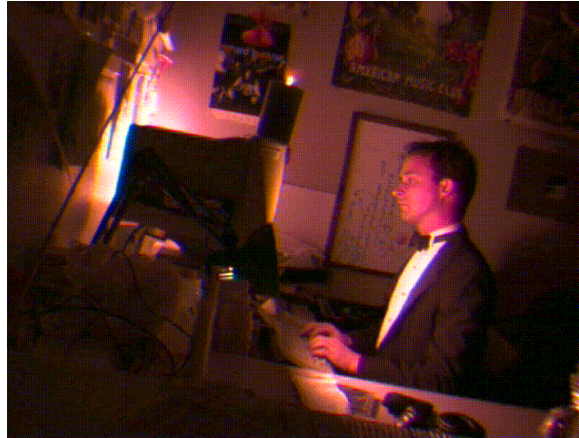


## Today's Plan

---

### Chapel

- The Cascade High Productivity Language



Thanks to Brad Chamberlain for most of the contents of these slides

## Chapel Generalizes ZPL

---

### Chapel builds off of ZPL's successes

- Extends first class index sets
    - Unifies local and distributed arrays
    - Generalizes the idea to support richer data types (sets, graphs, maps)
  - Relaxes constraints on array alignment
    - Preserves compiler's ability to reason about aligned arrays
  - Supports user-defined distributions
  - Supports more general parallelism
- } No syntactic performance model

## Chapel Goals

---

### Goals

- Support general parallel programming

## General Parallel Programming

---

### Should be able to express any parallel computation

- Should never hit a limitation requiring the user to return to MPI

### Supports both data-parallelism and task-parallelism

- As well as the ability to compose these naturally

### Provides multiple levels of software parallelism

- Module-level, function-level, loop-level, statement-level, ...

### Supports general levels of hardware parallelism

- Inter-machine, inter-node, inter-core, vectorization, multithreading, ...

## Chapel Goals

---

### Goals

- Support **general parallel programming**
- Provide global view abstractions
- Provide support for locality
- Reduce gap between mainstream languages and parallel languages

## History– High Productivity Computing Systems Program

---

### DARPA HPCS Program (2002)

- Realization that programmer productivity is critical
  - Sought to increase programmer productivity by 10× by 2010
    - Productivity = Performance +  
Programmability +  
Portability +  
Robustness
- } **Includes both hardware and language design**

### Phased competition

- Phase 2 (2003-2006)
  - Cray (Chapel)
  - IBM (X10)
  - Sun (Fortress)
- Phase 3 (2006-2010)
  - Cray, IBM

## Chapel's Productivity Goals

### Vastly improve **programmability** over current languages/models

- Writing parallel codes
- Reading, modifying, porting, tuning, maintaining, evolving them

### Support **performance** at least as good as MPI

- Competitive with MPI on generic clusters
- Better than MPI on more capable architectures

### Improve **portability** compared to current languages/models

- As ubiquitous as MPI, but with fewer architectural assumptions
- More portable than OpenMP, Unified Parallel C, Co-Array Fortran, ...

### Improve **code robustness** via improved semantics and concepts

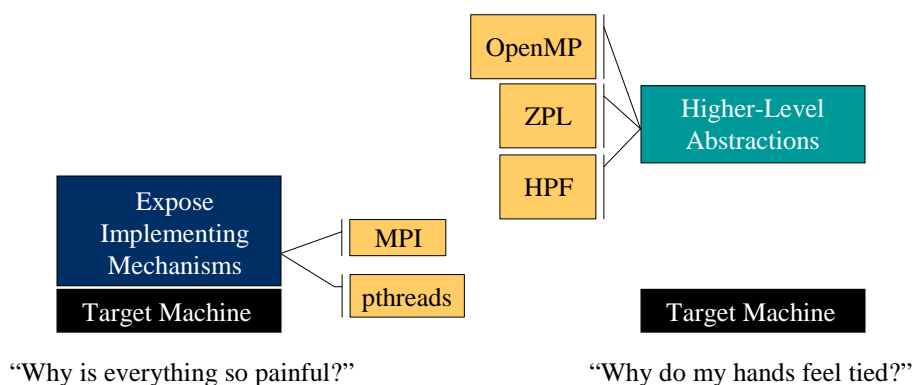
- Eliminate common error cases altogether
- Better abstractions to help avoid other errors

CS380P Lecture 24

Chapel

7

## Previous Languages– Two Extremes



CS380P Lecture 24

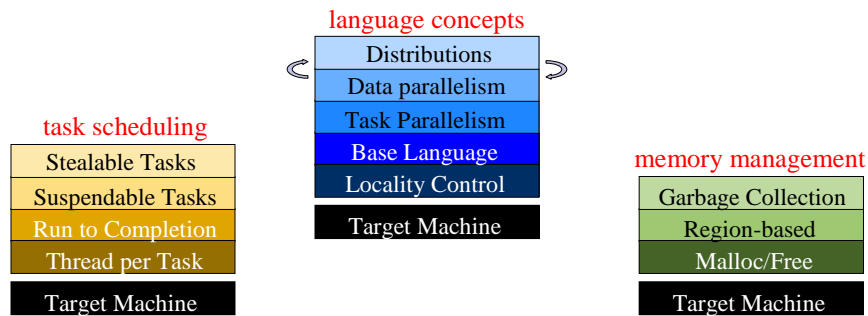
Chapel

8

## Multi-Resolution Language Design

### The Chapel approach

- Allow the language to be used at multiple levels of abstraction
  - Provide high-level features and automation for convenience
  - Provide the ability to drop down to lower, more manual levels
  - Use appropriate separation of concerns to keep these layers clean

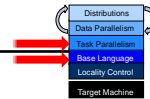


CS380P Lecture 24

Chapel

9

## Chapel In a Nutshell



### Base language

- **Standard stuff:** types, expressions, statements, functions, modules
- **Object-orientation:** value- and reference-based classes (optional)
- **Iterators:** functions that generate a stream of return values
- **Latent types:** ability to omit types of variables, arguments, etc.

### Task parallelism

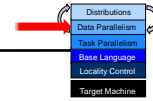
- **Task creation:** structured and unstructured task creation
- **Synchronization:** through sync variables, transactional memory

CS380P Lecture 24

Chapel

10

## Chapel In a Nutshell (cont)



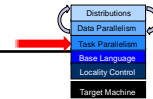
### Data parallelism

- **Data structures:** global view of dense, sparse, associative arrays
- **Operators:** forall loops, promotion of scalar operators/functions, ...

### Locality

- **Locales:** language concept for reasoning about machine locality
- **On clauses:** ability to place tasks, variables on specific locales
- **Distributions:** recipes for implementing distributed arrays on locales

## Task Parallelism– Creating Tasks



### Spawn a task: **begin**

```
begin DoThisTask();
WhileContinuing();
TheOriginalThread();
```

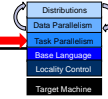
### Wait for tasks to complete: **sync**

- Wait for all tasks created within a dynamic scope

```
sync {
  begin treeSearch(root);
}

def treeSearch(node) {
  if node == nil then return;
  begin treeSearch(node.right);
  begin treeSearch(node.left);
}
```

## Task Parallelism: Task Coordination



### Full/Empty variables

- Maintain full/empty state along with a value

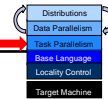
```
var result$: sync real; // result is initially empty
sync {
  begin ... = result$; // block until full, leave empty
  begin result$ = ...; // block until empty, leave full
}
result$.readXX(); // read value, leave state
unchanged; // other variations also supported
```

CS380P Lecture 24

Chapel

13

## Task Parallelism: Task Coordination



### single-assignment variables — Write once only

```
var result$: single real = begin f(); // initially empty
... // do some other things
total += result$; // block until f() has completed
```

### Atomic sections

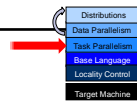
```
atomic {
  newnode.next = insertpt;
  newnode.prev = insertpt.prev;
  insertpt.prev.next = newnode;
  insertpt.prev = newnode;
}
```

CS380P Lecture 24

Chapel

14

## Task Parallelism: Structured Tasks



### Cobegin – Create one task per statement

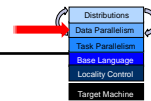
```
computePivot(lo, hi, data);
cobegin {
    Quicksort(lo, pivot, data);
    Quicksort(pivot, hi, data);
} // implicit join here
```

```
cobegin {
    computeTaskA(...);
    computeTaskB(...);
    computeTaskC(...);
} // implicit join
```

### Coforall – Create one task per iteration

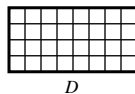
```
coforall e in Edges {
    exploreEdge(e);
} // implicit join here
```

## Domains

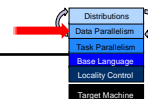


*domain*: a first-class index set

```
var m = 4, n = 8;
var D: domain(2) = [1..m, 1..n];
```

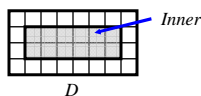


## Domains

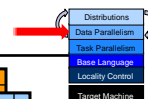


*domain*: a first-class index set

```
var m = 4, n = 8;
var D: domain(2) = [1..m, 1..n];
var Inner: subdomain(D) = [2..m-1, 2..n-1];
```

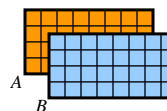


## Domains: Some Uses



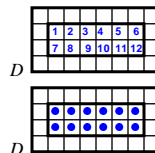
### Declaring arrays:

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



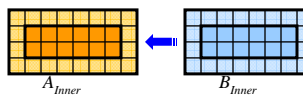
### Iteration (sequential or parallel):

```
for ij in Inner { ... }
or: forall ij in Inner { ... }
```



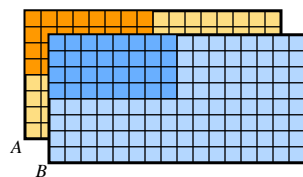
### Array Slicing:

```
A[Inner] = B[Inner];
```



### Array reallocation:

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



## Global View (Chapel) vs. Fragmented View

### Example

- 3 point stencil of a vector


#### Global View

```

def main() {
  var n: int = 1000;
  var a, b: [1..n] real;

  forall i in 2..n-1 {
    b(i) = (a(i-1) + a(i+1))/2;
  }
}

```



**B = (A@east + A@west)/2;**

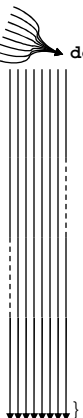
#### Fragmented View

```

def main() {
  var n: int = 1000;
  var locN: int = n/numProcs;
  var a, b: [0..locN+1] real;

  if (iHaveRightNeighbor) {
    send(right, a(locN));
    rcv(right, a(locN+1));
  }
  if (iHaveLeftNeighbor) {
    send(left, a(1));
    rcv(left, a(0));
  }
  forall i in 1..locN {
    b(i) = (a(i-1) + a(i+1))/2;
  }
}

```



## Global View (ZPL) vs. Fragmented View

### Example


- 3 point stencil of a vector

#### Global View

```

begin
  region R = [1..n];
  var n: int;
  var A, B: [R] real;
  procedure main()
  [R] begin
    B := (A@west+A@east)/2;
  end;
end;

```



**B = (A@east + A@west)/2;**

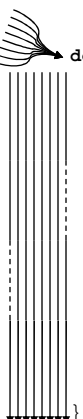
#### Fragmented View

```

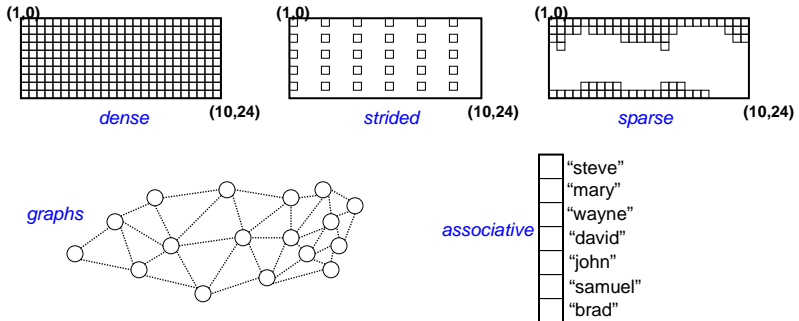
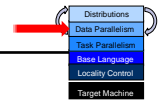
def main() {
  var n: int = 1000;
  var locN: int = n/numProcs;
  var a, b: [0..locN+1] real;

  if (iHaveRightNeighbor) {
    send(right, a(locN));
    rcv(right, a(locN+1));
  }
  if (iHaveLeftNeighbor) {
    send(left, a(1));
    rcv(left, a(0));
  }
  forall i in 1..locN {
    b(i) = (a(i-1) + a(i+1))/2;
  }
}

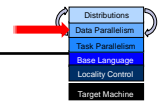
```



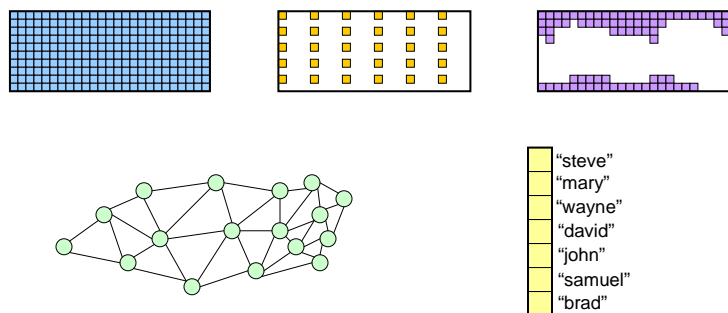
## Data Parallelism: Other Domains



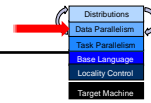
## Data Parallelism: Domain Uses



Domains are used to declare arrays...

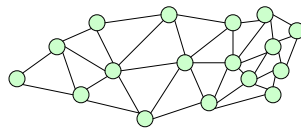
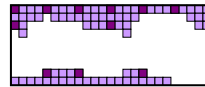
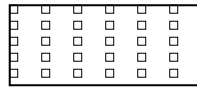
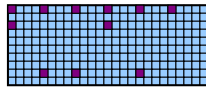


## Data Parallelism: Domain Uses



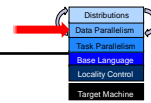
...to iterate over index sets...

```
forall ij in StrDom {
  DnsArr(ij) += SpsArr(ij);
}
```



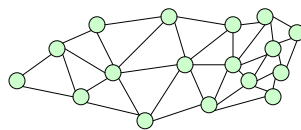
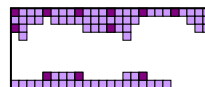
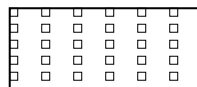
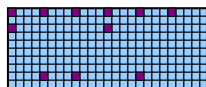
- “steve”
- “mary”
- “wayne”
- “david”
- “john”
- “samuel”
- “brad”

## Data Parallelism: Domain Uses



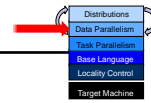
...to slice arrays...

```
DnsArr[StrDom] += SpsArr[StrDom];
```



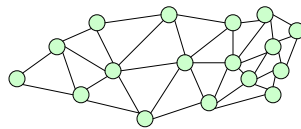
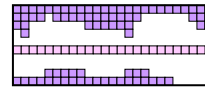
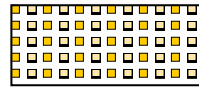
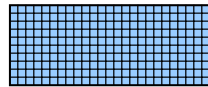
- “steve”
- “mary”
- “wayne”
- “david”
- “john”
- “samuel”
- “brad”

## Data Parallelism: Domain Uses



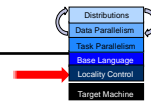
...and to reallocate arrays

```
StrDom = DnsDom by (2,2);
SpsDom += genEquator();
```



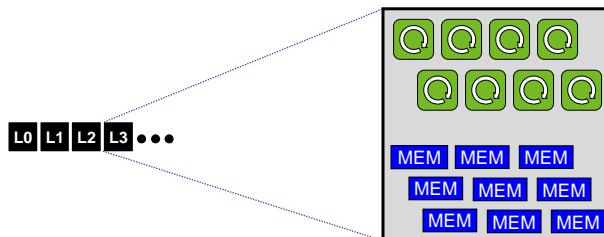
- “steve”
- “mary”
- “wayne”
- “david”
- “john”
- “samuel”
- “brad”

## Locality: Locales

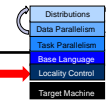


*locale*: architectural unit of locality

- Represents both a processor and local memory
- Threads within a locale have ~uniform access to local memory
- Memory within other locales is accessible, but at a price
- *e.g.*, a multicore processor or SMP node could be a locale



## Locality: Locales



User specifies # locales on executable command-line

```
prompt> myChapelProg -nl=8
```

Chapel programs have built-in locale variables:

```
config const numLocales: int;
const LocaleSpace = [0..numLocales-1],
      Locales: [LocaleSpace] locale;
```



Programmers can create their own locale views:

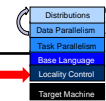
```
var CompGrid = Locales.reshape([1..GridRows,
                               1..GridCols]);
```



```
var TaskALocs = Locales[..numTaskALocs];
var TaskBLocs = Locales[numTaskALocs+1..]
```



## Locality: Task Placement



*on* clauses: indicate where tasks should execute

Either in a data-driven manner...

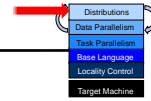
```
computePivot(lo, hi, data);
cobegin {
  on data(lo) do Quicksort(lo, pivot, data);
  on data(pivot) do Quicksort(pivot, hi, data);
}
```

...or by naming locales explicitly

```
cobegin {
  on TaskALocs do computeTaskA(...);
  on TaskBLocs do computeTaskB(...);
  on Locales(0) do computeTaskC(...);
}
```

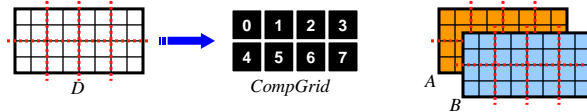


## Locality: Domain Distribution



### Domains may be distributed across locales

```
var D: domain(2) distributed Block on CompGrid = ...;
```



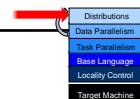
### A distribution implies...

- ...ownership of the domain's indices (and its arrays' elements)
- ...the default work ownership for operations on the domains/arrays

### Chapel provides...

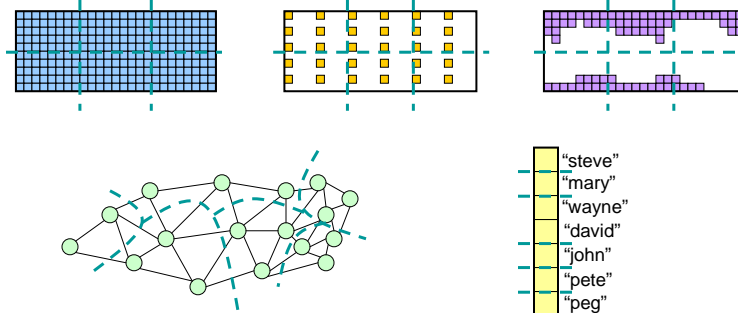
- ...a standard library of distributions (Block, Recursive Bisection, ...)
- ...the means for advanced users to author their own distributions

## Locality: Domain Distributions

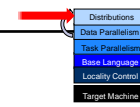


### A distribution must implement

- The mapping from indices to locales
- The per-locale representation of domain indices and array elements
- The compiler's target interface for lowering global-view operations



## Locality: Distributions Overview



### Distributions define a mapping

- From the user's global view operations to the fragmented implementation for a distributed memory machine

### Users can implement custom distributions

- Written using task parallel features, on clauses, domains/arrays
- Must implement standard interface:
  - **Allocation/reallocation** of domain indices and array elements
  - **Mapping functions** (e.g., index-to-locale, index-to-value)
  - **Iterators:** parallel/serial  $\times$  global/local
  - Optionally, communication idioms

### Chapel's standard library of distributions

- Written using the same mechanism as user-defined distributions
- Tuned for different platforms to maximize performance

## Distributions vs. Domains

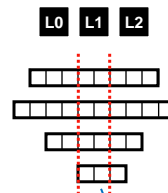
**Q1:** Why distinguish between distributions and domains?

**Q2:** Why do distributions map an index *space* rather than a fixed index set?

**A:** To permit several domains to share a single distribution

- Amortizes the overheads of storing a distribution
- Supports trivial domain/array alignment and compiler optimizations

```
const D      : ...distributed B1 = [1..8],
  outerD: ...distributed B1 = [0..9],
  innerD: subdomain(D)      = [2..7],
  slideD: subdomain(D)      = [4..6];
```



Shared distributions support trivial alignment of these domains

## Distributions vs. Domains

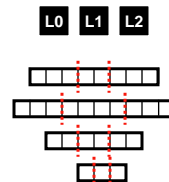
**Q1:** Why distinguish between distributions and domains?

**Q2:** Why do distributions map an index *space* rather than a fixed index set?

**A:** To permit several domains to share a single distribution

- Amortizes the overheads of storing a distribution
- Supports trivial domain/array alignment and compiler optimizations

```
const D      : ...distributed B1 = [1..8],
outerD: ...distributed B1 = [0..9],
innerD: subdomain(D)      = [2..7],
slideD: subdomain(D)      = [4..6];
```



When each domain is given its own distribution, the compiler cannot reason about alignment of indices

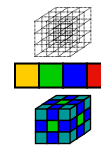
CS380P Lecture 24

33

## Recall the NAS MG *rprj3* Stencil in ZPL

**The Issue:** 27 directions

```
procedure rprj3(var S,R: [, ,] double;
               d: array [] of direction);
begin
  S := 0.5 * R
    + 0.25 * (R@d[ 1, 0, 0] + R@d[ 0, 1, 0] + R@d[ 0, 0, 1] +
              R@d[-1, 0, 0] + R@d[ 0,-1, 0] + R@d[ 0, 0,-1])
    + 0.125 * (R@d[ 1, 1, 0] + R@d[ 1, 0, 1] + R@d[ 0, 1, 1] +
              R@d[ 1,-1, 0] + R@d[ 1, 0,-1] + R@d[ 0, 1,-1] +
              R@d[-1, 1, 0] + R@d[-1, 0, 1] + R@d[ 0,-1, 1] +
              R@d[-1,-1, 0] + R@d[-1, 0,-1] + R@d[ 0,-1,-1])
    + 0.0625 * (R@d[ 1, 1, 1] + R@d[ 1, 1,-1] +
                R@d[ 1,-1, 1] + R@d[ 1,-1,-1] +
                R@d[-1, 1, 1] + R@d[-1, 1,-1] +
                R@d[-1,-1, 1] + R@d[-1,-1,-1]);
end;
```



CS380P Lecture 24

Chapel

34

### NAS MG rprj3 stencil in Fortran+MPI

```

subroutine comm1(s1,s2,s3,s4)
use mpi
implicit none
include "mpif.h"
include "mpi.h"
integer :: s1, s2, s3, s4
double precision w(11,11,11)
integer :: i, j, k
if (.not. mod(s1,3) .eq. 0) then
  call mpi_send(s1, 1, MPI_INTEGER, 0, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
else
  call mpi_send(s1, 1, MPI_INTEGER, 1, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
endif
if (.not. mod(s2,3) .eq. 0) then
  call mpi_send(s2, 1, MPI_INTEGER, 0, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
else
  call mpi_send(s2, 1, MPI_INTEGER, 1, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
endif
if (.not. mod(s3,3) .eq. 0) then
  call mpi_send(s3, 1, MPI_INTEGER, 0, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
else
  call mpi_send(s3, 1, MPI_INTEGER, 1, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
endif
if (.not. mod(s4,3) .eq. 0) then
  call mpi_send(s4, 1, MPI_INTEGER, 0, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
else
  call mpi_send(s4, 1, MPI_INTEGER, 1, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
endif
end subroutine comm1

subroutine givw1(w, dir, n, m, k, w3d)
use mpi
implicit none
include "mpif.h"
include "mpi.h"
integer :: dir, n, m, k, w3d
double precision w(11,11,11)
integer :: i, j, k, i1, i2, i3, i4, i5, i6, i7, i8, i9, i10, i11, j1, j2, j3, j4, j5, j6, j7, j8, j9, j10, j11, k1, k2, k3, k4, k5, k6, k7, k8, k9, k10, k11
if (dir .eq. 1) then
  do i=1,11
    do j=1,11
      do k=1,11
        w(i,j,k) = w(i,j,k) + w(i-1,j,k)
        w(i,j,k) = w(i,j,k) + w(i+1,j,k)
        w(i,j,k) = w(i,j,k) + w(i,j-1,k)
        w(i,j,k) = w(i,j,k) + w(i,j+1,k)
        w(i,j,k) = w(i,j,k) + w(i,j,k-1)
        w(i,j,k) = w(i,j,k) + w(i,j,k+1)
      end do
    end do
  end do
else if (dir .eq. 2) then
  do i=1,11
    do j=1,11
      do k=1,11
        w(i,j,k) = w(i,j,k) + w(i,j,k-1)
        w(i,j,k) = w(i,j,k) + w(i,j,k+1)
      end do
    end do
  end do
else if (dir .eq. 3) then
  do i=1,11
    do j=1,11
      do k=1,11
        w(i,j,k) = w(i,j,k) + w(i,j,k-1)
        w(i,j,k) = w(i,j,k) + w(i,j,k+1)
      end do
    end do
  end do
else if (dir .eq. 4) then
  do i=1,11
    do j=1,11
      do k=1,11
        w(i,j,k) = w(i,j,k) + w(i,j,k-1)
        w(i,j,k) = w(i,j,k) + w(i,j,k+1)
      end do
    end do
  end do
else if (dir .eq. 5) then
  do i=1,11
    do j=1,11
      do k=1,11
        w(i,j,k) = w(i,j,k) + w(i,j,k-1)
        w(i,j,k) = w(i,j,k) + w(i,j,k+1)
      end do
    end do
  end do
else if (dir .eq. 6) then
  do i=1,11
    do j=1,11
      do k=1,11
        w(i,j,k) = w(i,j,k) + w(i,j,k-1)
        w(i,j,k) = w(i,j,k) + w(i,j,k+1)
      end do
    end do
  end do
else if (dir .eq. 7) then
  do i=1,11
    do j=1,11
      do k=1,11
        w(i,j,k) = w(i,j,k) + w(i,j,k-1)
        w(i,j,k) = w(i,j,k) + w(i,j,k+1)
      end do
    end do
  end do
else if (dir .eq. 8) then
  do i=1,11
    do j=1,11
      do k=1,11
        w(i,j,k) = w(i,j,k) + w(i,j,k-1)
        w(i,j,k) = w(i,j,k) + w(i,j,k+1)
      end do
    end do
  end do
else if (dir .eq. 9) then
  do i=1,11
    do j=1,11
      do k=1,11
        w(i,j,k) = w(i,j,k) + w(i,j,k-1)
        w(i,j,k) = w(i,j,k) + w(i,j,k+1)
      end do
    end do
  end do
else if (dir .eq. 10) then
  do i=1,11
    do j=1,11
      do k=1,11
        w(i,j,k) = w(i,j,k) + w(i,j,k-1)
        w(i,j,k) = w(i,j,k) + w(i,j,k+1)
      end do
    end do
  end do
else if (dir .eq. 11) then
  do i=1,11
    do j=1,11
      do k=1,11
        w(i,j,k) = w(i,j,k) + w(i,j,k-1)
        w(i,j,k) = w(i,j,k) + w(i,j,k+1)
      end do
    end do
  end do
endif
end subroutine givw1

```

### NAS MG rprj3 stencil in Chapel

#### Chapel solution

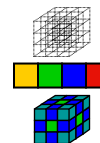
- Exploits first class domains

```

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));
  }
}

```



## Summary

---

### Generality

- Chapel extends the notion of data parallelism beyond dense arrays
  - (Some features not yet implemented, but the concepts fit within the design)
- Chapel supports task parallelism as well as data parallelism

### Philosophical difference from ZPL

- Gives up the notion of a strict performance model

## Next Time

---

### Reading

- Chapter 10: pp. 290 – 304

### Assignment 6

- Due Saturday April 1<sup>st</sup> 11:59pm