Chapel

the Cascade High-Productivity Language

LaR'04 – 10/28/2004

Brad Chamberlain

Cray Inc.

HPCS in one slide

HPCS = High Productivity Computing Systems
(a DARPA program)

Overall Goal: Increase productivity for HEC community by the year 2010

Productivity = Programmability
+ Performance
+ Portability
+ Robustness

Result must be…

…revolutionary not evolutionary

…marketable to people other than program sponsors

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

Why develop a new language?

- We believe current parallel languages are lacking:
  - tend to require fragmentation of data and control
  - tend to support a single parallel mode
    - data vs. task parallelism
  - fail to support composition of parallelism
  - have few data abstractions
    - distributed sparse arrays, graphs, hash tables
  - lack support for generic programming
  - fail to cleanly isolate computation from changes to…

    …virtual processor topology

    …data decomposition

    …communication details

    …choice of data structure

    …memory layout

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 parallel programming
  2) locality-aware programming
  3) object-oriented programming
  4) generic programming and type inference

1) Multithreaded Parallel Programming

- Global view of computation, data structures
- Abstractions for data and task parallelism
  - data: domains, arrays, iterators, …
  - task: cobegins, sync variables, atomic transactions, …
- Virtualization of threads
- Composition of parallelism

Global-view: Definition

- "Must programmer code on a per-processor basis?"
- **Data parallel example:** "Add 1000 x 1000 matrices"

global-view

```
var n: integer = 1000;
var a, b, c: [1..n, 1..n] float;

forall ij in [1..n, 1..n]
  c(ij) = a(ij) + b(ij);
```

fragmented

```
var n: integer = 1000;
var locX: integer = n/numProcRows;
var locY: integer = n/numProcCols;
var a, b, c: [1..locX, 1..locY] float;

forall ij in [1..locX, 1..locY]
  c(ij) = a(ij) + b(ij);
```

- **Task parallel example:** "Run Quicksort"

global-view

```
computePivot(lo, hi, data);
cobegin {
  Quicksort(lo, pivot, data);
  Quicksort(pivot, hi, data);
}
```

fragmented

```
if (iHaveParent)
  recv(parent, lo, hi, data);
computePivot(lo, hi, data);
if (iHaveChild)
  send(child, lo, pivot, data);
else
  LocalSort(lo, pivot, data);
LocalSort(pivot, hi, data);
if (iHaveChild)
  recv(child, lo, pivot, data);
if (iHaveParent)
  send(parent, lo, hi, data);
```

Global-view: Impact

- Fragmented languages…
  …obfuscate algorithms by interspersing per-processor management details in-line with the computation
  …require programmers to code with SPMD model in mind
- Global-view languages abstract the processors from the computation

global-view languages
OpenMP
HPF
ZPL
Sisal
NESL
MTA C/Fortran
Matlab
Chapel

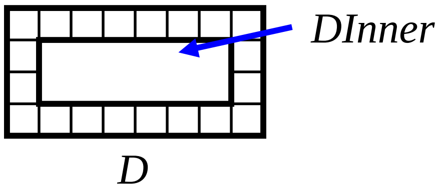
fragmented languages
MPI
SHMEM
Co-Array Fortran
UPC
Titanium

Data Parallelism: Domains

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

A Simple Domain Declaration

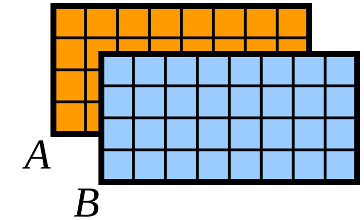
```
var m: integer = 4;
var n: integer = 8;

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



Domain Uses

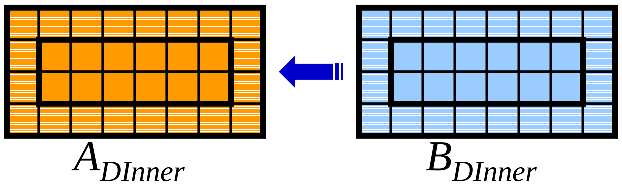
- Declaring arrays:

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



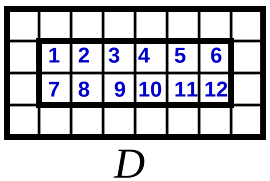
- Sub-array references:

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



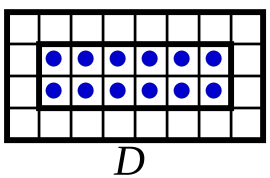
- Sequential iteration:

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



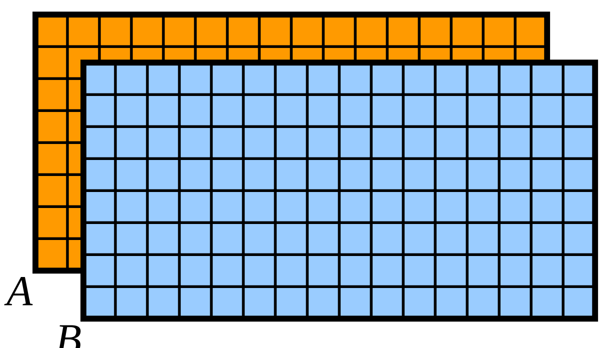
- Parallel iteration:

```
    forall ij in DInner { …A(ij)… }
or: [ij in DInner] …A(ij)…
```



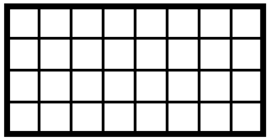
- Array reallocation:

```
D = (1..2*m, 1..2*n);
```



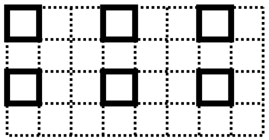
Other Arithmetic Domains

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



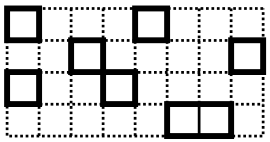
D2

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



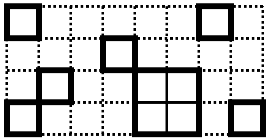
StridedD

```
function foo(ind: index(D)): boole { … }
var SparseD: sparse domain(D)
           = [ij in D] if foo(ij) then ij;
```



SparseD

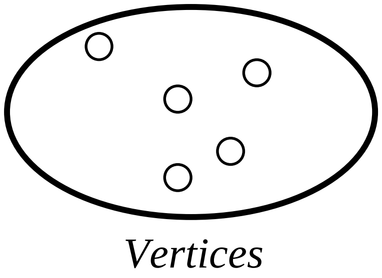
```
var indexList: [1..numInds] index(D) = …;
var SparseD2: sparse domain(D)
            = indexList;
```



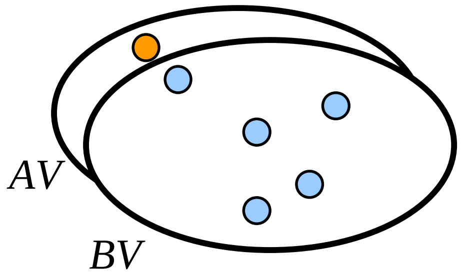
SparseD2

Opaque Domains

```
var Vertices: domain(opaque);

for i in (1..5) {
  Vertices.new();
}
```



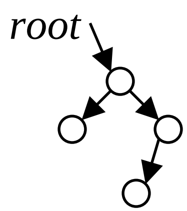

```
var AV, BV: [Vertices] float;
```



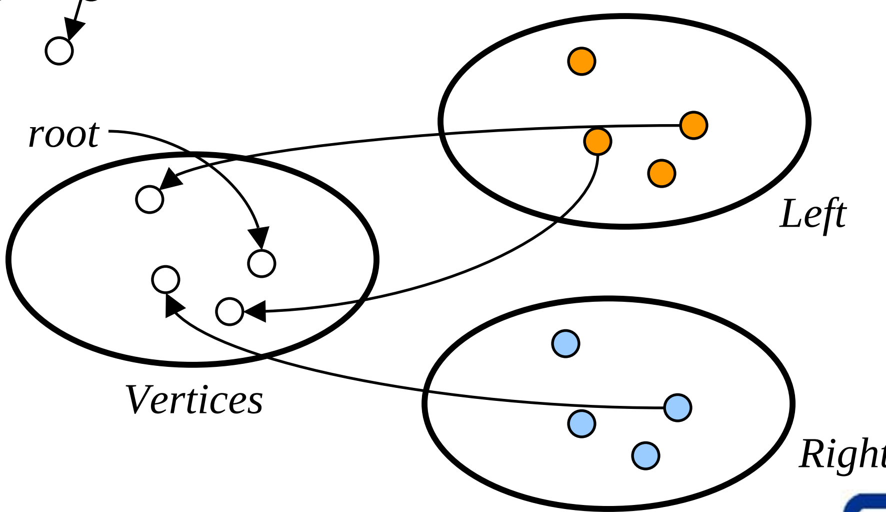
Opaque Domains II

```
var Vertices: domain(opaque);
var left, right: [Vertices] index(Vertices);
var root: index(Vertices);

root = Vertices.new();
left(root) = Vertices.new();
right(root) = Vertices.new();
left(right(root)) = Vertices.new();
```

conceptually:

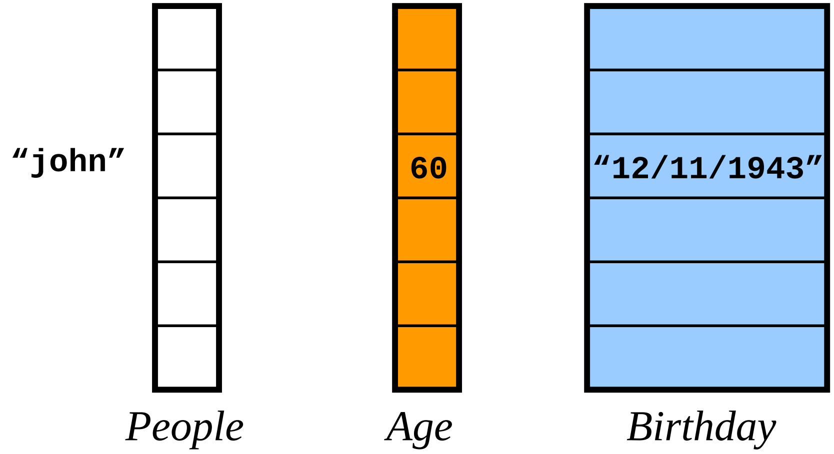


more precisely:



Indefinite Domains

```
var People: domain(string);
var Age: [People] integer;
var Birthdate: [People] string;

Age("john") = 60;
Birthdate("john") = "12/11/1943";
…
forall person in People {
  if (Birthdate(person) == today) {
    Age(person) += 1;
  }
}
```



Task Parallelism

- *co-begins:* indicate statements that may run in parallel:

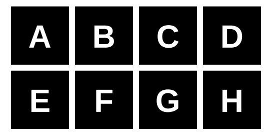
```
computePivot(lo, hi, data);
cobegin {
  Quicksort(lo, pivot, data);
  Quicksort(pivot, hi, data);
}

cobegin {
  ComputeTaskA(…);
  ComputeTaskB(…);
}
```

- *sync and single-assignment variables:* synchronize tasks
  - similar to Cray MTA C/Fortran
- *atomic sections:* provide atomic transactions

2) Locality-aware Programming

- *locale:* machine unit of storage and processing

```
var CompGrid: [1..GridRows, 1..GridCols] locale = …;
```



CompGrid

```
var TaskALocs: [1..numTaskALocs] locale = …;
var TaskBLocs: [1..numTaskBLocs] locale = …;
```

TaskALocs

TaskBLocs

- domains may be distributed across locales

```
var D: domain(2) distribute(block(2)) to CompGrid = …;
```

- "on" keyword binds computation to locale(s)

```
cobegin {
  on TaskALocs do ComputeTaskA(…);
  on TaskBLocs do ComputeTaskB(…);
}
```

3) Object-oriented Programming

- OOP can help manage program complexity
  - encapsulates related data and code
  - facilitates reuse
  - separates common interfaces from specific implementations
- Chapel supports traditional and value classes
- Many productivity-oriented decisions here:
  - anonymous value classes:

    ```
    myPoint = (x = 3, y = 5);
    ```
  - value class field concatenation:

    ```
    myColoredPoint = myPoint + (color = blue);
    ```
  - bound functions and with statements
- OOP is typically not required (user's preference)
- Advanced language features expressed using classes
  - user-defined reductions, distributions, …

4) Generic Programming and Type Inference

- Type Variables and Parameters

```
class Stack {
  type t;
  var buffsize: integer = 128;
  var data: [1..buffsize] t;
  function top(): t { … };
}
```

- Type Query Variables

```
function copyN(data: [?D] ?t; n: integer): [1..n] t {
  var newcopy: [D] t;
  forall i in D
    newcopy(i) = data(i);
  return newcopy;
}
```

- Elided Types

```
function inc(val): {
  var tmp = val;
  return tmp + 1;
}
```

- Chapel programs are statically-typed

Other Chapel Features

- Tuple types, type unions, and typeselect statements
- Sequences & user-defined iterators
- Curried function calls, default arguments & name-based parameter passing
- Support for user-defined…
  …reductions and parallel prefix operations
  …data distributions and memory layouts
  - row/column-major order, block-recursive, Morton order...
  - different sparse representations
- Modules (for namespace management)
- Interoperability with other languages
- Garbage Collection

Example: NAS CG conj_grad()

```
function conj_grad(A, X): {
  const cgitmax = 25;

  var Z = 0.0;
  var R = X;
  var P = R;
  var rho = sum R**2;

  for cgit in (1..cgitmax) {
    var Q = sum(dim=2) (A*P);

    var alpha = rho / sum (P*Q);
    Z += alpha*P;
    R -= alpha*Q;

    var rho0 = rho;
    rho = sum R**2;
    var beta = rho / rho0;
    P = R + beta*P;
  }
  R = sum(dim=2) (A*Z);
  var rnorm = sqrt(sum (X-R)**2);

  return (Z, rnorm);
}
```

Example: NAS CG conj_grad()

```
function conj_grad(A, X): {
  const cgitmax = 25;

  var Z = 0.0;
  var R = X;
  var P = R;
  var rho = sum R**2;

  for cgit in (1..cgitmax) {
    var Q = sum(dim=2) (A*P);

    var alpha = rho / sum (P*Q);
    Z += alpha*P;
    R -= alpha*Q;

    var rho0 = rho;
    rho = sum R**2;
    var beta = rho / rho0;
    P = R + beta*P;
  }
  R = sum(dim=2) (A*Z);
  var rnorm = sqrt(sum (X-R)**2);

  return (Z, rnorm);
}
```

Function return type elided (inferred from return statement)

Parameter types elided (inferred from callsite)

Local variable types elided (inferred from initializer, uses)

Whole-array operations ⇒ data parallel implementation

Built-in array reductions

Sequential iteration over an anonymous domain

Operate on sparse arrays as though dense, and independently of implementing data structures

and partial reductions

Global view ⇒ processors not exposed in computation, array sizes

Separation of concerns ⇒ locale views, domain/array distributions & alignments, and sparse data structures are expressed elsewhere

Composable parallelism ⇒ this (parallel) function could be called from a parallel task (which in turn could be called from another…)

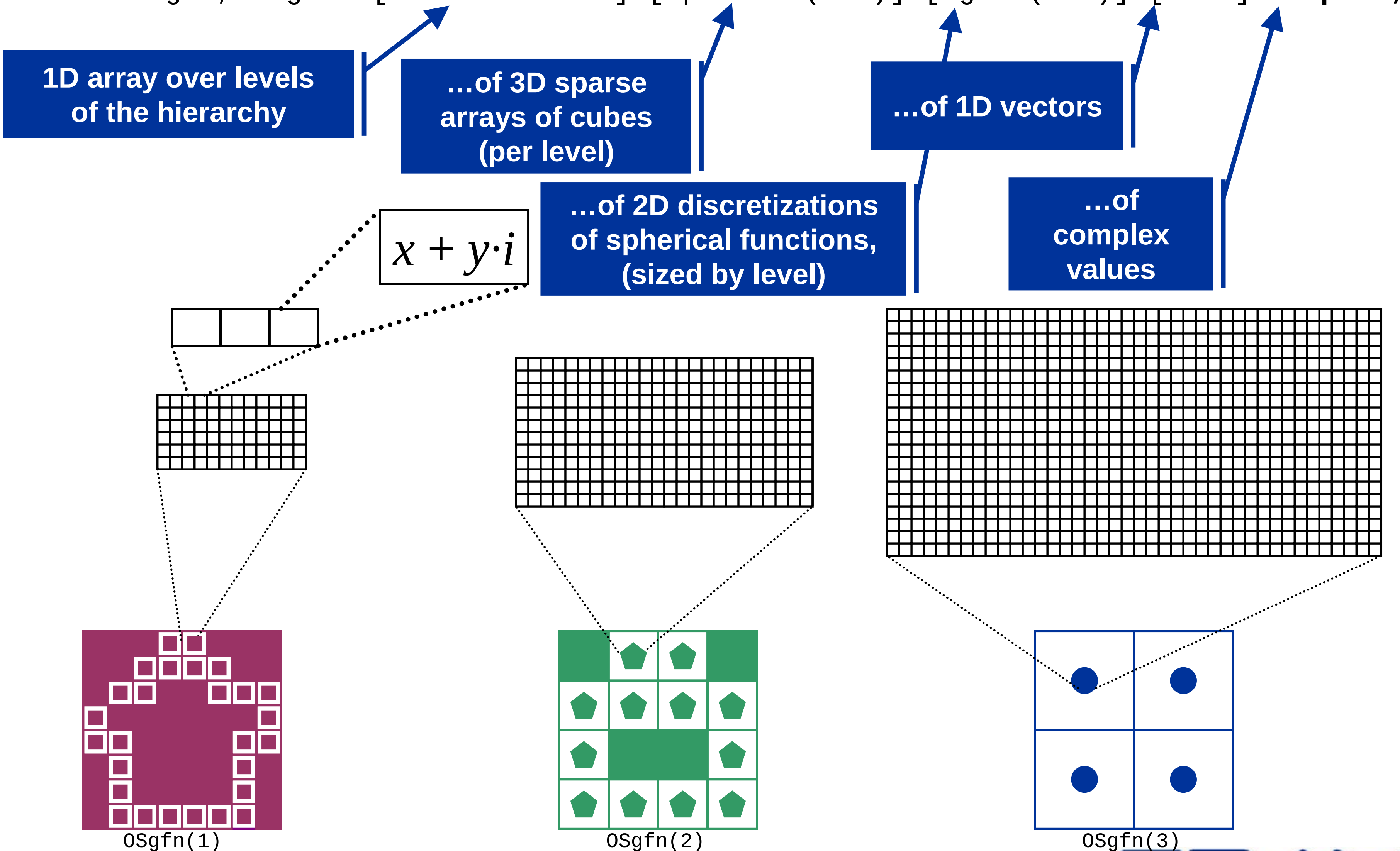
Promotion of scalar operators, values, and functions

Support for tuples

Fortran+MPI	=	173-288 lines	(1265 tokens)
Chapel	=	20 lines	(150 tokens)

Example: Fast Multipole Method

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

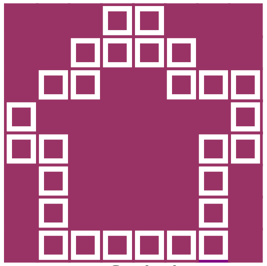


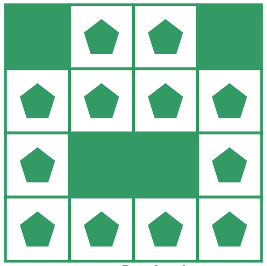
Example: Fast Multipole Method

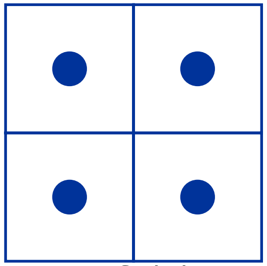
```
var OSgfn, ISgfn: [lvl in Levels] [SpsCubes(lvl)] [Sgfns(lvl)] [1..3] complex;
```

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)

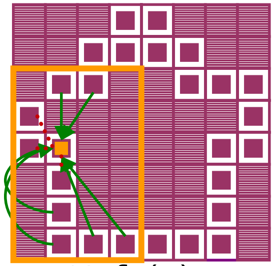

OSgfn(3)

Example: Fast Multipole Method

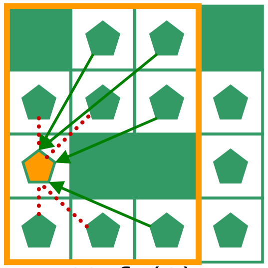
```
var OSgfn, ISgfn: [lvl in Levels] [SpsCubes(lvl)] [Sgfns(lvl)] [1..3] complex;
```

outer-to-inner translation:

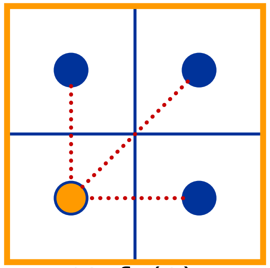
```
var o2iSiblings: [lvl in Levels] [SpsCubes(lvl)] seq(index(SpsCubes(lvl)));

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

- Code captures structure of data and computation far better than sequential Fortran/C versions (let alone MPI variations on them)
  - cleaner
  - more informative
  - 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)

Chapel Challenges

- User Acceptance
  - True of any new language
  - Quantity of features
  - Uniqueness of features
  - Skeptical parallel community
- Cascade Implementation
  - Type determination w/ OOP w/ overloading w/ …
  - Efficient user-defined domain distributions
  - Garbage Collection
- Commodity Architecture Implementation
  - Chapel designed with idealized architecture in mind
  - Clusters are not an ideal architecture
  - Result: implementation and performance challenges

Summary

- Chapel is being designed to…
  …enhance programmer productivity
  …address a wide range of workflows

- Via high-level, extensible abstractions for…
  …multithreaded parallel programming
  …locality-aware programming
  …object-oriented programming
  …generic programming and type inference

- Status
  - language specification currently undergoing editing
    - first draft will be released this winter
  - Open source implementation under way