NAS MG:

Productivity Challenges and a Chapel-based Solution

LaR 2005
November 12, 2005

Brad Chamberlain
Cray Inc.

(excerpts from) Brad's Wishlist for Parallel Languages / MG

CRAY

The SPMD Programming Model

- Most current parallel languages require code to be written on a per-processor basis
- One impact of this is that data structures must be manually *fragmented* between processors
  - *e.g.*, sequential MG pseudo-code:

```
var V: [1..nx, 1..ny, 1..nz] float;
```

  - fragmented pseudo-code:

```
var mynx: integer = nx/procsx,
    myny: integer = ny/procsy,
    mynz: integer = nz/procsz;

var V: [0..mynx+1, 0..myny+1, 0..mynz+1] float;
```

- Note that the code shown will only work if each dimension divides evenly
  - To make it general, more work/code/details would be required
  - For this reason, the NPB typically build-in assumptions about problem size & #processors (*e.g.*, powers-of-2)

The SPMD Programming Model

- Most current parallel languages require code to be written on a per-processor basis
- Another impact of this is that codes are typically written so that each processor will execute its own copy
  - *i.e.*, write one program, run *p* copies of it
  - compute and branch on the unique processor ID to get non-trivial behavior
  - specify communication explicitly between the program images
  - *e.g.*, fragmented pseudocode:

```
if (iHaveLeftNeighbor) {
  send(leftNeighbor, V[1..mynx, 1, 1..mynz];
  recv(leftNeighbor, V[1..mynx, 0, 1..mynz];
} // repeat for other 25 logical neighbors
```

  - iterate over local chunks of data

```
forall (i,j,k) in [1..mynx, 1..myny, 1..mynz] {
  V(i,j,k) = …;
}
```

CRAY

Global-view Languages

- In contrast, global-view parallel languages are written such that one copy of them is run
  - parallelism contained within program text itself
  - code looks more similar to sequential, less impacted by per-processor implementation details
  - burden of managing details and communication is put on the compiler and runtime
    - (potentially guided by the programmer)
  - e.g., previous example in global-view pseudocode

```
var FineGrid = [1..nx, 1..ny, 1..nz]
               distributed Block(3) to ProcGrid;

var V: [FineGrid] float;

forall (i,j,k) in FineGrid {
  V(i,j,k) = …;
```

Classifying Parallel Languages

fragmented languages	***global-view languages***
MPI	**HPF**
SHMEM	**ZPL**
Co-Array Fortran	**Sisal**
UPC	**NESL**
Titanium	**MTA C/Fortran**
	OpenMP (often)
	Matlab (trivially)
	Chapel

MG Desiderata: Global-View

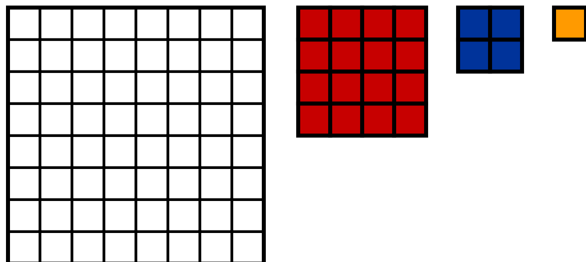
- Ability to describe algorithm using a global view
  - no need to chop arrays into per-processor pieces
  - no requirement to explicitly specify communication
  - No constraints or pressures on problem size due to #processors
- Support these features with minimal impact on performance

- *Most scalable implementations of MG are written in fragmented languages*
- *Most implementations of MG constrain the problem size*

CRAY

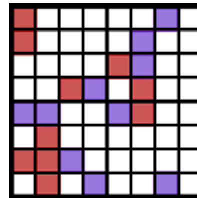
Arrays

- Good arrays are a must for scientific computation
  - multidimensional
  - dynamically-sized
  - resizable
  - rich support for array (*not matrix*) operations
  - composable with each other & other data structures
  - ability to allocate hierarchical & skyline arrays without playing games…
    - mapping everything to a 1D array
    - requiring pointers to pointers to …
  - distributable across a number of processors in various ways
  - distributed and stored in a user-specified manner?



Sparse Arrays

- Good *sparse* arrays are also a must for scientific computation
  - should be built-in
  - support a multitude of sparse storage formats
  - support user-defined sparse formats?
  - code reuse between sparse and dense arrays
  - composable with each other, dense arrays



MG Desiderata: Arrays

- Ability to allocate 3D arrays
- Ability to specify problem size dynamically
- Ability to specify #levels of hierarchy dynamically
- Ability to create hierarchical arrays easily
- Ability to distribute arrays in a multidimensional blocked manner (or otherwise?)
- Ability to create 3D sparse arrays
- Ability to write code generically for sparse or dense arrays
- Support these features with a minimal impact on performance

- *Most implementations of MG…*
  …require problem size to be specified at compile-time
  …use arcane hierarchical array formats
  …do not utilize sparse arrays for the input

CRAY

Processors

- Requiring #processors to be specified at compile-time is onerous
  - requires recompilation for each configuration
  - doesn't support dynamicism of real machines
  - parallel languages shouldn't require it
  - programming models shouldn't make it virtually necessary
- Virtual topology of processors should be specifiable at runtime
  - given p^3 processors, could think of them as…
    - a p^3 x 1 x 1 grid (or 1 x p^3 x 1, or …) for 1D decompositions
    - a p^2 x p x 1 grid (or …) for 2D decompositions
    - a p x p x p grid for 3D decompositions
      - (or any arrangement that makes sense…)
    - multiple disjoint grids for two concurrent algorithms
    - etc.

A B C D

A B
C D

A B
D
C

CRAY

MG Desiderata: Processors

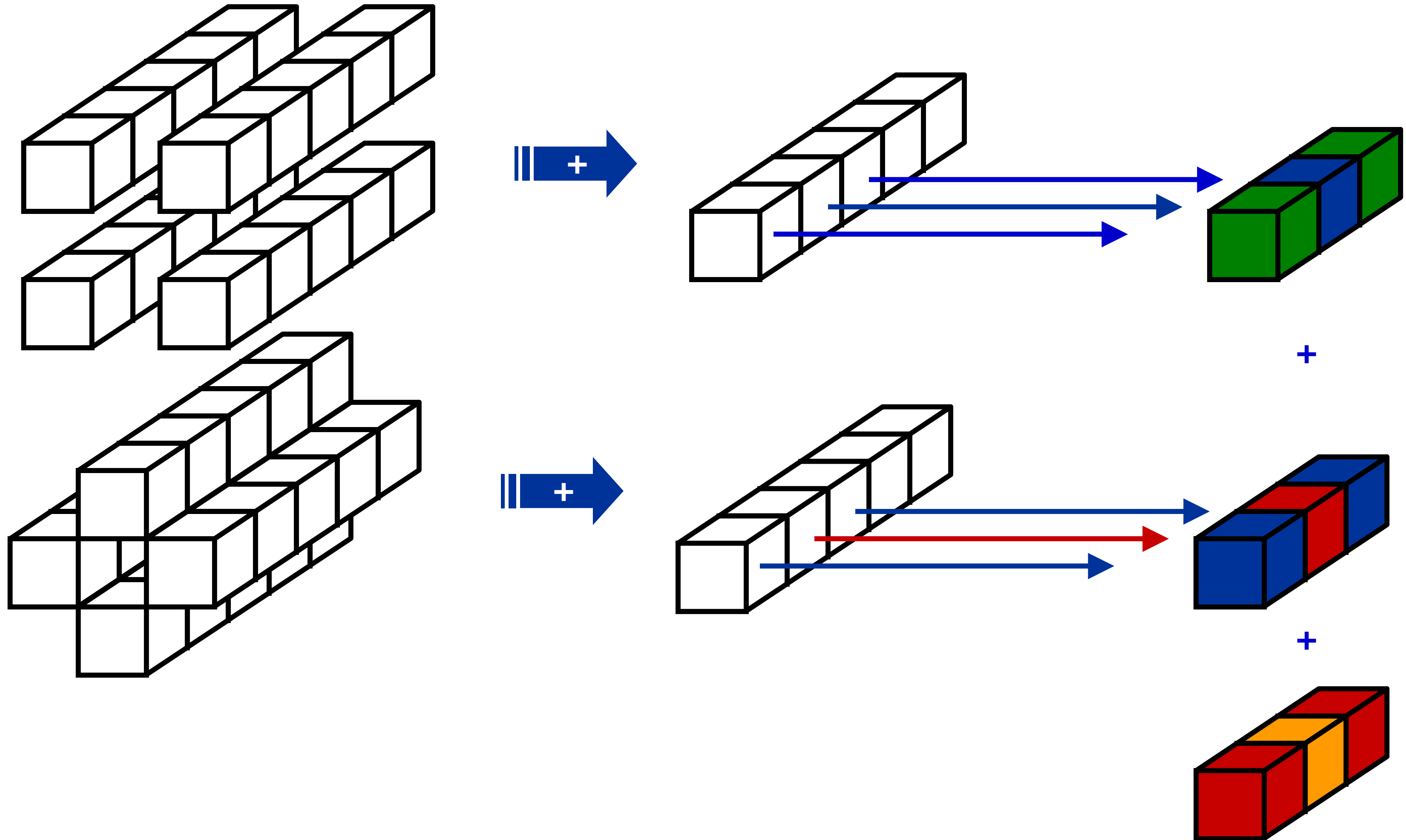
- Ability to specify # processors on command-line
- No constraints on # processors used to run program
- Ability to specify virtual processor grid topologies at runtime

- Support these features with a minimal impact on performance

- *Many implementations of MG require # processors to be specified at compile time*
- *Most…*
  …constrain # processors to be a power of two
  …can only support a uniform 1D/3D virtual processor grid

CRAY

Stencil Optimizations



MG Desiderata: Stencils

- Compiler-supported optimization of stencils
  - caching of adjacent sub-computation reuse
  - automatic elimination of 0-weight values
  - automatic choice between vector-, circular-queue, or register-based scheme based on architecture, stencil
- Scalable expression of stencils w.r.t. size
  - e.g., ability to create 216-point stencils without typing 216 expressions, debugging 648 indexing expressions
- Support these features with a minimal impact on performance

- *Most implementations of MG's 27-point stencils…*
  …contain hand-coded optimizations
  …have θ(27) cost in expression

CRAY

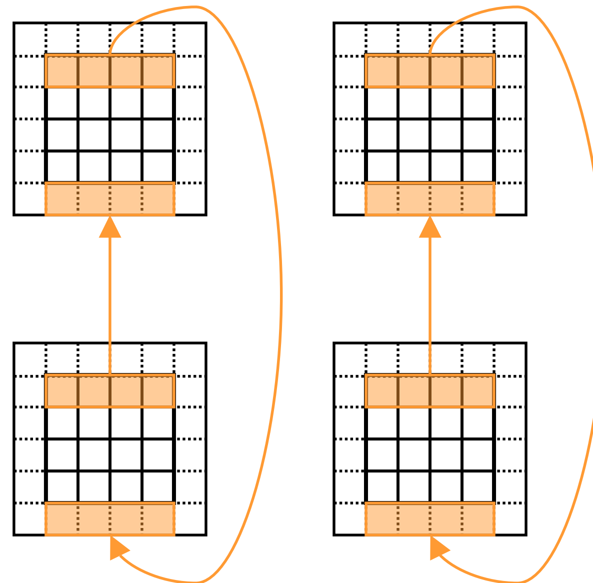
Periodic Boundary Conditions

- Even in global-view languages, periodic boundary conditions are troublesome:
  - require users to allocate extra storage at problem boundaries
  - and to write code to copy values to boundaries

```
var FineGrid = [0..nx+1, 0..ny+1, 0..nz+1]
                    distributed Block(3) to ProcGrid;
var FineGridInterior = [1..nx, 1..ny, 1..nz];

var V: [FineGrid] float;

V[0, 1..ny, 1..nz] = V[nx, 1..ny, 1..nz];
// repeat for other 26 global boundaries
forall (i,j,k) in FineGridInterior {
  V(i,j,k) = V(i-1,j,k) + …;
}
```

Periodic Boundary Conditions

- Yet, as we saw, the global boundaries can/should be handled systolically along with each processor's local boundaries



- Since the compiler is handling inter-processor boundaries, can it be coerced into managing global boundaries as well?

MG Desiderata: Boundaries

- Find some way to have the compiler take the burden of global boundary management from the user by-and-large
- Have these global boundary conditions be updated systolically with the local boundary conditions
- Support these features with a minimal impact on performance

- *Most implementations of MG are fragmented and therefore handle the global boundaries explicitly with the local*
- *Most global-view implementations of MG require the user to manage the boundary conditions explicitly*

CRAY

Reductions

- Languages should have good support for standard reductions
  - *e.g.*, sum, product, min, max, ands, ors, minloc, etc.
- Should also support ability for users to create their own reductions
  - *e.g.*, “smallest-*k* values”, “histogram”, …
- As a potential optimization, compilers should combine reductions
  - if not, user-defined reductions provide a workaround

MG Desiderata: Reductions

- Ability to do min, max, and sum reductions
- Potential for combining reductions in *norm2u3*
  - but gains likely to be minimal
- Good opportunity for min-10, max-10 reductions in initialization
  - 20 non-zeroes determined by finding 10 largest and smallest values in pseudo-random matrix
  - Can code using normal min and max, but rather expensive, awkward

- *Most languages implementing MG have reasonable built-in reductions, but don't have good support for collapsing them or for user reductions*

CRAY

MG Desiderata: Miscellaneous

- Ability to express algorithm recursively as well as iteratively
  - recursion is natural for MG; most implementations are iterative
- Ability to express algorithm in a rank-neutral manner
  - write one code but compile (or run?) for 1D, 2D, 3D
  - helpful for debugging – 3D produces lots of data quickly; more difficult to visualize, especially via test-based console I/O
  - 2D easier, but maintaining two codes is annoying

Chapel and MG

CRAY

HPCS in one slide

HPCS = High Productivity Computing Systems
(a DARPA program)

Overall Goal: Increase productivity for HEC community by the year 2010 (via HW, arch., OS, compilers, tools, …)

Productivity = Programmability
+ Performance
+ Portability
+ Robustness

Result must be…
…revolutionary not evolutionary
…marketable to users other than program sponsors

Phase II Competitors (7/03-7/06): Cray (Cascade), IBM, Sun

What is Chapel?

- *Chapel:* Cascade High-Productivity Language
- Overall goal: Solve the parallel programming problem
  - simplify the creation of parallel programs
  - support their evolution to extreme-performance, production-grade codes
  - emphasize generality
- Motivating Language Technologies:
  1) multithreaded global-view parallel programming
  2) locality-aware programming
  3) object-oriented programming (optionally)
  4) generic programming and type inference

CRAY

Other HPCS Language Efforts

- Other vendor teams
  - IBM: X10
  - Sun: Fortress
- HPLS team (academics, labs) led by Rusty Lusk evaluating HPCS languages for the December 2007 timeframe

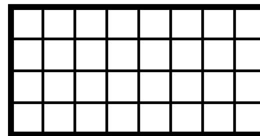
Data Parallelism: Domains

- *domain:* an index set
  - specifies size and shape of “arrays”
  - potentially decomposed across locales
  - supports sequential and parallel iteration
- Three main classes:
  - *arithmetic:* indices are Cartesian tuples
    - rectilinear, multidimensional
    - optionally strided and/or sparse
  - *indefinite:* indices serve as hash keys
    - supports hash tables, associative arrays, dictionaries
  - *opaque:* indices are anonymous
    - supports sets, graph-based computations
- Fundamental Chapel concept for data parallelism
- A generalization of ZPL’s *region* concept

CRAY

A Simple Domain Declaration

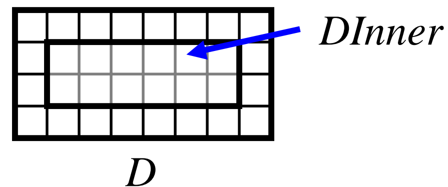
```
var m: integer = 4;
var n: integer = 8;

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



CRAY

A Simple Domain Declaration

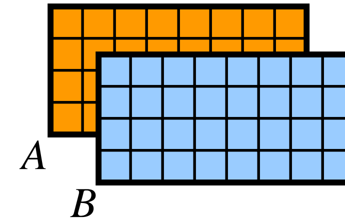
```
var m: integer = 4;
var n: integer = 8;

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



Domain Uses

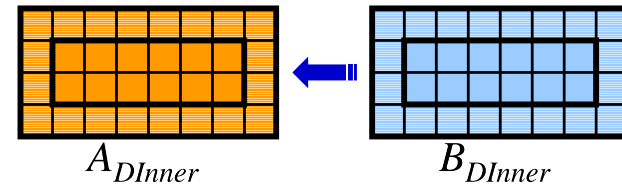
- **Declaring arrays:**

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



- **Sub-array references:**

```
A(DInner) = B(DInner);
```

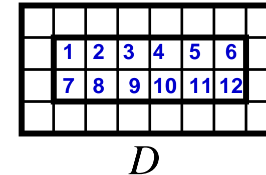


- **Sequential iteration:**

```
for (i,j) in DInner { …A(i,j)… }
```

or:

```
for ij in DInner { …A(ij)… }
```

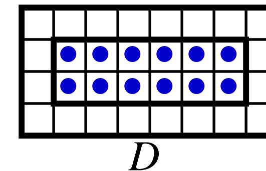


- **Parallel iteration:**

```
forall ij in DInner { …A(ij)… }
```

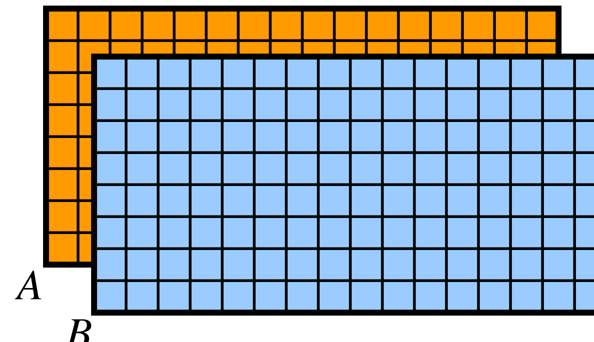
or:

```
[ij in DInner] …A(ij)…
```



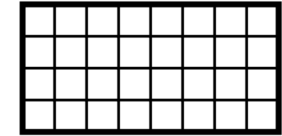
- **Array reallocation:**

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



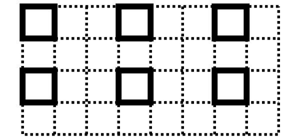
Other Arithmetic Domains

```
var D2: domain(2) = (1,1)..(m,n);
```



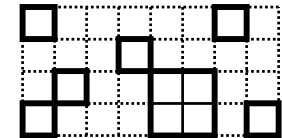
D2

```
var StridedD: domain(D) = D by (2,3);
```



StridedD

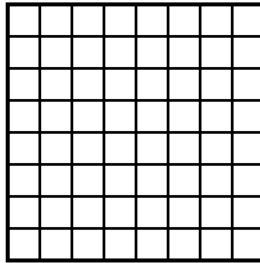
```
var indexList: seq(index(D)) = …;
var SparseD: sparse domain(D) = indexList;
```

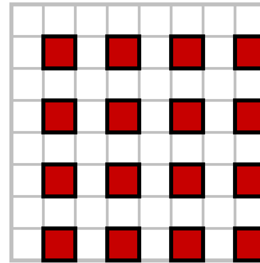


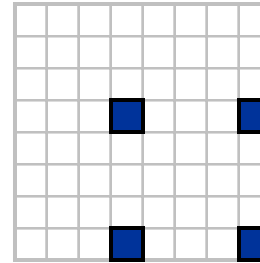
SparseD

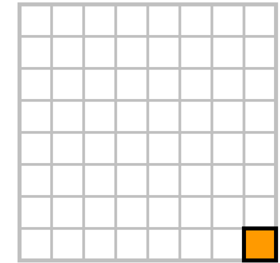
Hierarchical Arrays in Chapel

strided indexing:





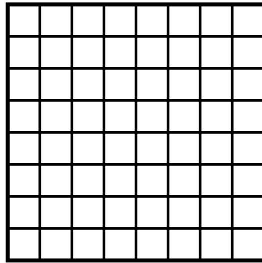


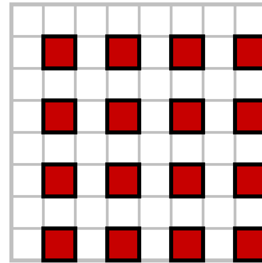


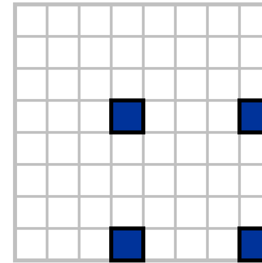
Observation: a hierarchical array is an array of arrays, where each array is parameterized by its level in the hierarchy

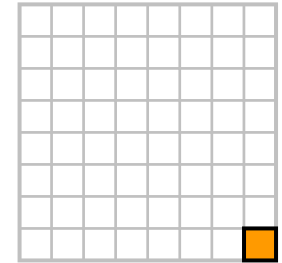
Hierarchical Arrays in Chapel

strided indexing:









`(1:8:-1,1:8:-1)` `(1:8:-2,1:8:-2)` `(1:8:-4,1:8:-4)` `(1:8:-8,1:8:-8)`

```
const BaseD: domain(3) = [1..n, 1..n, 1..n]
                          distributed Block(3);
var V: [BaseD] float;

const Levels: domain(1) = [1..numLevels];
const HierD: [lvl in Levels] domain(BaseD)
           = BaseD by (-2**lvl, -2**lvl, -2**lvl);

var U, R: [lvl in Levels] [HierD(lvl)] float;
// or…   : [HierD] float?
```

CRAY

Reductions in Chapel

```
// support for standard reductions
var biggestV: float = max reduce V;
var totalV: float = sum reduce V;

// also for user-defined reductions:
class min_k : Reduction {
  var k: integer;

  typedef inType;
  typedef stateType = …;
  typedef outType = [1..k] inType;

  function identity() {…}
  function accumulate() {…}
  function combine(…) {…}
  function generate(…) {…}
}

var smallest10Vs: [1..10] float = min_k(k=10) reduce V;
```

CRAY

Stencils in Chapel

- Could write them out explicitly, as in Fortran, ZPL…
- But, note that a stencil is simply a reduction over a small subarray expression
- Thus, stencils can be written scaleably via reductions:

```
const coeff: domain(1) = [0..3]; // for 4 unique weight values
const Stencil: domain(3) = [-1..1, -1..1, -1..1]; // 27-points

function rprj3(S, R) {
  const w: coeff = (/0.5, 0.25, 0.125, 0.0625/);
  const w3d: [(i,j,k) in Stencil] float
           = w((i!=0) + (j!=0) + (k!=0));
  const SD = S.Domain,
        Rstr = R.stride;

  S = [ijk in SD] sum reduce [off in Stencil]
                                (w3d(off) * R(ijk + Rstr*off));
}
```

CRAY

Global Boundary Conditions

- Still TBD…
- Would like to support the ability to associate a function to handle out-of-bounds indexing
  - standard functions built-in for errors, 0's, periodic…
  - compiler implements much like inter-processor ghost cells

```
V.setBoundaryConditions(error);
forall lvl in Levels {
  U(lvl).setBoundaryConditions(wrap);
  R(lvl).setBoundaryConditions(wrap);
}
```

CRAY

Remaining Desiderata

- Processors: user can create arrays of processors ("locales") in Chapel to describe processor resource
  - use these for domain distribution, data allocation
  - can also specify which locales computation should execute on
- Sparse: got a good start at this in ZPL, continuing in Chapel via sparse domains
- Rank-independent: designing Chapel to support rank-independent expressions wherever possible as we go

Other Chapel Features

- Other domain, array types; user-defined distributions
- Task-parallelism abstractions and nested parallelism
- Tuple types, type unions, and typeselect statements
- Sequences, user-defined iterators
- Default arguments, name-based argument passing
- Function and operator overloading
- Curried function calls
- Modules (for namespace management)
- Interoperability with other parallel languages
- Garbage Collection

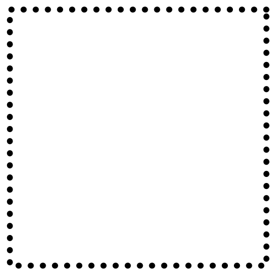
CRAY

Life After MG

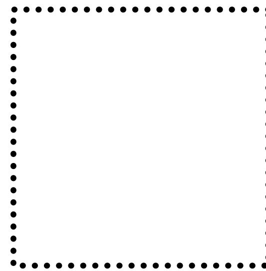
Example: Fast Multipole Method

```
var OSgfn, ISgfn: [lvl in Levels] [SpsCubes(lvl)] [Sgfns(lvl)] [1..3] complex;
```

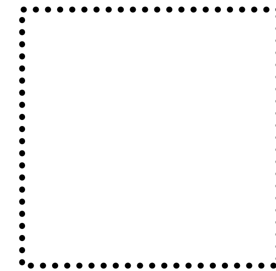
1D array over levels of the hierarchy



OSgfn(1)



OSgfn(2)



OSgfn(3)

Example: Fast Multipole Method

```
var OSgfn, ISgfn: [lvl in Levels] [SpsCubes(lvl)] [Sgfns(lvl)] [1..3] complex;
```

1D array over levels of the hierarchy

…of 3D sparse arrays of cubes (per level)

…of 1D vectors

…of 2D discretizations of spherical functions, (sized by level)

…of complex values

$x + y \cdot i$

OSgfn(1)

OSgfn(2)

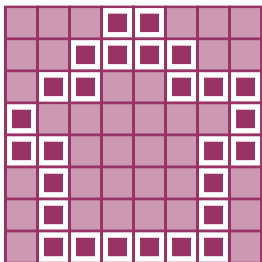
OSgfn(3)

Example: Fast Multipole Method

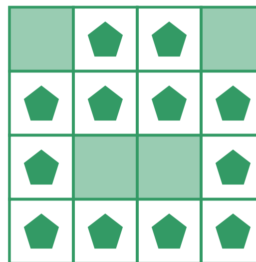
```
var OSgfn, ISgfn: [lvl in Levels] [SpsCubes(lvl)] [Sgfns(lvl)] [1..3] complex;
```

previous definitions:

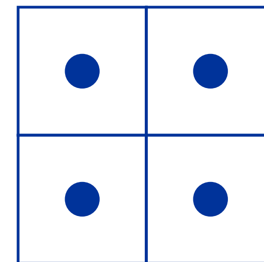
```
var n: integer = …;
var numLevels: integer = …;

var Levels: domain(1) = (1..numLevels);

var scale: [lvl in Levels] integer = 2**(lvl-1);
var SgFnSize: [lvl in Levels] integer = computeSgFnSize(lvl);

var LevelBox: [lvl in Levels] domain(3) = (1,1,1)..(n,n,n) by scale(lvl);
var SpsCubes: [lvl in Levels] sparse domain(LevelBox) = …;

var Sgfns: [lvl in Levels] domain(2) = (1..SgFnSize(lvl), 1..2*SgFnSize(lvl));
```



OSgfn(1)



OSgfn(2)



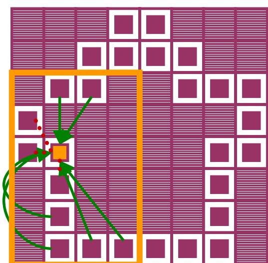
OSgfn(3)

Example: Fast Multipole Method

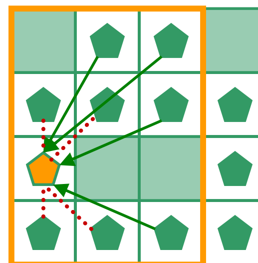
```
var OSgfn, ISgfn: [lvl in Levels] [SpsCubes(lvl)] [Sgfns(lvl)] [1..3] complex;
```

outer-to-inner translation:

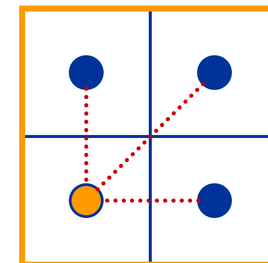
```
var o2iSiblings: [lvl in Levels] [SpsCubes(lvl)] seq(index(SpsCubes(lvl)));

for lvl in Levels by -1 {
  …
  forall cube in SpsCubes(lvl) {
    forall sib in o2iSiblings(lvl)(cube) {
      var Trans: [Sgfns(lvl)] [1..3] complex = lookupXlateTab(cube, sib);

      ISgfn(lvl)(cube) += OSgfn(lvl)(sib) * Trans;
    }
  }
  …
}
```



OSgfn(1)



OSgfn(2)



OSgfn(3)

Fast Multipole Method: Summary

HPCS

- Code captures structure of data and computation far better than sequential Fortran/C versions (let alone MPI variations on them)
  - cleaner
  - more informative use of data structures
  - more succinct
- Parallelism changes at different levels of hierarchy
  - Global view and syntactic separation of concerns helps here
- Good feedback from Boeing engineer who codes FMM
- Yet, I've elided some non-trivial code (data distribution)

CRAY