An Example-Based Introduction to Global-view Programming in Chapel

Brad Chamberlain
Cray Inc.

User Experience and Advances in Bridging Multicore's Programmability Gap

November 16, 2009
SC09 -- Portland

What is Chapel?

- A new parallel language being developed by Cray Inc.
- Part of Cray's entry in DARPA's HPCS program
- **Main Goal:** Improve programmer productivity
  - Improve the programmability of parallel computers
  - Match or beat the performance of current programming models
  - Provide better portability than current programming models
  - Improve robustness of parallel codes
- Target architectures:
  - multicore desktop machines
  - clusters of commodity processors
  - Cray architectures
  - systems from other vendors
- A work in progress

Chapel's Setting: HPCS

HPCS: High *Productivity* Computing Systems (DARPA *et al.*)

- Goal: Raise productivity of high-end computing users by 10×
- Productivity = Performance
  + Programmability
  + Portability
  + Robustness

- **Phase II**: Cray, IBM, Sun (July 2003 – June 2006)
  - Evaluated the entire system architecture's impact on productivity…
    - processors, memory, network, I/O, OS, runtime, compilers, tools, …
    - …and new languages:
      Cray: Chapel   IBM: X10   Sun: Fortress

- **Phase III**: Cray, IBM (July 2006 –)
  - Implement the systems and technologies resulting from phase II
  - (Sun also continues work on Fortress, without HPCS funding)

Chapel: Motivating Themes

1) general parallel programming
2) *global-view* abstractions
3) *multiresolution* design
4) control of locality/affinity
5) reduce gap between mainstream & parallel languages

What I intend for this talk to do

- Illustrate Chapel's feature set and design as motivated by realistic computational kernels

What this talk will not do

- Illustrate that Chapel performs well for these codes today
  - to date, our implementation has focused primarily on completeness and correctness
  - in many cases, our experiences with ZPL give us confidence that compilers can achieve competitive performance for these codes
  - in other cases, we have some research and work ahead of us

come to tomorrow's HPC Challenge BOF (12:15pm) to get an up-to-date report on Chapel performance for the HPCC benchmarks

Disclaimers

- some of these examples use slightly outdated syntax
  - particularly when it comes to declaring distributions
    - in part because I got lazy last night at midnight
    - in part because it's still in a bit of flux
- other examples use features that aren't fully implemented
  - particularly cases that use arrays of arrays of varying size
  - but I think it's important to show you Chapel's motivators and future

Outline

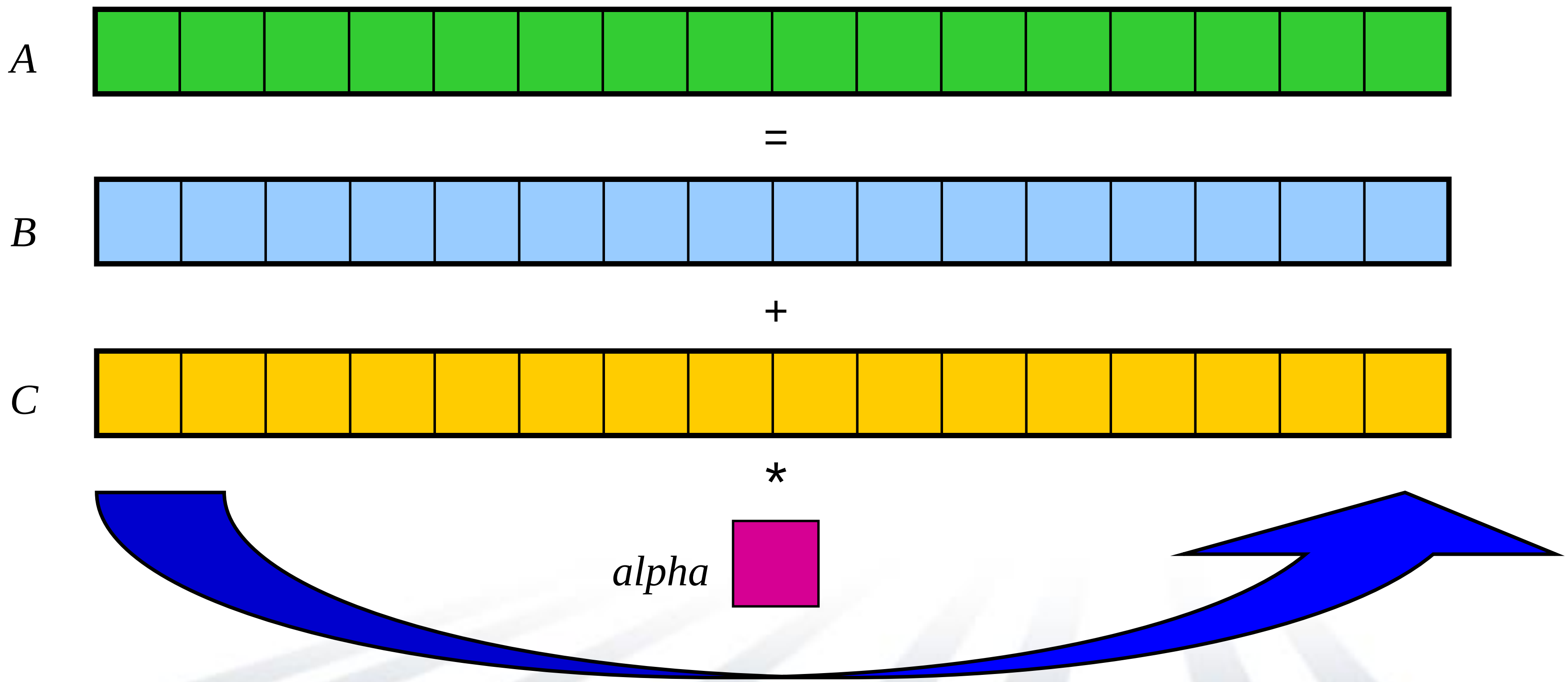
- ✓ Chapel Overview
- ➢ Chapel computations
  - **your first Chapel program:** STREAM Triad
  - ***the stencil ramp*****:** from jacobi to finite element methods
  - **graph-based computation in Chapel:** SSCA #2
  - **task-parallelism:** producer-consumer to MADNESS
  - **GPU computing in Chapel:** STREAM revisited and CP
- Status, Summary, and Future Work

Introduction to STREAM Triad

Given: m-element vectors A, B, C

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

Visually:

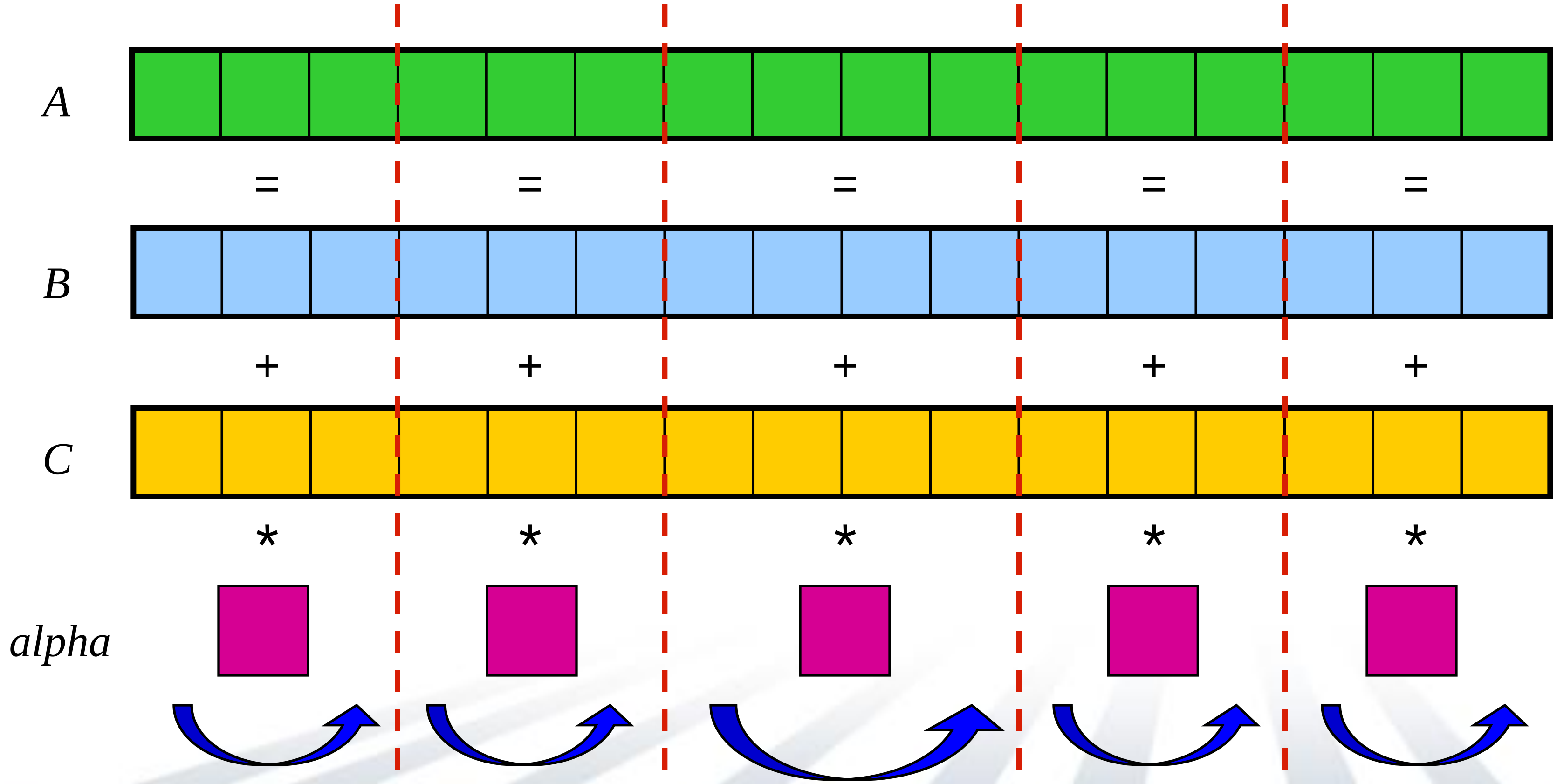


Introduction to STREAM Triad

Given: m-element vectors A, B, C

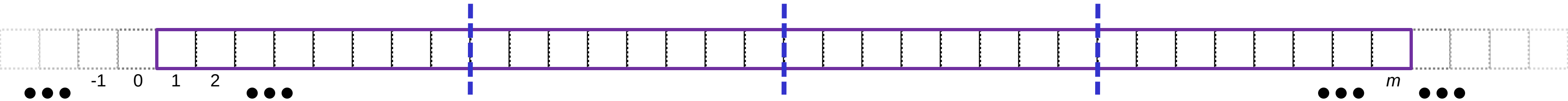
Compute: $\forall i \in 1..m,\ A_i = B_i + \alpha \cdot C_i$

Pictorially (in parallel):

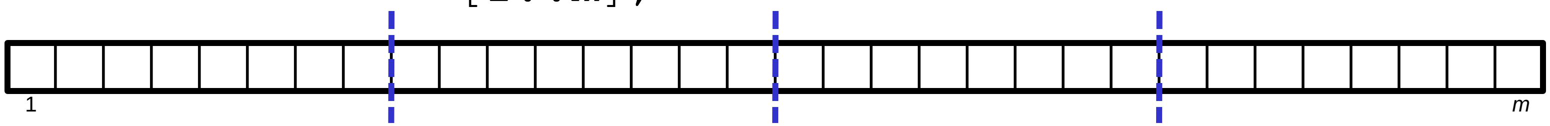


STREAM Triad in Chapel

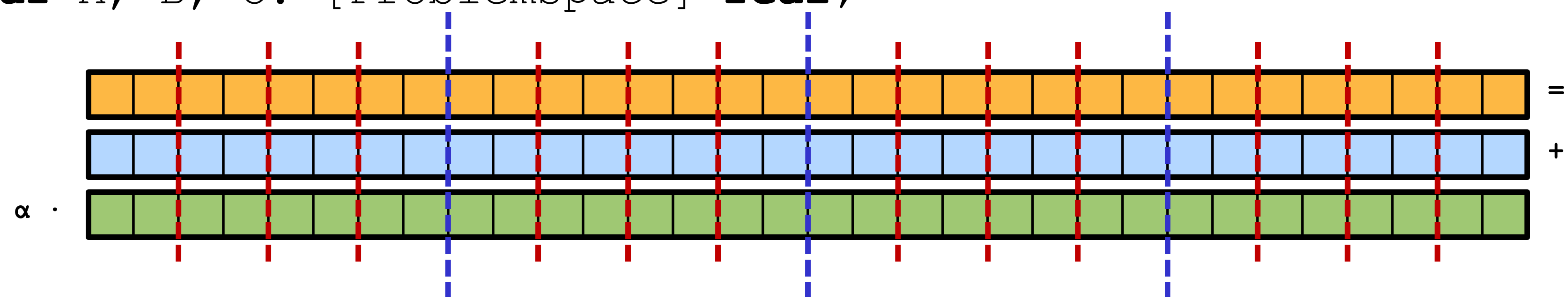
```
const BlockDist = new Block1D(bbox=[1..m], tasksPerLocale=…);
```




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



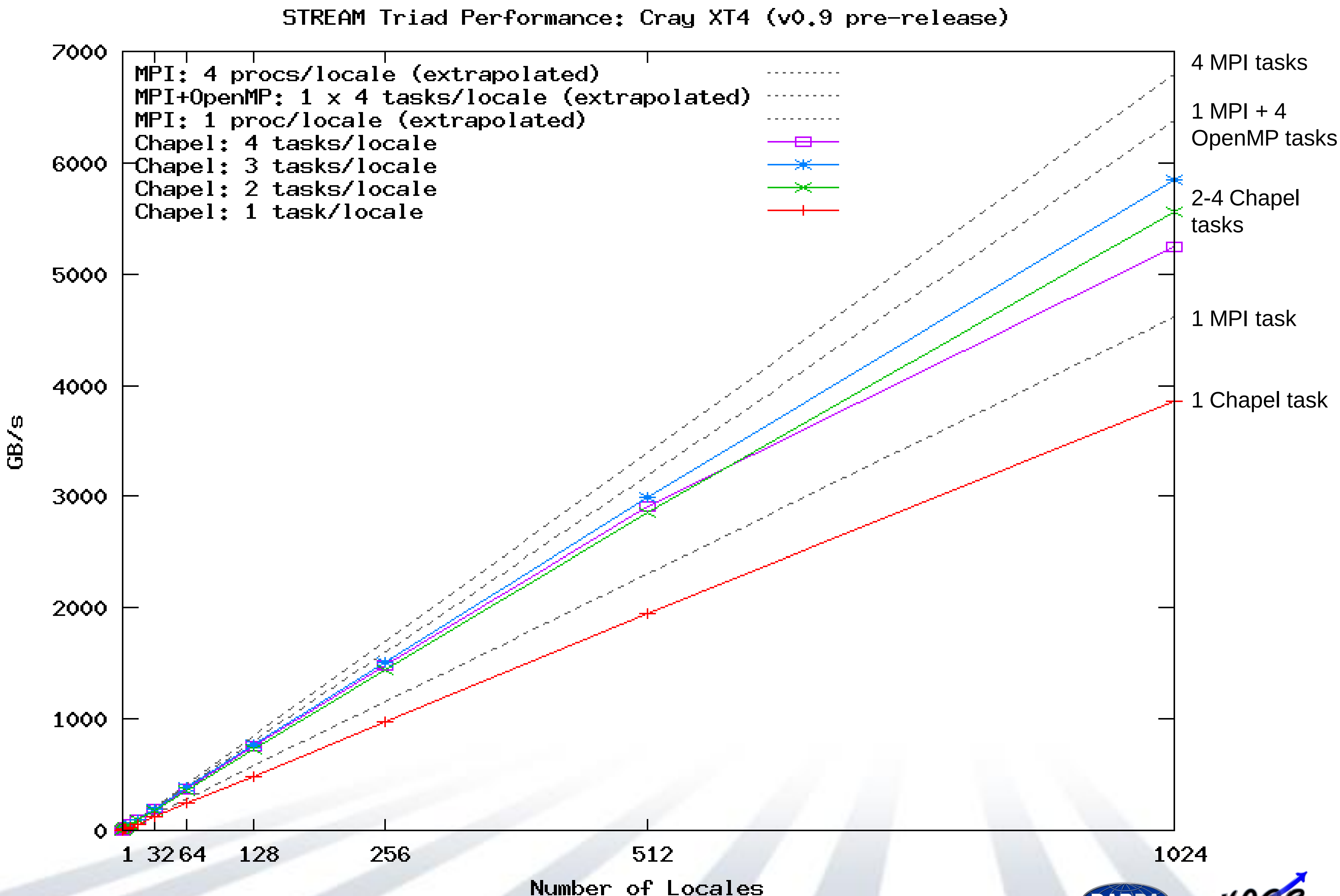

```
var A, B, C: [ProblemSpace] real;
```




```
forall (a, b, c) in (A, B, C) do
  a = b + alpha * c;
```

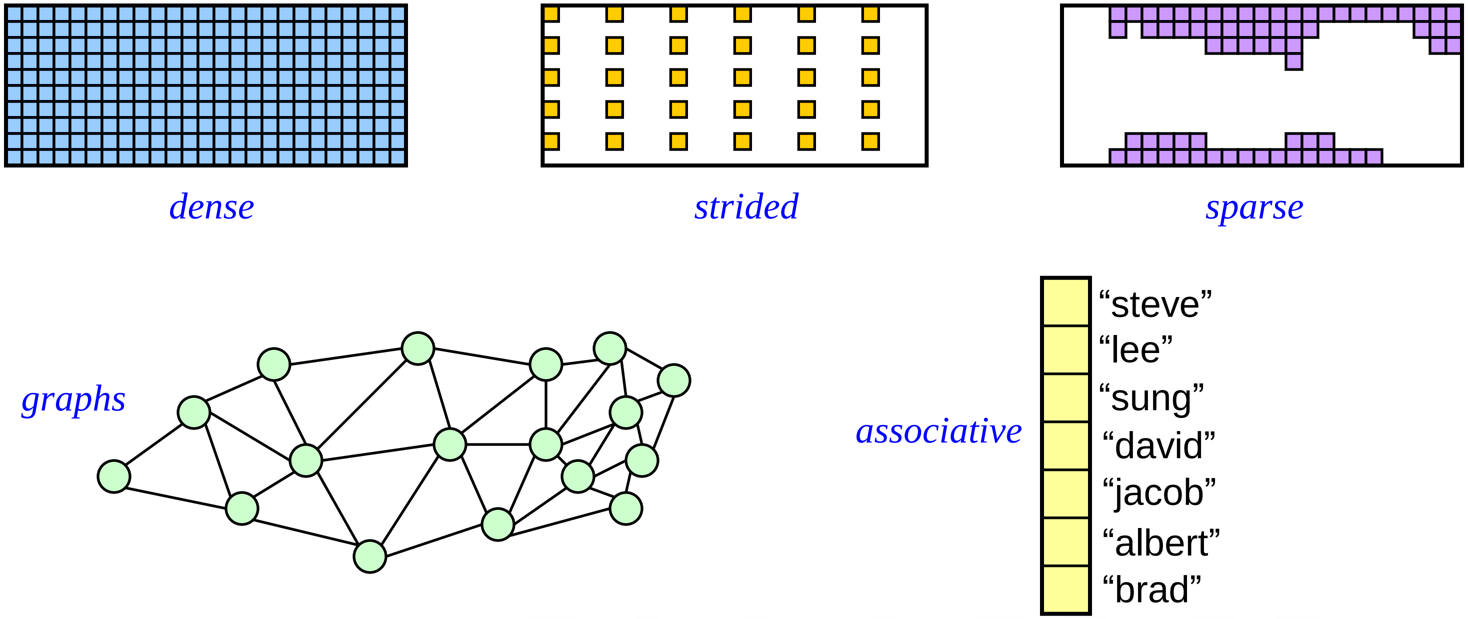
Chapel (10)

STREAM Performance, Cray XT4 (April 2009)



Chapel Domains and Arrays

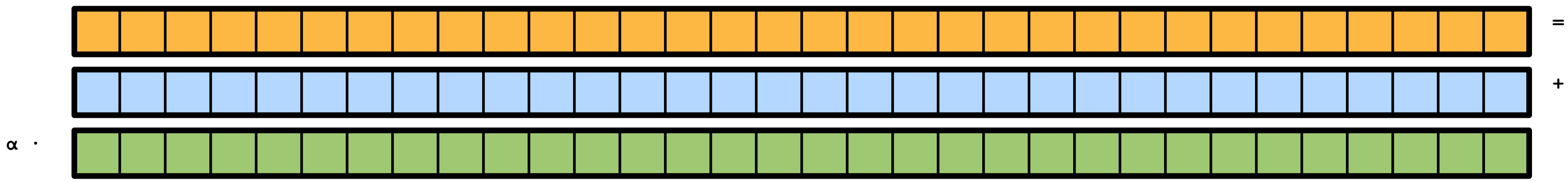
Chapel supports several domain and array types…



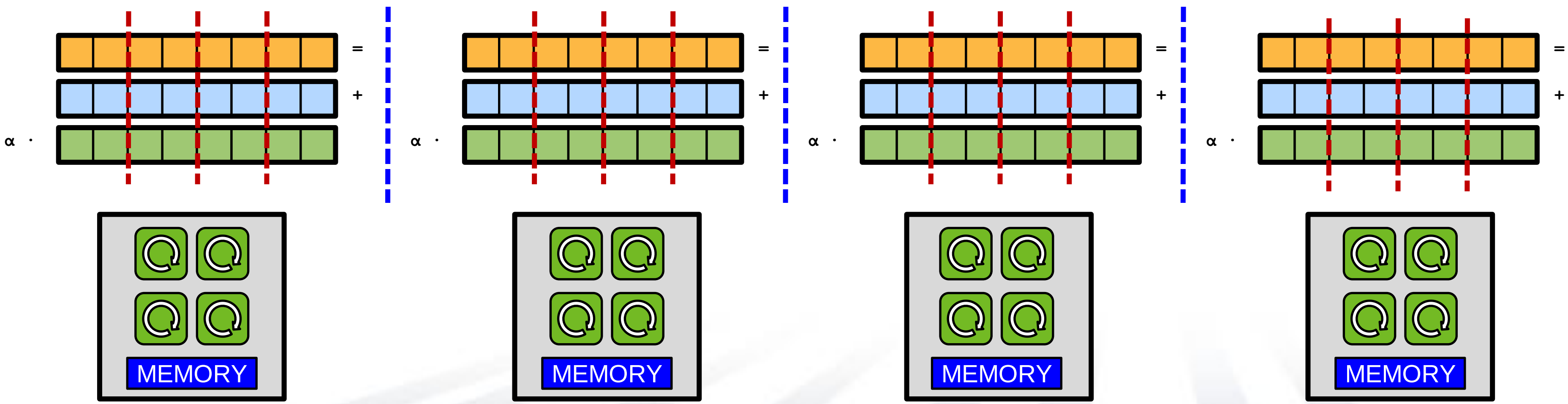
Chapel Distributions

Distributions: "Recipes for parallel, distributed arrays"

- help the compiler map from the computation's global view…

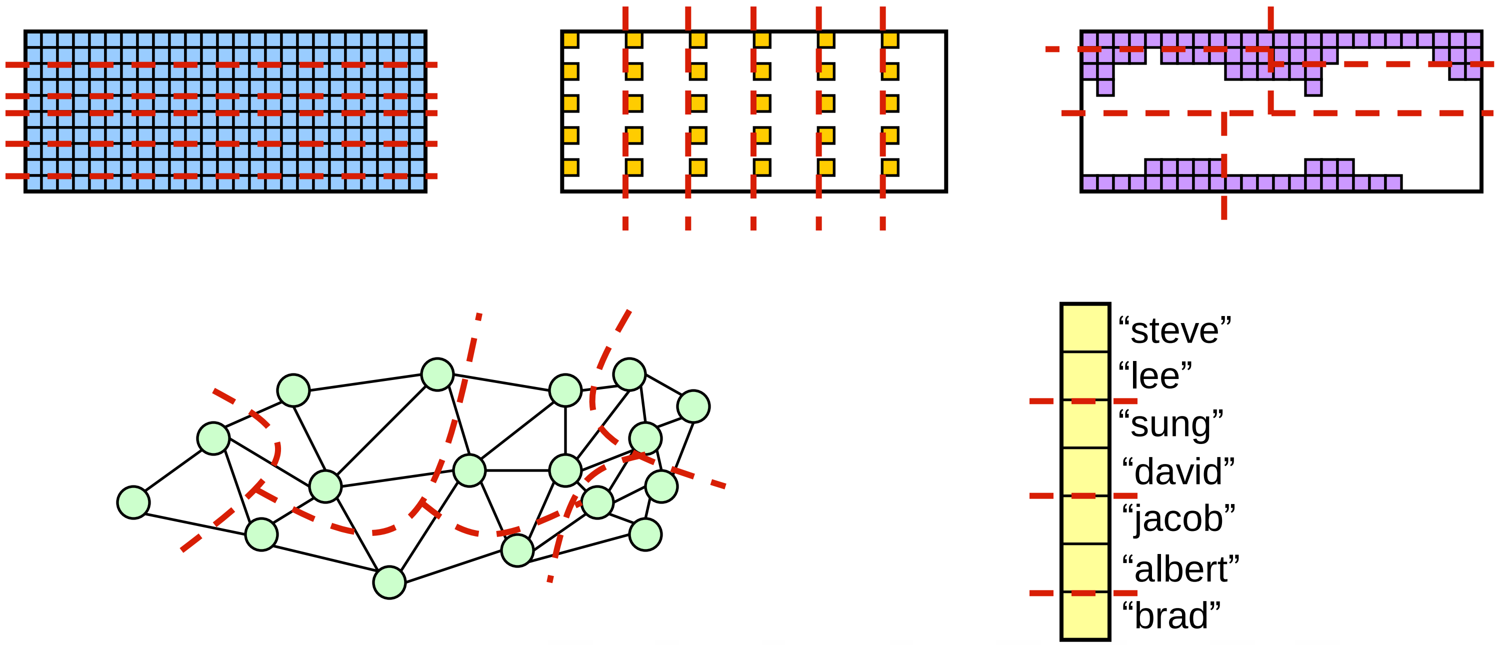


…down to the *fragmented*, per-processor implementation



Domain Distributions

- Any domain type may be distributed
- Distributions do not affect program semantics
  - only implementation details and therefore performance



Distributions: Goals & Research

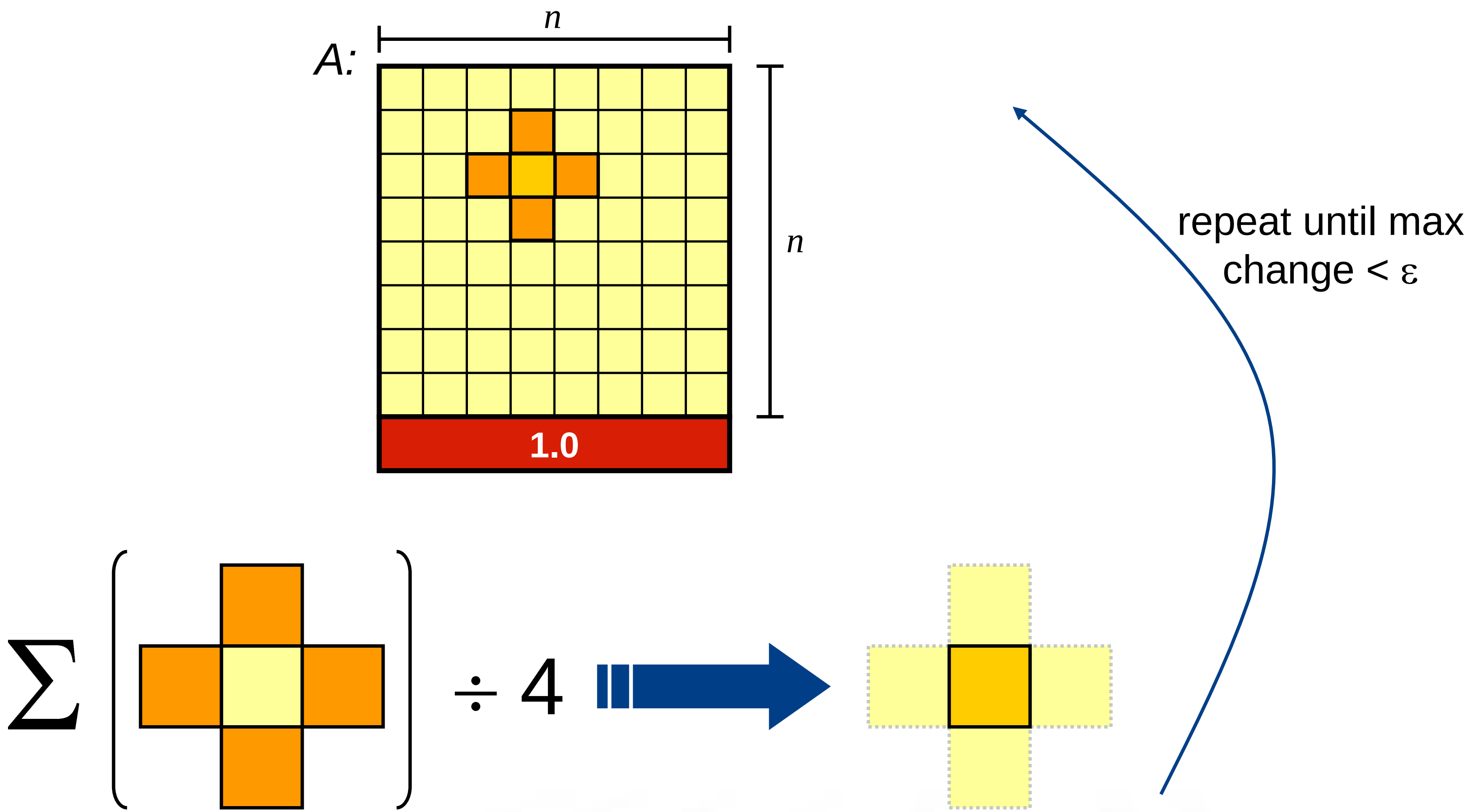
- Advanced users can write their own distributions
  - specified in Chapel using lower-level language features
- Chapel will provide a standard library of distributions
  - written using the same user-defined distribution mechanism

(Draft paper describing user-defined distribution strategy available by request)

Outline

- ✓ Chapel Overview
- ➢ Chapel computations
  - **your first Chapel program:** STREAM Triad
  - ***the stencil ramp*****:** from jacobi to finite element methods
  - **graph-based computation in Chapel:** SSCA #2
  - **task-parallelism:** producer-consumer to MADNESS
  - **GPU computing in Chapel:** STREAM revisited and CP
- Status, Summary, and Future Work

Stencil 1: Jacobi Iteration



Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
         D: subdomain(BigD) = [1..n, 1..n],
   LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
  [(i,j) in D] Temp(i,j) = (A(i-1,j) + A(i+1,j)
                          + A(i,j-1) + A(i,j+1)) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
         D: subdomain(BigD) = [1..n, 1..n],
   LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;

A[Las

do {
  [(i


  con
  A[D
} whi

writeln(A);
```

Declare program parameters

const ⇒ can't change values after initialization

config ⇒ can be set on executable command-line

prompt> `jacobi -sn=10000 -sepsilon=0.0001`

note that no types are given; inferred from initializer

n ⇒ **integer** (current default, 32 bits)

epsilon ⇒ **floating-point** (current default, 64 bits)

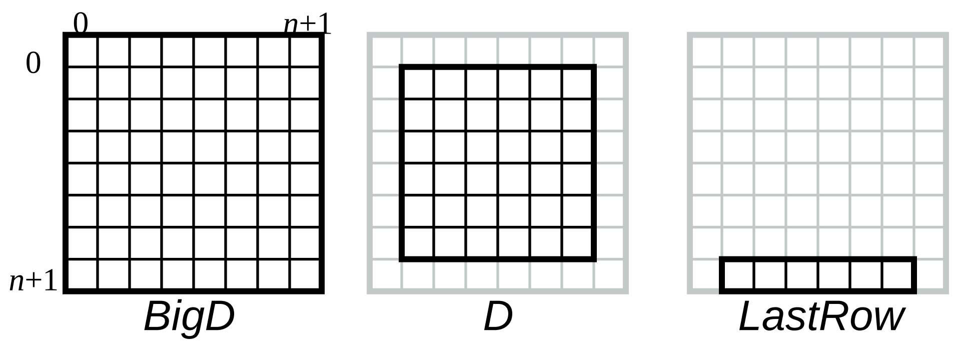
Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
         D: subdomain(BigD) = [1..n, 1..n],
   LastRow: subdomain(BigD) = D.exterior(1,0);
```

Declare domains (first class index sets)

domain(2) ⇒ 2D arithmetic domain, indices are integer 2-tuples

subdomain(*P*) ⇒ a domain of the same type as *P* whose indices are guaranteed to be a subset of *P*'s



exterior ⇒ one of several built-in domain generators

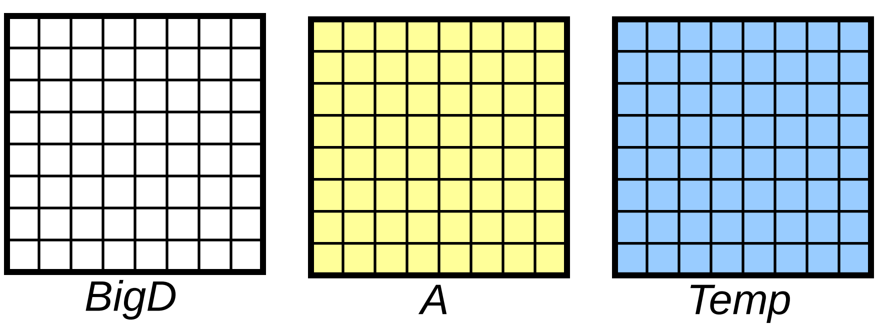
4;

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
           D: subdomain(BigD) = [1..n, 1..n],
     LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;
```

Declare arrays

var ⇒ can be modified throughout its lifetime
: *T* ⇒ declares variable to be of type *T*
: *[D] T* ⇒ array of size *D* with elements of type *T*
(no initializer) ⇒ values initialized to default value (0.0 for reals)



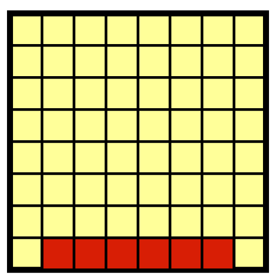
4;

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
         D: subdomain(BigD) = [1..n, 1..n],
   LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;
```

<u>Set Explicit Boundary Condition</u>

indexing by domain ⇒ slicing mechanism
array expressions ⇒ parallel evaluation



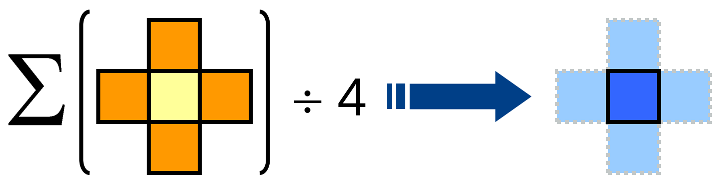
A

Jacobi Iteration in Chapel

Compute 5-point stencil

[(*i,j*) in *D*] ⇒ parallel forall expression over *D*'s indices, binding them to new variables *i* and *j*

Note: since (*i,j*) $\in$ *D* and *D* $\subseteq$ *BigD* and *Temp*: [*BigD*]
⇒ no bounds check required for *Temp(i,j)*
with compiler analysis, same can be proven for A's accesses




```
    [(i,j) in D] Temp(i,j) = (A(i-1,j) + A(i+1,j)
                            + A(i,j-1) + A(i,j+1)) / 4;

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
  } while (delta > epsilon);

  writeln(A);
```

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
```

Compute maximum change

***op* reduce** ⇒ collapse aggregate expression to scalar using *op*

Promotion: *abs()* and – are scalar operators, automatically promoted to work with array operands

```
do {
  [(i,j) in D] Temp(i,j) = (A(i-1,j) + A(i+1,j)
                          + A(i,j-1) + A(i,j+1)) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
         D: subdomain(BigD) = [1..n, 1..n],
   LastRow: subdomain(BigD) = D.exterior(1,0);

var

A[La

do {
  [(i,j) in D] Temp(i,j) = (A(i-1,j) + A(i+1,j)
                          + A(i,j-1) + A(i,j+1)) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Copy data back & Repeat until done

uses slicing and whole array assignment
standard *do…while* loop construct

Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
         D: subdomain(BigD) = [1..n, 1..n],
   LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
  [(                                        j)
                                            1)) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

<u>Write array to console</u>

If written to a file, parallel I/O could be used

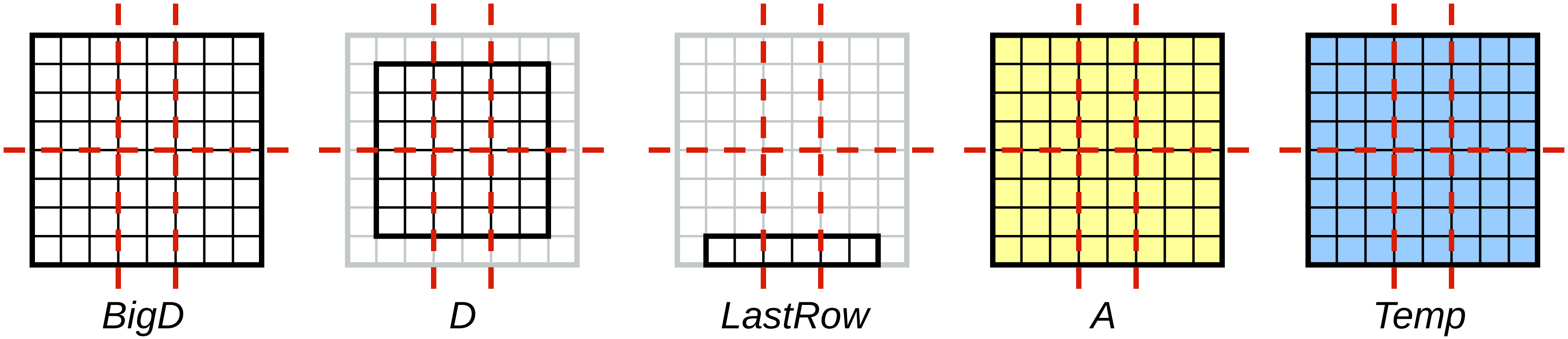
Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD: domain(2) distributed Block = [0..n+1, 0..n+1],
         D: subdomain(BigD) = [1..n, 1..n],
   LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;
```

With this change, same code runs in a distributed manner
Domain distribution maps indices to *locales*

⇒ decomposition of arrays & default location of iterations over locales

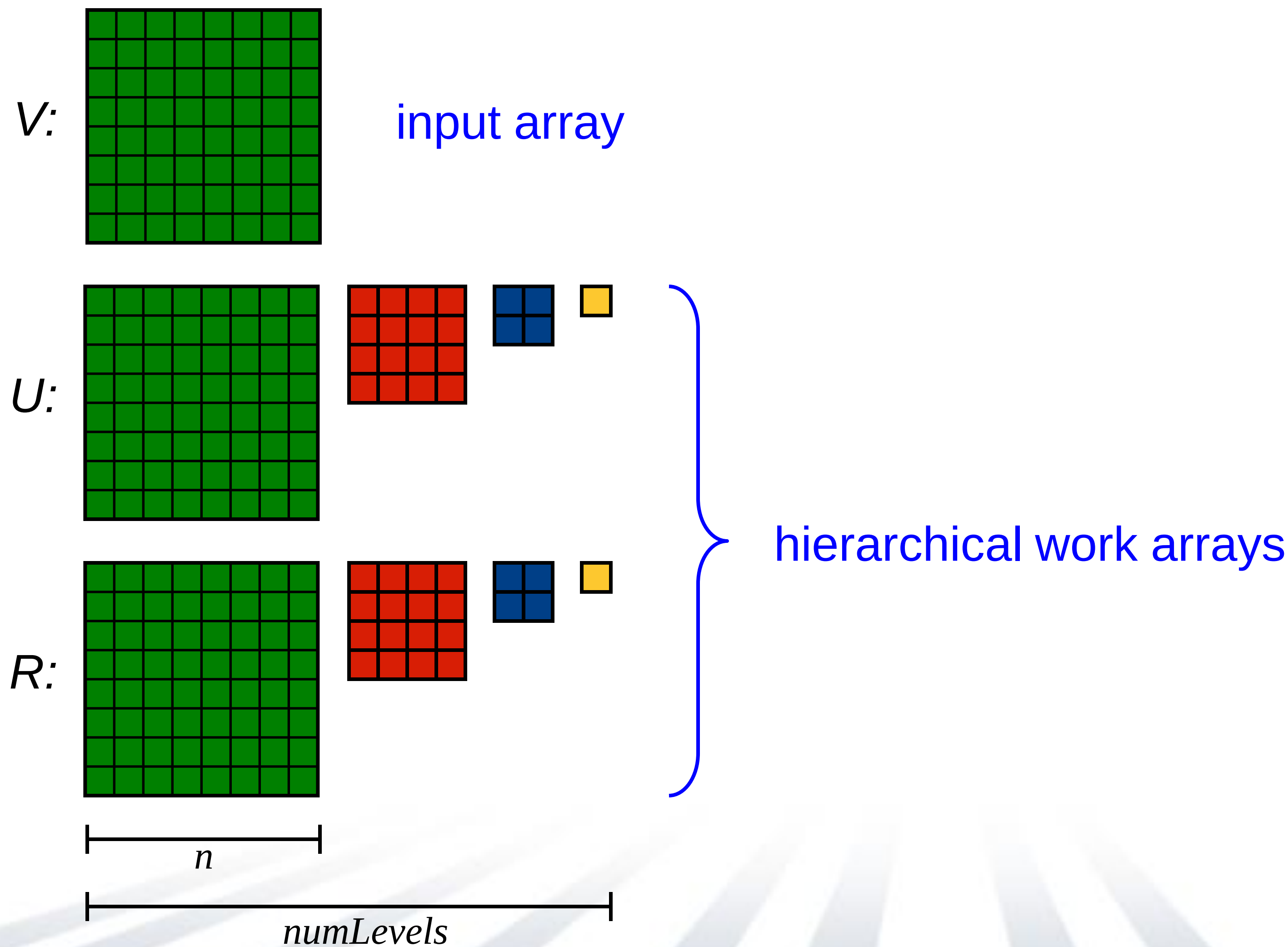
Subdomains inherit parent domain's distribution



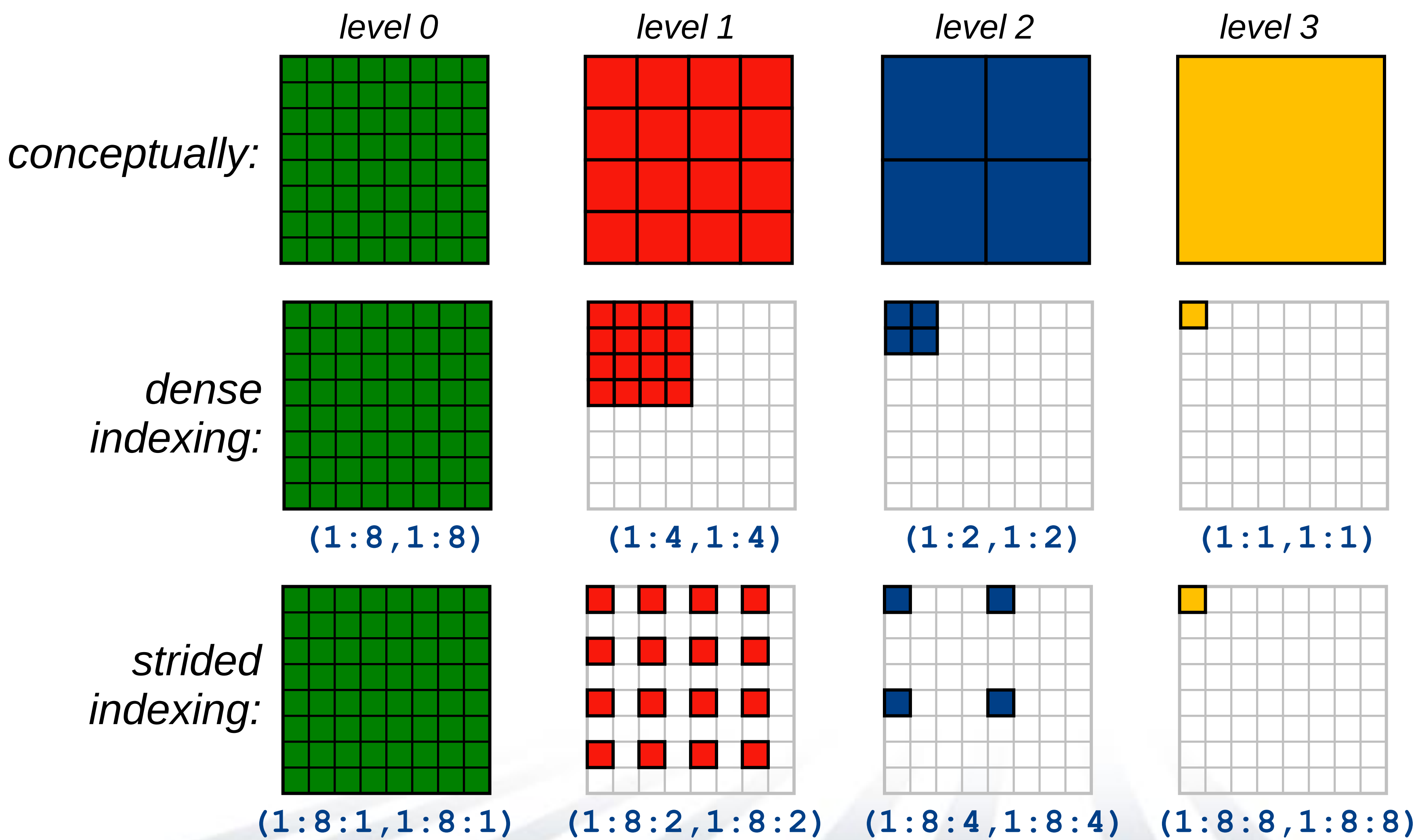
Jacobi Iteration in Chapel

```
config const n = 6,
             epsilon = 1.0e-5;

const BigD: domain(2) distributed Block = [0..n+1, 0..n+1],
         D: subdomain(BigD) = [1..n, 1..n],
   LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
  [(i,j) in D] Temp(i,j) = (A(i-1,j) + A(i+1,j)
                          + A(i,j-1) + A(i,j+1)) / 4;

  const delta = max reduce abs(A[D] - Temp[D]);
  A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

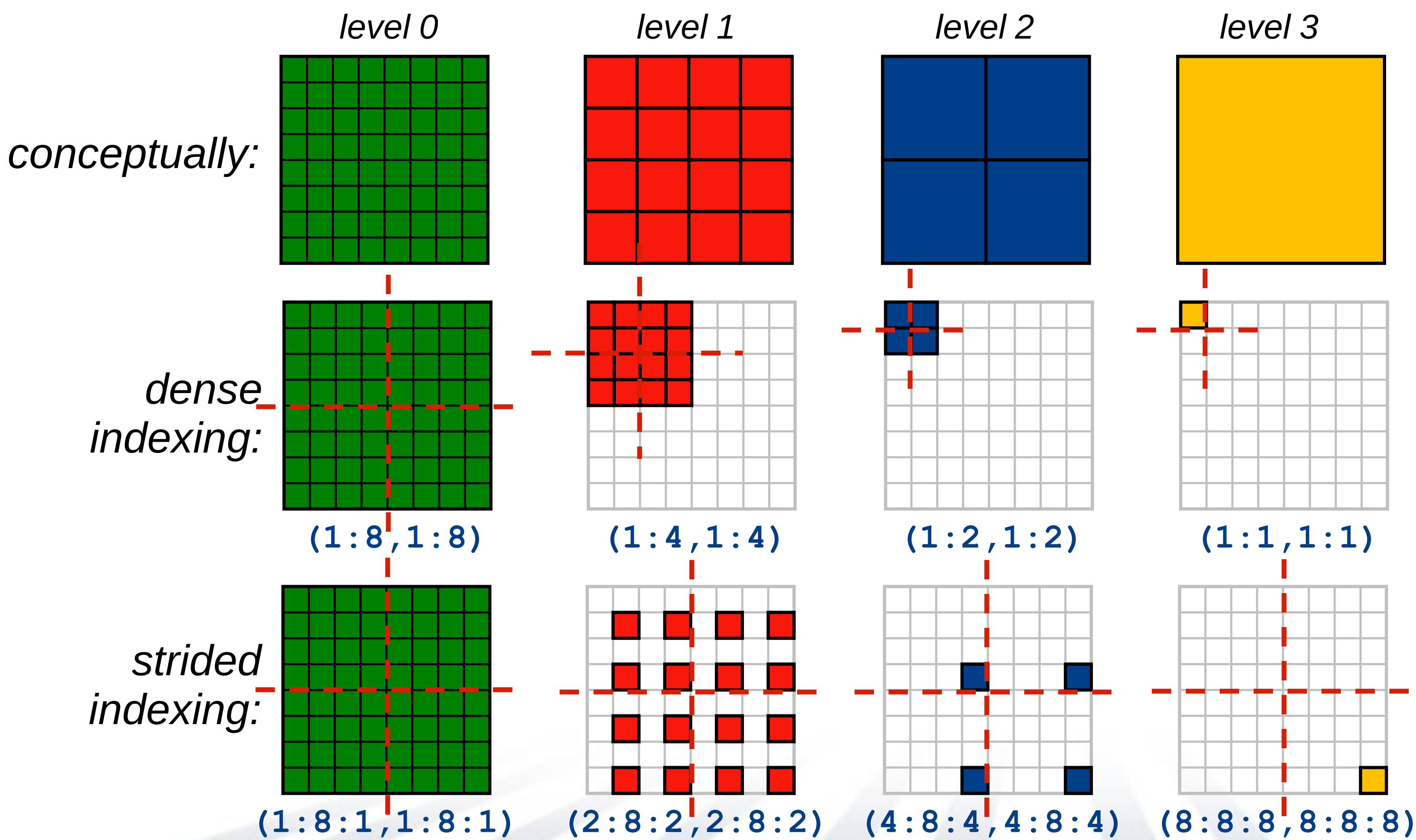
Stencil 2: Multigrid



Hierarchical Arrays



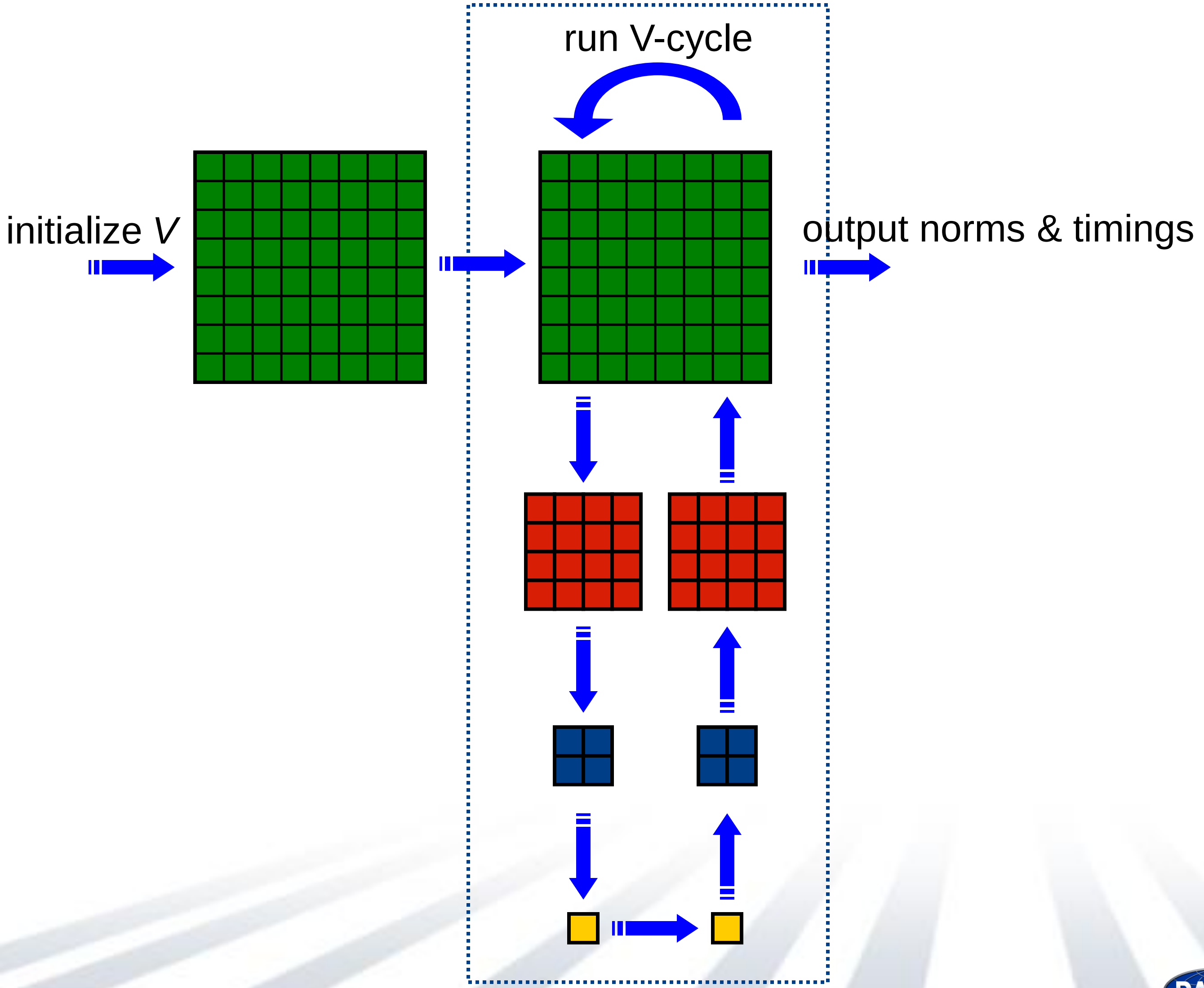
Hierarchical Arrays



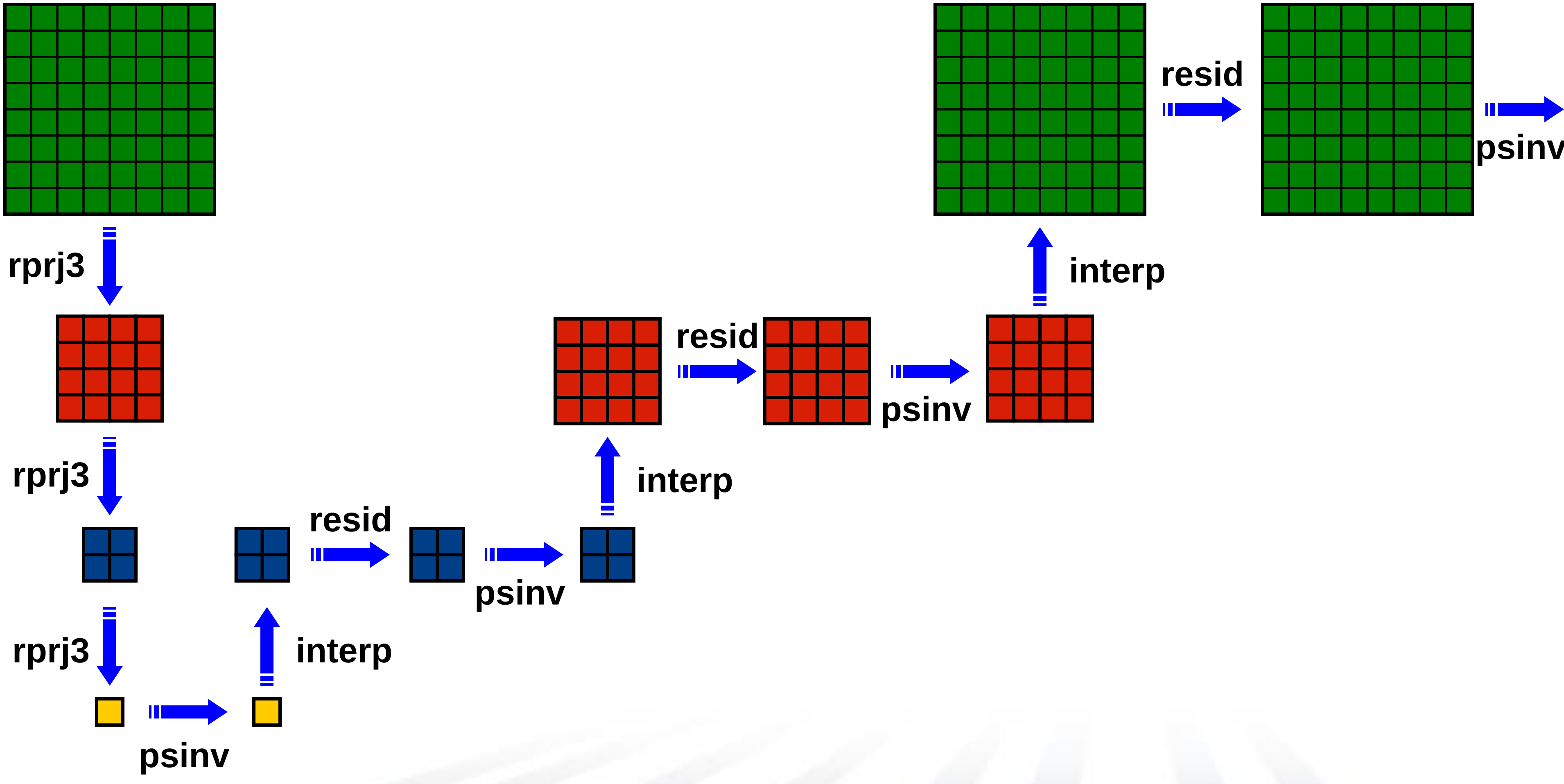
Hierarchical Array Declarations in Chapel

```
config const n = 1024,
             numLevels = lg2(n);

const Levels = [0..#numLevels];
const ProblemSpace: domain(1) distributed Block = [1..n]**3;

var V: [ProblemSpace] real;

const HierSpace: [lvl in Levels] subdomain(ProblemSpace)
               = ProblemSpace by -2**lvl;

var U, R: [lvl in Levels] [HierSpace(lvl)] real;
```

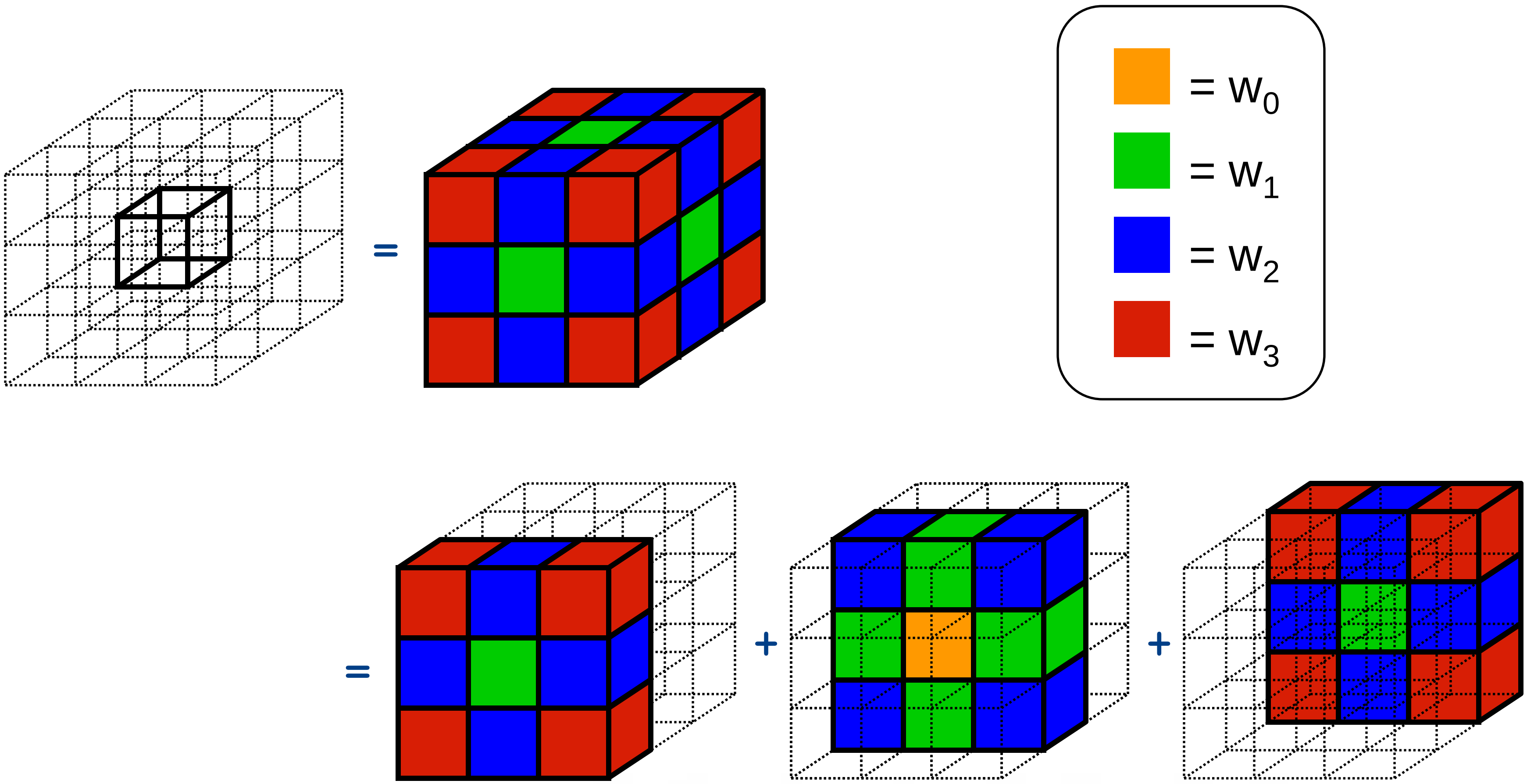
Overview of NAS MG



MG's projection/interpolation cycle



Multigrid: 27-Point Stencils



Multigrid: Stencils in Chapel

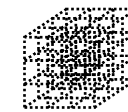
- Can write them out explicitly…

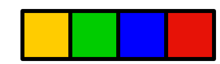
```
def rprj3(S, R) {
  param w: [0..3] real = (0.5, 0.25, 0.125, 0.0625);
  const Rstr = R.stride;

  forall ijk in S.domain do
    S(ijk) = w(0) * R(ijk)
           + w(1) * (R(ijk+Rstr*(1,0,0)) + R(ijk+Rstr*(-1,0,0))
                    + R(ijk+Rstr*(0,1,0)) + R(ijk+Rstr*(0,-1,0))
                    + R(ijk+Rstr*(0,0,1)) + R(ijk+Rstr*(0,0,-1)))
           + w(2) * (R(ijk+Rstr*(1,1,0)) + R(ijk+Rstr*(1,-1,0))
                    + R(ijk+Rstr*(-1,1,0)) + R(ijk+Rstr*(-1,-1,0))
                    + R(ijk+Rstr*(1,0,1)) + R(ijk+Rstr*(1,0,-1))
                    + R(ijk+Rstr*(-1,0,1)) + R(ijk+Rstr*(-1,0,-1))
                    + R(ijk+Rstr*(0,1,1)) + R(ijk+Rstr*(0,1,-1))
                    + R(ijk+Rstr*(0,-1,1)) + R(ijk+Rstr*(0,-1,-1)))
           + w(3) * (R(ijk+Rstr*(1,1,1) + R(ijk+Rstr*(1,1,-1))
                    + R(ijk+Rstr*(1,-1,1) + R(ijk+Rstr*(1,-1,-1))
                    + R(ijk+Rstr*(-1,1,1) + R(ijk+Rstr*(-1,1,-1))
                    + R(ijk+Rstr*(-1,-1,1) + R(ijk+Rstr*(-1,-1,-1)));
}
```

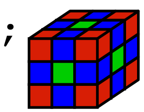
Multigrid: Stencils in Chapel

…or, note that a stencil is simply a reduction over a small subarray leading to a “syntactically scalable” version:

```
def rprj3(S, R) {
  const Stencil = [-1..1, -1..1, -1..1]
        w: [0..3] real = (0.5, 0.25, 0.125, 0.0625),
        w3d = [(i,j,k) in Stencil] w((i!=0) + (j!=0) + (k!=0));

  forall ijk in S.domain do
    S(ijk) = + reduce [offset in Stencil]
                         (w3d(offset) * R(ijk + offset*R.stride));
}
```







Our previous work in ZPL showed that compact, global-view codes like these can result in performance that matches or beats hand-coded Fortran+MPI while also supporting more runtime flexibility

Fortran+MPI NAS MG *rprj3* stencil

```fortran
subroutine comm3(u,n1,n2,n3,kk)
use caf_intrinsics

implicit none

include 'cafnpb.h'
include 'globals.h'

integer n1, n2, n3, kk
double precision u(n1,n2,n3)
integer axis

if( .not. dead(kk) )then
   do  axis = 1, 3
      if( nprocs .ne. 1) then
         call sync_all()
         call give3( axis, +1, u, n1, n2, n3, kk )
         call give3( axis, -1, u, n1, n2, n3, kk )
         call sync_all()
         call take3( axis, -1, u, n1, n2, n3 )
         call take3( axis, +1, u, n1, n2, n3 )
      else
         call comm1p( axis, u, n1, n2, n3, kk )
      endif
   enddo
else
   do  axis = 1, 3
      call sync_all()
      call sync_all()
   enddo
   call zero3(u,n1,n2,n3)
endif
return
end


subroutine give3( axis, dir, u, n1, n2, n3, k )
use caf_intrinsics


implicit none

include 'cafnpb.h'
include 'globals.h'

integer axis, dir, n1, n2, n3, k, ierr
double precision u( n1, n2, n3 )

integer i3, i2, i1, buff_len,buff_id

buff_id = 2 + dir
buff_len = 0

if( axis .eq.  1 )then
   if( dir .eq. -1 )then

      do  i3=2,n3-1
         do  i2=2,n2-1
            buff_len = buff_len + 1
            buff(buff_len,buff_id ) = u( 2,  i2,i3)
         enddo
      enddo

      buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
>     buff(1:buff_len,buff_id)

   else if( dir .eq. +1 ) then

      do  i3=2,n3-1
         do  i2=2,n2-1
            buff_len = buff_len + 1
            buff(buff_len, buff_id ) = u( n1-1, i2,i3)
         enddo
      enddo

      buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
>     buff(1:buff_len,buff_id)

   endif
endif

if( axis .eq.  2 )then
   if( dir .eq. -1 )then
      do  i3=2,n3-1
         do  i1=1,n1
            buff_len = buff_len + 1
            buff(buff_len, buff_id ) = u( i1,  2,i3)
         enddo
      enddo

  buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
>     buff(1:buff_len,buff_id)

else if( dir .eq. +1 ) then

      do  i3=2,n3-1
         do  i1=1,n1
            buff_len = buff_len + 1
            buff(buff_len,  buff_id )= u( i1,n2-1,i3)
         enddo
      enddo

      buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
>     buff(1:buff_len,buff_id)

   endif
endif

if( axis .eq.  3 )then
   if( dir .eq. -1 )then

      do  i2=1,n2
         do  i1=1,n1
            buff_len = buff_len + 1
            buff(buff_len, buff_id ) = u( i1,i2,2)
         enddo
      enddo

      buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
>     buff(1:buff_len,buff_id)

   else if( dir .eq. +1 ) then

      do  i2=1,n2
         do  i1=1,n1
            buff_len = buff_len + 1
            buff(buff_len, buff_id ) = u( i1,i2,n3-1)
         enddo
      enddo

      buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
>     buff(1:buff_len,buff_id)

   endif
endif

return
end


subroutine take3( axis, dir, u, n1, n2, n3 )
use caf_intrinsics


implicit none

include 'cafnpb.h'
include 'globals.h'

integer axis, dir, n1, n2, n3
double precision u( n1, n2, n3 )

integer buff_id, indx

integer i3, i2, i1

buff_id = 3 + dir
indx = 0

if( axis .eq.  1 )then
   if( dir .eq. -1 )then

      do  i3=2,n3-1
         do  i2=2,n2-1
            indx = indx + 1
            u(n1,i2,i3) = buff(indx, buff_id )
         enddo
      enddo

   else if( dir .eq. +1 ) then

      do  i3=2,n3-1
         do  i2=2,n2-1
            indx = indx + 1
            u(1,i2,i3) = buff(indx, buff_id )
         enddo
      enddo

   endif
endif

if( axis .eq.  2 )then
   if( dir .eq. -1 )then

      do  i3=2,n3-1
         do  i1=1,n1
            indx = indx + 1
            u(i1,n2,i3) = buff(indx, buff_id )
         enddo
      enddo

else if( dir .eq. +1 ) then

      do  i3=2,n3-1
         do  i1=1,n1
            indx = indx + 1
            u(i1,1,i3) = buff(indx, buff_id )
         enddo
      enddo

   endif
endif

if( axis .eq.  3 )then
   if( dir .eq. -1 )then

      do  i2=1,n2
         do  i1=1,n1
            indx = indx + 1
            u(i1,i2,n3) = buff(indx, buff_id )
         enddo
      enddo

   else if( dir .eq. +1 ) then

      do  i2=1,n2
         do  i1=1,n1
            indx = indx + 1
            u(i1,i2,1) = buff(indx, buff_id )
         enddo
      enddo

   endif
endif

return
end


subroutine comm1p( axis, u, n1, n2, n3, kk )
use caf_intrinsics


implicit none

include 'cafnpb.h'
include 'globals.h'

integer axis, dir, n1, n2, n3
double precision u( n1, n2, n3 )

integer i3, i2, i1, buff_len,buff_id
integer i, kk, indx

dir = -1

buff_id = 3 + dir
buff_len = nm2

do  i=1,nm2
   buff(i,buff_id) = 0.0D0
enddo


dir = +1

buff_id = 3 + dir
buff_len = nm2

do  i=1,nm2
   buff(i,buff_id) = 0.0D0
enddo

dir = +1

buff_id = 2 + dir
buff_len = 0

if( axis .eq.  1 )then
   do  i3=2,n3-1
      do  i2=2,n2-1
         buff_len = buff_len + 1
         buff(buff_len, buff_id ) = u( n1-1,
 i2,i3)
      enddo
   enddo
endif

if( axis .eq.  2 )then
   do  i3=2,n3-1
      do  i1=1,n1
         buff_len = buff_len + 1
         buff(buff_len,  buff_id )= u( i1,n2-
 1,i3)
      enddo
   enddo
endif

if( axis .eq.  3 )then
   do  i2=1,n2
      do  i1=1,n1
         buff_len = buff_len + 1
         buff(buff_len, buff_id ) = u( i1,i2,n3-
 1)
      enddo
   enddo
endif

dir = -1

buff_id = 2 + dir
buff_len = 0

if( axis .eq.  1 )then
   do  i3=2,n3-1
      do  i2=2,n2-1
         buff_len = buff_len + 1
         buff(buff_len,buff_id ) = u( 2,  i2,i3)
      enddo
   enddo
endif

if( axis .eq.  2 )then
   do  i3=2,n3-1
      do  i1=1,n1
         buff_len = buff_len + 1
         buff(buff_len, buff_id ) = u( i1,
 2,i3)
      enddo
   enddo
endif

if( axis .eq.  3 )then
   do  i2=1,n2
      do  i1=1,n1
         buff_len = buff_len + 1
         buff(buff_len, buff_id ) = u( i1,i2,2)
      enddo
   enddo
endif

do  i=1,nm2
   buff(i,4) = buff(i,3)
   buff(i,2) = buff(i,1)
enddo

dir = -1

buff_id = 3 + dir
indx = 0

if( axis .eq.  1 )then
   do  i3=2,n3-1
      do  i2=2,n2-1
         indx = indx + 1
         u(n1,i2,i3) = buff(indx, buff_id )
      enddo
   enddo
endif

if( axis .eq.  2 )then
   do  i3=2,n3-1
      do  i1=1,n1
         indx = indx + 1
         u(i1,n2,i3) = buff(indx, buff_id )
      enddo
   enddo
endif

if( axis .eq.  3 )then
   do  i2=1,n2
      do  i1=1,n1
         indx = indx + 1
         u(i1,i2,n3) = buff(indx, buff_id )
      enddo
   enddo
endif


dir = +1

buff_id = 3 + dir
indx = 0

if( axis .eq.  1 )then
   do  i3=2,n3-1
      do  i2=2,n2-1
         indx = indx + 1
         u(1,i2,i3) = buff(indx, buff_id )
      enddo
   enddo
endif

if( axis .eq.  2 )then
   do  i3=2,n3-1
      do  i1=1,n1
         indx = indx + 1
         u(i1,1,i3) = buff(indx, buff_id )
      enddo
   enddo
endif

if( axis .eq.  3 )then
   do  i2=1,n2
      do  i1=1,n1
         indx = indx + 1
         u(i1,i2,1) = buff(indx, buff_id )
      enddo
   enddo
endif

return
end

subroutine rprj3(r,m1k,m2k,m3k,s,m1j,m2j,m3j,k)
implicit none
include 'cafnpb.h'
include 'globals.h'

integer m1k, m2k, m3k, m1j, m2j, m3j,k

double precision r(m1k,m2k,m3k), s(m1j,m2j,m3j)
integer j3, j2, j1, i3, i2, i1, d1, d2, d3, j
double precision x1(m), y1(m), x2,y2

if(m1k.eq.3)then
  d1 = 2
else
  d1 = 1
endif

if(m2k.eq.3)then
  d2 = 2
else
  d2 = 1
endif

if(m3k.eq.3)then
  d3 = 2
else
  d3 = 1
endif

do  j3=2,m3j-1
  i3 = 2*j3-d3
  do  j2=2,m2j-1
    i2 = 2*j2-d2
    do j1=2,m1j
      i1 = 2*j1-d1
      x1(i1-1) = r(i1-1,i2-1,i3  ) + r(i1-1,i2+1,i3  )
>                + r(i1-1,i2,  i3-1) + r(i1-1,i2,  i3+1)
      y1(i1-1) = r(i1-1,i2-1,i3-1) + r(i1-1,i2-1,i3+1)
>                + r(i1-1,i2+1,i3-1) + r(i1-1,i2+1,i3+1)
    enddo
    do  j1=2,m1j-1
      i1 = 2*j1-d1
      y2 = r(i1,  i2-1,i3-1) + r(i1,  i2-1,i3+1)
>        + r(i1,  i2+1,i3-1) + r(i1,  i2+1,i3+1)
      x2 = r(i1,  i2-1,i3  ) + r(i1,  i2+1,i3  )
>        + r(i1,  i2,  i3-1) + r(i1,  i2,  i3+1)
      s(j1,j2,j3) =
>         0.5D0 * r(i1,i2,i3)
>       + 0.25D0 * (r(i1-1,i2,i3) + r(i1+1,i2,i3) + x2)
>       + 0.125D0 * ( x1(i1-1) + x1(i1+1) + y2)
>       + 0.0625D0 * ( y1(i1-1) + y1(i1+1) )
      enddo
    enddo
  enddo
  j = k-1
  call comm3(s,m1j,m2j,m3j,j)
  return
  end
```

Stencil 3: Fast Multipole Method (FMM)

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

1D array over levels of the hierarchy

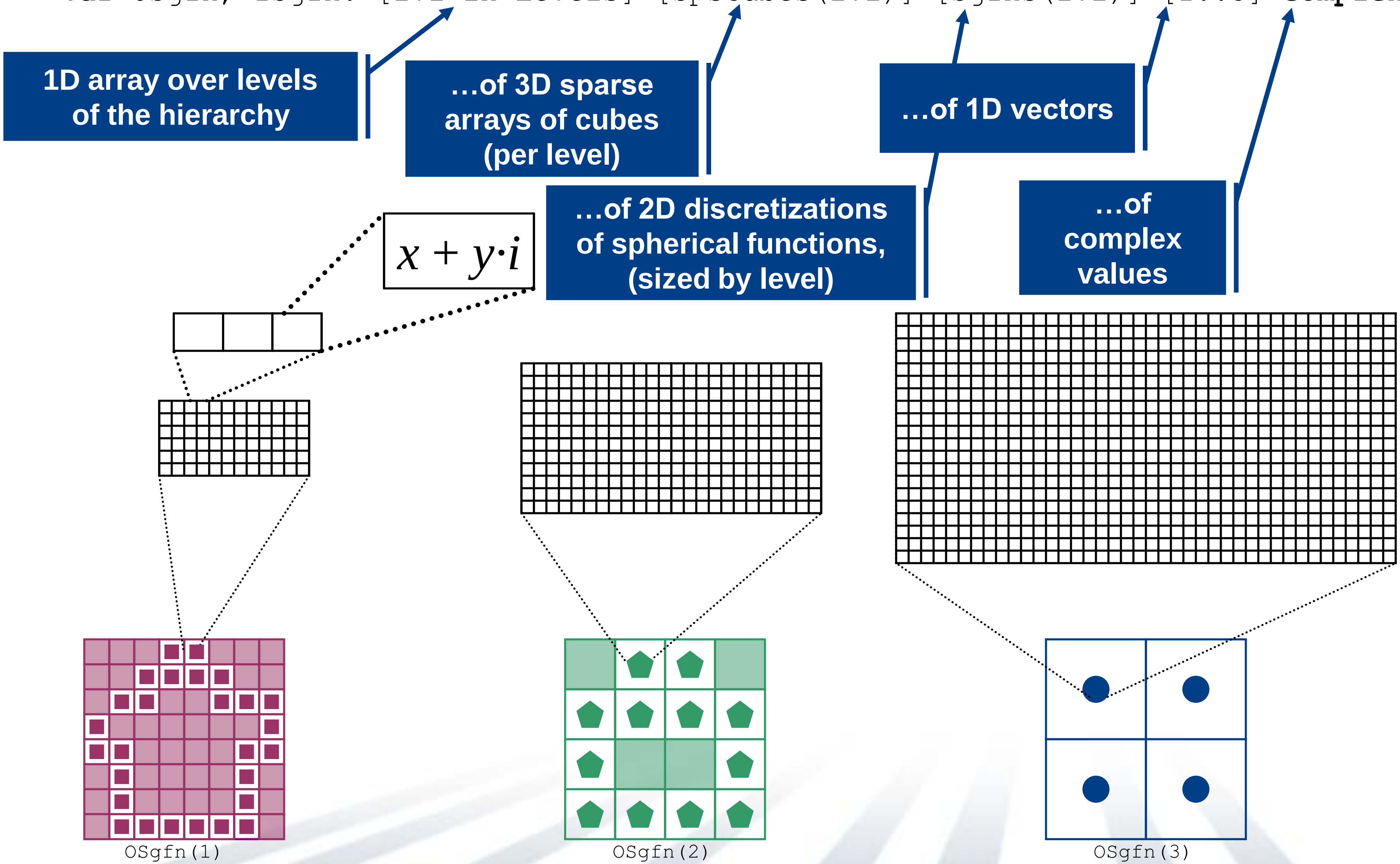
OSgfn(1)

OSgfn(2)

OSgfn(3)

Stencil 3: Fast Multipole Method (FMM)

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

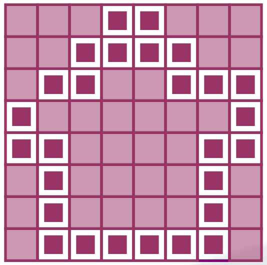


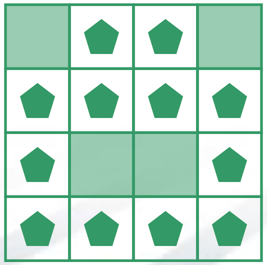
FMM: Supporting Declarations

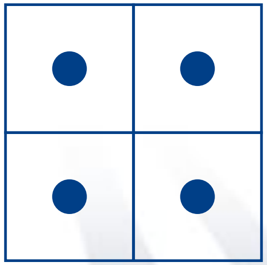
```
var OSgfn, ISgfn: [lvl in Levels] [SpsCubes(lvl)] [Sgfns(lvl)] [1..3] complex;
```

previous definitions:

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

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

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

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

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


OSgfn(1)


OSgfn(2)

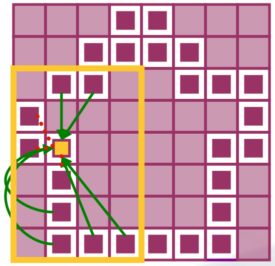

OSgfn(3)

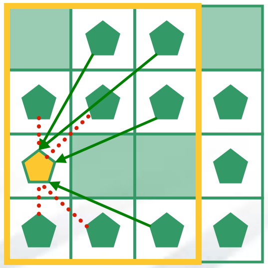
FMM: Computation

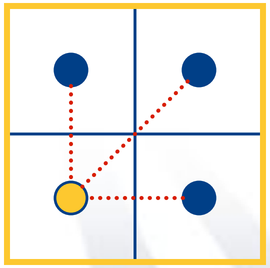
```
var OSgfn, ISgfn: [lvl in Levels] [SpsCubes(lvl)] [Sgfns(lvl)] [1..3] complex;
```

outer-to-inner translation:

```
for lvl in 1..numLevels-1 by -1 {
  …
  forall cube in SpsCubes(lvl) {
    forall sib in out2inSiblings(lvl, cube) {
      const Trans = lookupXlateTab(cube, sib);

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


OSgfn(1)


OSgfn(2)

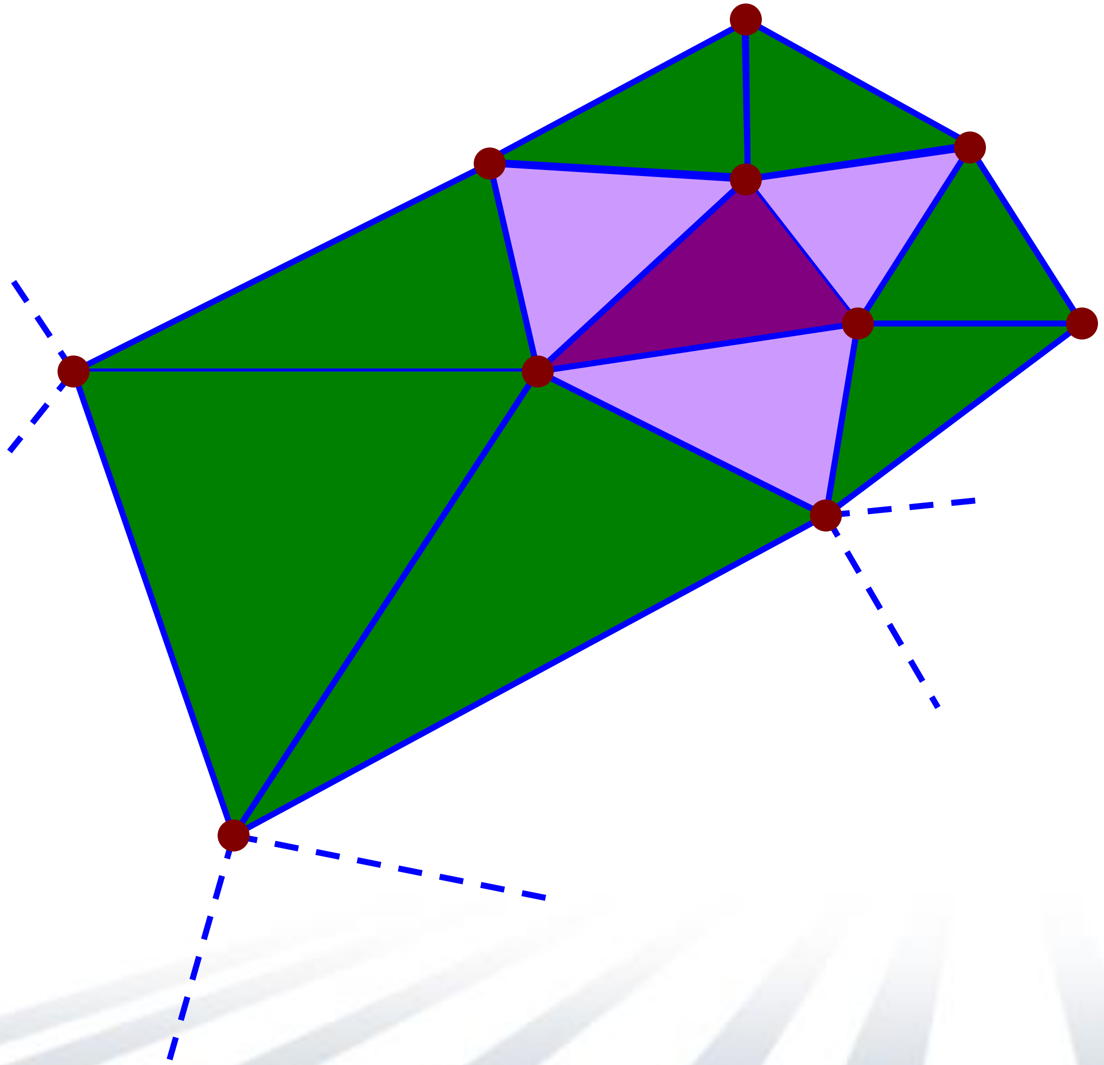

OSgfn(3)

Fast Multipole Method: Summary

- Chapel code captures structure of data and computation far better than sequential Fortran/C versions (to say nothing of the MPI versions)
  - cleaner, more succinct, more informative
  - rich domain/array support plays a big role in this
- Parallelism shifts at different levels of hierarchy
  - Aided by global-view programming and nested parallelism
- Boeing FMM expert was able to find bugs in my implementation when seeing Chapel for the first time
- Yet, I’ve elided some non-trivial code (the distributions)

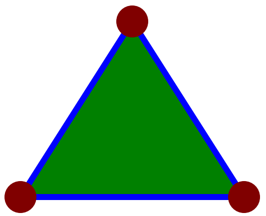
Stencil 4: Stencils on Unstructured Grids

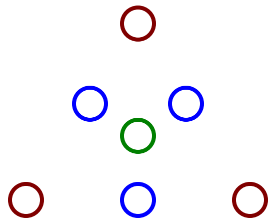
- e.g., Finite Element Methods (FEM)

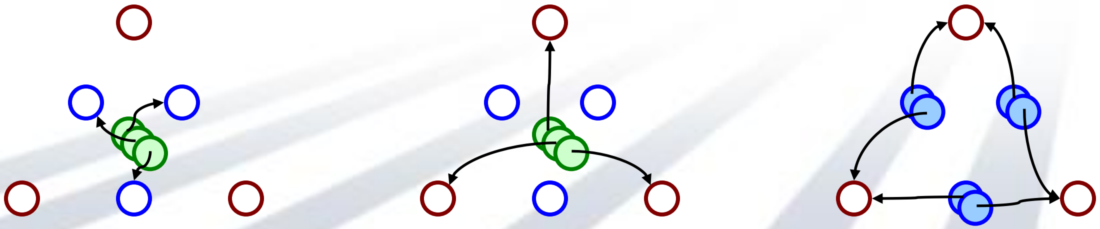


FEM Declarations

```
config param numdims = 2;
const facesPerElem = numdims+1,
      vertsPerFace = 3,
      vertsPerElem = numdims+1;

var Elements: domain(opaque),
    Faces: domain(opaque),
    Vertices: domain(opaque);

var element: index(Elements),
    face: index(Faces),
    vertex: index(Vertices);

var elementFaces: [Elements] [1..facesPerElem] face,
    elemVertices: [Elements] [1..vertsPerElem] vertex,
    faceVertices: [Faces] [1..vertsPerFace] vertex;
```







FEM Computation

- Sample Idioms:

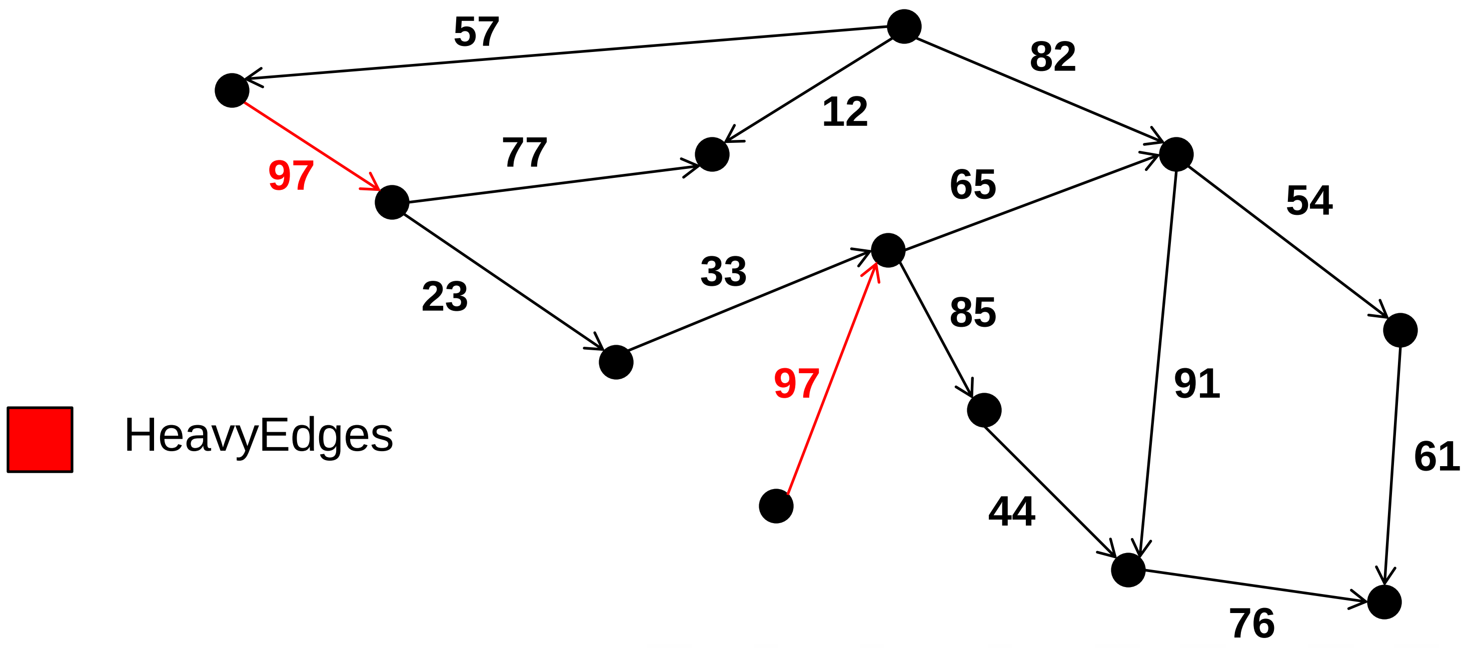
```
var a, b, c, f: [Vertices] real;
var p: [1..2, Vertices] real;

function PoissonComputeA {
  forall e in Elements {
    const c = 0.10 * volume(e);
    for v in elemVertices(e) {
      a(v1) += c*f(v1);
      for v2 in elemVertices(e) do
        if (v1 != v2) then
          a(v2) += 0.5*c*f(v2);
    }
  }
}

function computePressure(pressure: [Vertices] real) {
  pressure = (a - b) / c;
}
```

Outline

- ✓ Chapel Overview
- ➢ Chapel computations
  - **your first Chapel program:** STREAM Triad
  - ***the stencil ramp*****:** from jacobi to finite element methods
  - **graph-based computation in Chapel:** SSCA #2
  - **task-parallelism:** producer-consumer to MADNESS
  - **GPU computing in Chapel:** STREAM revisited and CP
- Status, Summary, and Future Work

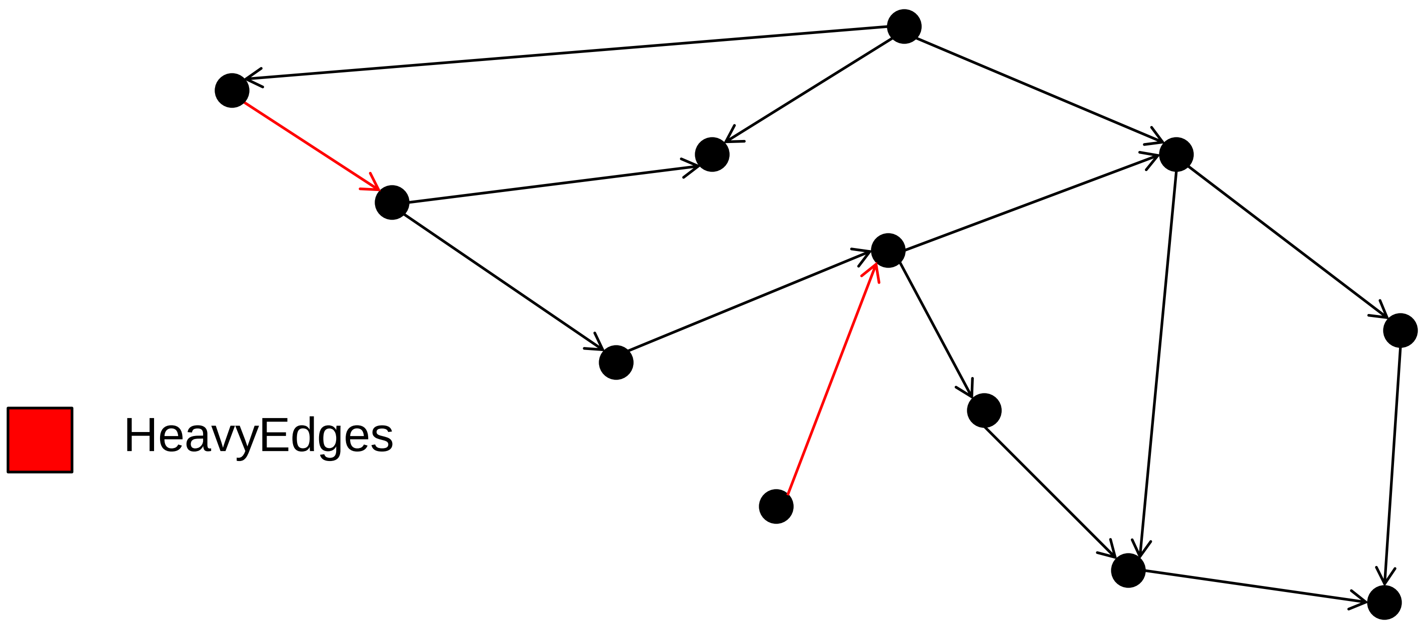
HPCS SSCA #2, kernel 2

Definition: Given a set of heavy root edges (*HeavyEdges*) in a directed graph *G*, find the subgraphs formed by outgoing paths with length ≤ *maxPathLength*



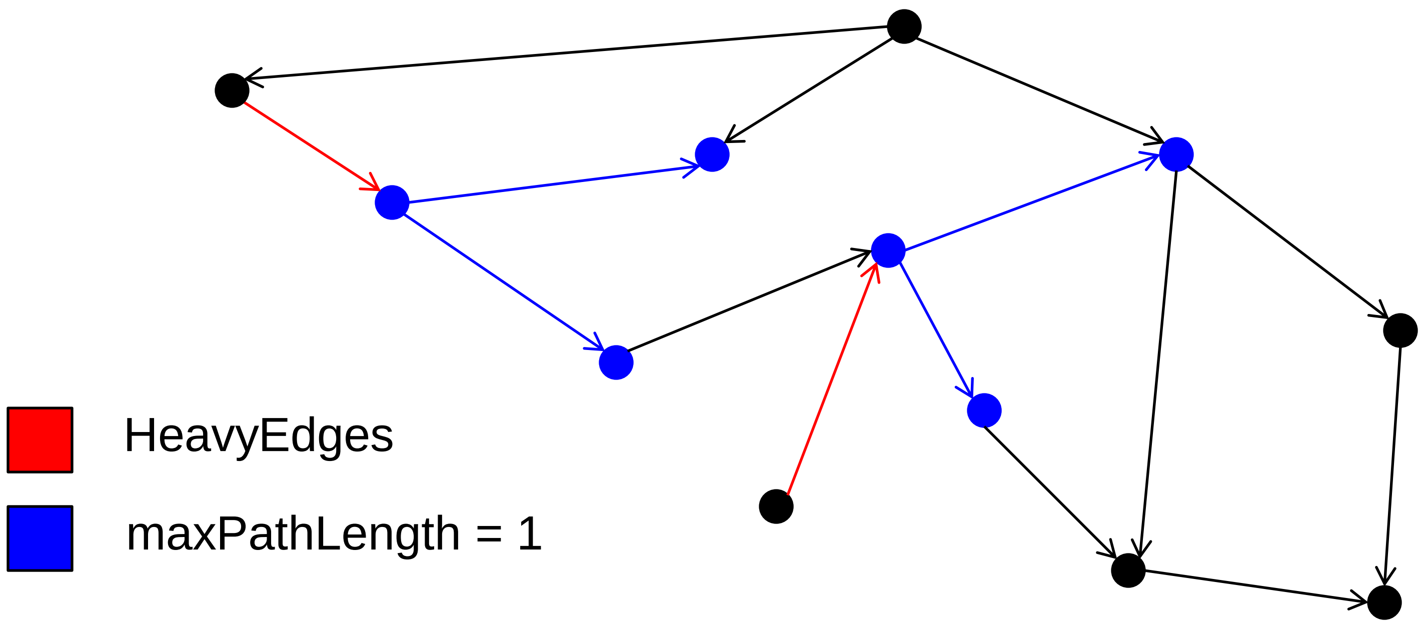
HPCS SSCA #2, kernel 2

Definition: Given a set of heavy root edges (*HeavyEdges*) in a directed graph *G*, find the subgraphs formed by outgoing paths with length ≤ *maxPathLength*



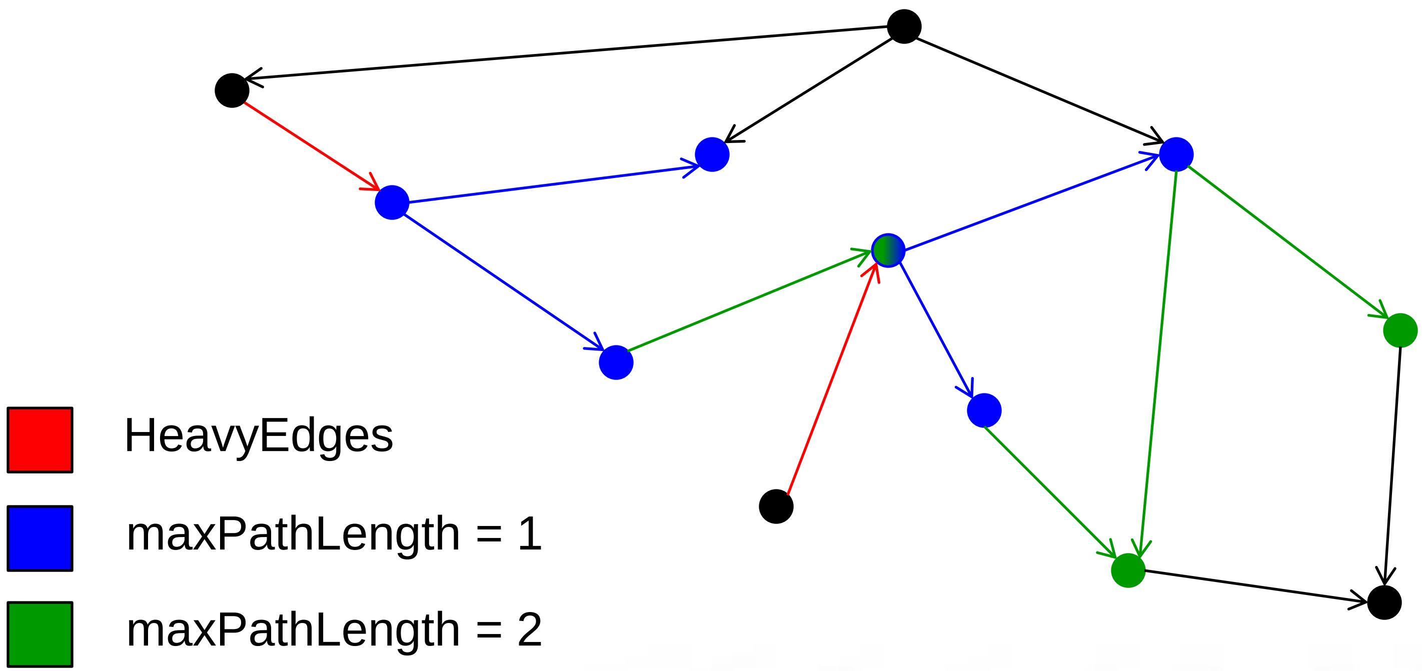
HPCS SSCA #2, kernel 2

Definition: Given a set of heavy root edges (*HeavyEdges*) in a directed graph *G*, find the subgraphs formed by outgoing paths with length ≤ *maxPathLength*



HPCS SSCA #2, kernel 2

Definition: Given a set of heavy root edges (*HeavyEdges*) in a directed graph *G*, find the subgraphs formed by outgoing paths with length ≤ *maxPathLength*



HPCS SSCA #2, kernel 2

```
def rootedHeavySubgraphs(
      G,
      type vertexSet;
      HeavyEdges       : domain,
      HeavyEdgeSubG    : [],
      in maxPathLength: int ) {

  forall (e, subgraph)
      in (HeavyEdges, HeavyEdgeSubG) {

    const (x,y) = e;
    var ActiveLevel: vertexSet;

    ActiveLevel += y;

    subgraph.edges += e;
    subgraph.nodes += x;
    subgraph.nodes += y;

      for pathLength in 1..maxPathLength {
        var NextLevel: vertexSet;
        forall v in ActiveLevel do
          forall w in G.Neighbors(v) do
            atomic {
              if !subgraph.nodes.member(w) {
                NextLevel += w;
                subgraph.nodes += w;
                subgraph.edges += (v, w);
              }
            }

        if (pathLength < maxPathLength) then
          ActiveLevel = NextLevel;
      }
    }
  }
```


Original code courtesy of John Lewis, Cray Inc.

HPCS SSCA #2, kernel 2

```
def rootedHeavySubgraphs(
      G,
      type vertexSet;
      HeavyEdges      : domain,
      HeavyEdgeSubG   : [],
      in maxPathLength: int ) {
```

```
for pathLength in 1..maxPathLength {
  var NextLevel: vertexSet;
  forall v in ActiveLevel do
    forall w in G.Neighbors(v) do
      atomic {
f
.nodes.member(w) {
+= w;
odes += w;
dges += (v, w);

axPathLength) then
xtLevel;
```

<u>Generic Implementation of Graph G</u>

G.Vertices: a domain whose indices represent the vertices
- for toroidal graphs, a domain(*d*), so vertices are *d*-tuples
- for other graphs, a domain(1), so vertices are integers

G.Neighbors: an array over G.Vertices
- for toroidal graphs, a fixed-size array over the domain [1..2**d*]
- for other graphs…
  - …an associative domain with indices of type index(G.vertices)
  - …a sparse subdomain of G.Vertices

This kernel and the others are generic w.r.t. these decisions!

Original code courtesy of John Lewis, Cray Inc.

HPCS SSCA #2, kernel 2

```
def rootedHeavySubgraphs(
      G,
      type vertexSet;
      HeavyEdges       : domain,
      HeavyEdgeSubG    : [],
      in maxPathLength: int ) {
```

```
for pathLength in 1..maxPathLength {
  var NextLevel: vertexSet;
  forall v in ActiveLevel do
    forall w in G.Neighbors(v) do
      atomic {
        if !subgraph.nodes.member(w) {
          NextLevel += w;
          subgraph.nodes += w;
          subgraph.edges += (v, w);
        }
      }

  if (pathLength < maxPathLength) then
    ActiveLevel = NextLevel;
```

Generic with respect to vertex sets

vertexSet: a type argument specifying how to represent vertex subsets

Requirements:
- parallel iteration
- ability to add members, test for membership

Options:
- an associative domain over vertices
  `domain(index(G.vertices))`
- a sparse subdomain of the vertices
  `sparse subdomain(G.vertices)`

Original code courtesy of John Lewis, Cray Inc.

HPCS SSCA #2, kernel 2

```
def rootedHeavySubgraphs(
      G,
      type vertexSet;
      HeavyEdges       : domain,
      HeavyEdgeSubG    : [],
      in maxPathLength: int ) {

  forall (e, subgraph)
      in (HeavyEdges, HeavyEdgeSubG) {

    const (x,y) = e;
    var ActiveLevel: vertexSet;

    ActiveLevel += y;

    subgraph.edges += e;
    subgraph.nodes += x;
    subgraph.nodes += y;
```

```
    for pathLength in 1..maxPathLength {
      var NextLevel: vertexSet;
      forall v in ActiveLevel do
        forall w in G.Neighbors(v) do
          atomic {
            if !subgraph.nodes.member(w) {
              NextLevel += w;
              subgraph.nodes += w;
              subgraph.edges += (v, w);
            }
          }
                        < maxPathLength) then
        ActiveLevel = NextLevel;
    }
  }
}
```

Ditto for Subgraphs

Original code courtesy of John Lewis, Cray Inc.

HPCS SSCA #2, kernel 2

```
def rootedHeavySubgraphs(
      G,
      type vertexSet;
      HeavyEdges       : domain,
      HeavyEdgeSubG    : [],
      in maxPathLength: int ) {

  forall (e, subgraph)
      in (HeavyEdges, HeavyEdgeSubG) {

    const (x,y) = e;
    var ActiveLevel: vertexSet;

    ActiveLevel += y;

    subgraph.edges += e;
    subgraph.nodes += x;
    subgraph.nodes += y;

    for pathLength in 1..maxPathLength {
      var NextLevel: vertexSet;
      forall v in ActiveLevel do
        forall w in G.Neighbors(v) do
          atomic {
            if !subgraph.nodes.member(w) {
              NextLevel += w;
              subgraph.nodes += w;
              subgraph.edges += (v, w);
            }
          }

      if (pathLength < maxPathLength) then
        ActiveLevel = NextLevel;
    }
  }
}
```


Original code courtesy of John Lewis, Cray Inc.

Outline

- ✓ Chapel Overview
- ➢ Chapel computations
  - **your first Chapel program:** STREAM Triad
  - ***the stencil ramp*: from jacobi to finite element methods**
  - **graph-based computation in Chapel:** SSCA #2
  - **task-parallelism:** producer-consumer to MADNESS
  - **GPU computing in Chapel:** STREAM revisited and CP
- Status, Summary, and Future Work

Task Parallelism: Producer/Consumer

```
var buff$: [0..buffersize-1] sync int;

cobegin {
  producer();
  consumer();
}

def producer() {
  var i = 0;
  for … {
    i = (i+1) % buffersize;
    buff$(i) = …;
  }
}

def consumer() {
  var i = 0;
  while {
    i = (i+1) % buffersize;
    …buff$(i)…;
  }
}
```

Task Parallelism: Producer/Consumer

```
var buff$: [0..buffersize-1] sync int;

cobegin {
  producer();
  consumer();
}

def producer() {
  var i = 0;
  for … {
    i = (i+1) % bu
    buff$(i) = …;
  }
}

def consumer() {
  var i = 0;
  while {
    i = (i+1) % bu
    …buff$(i)…;
  }
}
```

Synchronization Variables

- Store full/empty state along with value
- By default…
  …reads block until full, leave empty
  …writes block until empty, leave full
- methods provide other forms of read/write
  e.g., buff$[0].readXX(); => read, ignoring state
- Chapel also has single-assignment variables
  - write once, read many times

Task Parallelism: Producer/Consumer

```
var buff$: [0..buffersize-1] sync int;

cobegin {
  producer();
  consumer();
}

def producer() {
  var i =
  for … {
    i = (
    buff$
  }
}

def consu
  var i =
  while {
    i = (i+1) % buffersize;
    …buff$(i)…;
  }
}
```

Cobegins

- Spawn a task for each component statement
- Original task waits until the tasks have finished
- Chapel also supports other flavors of structured & unstructured task creation

Task Parallelism: Producer/Consumer

```
var buff$: [0..buffersize-1] sync int;

cobegin {
  producer();
  consumer();
}

def producer() {
  var i = 0;
  for … {
    i = (i+1) % buffersize;
    buff$(i) = …;
  }
}

def consumer() {
  var i = 0;
  while {
    i = (i+1) % buffersize;
    …buff$(i)…;
  }
}
```

MADNESS

- ***MADNESS:***
  - Multiresolution ADaptive NumErical Scientific Simulation
  - a framework for scientific simulation in many dimensions using adaptive multiresolution methods in multiwavelet bases
- People:
  - Gregory Beylkin (University of Colorado), George Fann (Oak Ridge National Laboratory), Zhenting Gan (CCSG), **Robert Harrison** (CCSG), Martin Mohlenkamp (Ohio University), Fernando Perez (University of Colorado), P. Sadayappan (The Ohio State University), Takeshi Yanai (CCSG)

contents adapted from James Dinan, the Ohio State University

What does Madness do?

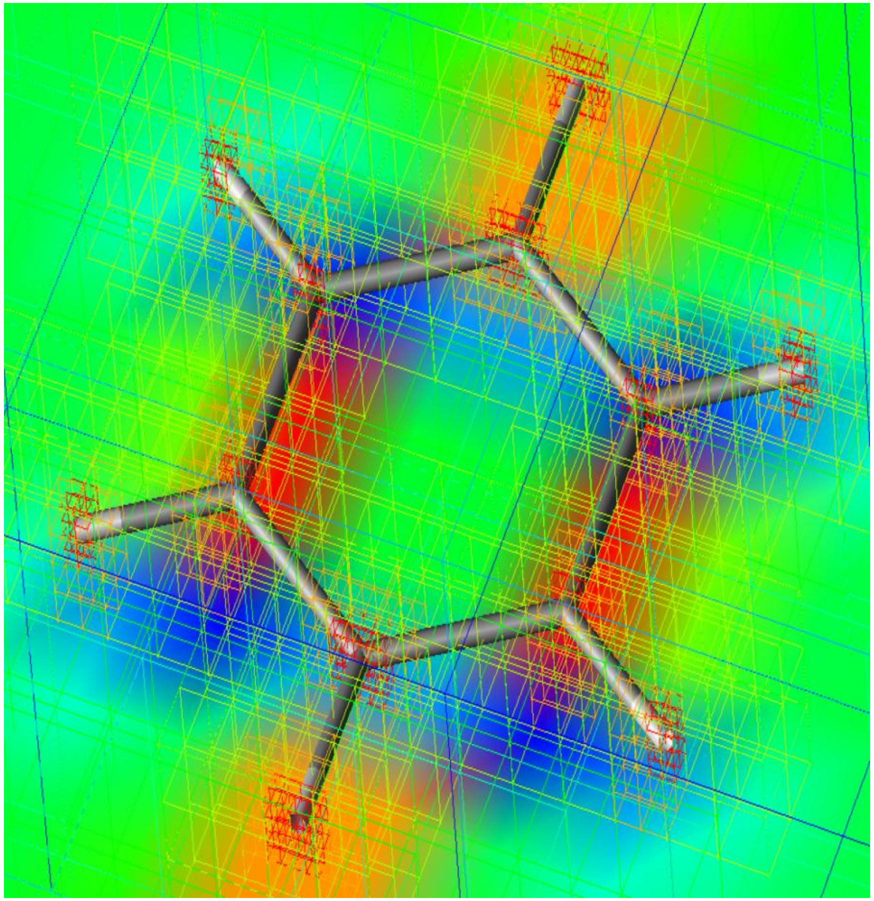
- Think of Madness as a math library
- Numerical representations for analytic functions
  - Stored in the scaling function (Gauss Legendre Polynomial) and Multiwavelet bases
  - Operations on functions become fast with guaranteed precision
  - Differential and Integral operators become O(n) in numerical representation

- Applications that can benefit from Madness include:
  - Density Functional Theory (DFT) (Quantum chemistry domain)
    - Explore electronic structure of many-body systems
  - Fluid dynamics
  - Climate modeling
  - Etc …

contents adapted from James Dinan, the Ohio State University

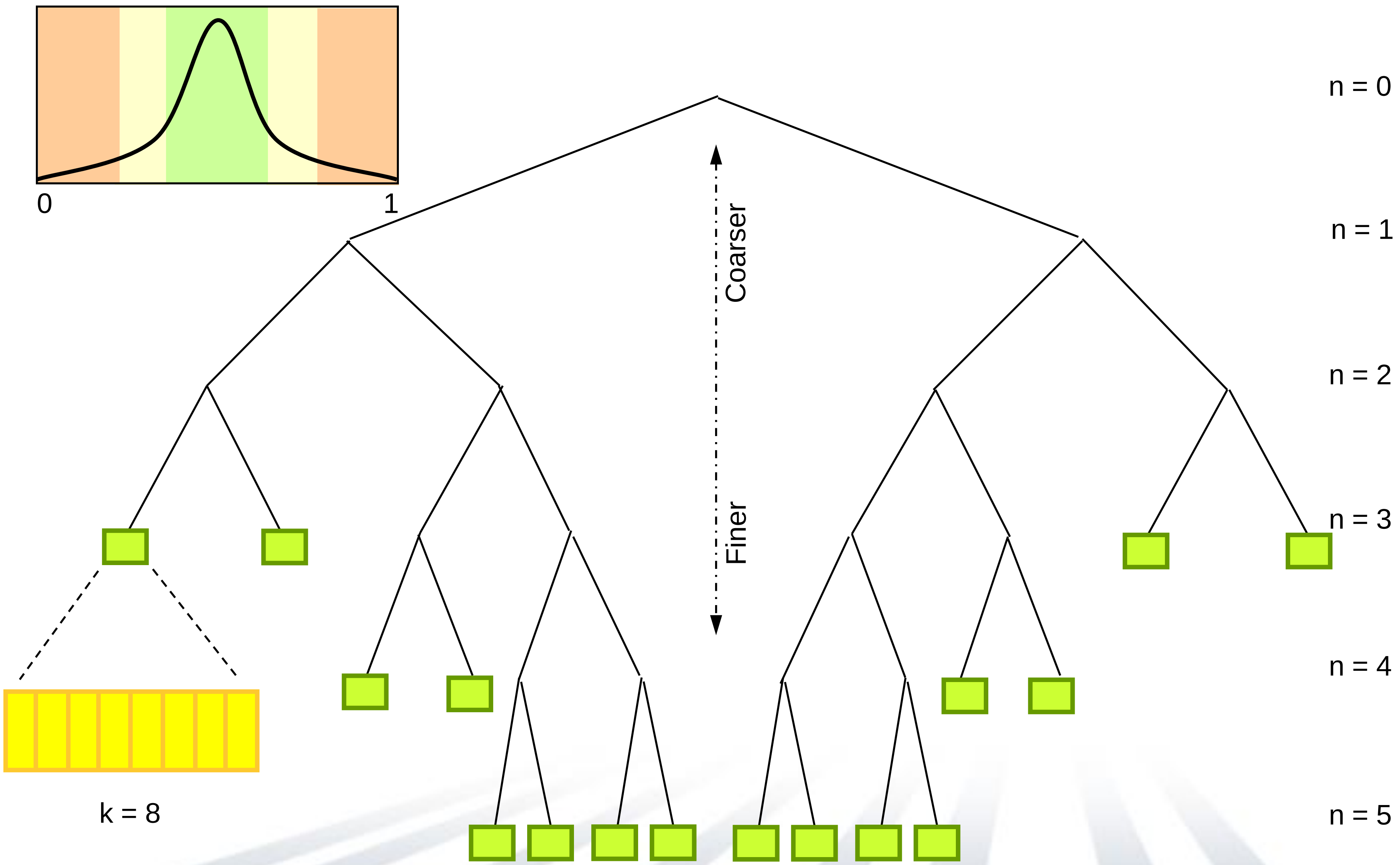
Numerical Representation for Functions

- Analytic function is *projected* into the numerical representation
- Approximate the function using basis functions
  - Similar to Fourier, but basis functions have compact support
  - Approximation is over a closed interval of interest
- Recursively subdivide the analytic function spatially to achieve desired accuracy
- Avoid extra computation in uninteresting areas
- Store the result in a *Function Tree*
  - 1d: Binary Tree
  - 2d: Quad Tree
  - 3d: Oct Tree



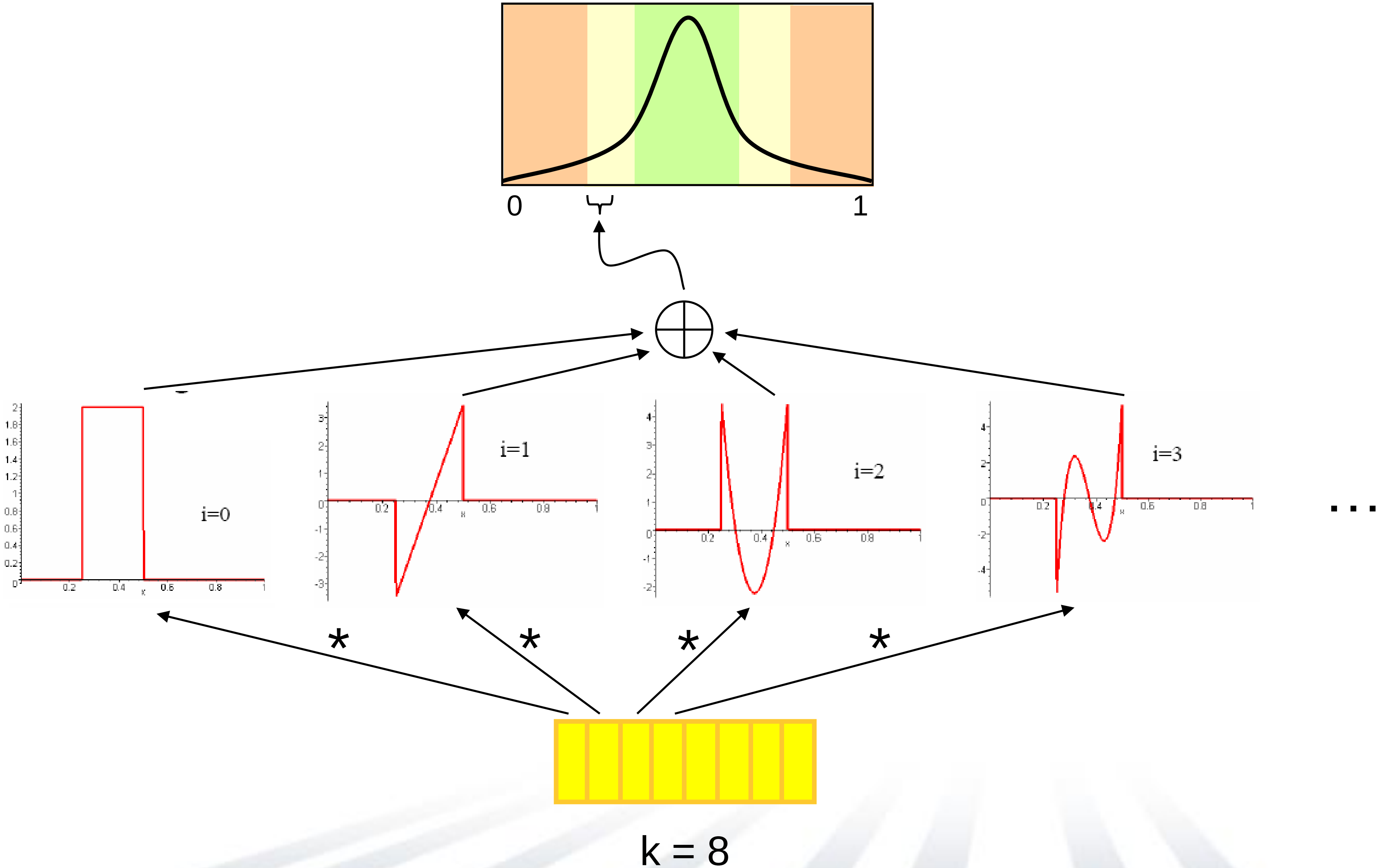
contents adapted from James Dinan, the Ohio State University

The 1d Function Tree of a Gaussian



contents adapted from James Dinan, the Ohio State University

Function Evaluation in the Numerical Representation



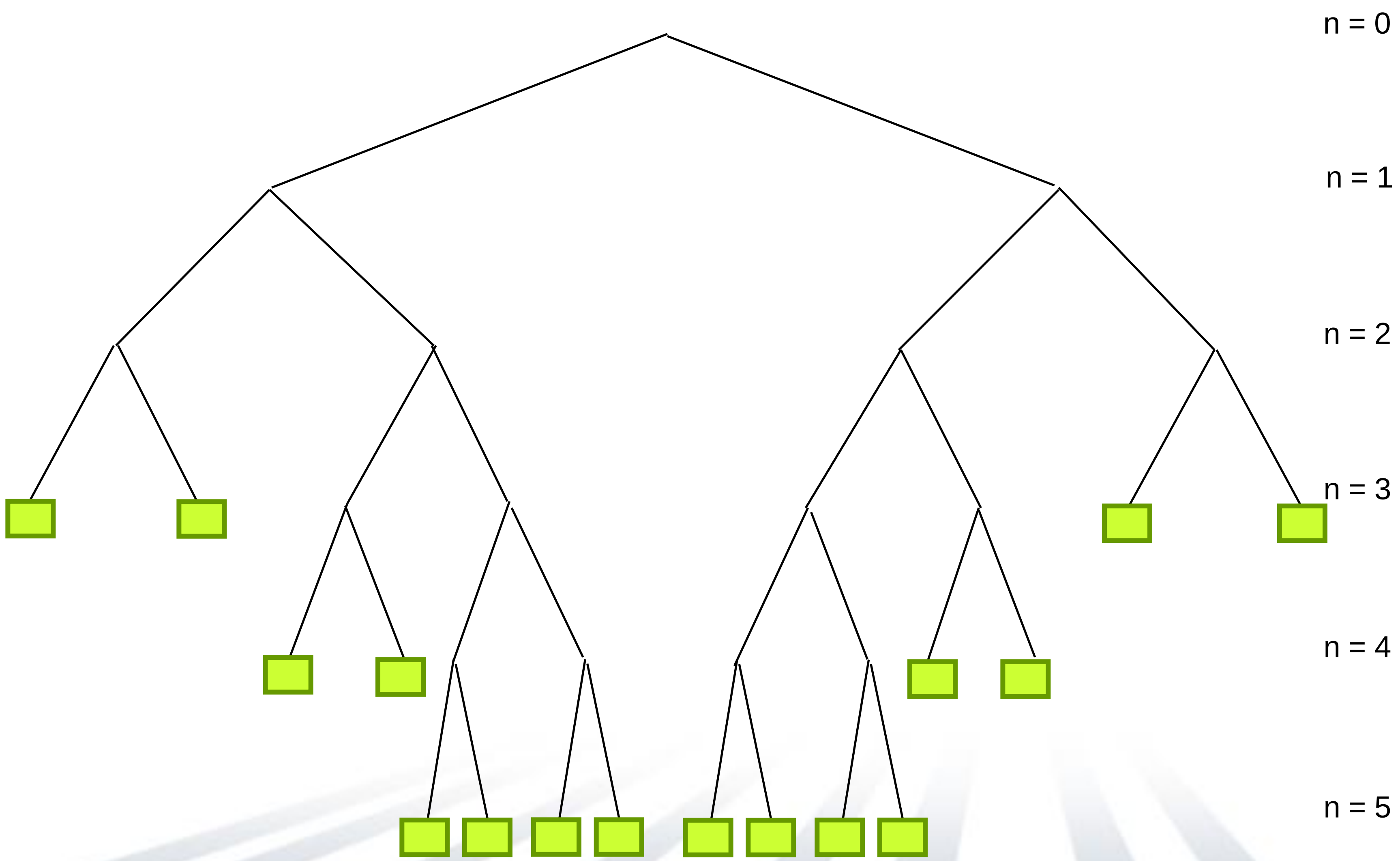
contents adapted from James Dinan, the Ohio State University

Core Algorithm: Differentiation

- Perform: `df = f.diff()`
- Walk down the tree and everywhere that we have coefficients, perform differentiation
- Performing differentiation involves getting our left and right neighbors and applying the derivative operator

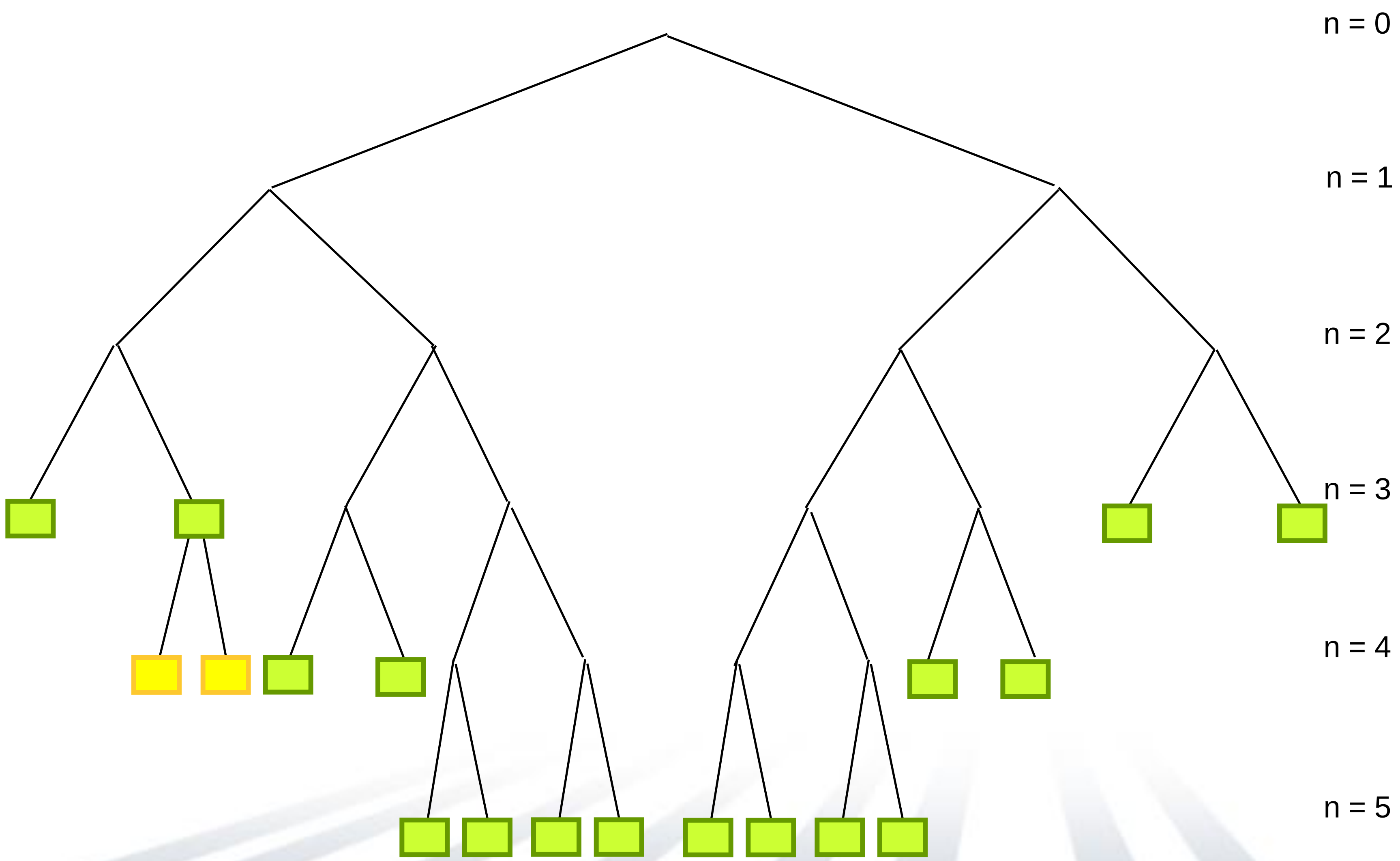
contents adapted from James Dinan, the Ohio State University

Differentiation: I have neighbors



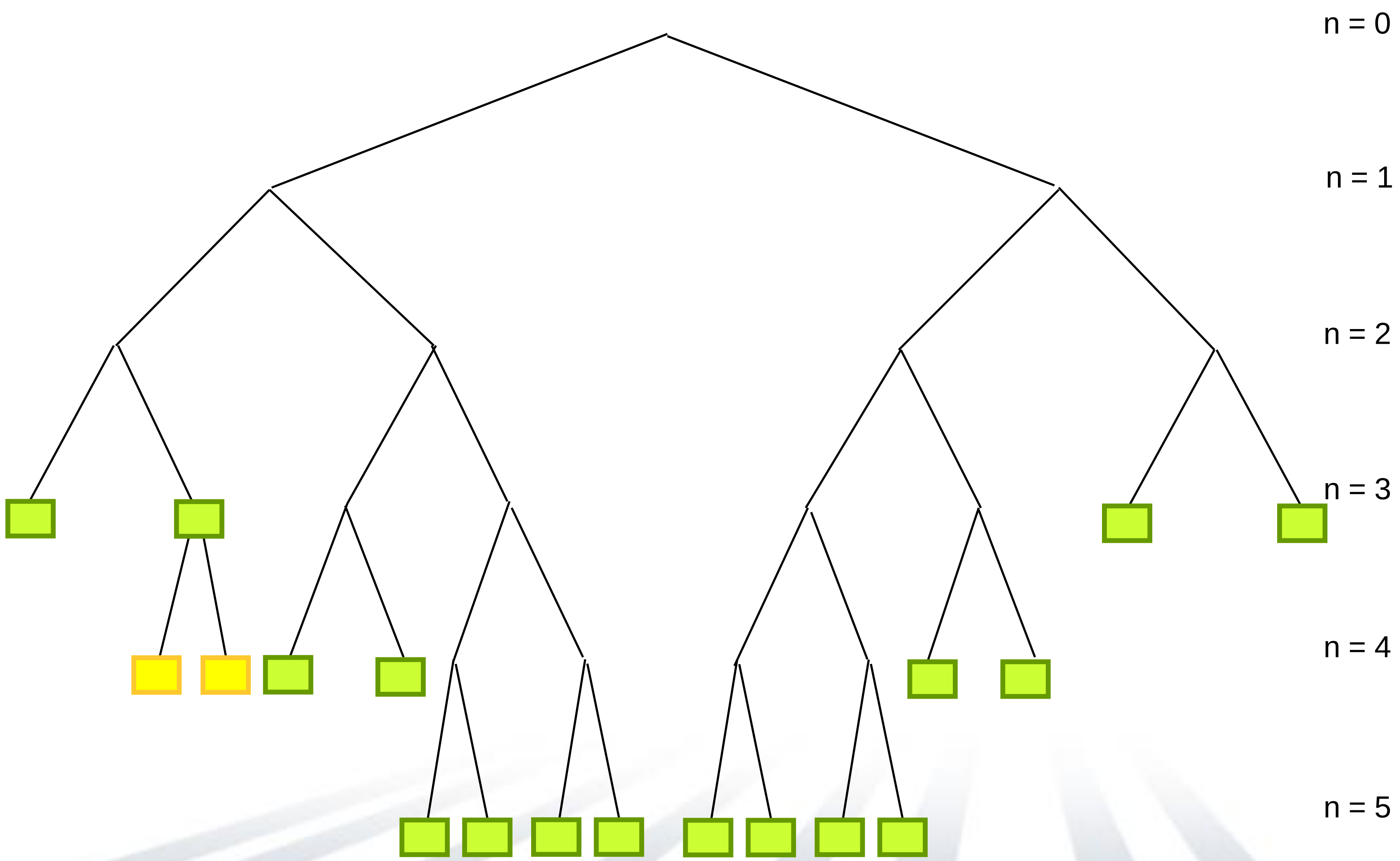
contents adapted from James Dinan, the Ohio State University

Differentiation: I'm too fine



contents adapted from James Dinan, the Ohio State University

Differentiation: I'm too coarse



contents adapted from James Dinan, the Ohio State University

Serial Differentiation Code

```
def diff (n = 0, l = 0, result) {
    if !s.has_coeffs(n, l) {
        // Run down tree until we hit scaling function coefficients

            diff(n+1, 2*l  , result);
            diff(n+1, 2*l+1, result);

    } else {

            var sm = get_coeffs(n, l-1);
            var sp = get_coeffs(n, l+1);
            var s0 = s[n, l];


        // We have s0, check if we found sm and sp at this level
        if !isNone(sm) && !isNone(sp) {
            var r = rp*sm + r0*s0 + rm*sp;
            result.s[n, l] = r * 2.0**n;
        } else {
            recur_down(n, l);

                diff(n+1, 2*l  , result);
                diff(n+1, 2*l+1, result);
}   }   }
```


contents adapted from James Dinan, the Ohio State University

Parallel Differentiation Code

```
def diff (n = 0, l = 0, result) {
    if !s.has_coeffs(n, l) {
        // Run down tree until we hit scaling function coefficients
        cobegin {
            diff(n+1, 2*l  , result);
            diff(n+1, 2*l+1, result);
        }
    } else {
        cobegin {
            var sm = get_coeffs(n, l-1);
            var sp = get_coeffs(n, l+1);
            var s0 = s[n, l];
        }

        // We have s0, check if we found sm and sp at this level
        if !isNone(sm) && !isNone(sp) {
            var r = rp*sm + r0*s0 + rm*sp;
            result.s[n, l] = r * 2.0**n;
        } else {
            recur_down(n, l);
            cobegin {
                diff(n+1, 2*l  , result);
                diff(n+1, 2*l+1, result);
}   }   }   }
```

Perform recursive calls in parallel

Get neighboring coefficients in parallel

Perform recursive calls in parallel

contents adapted from James Dinan, the Ohio State University

Outline

- ✓ Chapel Overview
- ➢ Chapel computations
  - **your first Chapel program:** STREAM Triad
  - ***the stencil ramp*****:** from jacobi to finite element methods
  - **graph-based computation in Chapel:** SSCA #2
  - **task-parallelism:** producer-consumer to MADNESS
  - **GPU computing in Chapel:** STREAM revisited and CP
- Status, Summary, and Future Work

Current Parallel Models and GPUs

- **MPI, Co-Array Fortran, Unified Parallel C**
  - SPMD model is too coarse-grain and heavy-weight for GPUs
- **Java, C#, pthreads**
  - Thread fork/join models not a good match for SIMD nature of GPUs
- **CUDA (C or API), OpenCL**
  - Low-level models impact productivity
  - Better suited as a compiler/library target
- **directive-based approaches (OpenMP, PGI, CAPS)**
  - Probably the most sensible evolutionary approach
  - But potentially a blunt tool -- lots of reliance on the compiler
  - Can't we do better?

GPU Programming Wishlist

- *general parallelism*
  - task parallelism to fire kernels off to the accelerator
  - data parallelism to express SIMD/SIMT computations
  - nested parallelism to handle inter-/intra-node parallelism (many kinds)
- *locality control:* the ability to say where things are run/stored:
  - one node vs. another
  - CPU vs. GPU
  - individual thread blocks
  - types of memory within the GPU
- *multiresolution design:* the ability to…

  …use high-level abstractions when convenient/appropriate

  …get as close to the hardware as necessary *within the language*

  …interoperate with other programming models

Conventional solutions will likely result in a notational mash-up

Chapel's concepts/themes already support all these goals

Traditional STREAM (single-node version)

By default, domains and arrays are implemented using the current locale

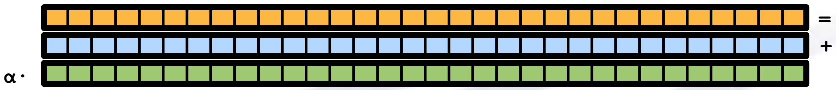
```
config const m = 1000;

const alpha = 3.0;

const ProbSpace: domain(1) = [1..m];

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

forall (a,b,c) in (A,B,C) do
  a = b + alpha * c;
```

Default problem size; user can override on executable's command-line

Domain representing the problem space

Three vectors of floating point values

Parallel loop specifying the computation



CPU+GPU STREAM

```
config const m = 1000, tpb = 256;
const alpha = 3.0;

const gpuDist = new GPUDist(rank=1, tpb);

const ProbSpace: domain(1) = [1..m];
const GPUProbSpace: domain(1) distributed gpuDist = ProbSpace;

var hostA, hostB, hostC: [ProbSpace] real;
var gpuA, gpuB, gpuC: [GPUProbSpace] real;

hostB = …;
hostC = …;

gpuB = hostB;
gpuC = hostC;

forall (a, b, c) in (gpuA, gpuB, gpuC) do
  a = b + alpha * c;

hostA = gpuA;
```

Create vectors on both host (CPU) and GPU

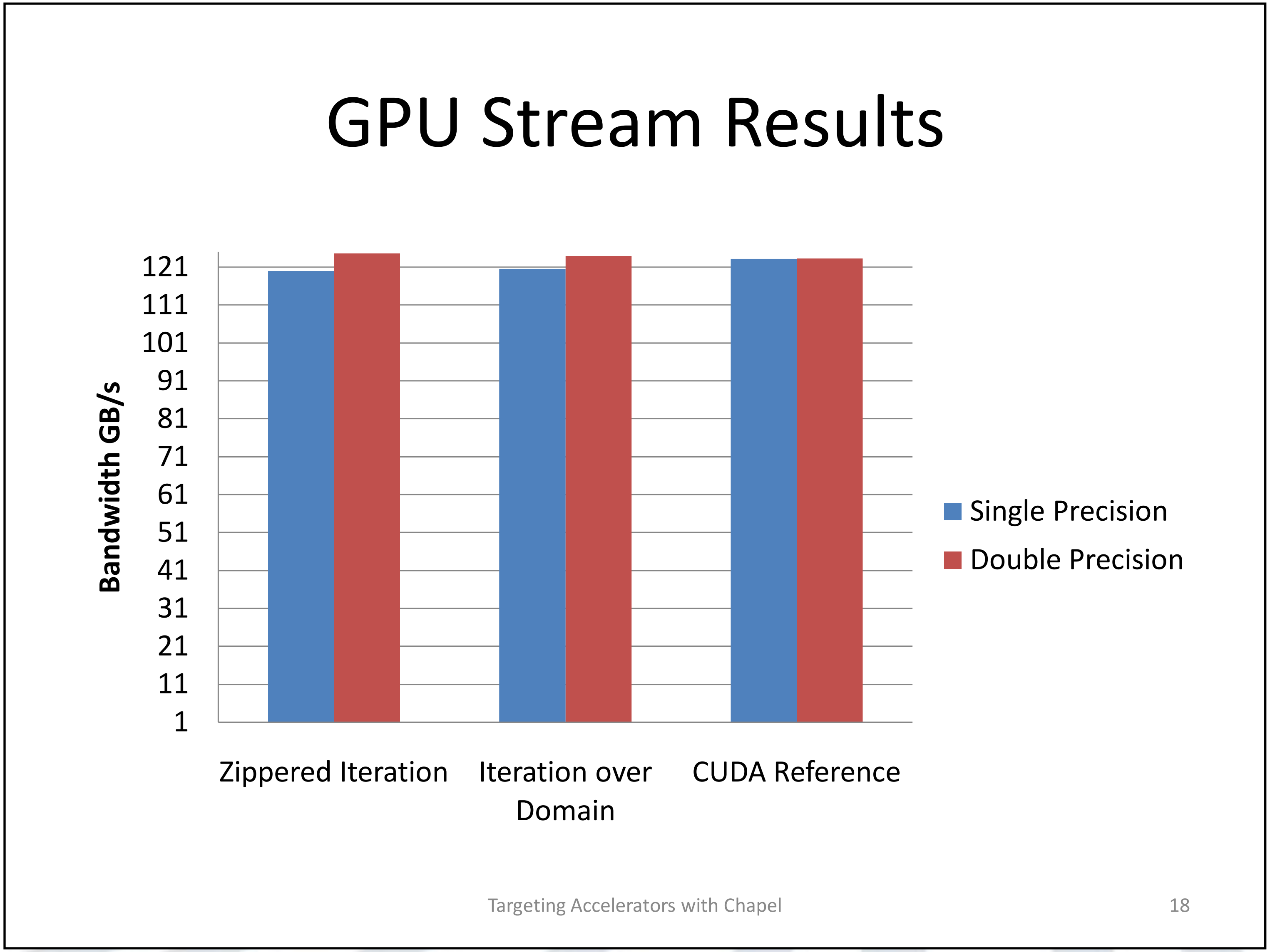
Perform vector initializations on the host

Assignments between host and GPU arrays implemented using CUDA's memcpy

Computation executed by GPU

Copy result back from GPU to host memory

Experimental results (NVIDIA GTX 280)



Case Study: STREAM (current practice)

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];
}
```

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;
}
```

Case Study: STREAM (current practice)

CUDA

```
#define N       2000000

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

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

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 );
  ...
```

```
#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;
}
```

Chapel (today)

```
config const m = 1000,
             tpb = 256;
const alpha = 3.0;

const gpuDist = new GPUDist(rank=1, tpb);

const ProbSpace: domain(1) = [1..m];
const GPUProbSpace: domain(1) distributed gpuDist
                  = ProbSpace;

var hostA, hostB, hostC: [ProbSpace] real;
var gpuA, gpuB, gpuC: [GPUProbSpace] real;

hostB = …;
hostC = …;

gpuB = hostB;
gpuC = hostC;

forall (a, b, c) in (gpuA, gpuB, gpuC) do
  a = b + alpha * c;

hostA = gpuA;
```

Chapel (ultimate goal)

```
config const m = 1000,
             tpl = here.numCores,
             tpb = 256;

const alpha = 3.0;

const ProbDist = new BlockCPUGPU(rank=1, tpl, tpb);

const ProbSpace: domain(1) distributed ProbDist
               = [1..m];

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

B = …;
C = …;

forall (a,b,c) in (A,B,C) do
  a = b + alpha * c;
```

Chapel Parboil Benchmark Suite Study

Parboil Benchmark Suite: GPU-oriented benchmarks from Wen-Mei Hwu (UIUC) with CPU and GPU (CUDA) versions

- http://impact.crhc.illinois.edu/parboil.php

This study: Rewrite the suite in Chapel to compare performance and programmability relative to CUDA

One benchmark: Coulombic Potential (CP)

- computes the Coulombic potential over a discretized plane within a 3D space of randomly-placed charges
- adapted from the *cionize* benchmark in VMD.

Team: Albert Sidelnik, David Padua, Maria Garzarán

CP Excerpts: Declarations

```
const GPUMem = distributionValue(new GPUDist(rank=2,
                 tbSizeX=BLOCKSIZEX, tbSizeY=BLOCKSIZEY));
const space: domain(2, int(64)) distributed GPUMem
           = [0..#VOLSIZEY, 0..#VOLSIZEX];

const atomspace_host = [0..#MAXATOMS];
var atominfo_host: [atomspace_host] float4;
const atomspace_gpu: domain(1, int(64)) distributed GPUMem
                   = atomspace_host;
var atominfo_gpu: [atomspace_gpu] float4;

const energyspace_cpu = [0..#volmemsz];
var energy_host: [energyspace_cpu] real(32);
const energyspace_gpu: domain(1, int(64)) distributed GPUMem
                     = energyspace_cpu;
var energy_gpu: [energyspace_gpu] real(32);
```

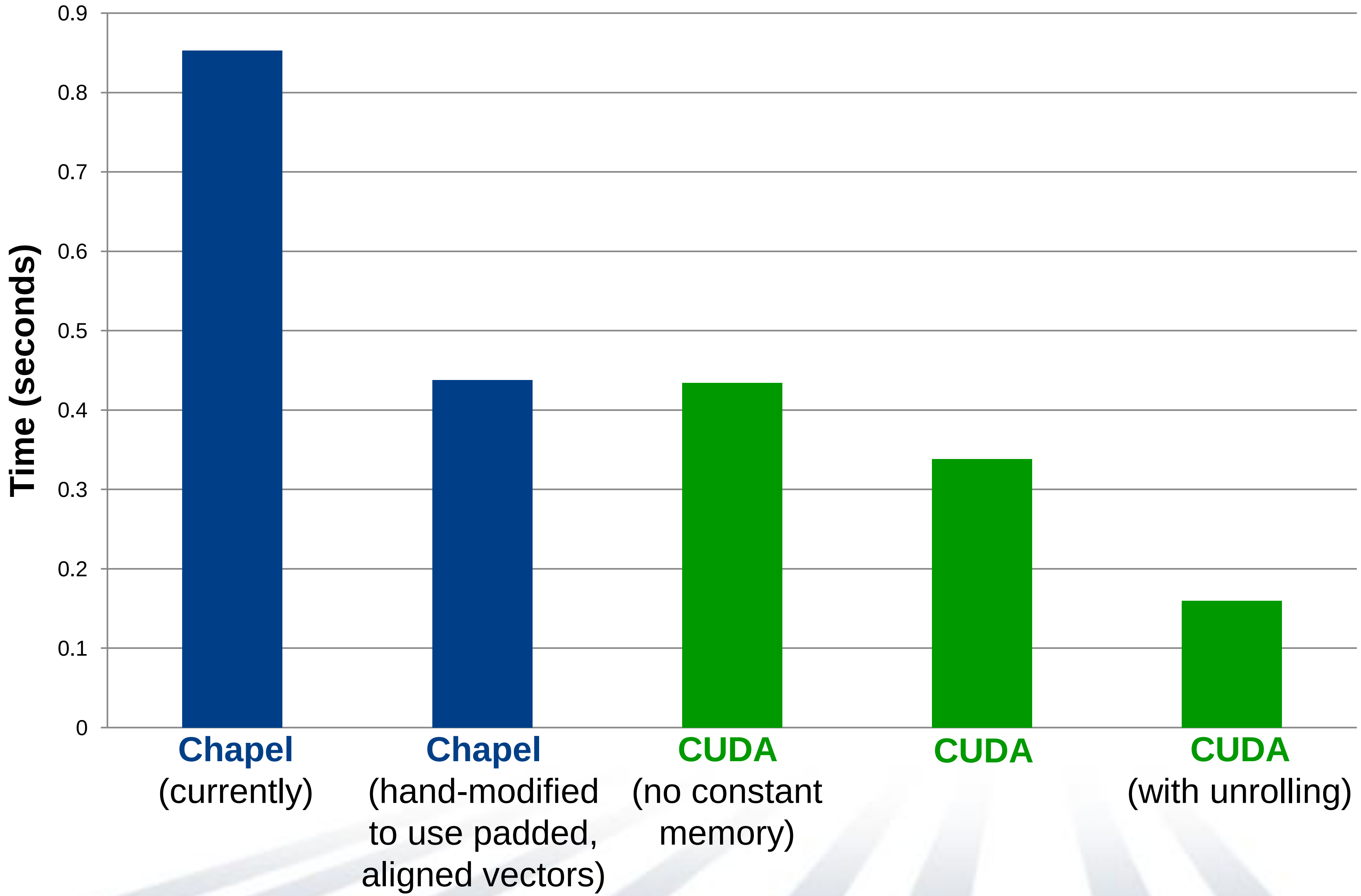

Original code courtesy of Albert Sidelnik, UIUC

CP Excerpts: Computation

```
atominfo_gpu = atominfo_host;
energy_gpu = energy_host;

forall (xindex, yindex) in space {
  const coorx = gridspacing * xindex,
        coory = gridspacing * yindex;
  var energyval: real(32);
  for atomid in 0..#runatoms {
    const dx = coorx - atominfo_gpu(atomid).x,
          dy = coory - atominfo_gpu(atomid).y;
    const r_1 = 1.0 : real(32)
                / sqrt(dx * dx + dy * dy + atominfo_gpu(atomid).z);
    energyval += atominfo_gpu(atomid).w * r_1;
  }
  energy_gpu(rowSizeX * yindex + xindex) += energyval;
}

energy_host = energy_gpu;
```


Original code courtesy of Albert Sidelnik, UIUC

Coulombic Potential: Execution Time



Results courtesy of Albert Sidelnik, UIUC

Chapel and GPUs: Next Steps

- **CP Benchmark:**
  - provide access to padded, aligned vector using external types
  - add support for using constant memory
    - explicitly via Chapel's *on-clauses*
    - automatically via compiler analysis
  - explore loop unrolling via Chapel's iterator functions and full unrolling
    - if infeasible look into adding language support or unrolling

- **GPU Programming in Chapel:**
  - continue studying additional benchmarks, Parboil and otherwise
  - create a distribution that spans CPU and GPU resources
    - to avoid duplicated declarations
    - to pipeline data between CPU and GPU to hide I/O latencies
  - combine CPU/GPU distributions with Block, Cyclic, … distributions
    - to target a cluster of CPU+GPU resources

Candidate CPU/GPU Distribution Concept

```
const CPUGPU = distributionValue(new CPUGPUDist(rank=2,
                 tbSizeX=BLOCKSIZEX, tbSizeY=BLOCKSIZEY));

const atomspace: domain(1, int(64)) distributed CPUGPU
                    =   [0..#MAXATOMS];;
var atominfo: [atomspace] float4;


// init on host

atominfo.setMode(gpu=true);

// compute on GPU;

atominfo.setMode(gpu=false);
```

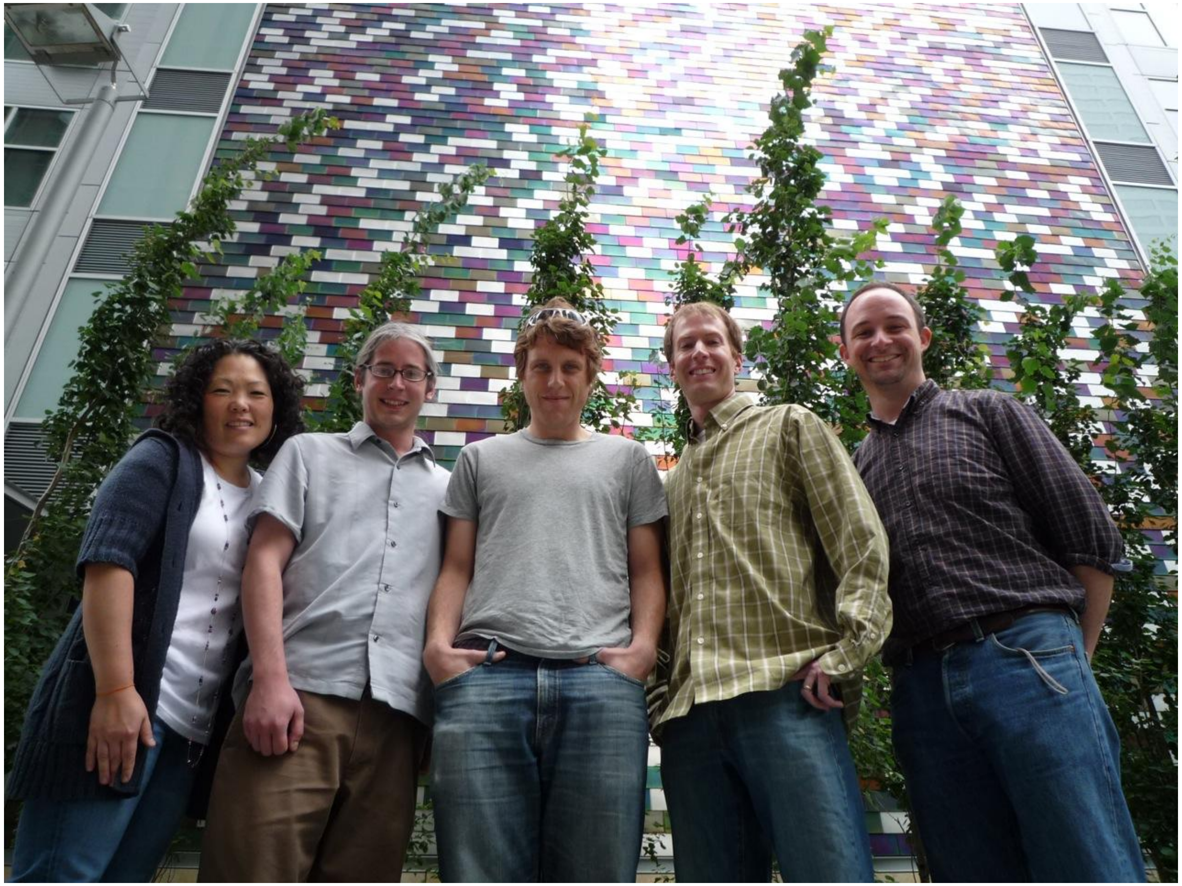
Outline

- ✓ Chapel Overview
- ➢ Chapel computations
  - ❑ **your first Chapel program:** STREAM Triad
  - ❑ ***the stencil ramp*****:** from jacobi to finite element methods
  - ❑ **graph-based computation in Chapel:** SSCA #2
  - ❑ **task-parallelism:** producer-consumer to MADNESS
  - ❑ **GPU computing in Chapel:** STREAM revisited and CP
- ❑ Status, Summary, and Future Work

Outline

- ✓ Chapel Overview
- ✓ Chapel computations
  - ❑ **your first Chapel program:** STREAM Triad
  - ❑ ***the stencil ramp*****:** from jacobi to finite element methods
  - ❑ **graph-based computation in Chapel:** SSCA #2
  - ❑ **task-parallelism:** producer-consumer to MADNESS
  - ❑ **GPU computing in Chapel:** STREAM revisited and CP
- ➢ Status, Summary, and Future Work

The Chapel Team



Sung-Eun Choi, David Iten, Lee Prokowich,
Steve Deitz, Brad Chamberlain

- Interns
  - Jacob Nelson (`09 – UW)
  - Albert Sidelnik (`09 – UIUC)
  - Andy Stone (`08 – Colorado St)
  - James Dinan (`07 – Ohio State)
  - Robert Bocchino (`06 – UIUC)
  - Mackale Joyner (`05 – Rice)
- Alumni
  - David Callahan
  - Roxana Diaconescu
  - Samuel Figueroa
  - Shannon Hoffswell
  - Mary Beth Hribar
  - Mark James
  - John Plevyak
  - Wayne Wong
  - Hans Zima

Chapel Release

- **Current release:** version 1.02 (November 12, 2009)
- Supported environments: UNIX/Linux, Mac OS X, Cygwin
- How to get started:
  1. Download from: http://sourceforge.net/projects/chapel
  2. Unpack tar.gz file
  3. See top-level README
     - for quick-start instructions
     - for pointers to next steps with the release
- Your feedback desired!
- Remember: a work-in-progress

  ⇒ it's likely that you will find problems with the implementation

  ⇒ this is still a good time to influence the language's design

Chapel Implementation Status (v1.02)

- **Base language:** stable (gaps and bugs remain)
- **Task parallel:**
  - stable multi-threaded implementation of tasks, sync variables
  - atomic sections are an area of ongoing research with U. Notre Dame
- **Data parallel:**
  - stable multi-threaded data parallelism for dense domains/arrays
  - other domain types have a single-threaded reference implementation
- **Locality:**
  - stable locale types and arrays
  - stable task parallelism across multiple locales
  - initial support for some distributions: Block, Cyclic, Block-Cyclic
- **Performance:**
  - has received much attention in designing the language
  - yet very little implementation effort to date

Chapel Collaborations

Notre Dame/ORNL (Peter Kogge, Srinivas Sridharan, Jeff Vetter): Asynchronous STM over distributed memory

UIUC (David Padua, Albert Sidelnik, Maria Garzarán): Chapel for hybrid CPU-GPU computing

OSU (Gagan Agrawal, Bin Ren): Data-intensive computing using Chapel's user-defined reductions

PNNL/CASS-MT (John Feo, Daniel Chavarria): Chapel extensions for hybrid computation; performance tuning for the Cray XMT; ARMCI port

Universitat Politècnica de Catalunya (Alex Duran): Chapel over Nanos

Universidad de Màlaga (Rafael Asenjo, Angeles Navarro, et al.): Parallel I/O, sparse distributions, …

ORNL (David Bernholdt *et al.*; Steve Poole *et al.*): Chapel code studies – Fock matrix computations, MADNESS, Sweep3D, coupled models, …

Berkeley (Dan Bonachea *et al.*): Chapel over GASNet; collectives

(Your name here?)

Collaboration Opportunities

- memory management policies/mechanisms
- exceptions
- dynamic load balancing: task throttling and stealing
- parallel I/O and checkpointing
- language interoperability
- application studies and performance optimizations
- index/subdomain semantics and optimizations
- targeting different back-end compilers/runtimes (LLVM, MS CLR, …)
- dynamic compilation
- library support
- tools
  - correctness debugging, visualizations, algorithm animations
  - performance debugging
  - IDE support
  - Chapel interpreter
- (your ideas here…)

Chapel: For More Information

chapel_info@cray.com

http://chapel.cray.com

http://sourceforge.net/projects/chapel/

SC08 tutorial slides

Parallel Programmability and the Chapel Language; Chamberlain, Callahan, Zima; International Journal of High Performance Computing Applications, August 2007, 21(3):291-312.