Chapel Generalizes ZPL

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

No syntactic performance model

Supports user-defined distributions

Supports more general parallelism

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 (cont)

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

CS380P Lecture 19 Chapel 5

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

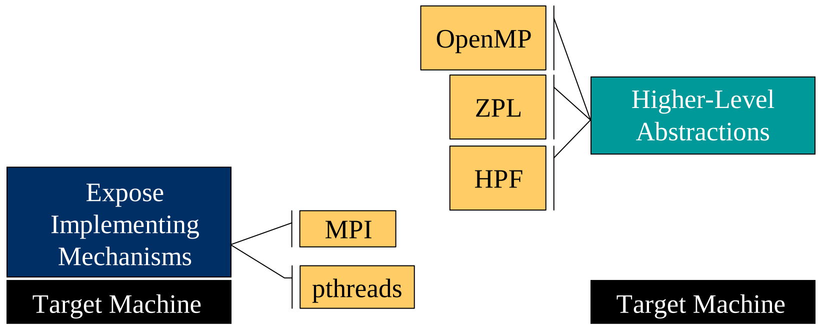
- As ubiquitous as MPI, but with fewer architectural assumptions
- More portable than OpenMP, Unified Parallel C, Co-Array Fortran, …

Improve code robustness via improved semantics and concepts

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

CS380P Lecture 19 Chapel 6

Previous Languages– Two Extremes



"Why is everything so painful?"

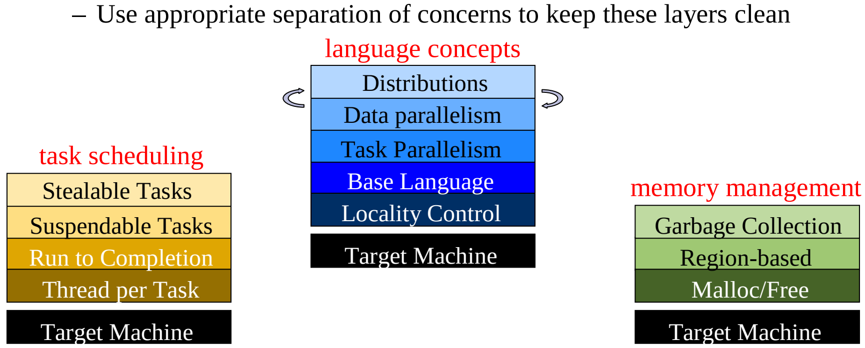
"Why do my hands feel tied?"

CS380P Lecture 19 Chapel 7

Multi-Resolution Language Design

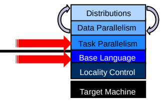
The Chapel approach

– 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 19 Chapel 8

Chapel In a Nutshell



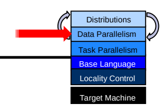
Base language

- **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

Chapel In a Nutshell (cont)



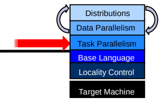
Data parallelism

- **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`

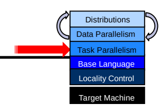
```
begin DoThisTask();
WhileContinuing();
TheOriginalThread();
```

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

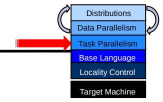


Full/Empty variables

– 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
```

Task Parallelism: Task Coordination



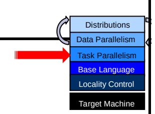
single-assignment variables — Write once only

```
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;
}
```


Task Parallelism: Structured Tasks



Cobegin – Create one task per statement

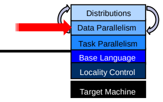
```
computePivot(lo, hi, data);
cobegin {
  Quicksort(lo, pivot, data);
  Quicksort(pivot, hi, data);
} // implicit join here
```

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

Coforall – Create one task per iteration

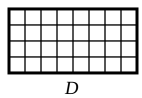
```
coforall e in Edges {
  exploreEdge(e);
} // implicit join here
```

Domains

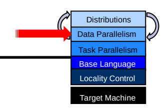


domain: a first-class index set

```
var m = 4, n = 8;

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

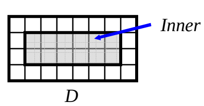


Domains

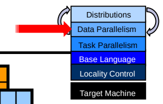


domain: a first-class index set

```
var m = 4, n = 8;

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

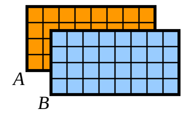


Domains: Some Uses



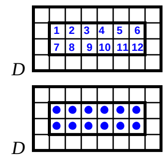
Declaring arrays:

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



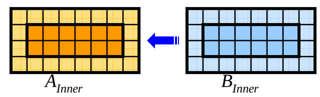
Iteration (sequential or parallel):

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



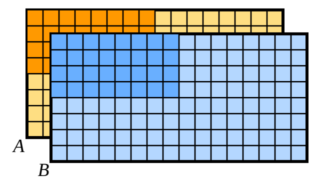
Array Slicing:

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



Array reallocation:

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



Global View (Chapel) vs. Fragmented View

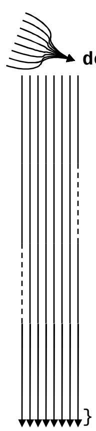
Example

– 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));
    recv(right, a(locN+1));
  }
  if (iHaveLeftNeighbor) {
    send(left, a(1));
    recv(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;
    A, B : [R] real;
procedure main()
[R] begin
    B := (A@west+A@east)/2;
    end;
end;
```

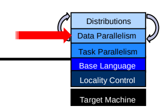
B = (A@east + A@west)/2;

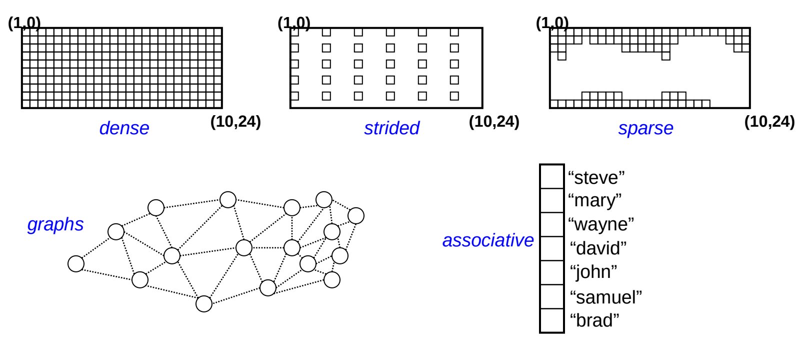
Fragmented View

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

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

CS380P Lecture 19 Chapel 19

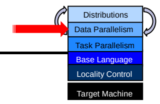
Data Parallelism: Other Domains



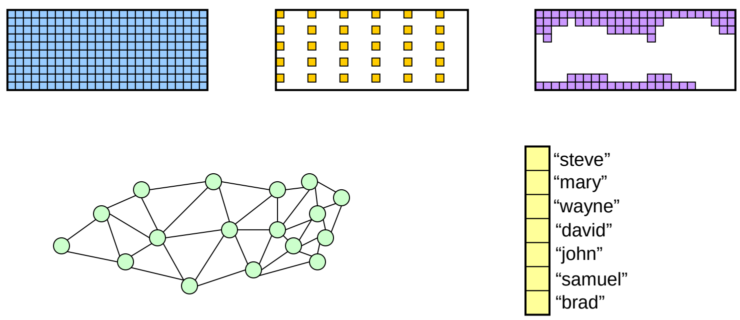


CS380P Lecture 19 Chapel 20

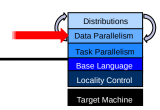
Data Parallelism: Domain Uses



Domains are used to declare arrays…

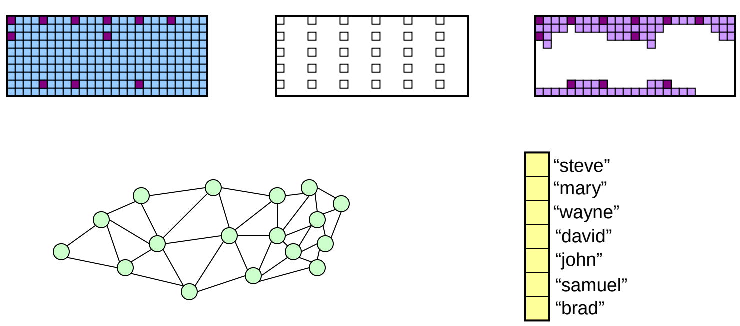


Data Parallelism: Domain Uses

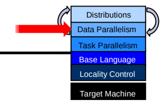


…to iterate over index sets…

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

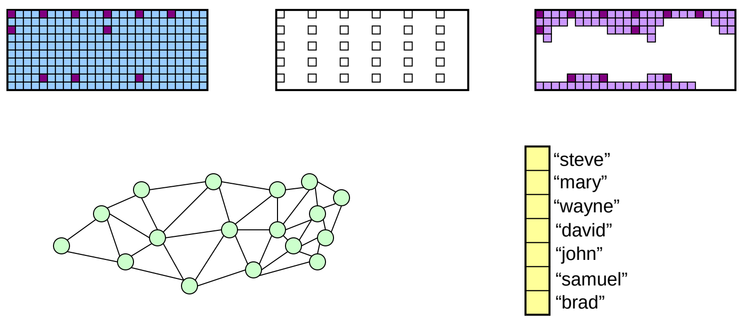


Data Parallelism: Domain Uses



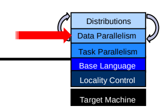
…to slice arrays…

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



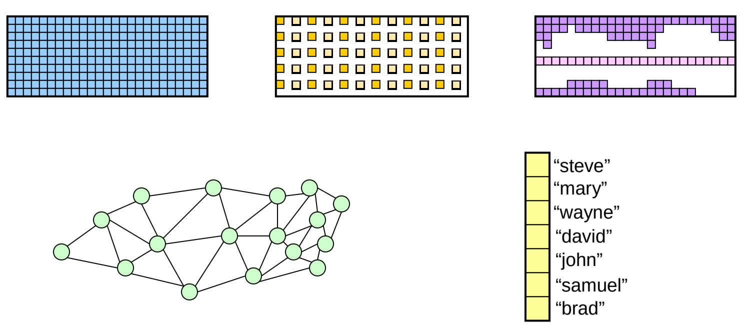
CS380P Lecture 19 Chapel 23

Data Parallelism: Domain Uses



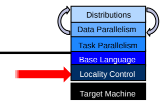
…and to reallocate arrays

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



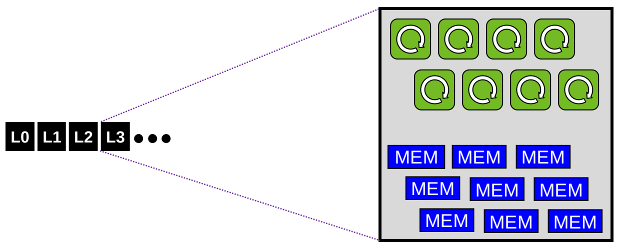
CS380P Lecture 19 Chapel 24

Locality: Locales

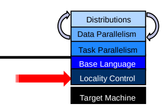


locale: **architectural unit of locality**

- 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

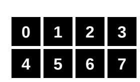
```
prompt> myChapelProg –nl=8
```

Chapel programs have built-in locale variables:

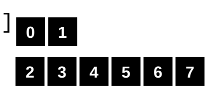
```
config const numLocales: int;
const LocaleSpace = [0..numLocales-1],
      Locales: [LocaleSpace] locale;
```

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

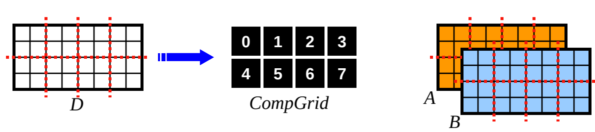
```
cobegin {
  on TaskALocs do computeTaskA(…);
  on TaskBLocs do computeTaskB(…);
  on Locales(0) do computeTaskC(…);
}
```

CS380P Lecture 19 Chapel 27

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

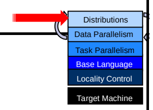
Chapel provides…

…a standard library of distributions (Block, Recursive Bisection, …)

…the means for advanced users to author their own distributions

CS380P Lecture 19 Chapel 28

Locality: Domain Distributions

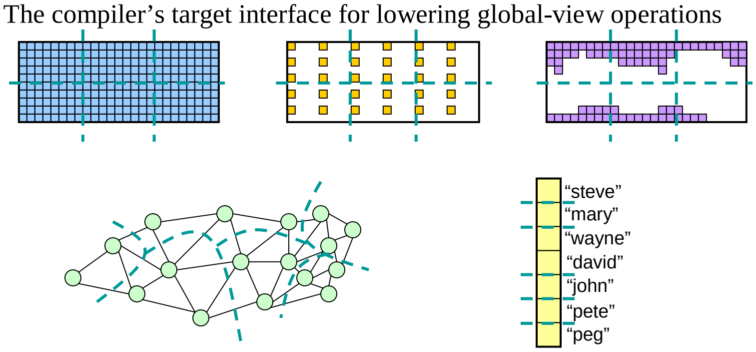


A distribution must implement

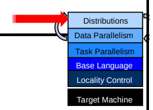
– The mapping from indices to locales

– 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 × 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

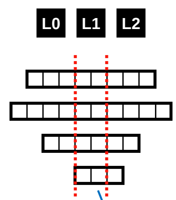
Q1: Why distinguish between distributions and domains?

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

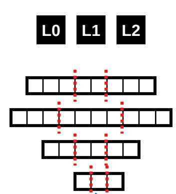
Q1: Why distinguish between distributions and domains?

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

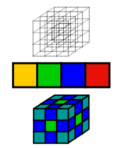
```
const D     : …distributed B1 = [1..8],
      outerD: …distributed B1 = [0..9],
      innerD: subdomain(D)    = [2..7],
      slideD: subdomain(D)    = [4..6];
```



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

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;
```

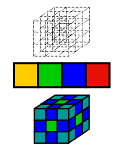

NAS MG *rprj3* stencil in Fortran+MPI

```
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
```

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

- None

Assignment 6

- Due Monday April 8th 11:59pm