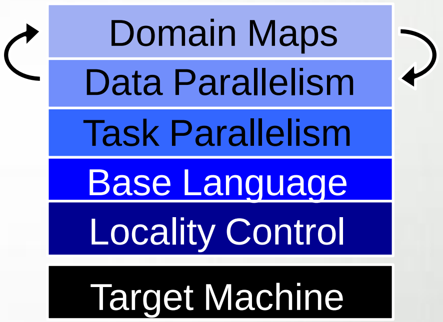
Chapel language concepts



- separate concerns appropriately for clean design
  - yet permit the user to intermix the layers arbitrarily

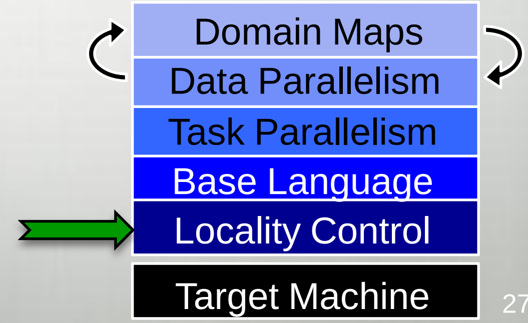
Outline

- ✓ Control- vs. data-centric motivation
- ✓ Up with data-centrism!
- Control-centric Chapel: Sample features from…
  - locality concepts
  - base language
  - task parallelism
- Implementing data-centric concepts
- Conclusion



Chapel's *Locale* Type

- **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



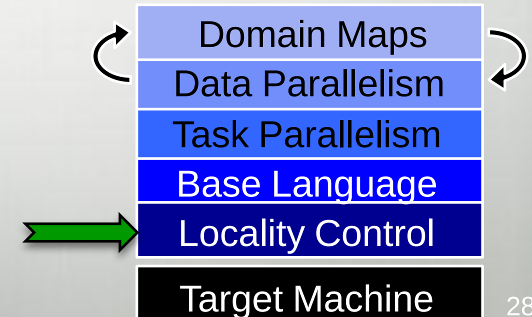
The On Statement

- Syntax

```
on-stmt:
  on expr do stmt
```

- Semantics
  - Executes *stmt* on the locale that stores *expr*
- Control-centric Example

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



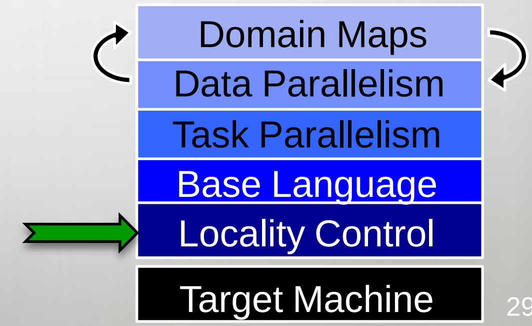
The On Statement

- Syntax

```
on-stmt:
  on expr do stmt
```

- Semantics
  - Executes *stmt* on the locale that stores *expr*
- Data-centric Example

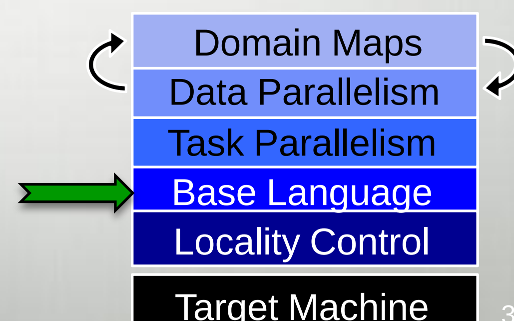
```
writeln("start on locale 0");
on A[i] do
  writeln("now on the locale that owns A[i]");
writeln("on locale 0 again");
```



Sample Base Language Features

```
//
// iterator to generate fibonacci numbers
//
iter fib(n) {                       // define an iterator
  var current = 0, next = 1;        // use type inference
  for 1..n {
    yield current;                  // generate next result
    current += next;
    next <=> current;               // swap operator
  }
}

for f in fib(10) do writeln(f);  // invoke iterator
```



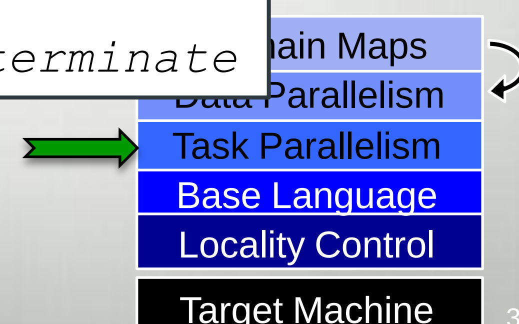
Loop-Based Tasking: Coforall

- Syntax

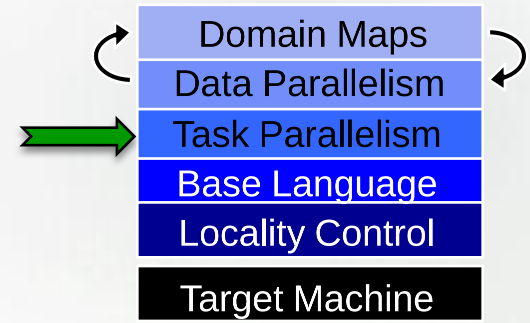
```
coforall-loop:
  coforall index-expr in iteratable-expr { stmt-list }
```

- Semantics
  - Create a task for each iteration in *iteratable-expr*
  - Parent task waits for all sub-tasks to complete
- Example

```
begin producer();
coforall i in 1..numConsumers {
  consumer(i);
}  // wait here for all consumers to terminate
```



Synchronization Variables



- Syntax

```
sync-type:
  sync type
```

- Semantics
  - Stores *full/empty* state along with normal value
  - Default read blocks until *full,* leaves *empty*
  - Default write blocks until *empty,* leaves *full*
  - Other variations supported via method calls (e.g., .readFF())
- Example: Capture future results

```
var future$: sync real;

begin future$ = compute();
computeSomethingElse();
useComputedResults(future$);   // data-centric synch.
```

Global-View Need Not Preclude Control

A language can support both global- and local-view programming

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

proc MySPMDProgram(me, numCopies) {
  ...
}
```

Global-View Need Not Preclude Control

A language can support both global- and local-view programming (and even message passing)

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

proc MySPMDProgram(me, numCopies) {
  MPI_Reduce(mySumOfSquares, sumOfSquares,
             MPI_SUM, MPI_DOUBLE, 0,
             MPI_COMM_WORLD);
}
```

Outline

- ✓ Control- vs. data-centric motivation
- ✓ Up with data-centrism!
- ✓ Control-centric Chapel
- Implementing data-centric concepts
- Conclusion

Global-View: How Implemented in Chapel?

```
config const n = computeProblemSize();
const D = [1..n];
var A, B: [D] real;


const sumOfSquares = + reduce (A**2 + B**2);
```

Chapel: Defined in terms of *zippered iteration* semantics

Global-View: How Implemented in Chapel?

```
config const n = computeProblemSize();
const D = [1..n];
var A, B: [D] real;


const sumOfSquares = + reduce forall (a,b) in (A,B) do
                                   (a**2 + b**2);
```

Since A is first array in zippering, it is the *leader*.

Chapel: Defined in terms of *zippered iteration* semantics

…which in turn are defined using *leader/follower iterators* and *domain maps*

Leader/Follower Iterators

- All zippered forall loops are defined in terms of leader/follower iterators:
  - *leader iterators:* specify parallelism, assign iterations to tasks
  - *follower iterators:* serially execute work generated by leader

- *Conceptually*, the Chapel compiler translates:

```
forall (a,b) in (A,B) do
  (a**2 + b**2);
```

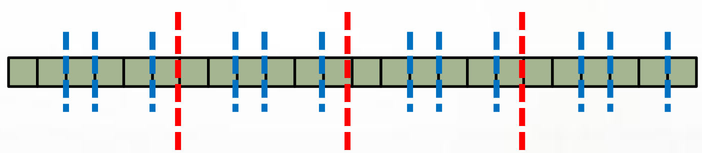
into:

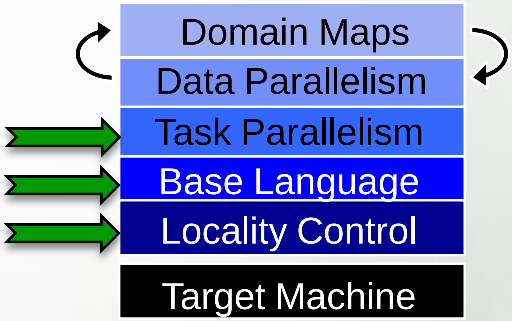
```
for work in A.lead() do
  for (a,b) in (A.follow(work), B.follow(work)) do
    yield a**2 + b**2;
```

Defining Leaders and Followers

Leader iterators are defined using task/locality features:

```
iter BlockArr.lead() {
  coforall loc in Locales do
    on loc do
      coforall tid in here.numCores do
        yield computeMyBlock(loc.id, tid);
}
```





Follower iterators simply use serial features:

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

Benefits of Zippered Iteration Semantics

- Chained whole-array operations are implemented element-wise rather than operator-wise.

  ⇒ No temporary arrays required by semantics

`A**2 + B**2` ~~⇒~~ `T1 = A**2;`

```
               T2 = B**2;
               T3 = T1 + T2;
```

```
⇒ forall (a,b) in (A,B) do (a**2 + b**2);
```

- Provides an execution model that one can reason about and control using *domain maps*.

Domain Maps

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

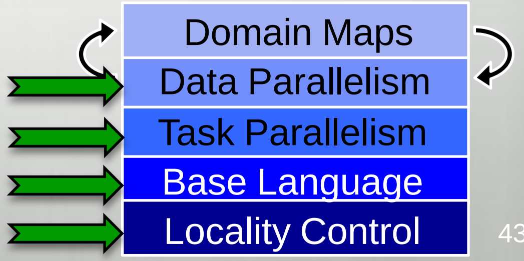
Domain maps define:

- 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

    ^^^ including leader/follower iterators!

Domain maps are built using Chapel concepts

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



Layouts and Distributions

Domain Maps fall into two major categories:

layouts: target a single locale (memory)

- e.g., a desktop machine or multicore node
- **examples:** row- and column-major order, tilings, compressed sparse row, space-filling curves

distributions: target distinct locales (memories)

- e.g., a distributed memory cluster or supercomputer
- **examples:** Block, Cyclic, Block-Cyclic, Recursive Bisection, …

Global-View: How Implemented in Chapel?

```
config const n = computeProblemSize();
const D = [1..n];
var A, B: [D] real;
```

No domain map ⇒ use default layout

```
const sumOfSquares = + reduce (A**2 + B**2);
```

Default Domain Map/Layout

```
config const n = computeProblemSize();
const D = [1..n];
var A, B: [D] real;
```

No domain map ⇒ use default layout

The default layout:

- targets local memory and processsors only
- its leader iterator…

  …by default, uses #tasks = #cores

  …decomposes indices/elements using static blocking

Controlling Data Parallelism

Q: *“But what if I don’t like the approach taken by an array’s leader iterator? (or rather, its domain map’s)”*

A: Several possibilities…

Controlling Data Parallelism

```
config const n = computeProblemSize();
const D = [1..n];
var A, B: [D] real;


const sumOfSquares = + reduce forall (b,a) in (B,A) do
                                        (a**2 + b**2);
```

Make something else the leader.

(moot in this case – B also uses default domain map)

Controlling Data Parallelism

```
config const n = computeProblemSize();
const D = [1..n];
var A, B: [D] real;


const sumOfSquares = + reduce forall (a,b)
                                in (myLdr(A,blk=64), B)
                              do (a**2 + b**2);
```

Invoke some other leader iterator explicitly (perhaps one that you wrote yourself).

Controlling Data Parallelism

```
config const n = computeProblemSize();
const D = [1..n] dmapped BlockCyclic(start=1,
                                    blocksize=64);
var A, B: [D] real;


const sumOfSquares = + reduce forall (a,b) in (A,B) do
                                (a**2 + b**2);
```

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

Controlling Data Parallelism: Hmm…

- We can still control an array's decomposition, layout
- We can still control parallelism and work mapping
  - even explicitly if we want to (SPMD-in-Chapel)

⇒ Data-centric programming can peacefully coexist with control-centric programming

- Yet, by using domain maps & iterators, we…
  - insulate our algorithm from its implementation details
  - make the code more portable, readable, maintainable, etc.
  - support distinct roles/levels of experties
    - parallel experts write domain maps
    - parallel-aware users utilize them
  - and really, isn't that what **productivity** is all about?

For More Information on Domain Maps

- HotPAR'10 paper: *User-Defined Distributions and Layouts in Chapel: Philosophy and Framework*
- Next week's CUG'11 paper/talk: *Authoring User-Defined Domain Maps in Chapel*
- For Chapel users…
  - 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

- ✓ Control- vs. data-centric motivation
- ✓ Up with data-centrism!
- ✓ Control-centric Chapel
- ✓ Implementing data-centric concepts
- Conclusion

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

Summary

Data-centric programming models help science to be insulated from implementation

- yet, without necessarily abandoning control
- supports 90/10 rule well

Building data-centric programming using control-centric features is beneficial

- Results in execution models that are more general, dynamic, and loosely-coupled than today's
- Separates concerns and programmer roles
- Serves as a good foundation for exascale
- Multiresolution philosophy is key here

For More Information

- **Chapel Home Page** (papers, presentations, tutorials):
  http://chapel.cray.com
- **Chapel Project Page** (releases, source, mailing lists):
  http://sourceforge.net/projects/chapel/
- **General Questions/Info:**
  chapel_info@cray.com (or chapel-users mailing list)

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