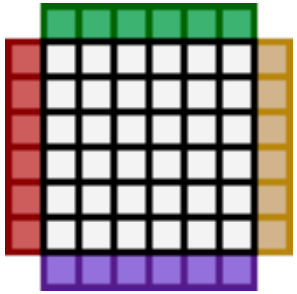
Lessons Learned in Array Programming: from ZPL to Chapel

Brad Chamberlain
Chapel Team, Cray Inc.
ARRAY 2016, June 14, 2016



Safe Harbor Statement

This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.

My Background

Education:

- Earned Ph.D. from University of Washington CSE in 2001
- Remain associated with UW CSE as an Affiliate Professor

Industrial Experience:

- Currently a Principal Engineer at Cray Inc.
- Also spent an educational year at a start-up between UW and Cray

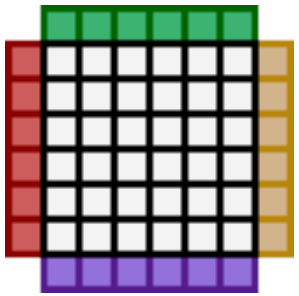
My R&D Interests

Designing and Implementing Parallel Languages

- targeting HPC (High-Performance Computing) system scales
- making use of array abstractions
  (as well as other abstractions)

Parallel Array Languages I've Worked on:

ZPL: at UW CSE

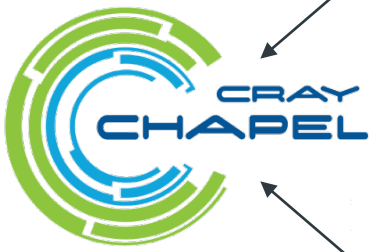


Chapel: at Cray

Chapel's Founders / Relationship to ZPL

Hans Zima
Vienna Fortran, HPF
Fortran-oriented
Performance
Minimal features

David Callahan
MTA C, Fortran
Multithreading
Generality
User-optimizable



Brad Chamberlain
ZPL
Array Programming
Syntactic performance model
SPMD execution

John Plevyak
D, Python, OCaml
Type inference
Generics
OOP

Today's Talk

Theme:

- Comparison of array features in ZPL vs. Chapel
  - and by extension, differences between academic vs. practical languages
    - **ZPL:** very pure, focused language ("array-based data parallelism")
    - **Chapel:** very general, intended for adoption ("any parallelism, including arrays")

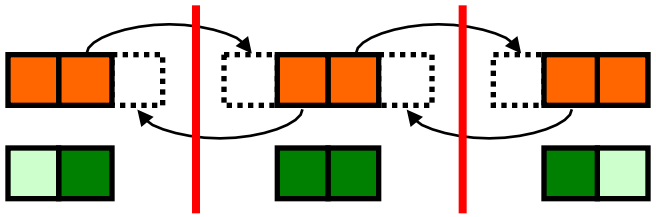
Outline:

- **Background**
- **ZPL**
  - Arrays in ZPL
  - ZPL's greatest hits
  - ZPL's drawbacks
- **Chapel arrays** (learning from ZPL's mistakes)
- **Q&A / Discussion**

Local-view vs. Global-view

Local-view:

- code describes per-task behavior
- programmer manages coordination details

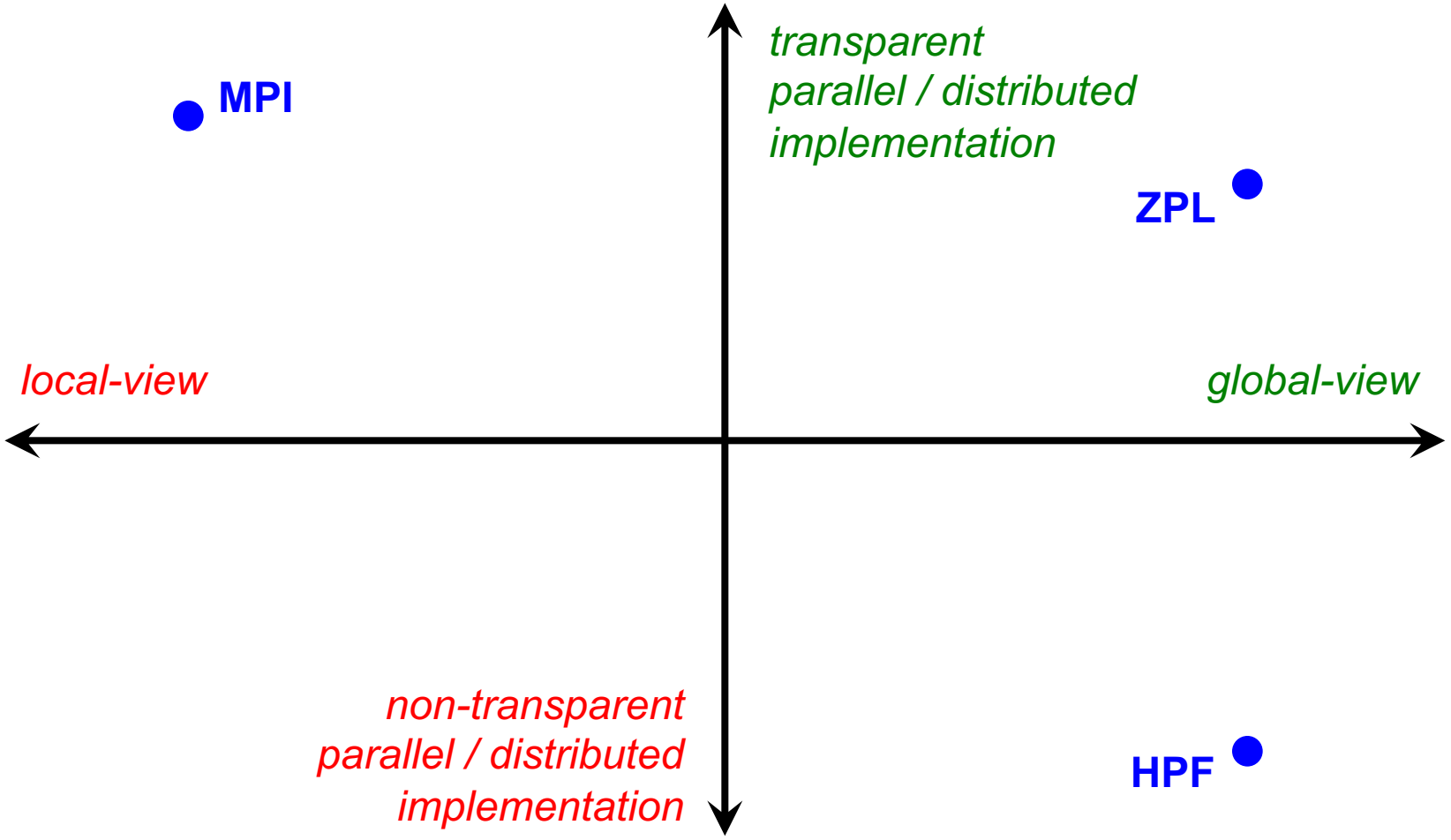


Global-view:

- code describes algorithm as a whole
- lower layers manage coordination

  (compiler, runtime, libraries)

Parallel Programming Stereotypes



ZPL Factsheet

ZPL: **an array-based data-parallel language**

Developed at: **University of Washington**

Timeframe: **1991 – 2003 (can still download today)**

Target systems: **1990's-era HPC systems**

- clusters of commodity processors or SMPs
- custom parallel architectures:
  - Cray T3E, KSR, SGI Origin, IBM SP2, Sun Enterprise, …

Related languages: **HPF**

Main concepts:

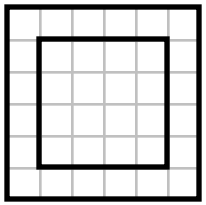
- abstract machine model: *the CTA*
- data parallelism via *regions* and arrays
- *WYSIWYG performance model*

Regions

Regions: index sets that…

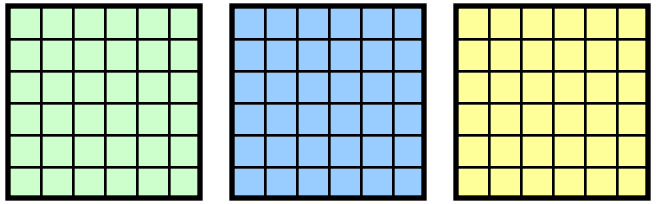
…can be named

```
region     R = [1..n  ,1..n  ];
       BigR = [0..n+1,0..n+1];
```



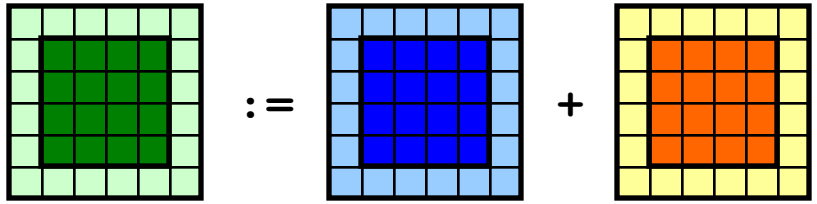
…are used to declare *parallel arrays*

```
var A, B, C:[BigR] integer;
```

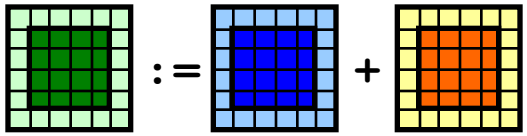


…specify indices for a statement's array references

```
[R] A := B + C;
```



Regions Eliminate Redundancy



C:

```
for (i=1; i<=n; i++) {
  for (j=1; j<=n; j++) {
    A[i][j] = B[i][j] + C[i][j];
  }
}
```

F90:

```
A(1:n,1:n) = B(1:n,1:n) + C(1:n,1:n)
```

ZPL:

```
[1..n,1..n] A := B + C;
```

or:

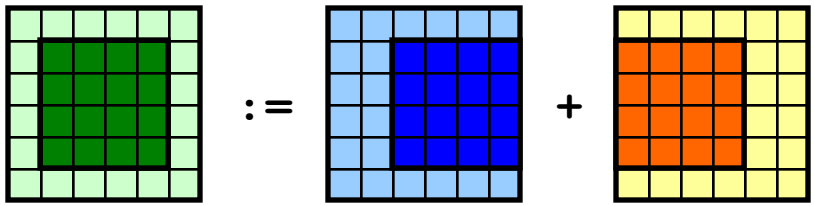
```
[R] A := B + C;
```

Array Operators

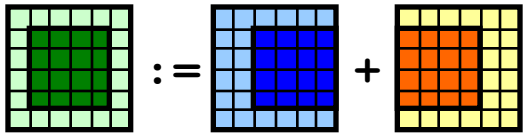
Array Operators: modify the current region's index set for an array reference

E.g., @-operator translates indices by a *direction*:

```
direction east = [0, 1];
          west = [0,-1];

[R] A := B@east + C@west;
```



Regions Emphasize Differences



C:

```
for (i=1; i<=n; i++) {
  for (j=1; j<=n; j++) {
    A[i][j] = B[i][j+1] + C[i][j-1];
  }
}
```

F90:

```
A(1:n,1:n) = B(1:n,2:n+1) + C(1:n,0:n-1)
```

ZPL:

```
[1..n,1..n] A := B@[0,1] + C@[0,-1];
```

or:

```
[R] A := B@east + C@west;
```

More on Regions / Parallel Arrays

- Region scopes can nest:

```
[R] begin
      A := Index1 + Index2 / 10.0;
      B := A@east;
      [i, 1..n] B := 0.0;
      foo(B);
    end;
```

- Constrained, by design:
  - Parallel arrays do not support “normal” indexing
  - Regions do not support direct iteration

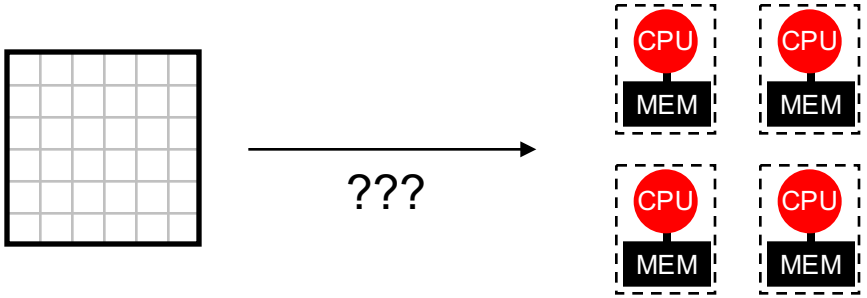
```
for (i,j) in R do A[i,j] = B[i,j+1];
```

Parallel Interpretation of Regions

- Users specify virtual processor grid at runtime

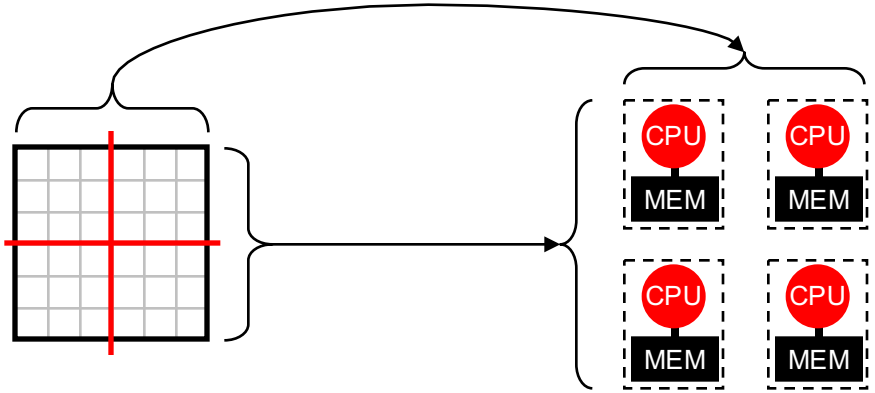
```
> ./a.out -p4          (uses a 2x2 grid  ⊡⊡ )
> ./a.out -p4 -r1      (uses a 1x4 grid  □□□□ )
```

- Region indices are distributed to this grid
  - defines data distribution for arrays
  - defines work distribution for computations



Region Distribution

1) Regions are block-distributed:



2) *Interacting regions* are distributed identically


```
[R] A := B + C;
```

ZPL's Performance Model

Distribution rules imply a performance model:

- Traditional operators are perfectly parallel

```
[R] A := B + C;
```

- Array operators indicate likely communication
  - of a particular style

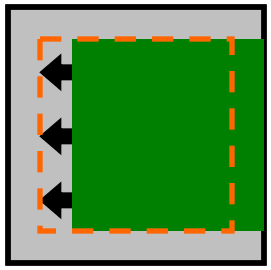
```
[R] A := B@east + C@west;
```

@-induced Communication

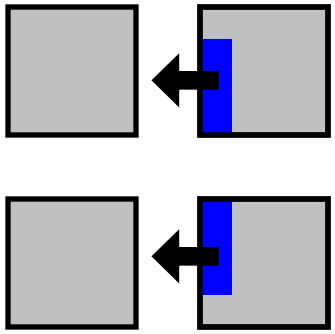
@ is used to refer to neighboring elements.

- data must be transferred to neighboring processors

```
[R] A := A@east;
```



global view



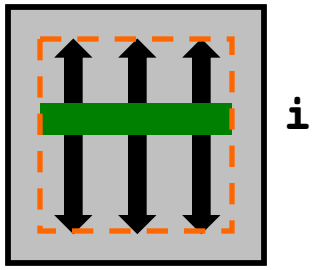
local view

Flood Communication

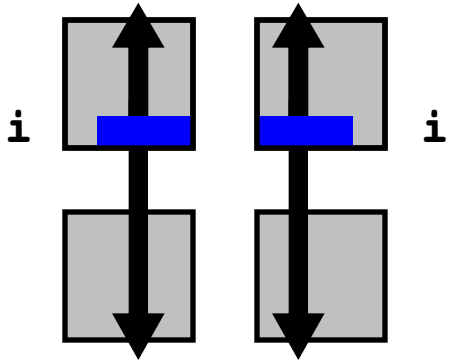
Floods replicate data across array dimensions

- data must be broadcast over sub-dimensions of the processor grid

```
[R] A := >>[i,] A;
```



global view



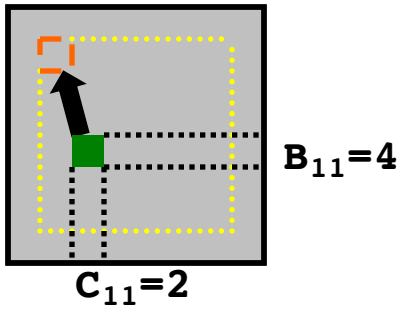
local view

Remap Communication

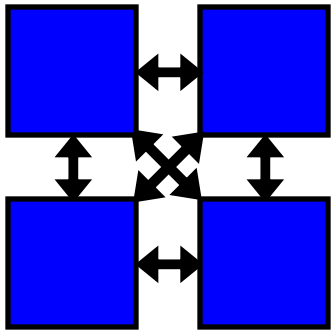
Remap arbitrarily re-arranges data

- indexes into an array with other arrays
- tends to require all-to-all communication

```
[R] A := A#[B,C];
```



global view



local view

Array Operator Summary

operator	effect	sample	communication
@	translate indices	`A@east`	point-to-point
flood	replicate array elements	`>>[i,] A`	sub-grid broadcast
reduction	collapse array elements	`+<<[R] A`	sub-grid reduction
scan	parallel scan	`+|| A`	parallel prefix
remap	arbitrary gather/scatter	`A#[X,Y]`	all-to-all

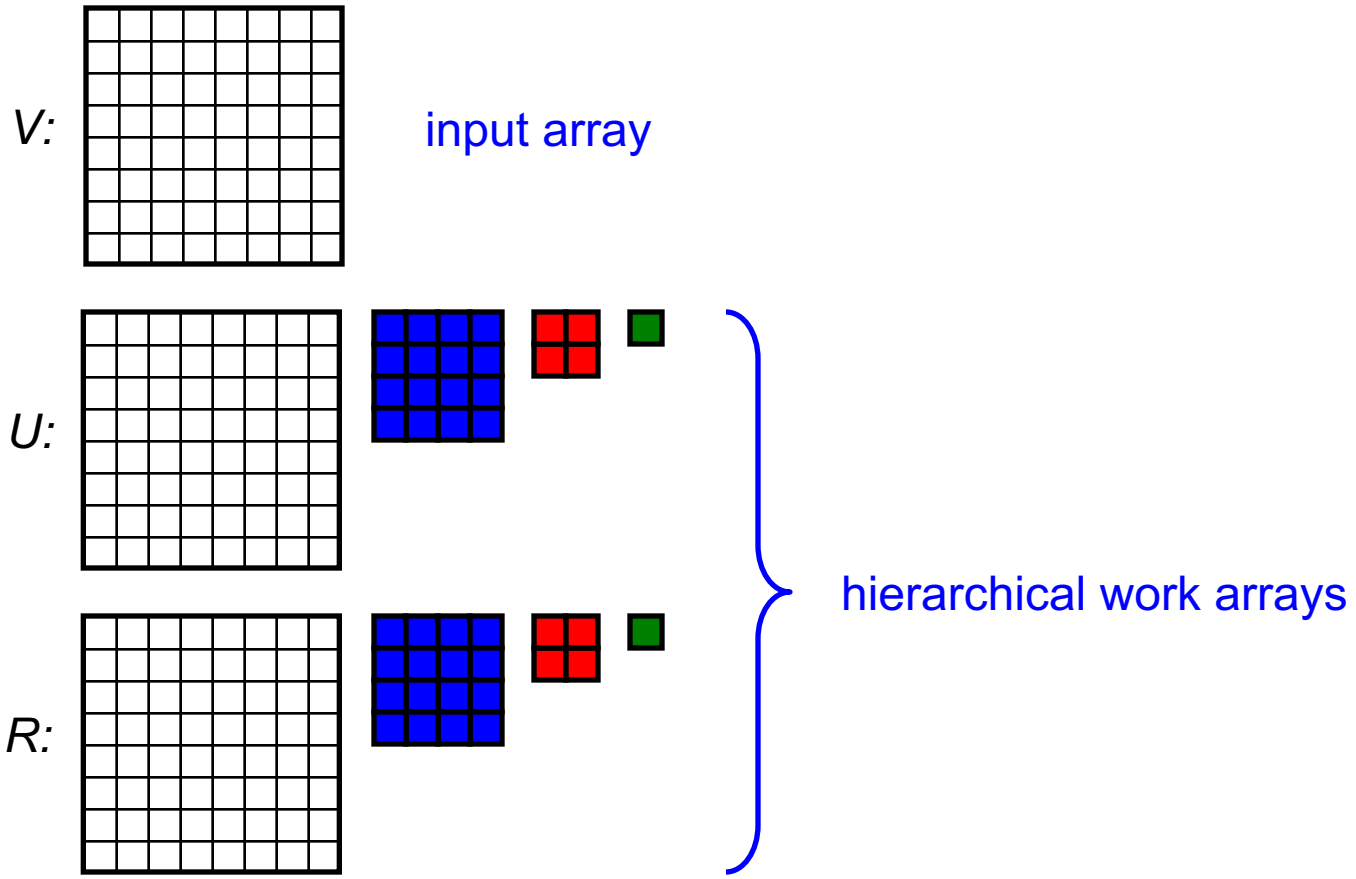
This is what we refer to as ZPL's *WYSIWYG performance model*:
Communication implications are visible in a program's syntax.

ZPL's Greatest Hits: Hierarchical Arrays in NAS MG

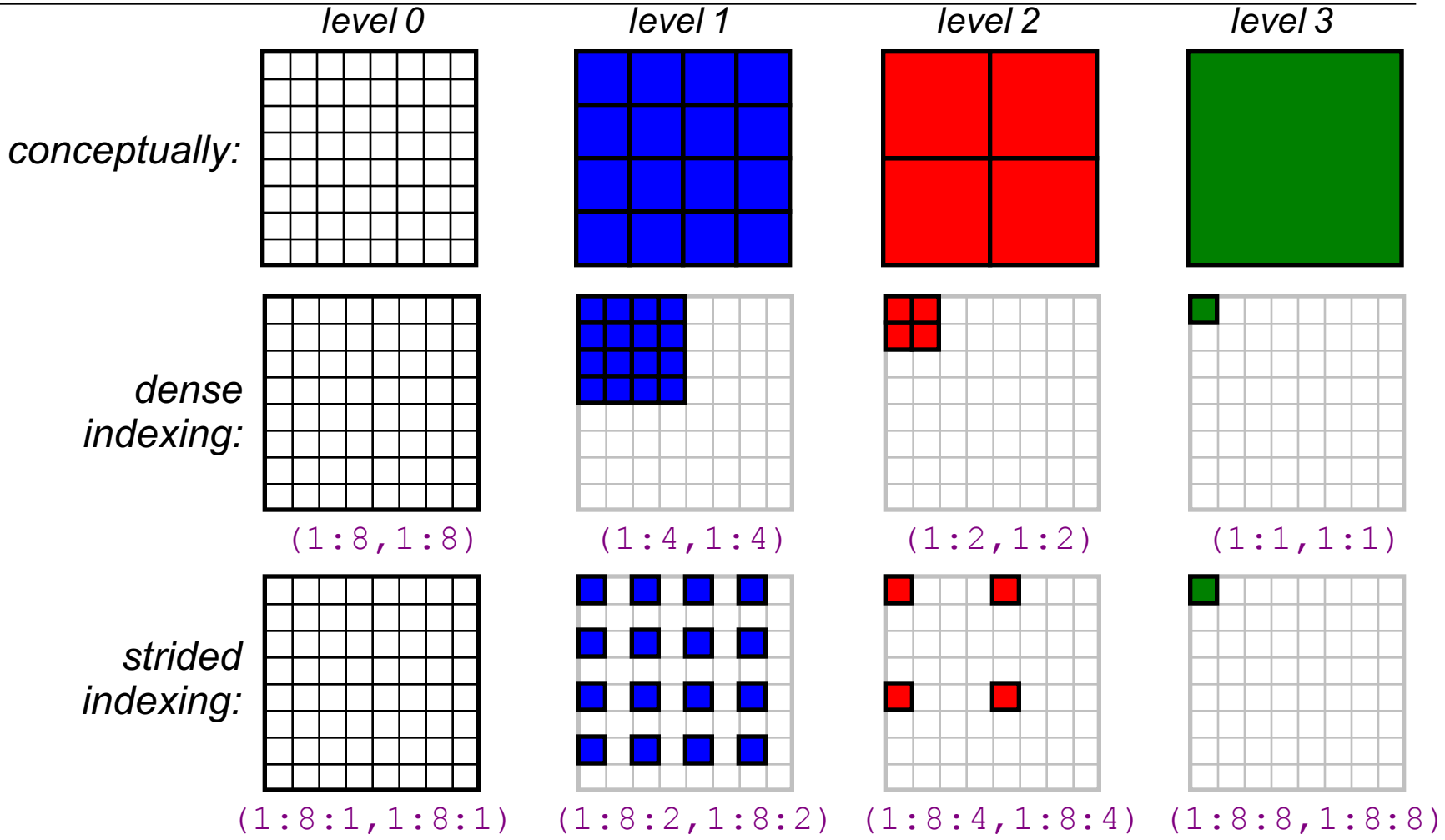
The NAS MG Benchmark

Mathematically: use a 3D multigrid method to find an approximate solution to a discrete Poisson problem ($\nabla^2 u = v$)

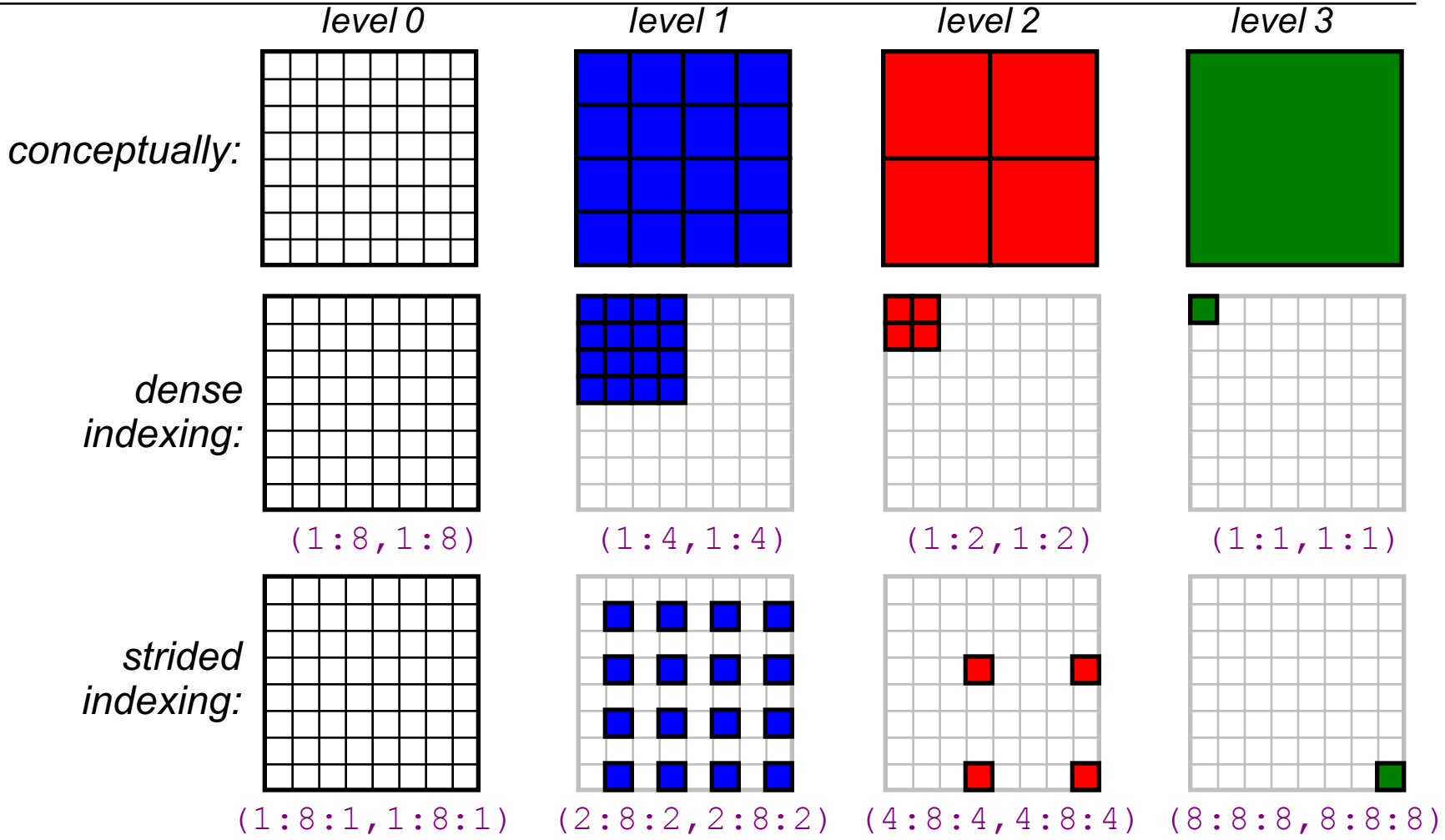
MG's arrays



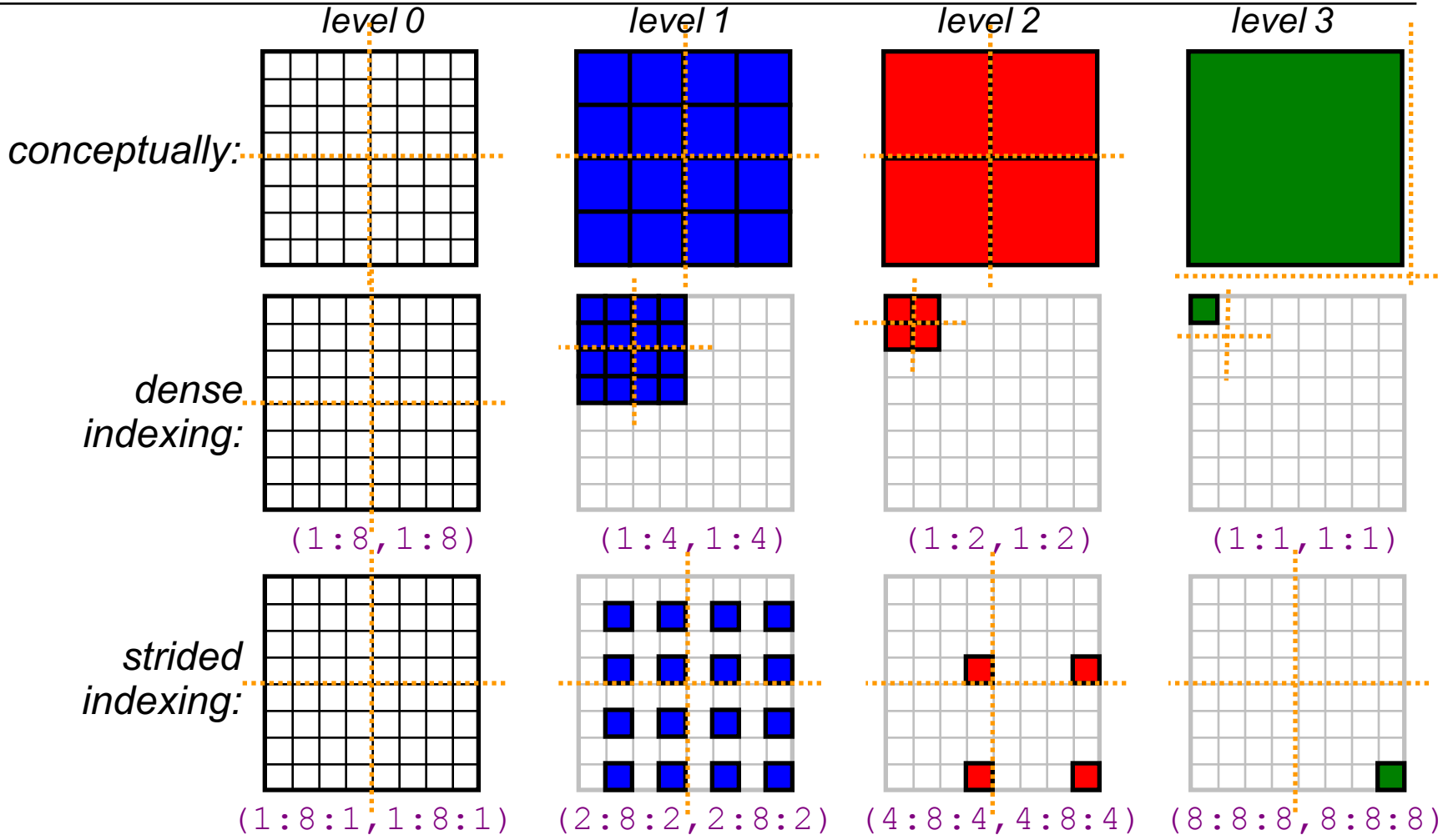
Hierarchical Arrays



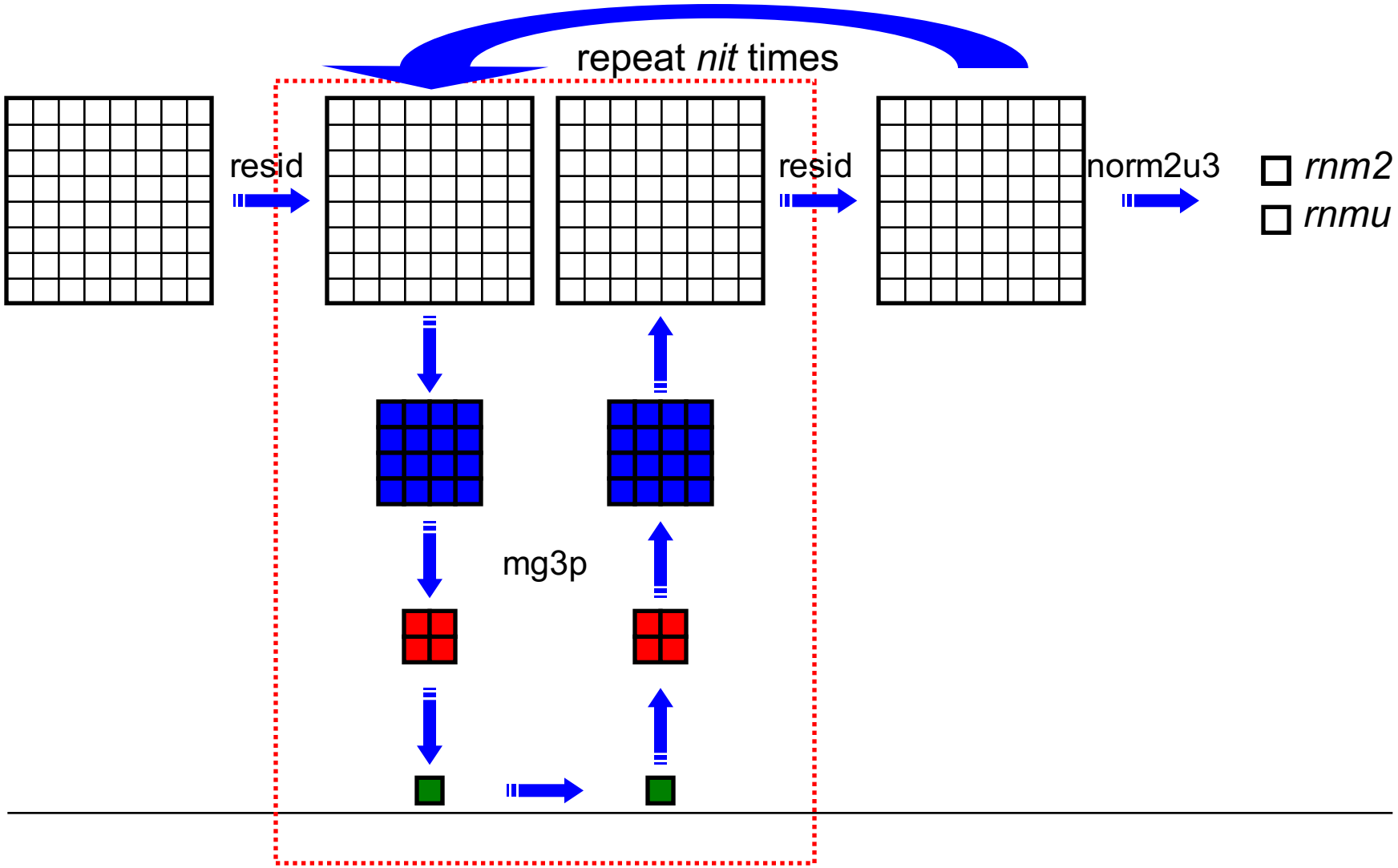
Hierarchical Arrays



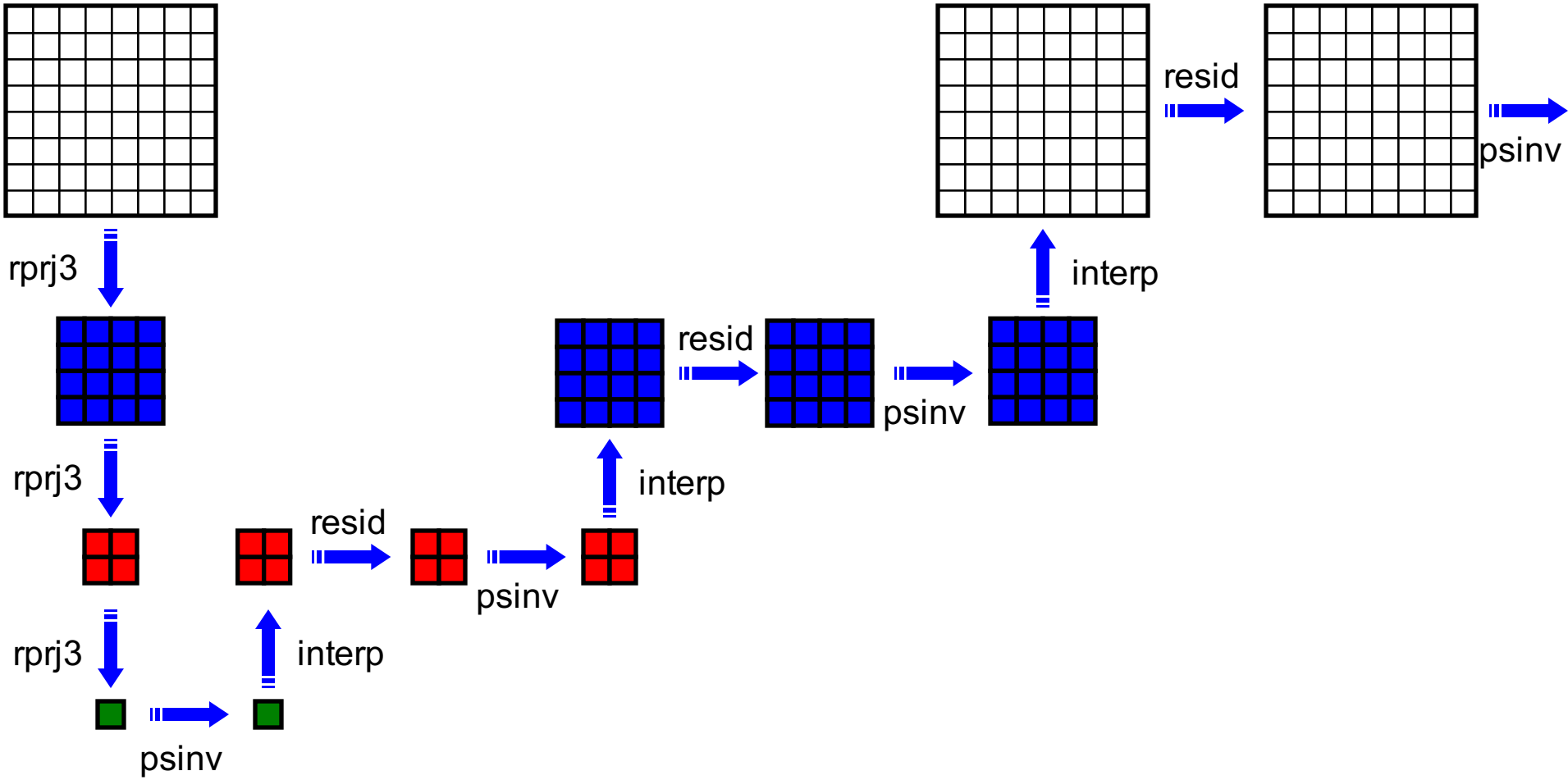
Distributed Hierarchical Arrays



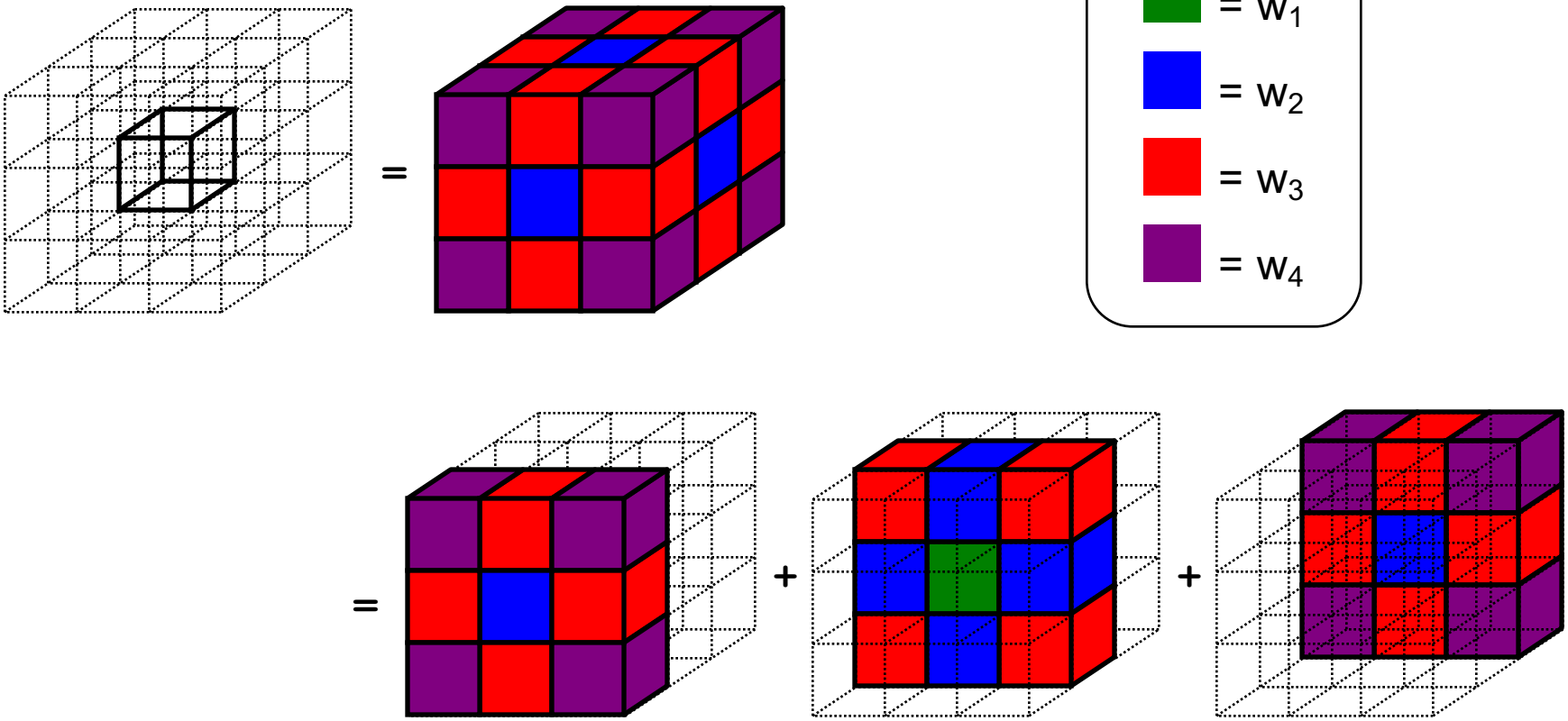
MG's Timed Portion



MG's Guts (*mg3P*)

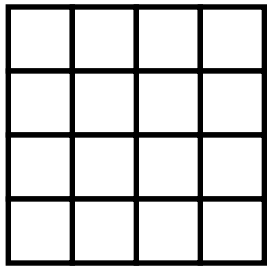


27-point stencils

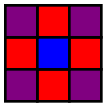


rprj3(S, R)

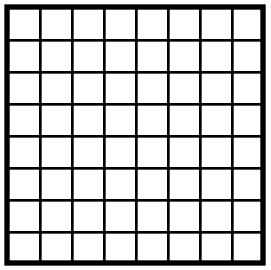
conceptually:



convolve(

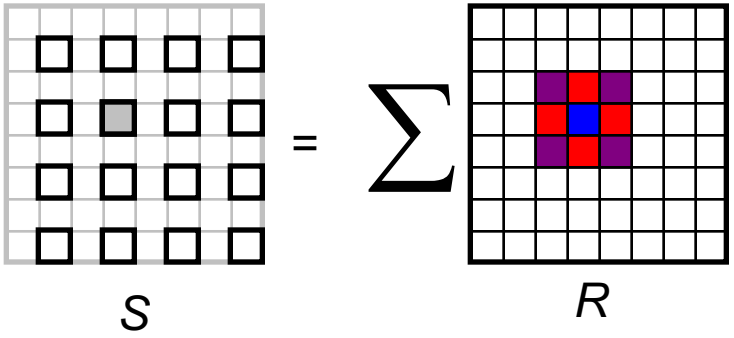


,



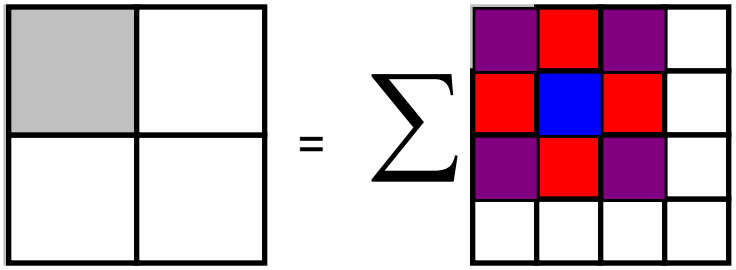
)

strided indexing:

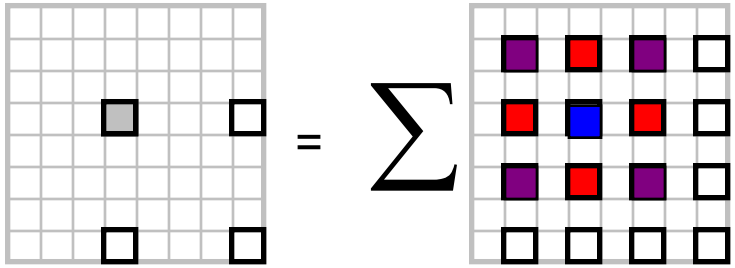


rprj3 at other levels

conceptually:



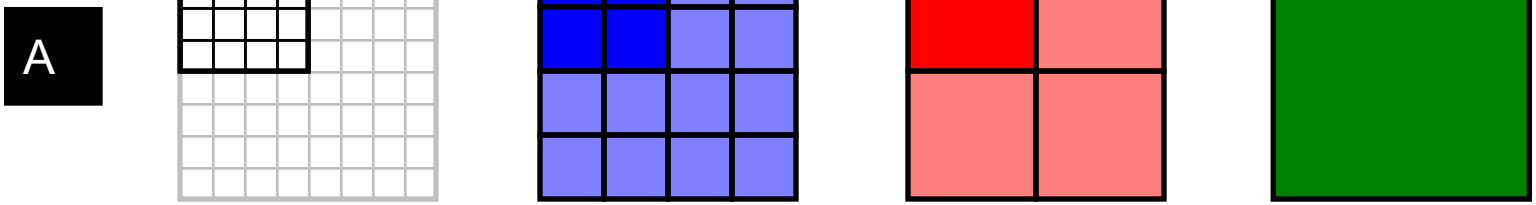
strided indexing:



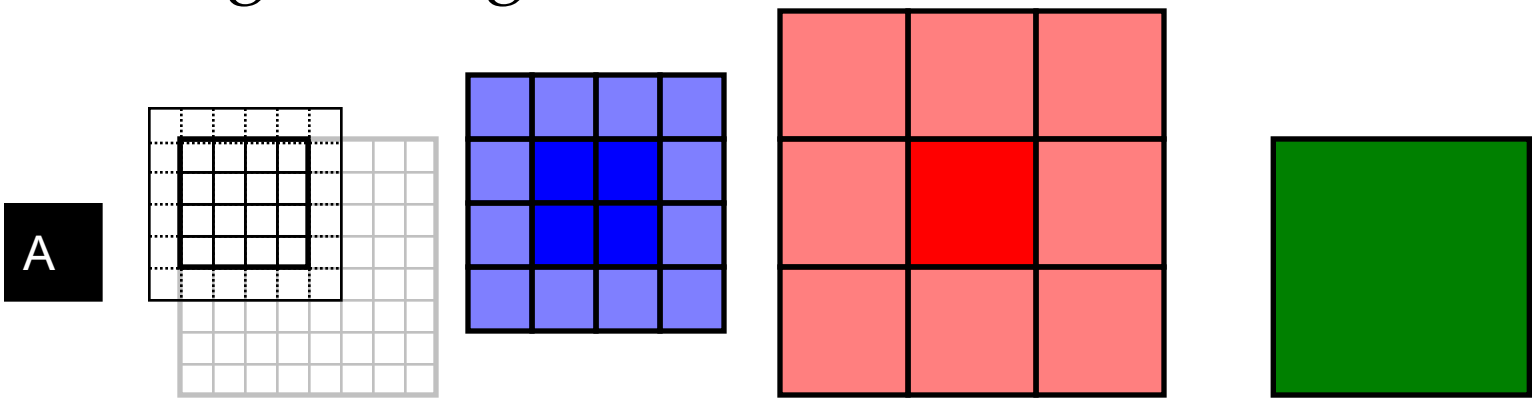
S *R*

Per-processor Data Allocation

In addition to its local block of values…

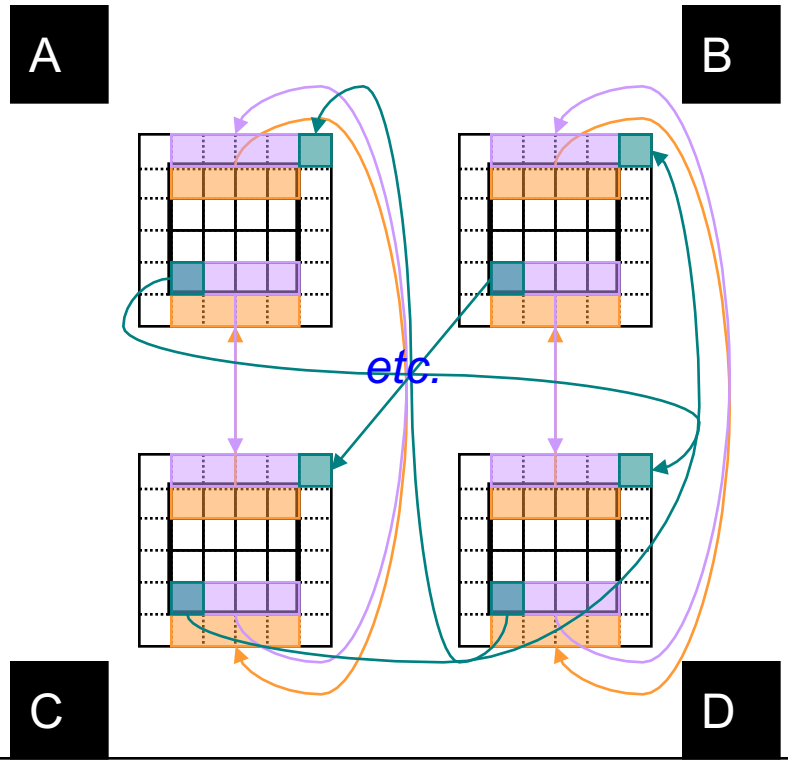


…each processor allocates ghost cells to cache neighboring values



Stencil Communication

Prior to computing a stencil, communication is typically required to refresh the ghost cells



Notes:

- Lots of optimization opportunities (e.g., comm/comp overlap, bulk transfers)
- Have to eventually start skipping processors at coarser levels

Distributed *rprj3* in Fortran + MPI

```
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
```

Local-view *rprj3* in Fortran + MPI

```
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
```

comm3: Communication in 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 commlp(  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  commlp(  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 i2=2,n2-1
         buff_len = buff_len + 1
         buff(buff_len, buff_id )= u( i1,n2-
     1,i3)
      enddo
   enddo
endif

if( axis .eq. 3 )then
   do i3=2,n3-1
      do i2=1,n2
         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
```

Global-view *rprj3* in ZPL

```
procedure rprj3(var S,R: [,,] double;
                d: array [] of direction);
begin
  S := 0.5000 * R +
       0.2500 * (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] +
       0.1250 * (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;
```

rprj3 kernel in Fortran, naively

```
   do  j3=2,m3j-1
    i3 = 2*j3-d3
    do  j2=2,m2j-1
      i2 = 2*j2-d2
      do  j1=2,m1j-1
       i1 = 2*j1-d1
       s(j1,j2,j3)  =
>          0.5D0 * r(i1,i2,i3)
>        + 0.25D0 * (r(i1-1,i2,i3)  + r(i1+1,i2,i3)
>                  + r(i1,   i2-1,i3   ) + r(i1,  i2+1,i3   )
>                  + r(i1,   i2,  i3-1) + r(i1,  i2,  i3+1))
>        + 0.125D0 * (r(i1-1,i2-1,i3   ) + r(i1-1,i2+1,i3   )
>                  + r(i1-1,i2,   i3-1) + r(i1-1,i2   ,i3+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))
>                  + r(i1,   i2-1,i3-1) + r(i1,   i2-1,i3+1)
>                  + r(i1,   i2+1,i3-1) + r(i1,   i2+1,i3+1))
>        + 0.0625D0 * (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)
>                   + 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
    enddo
  enddo
```

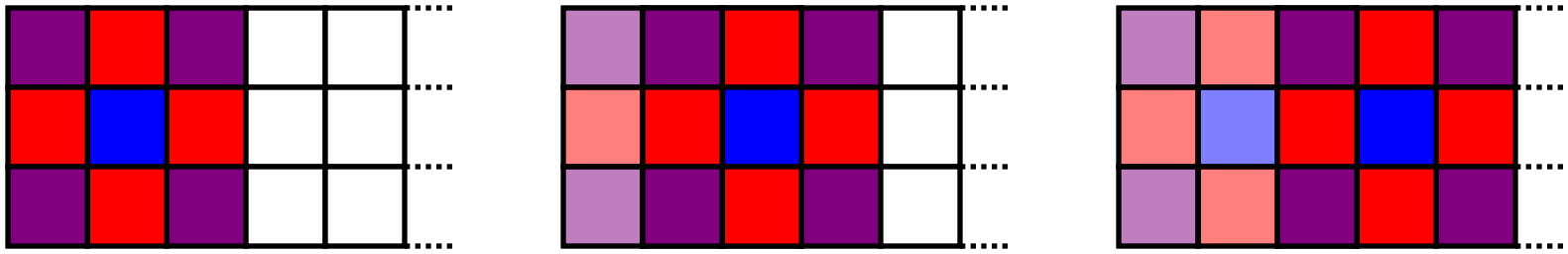
Actual *rprj3* Fortran code

```
subroutine rprj3(r,m1k,m2k,m3k,s,m1j,
implicit none
include 'cafnpb.h'
include 'globals.h'

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

double precision r(m1k,m2k,m3k), s(m1
integer j3, j2, j1, i3, i2, i1, d1, d
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
```

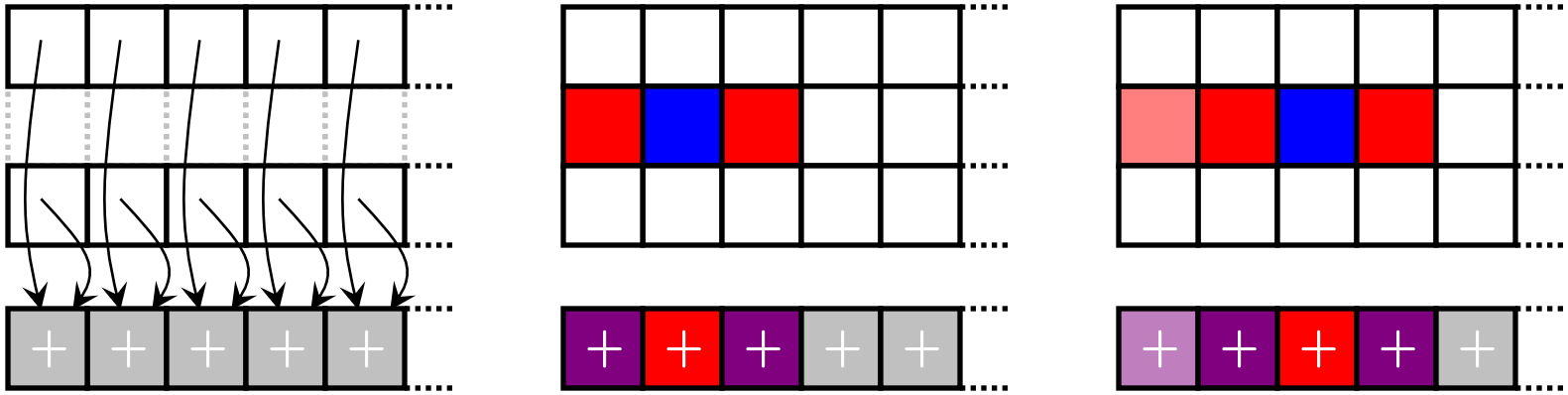
```
  do  j3=2,m3j-1
   i3 = 2*j3-d3
   do  j2=2,m2j-1
     i2 = 2*j2-d2
     do j1=2,m1j
      i1 = 2*j1-d1
      x1(i1-1) = r(i1-1,i2-1,i3   ) + r(i1-1,i2+1,i3   )
>              + r(i1-1,i2,   i3-1) + r(i1-1,i2,   i3+1)
      y1(i1-1) = r(i1-1,i2-1,i3-1) + r(i1-1,i2-1,i3+1)
>              + r(i1-1,i2+1,i3-1) + 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
```

Stencil Optimization (2D)

- Adjacent stencils use common subexpressions:

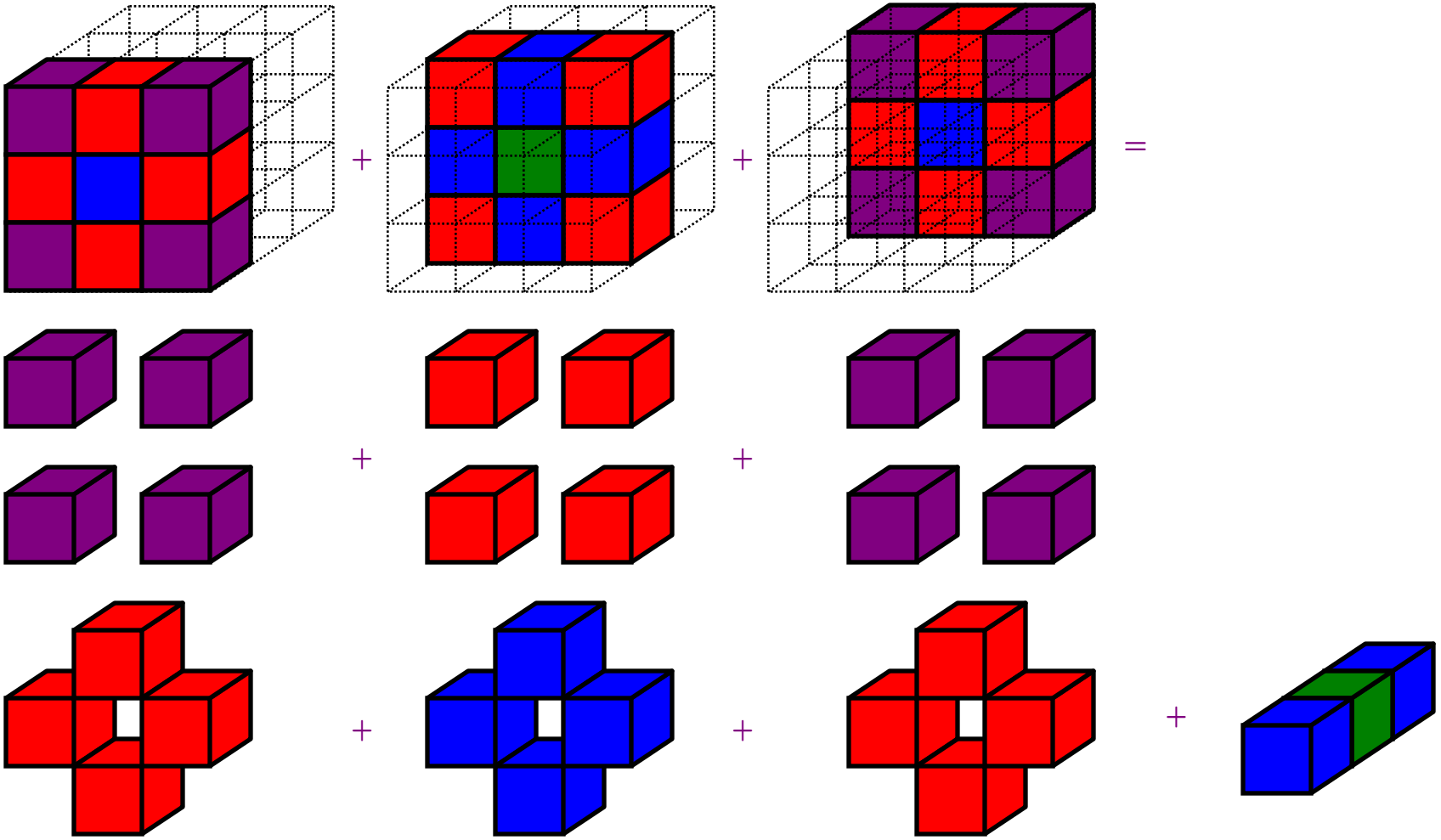


- Observation: Cache partial sums for reuse…

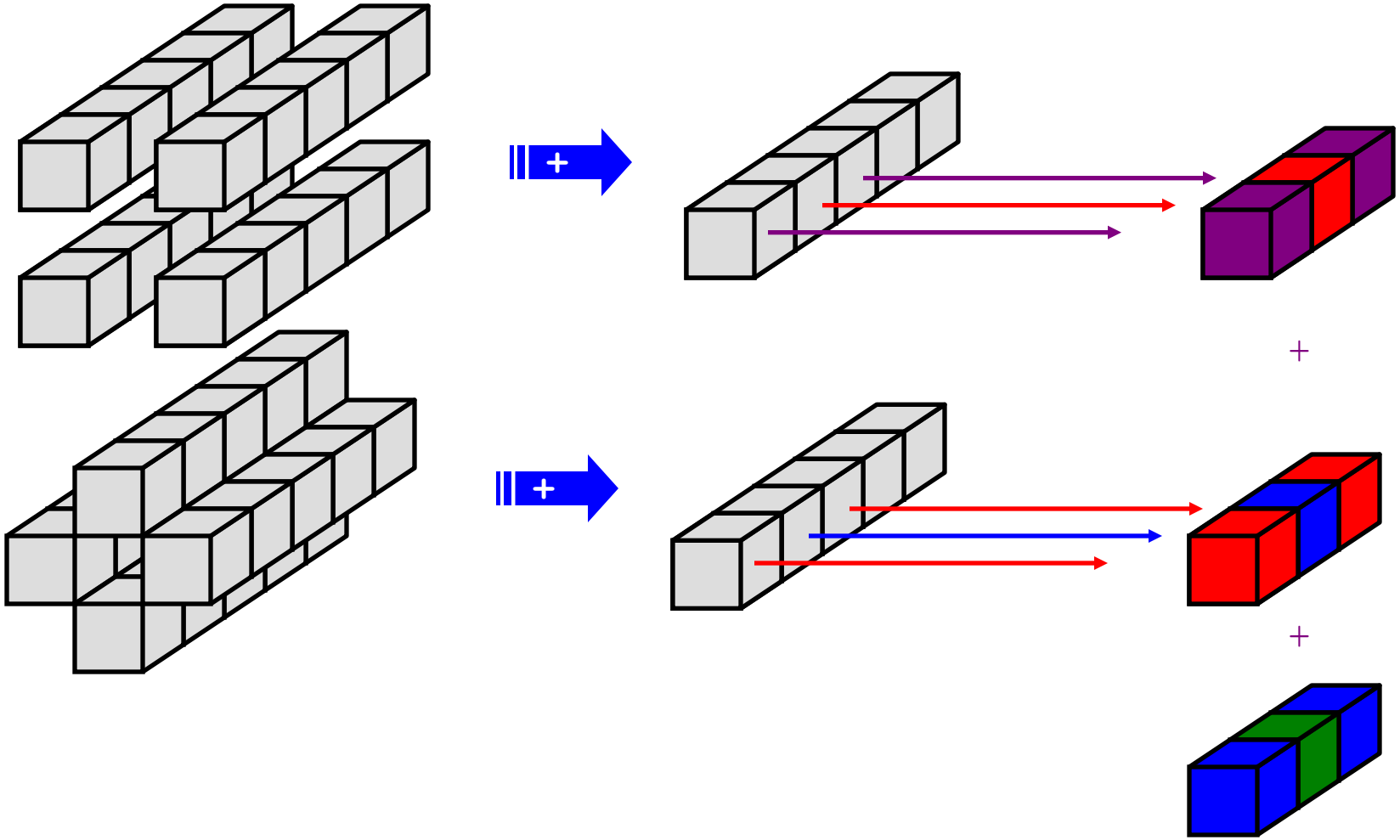


- Benefits are greater for 3D stencils…

MG Stencil Optimization



MG Stencil Optimization



Optimized stencil in Fortran

```
subroutine rprj3(r,m1k,m2k,m3k,s,m1j,
implicit none
include 'cafnpb.h'
include 'globals.h'

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

double precision r(m1k,m2k,m3k), s(m1
integer j3, j2, j1, i3, i2, i1, d1, d
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
```

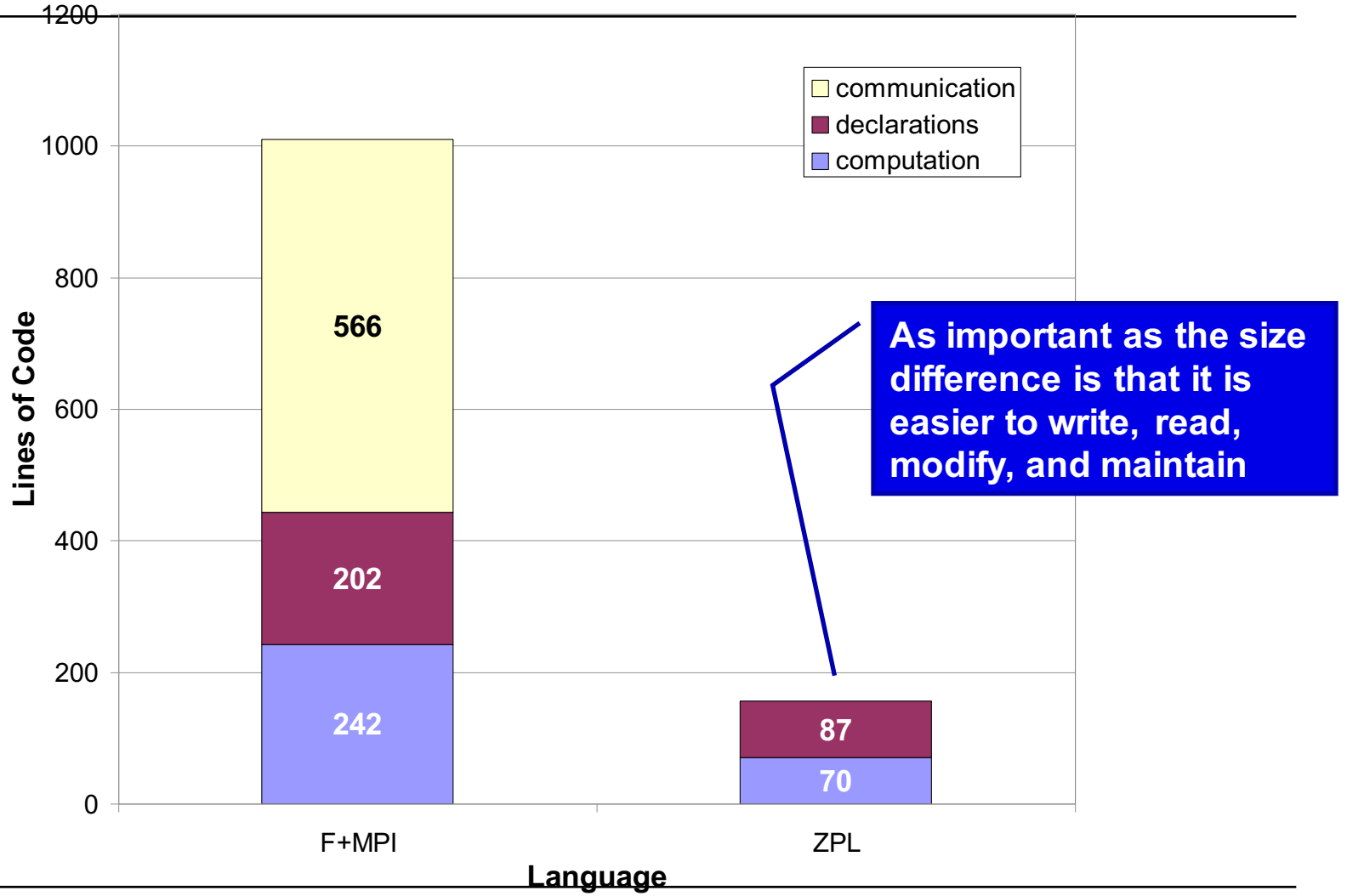
```
  do  j3=2,m3j-1
   i3 = 2*j3-d3
   do  j2=2,m2j-1
     i2 = 2*j2-d2
     do j1=2,m1j
      i1 = 2*j1-d1
      x1(i1-1) = r(i1-1,i2-1,i3   ) + r(i1-1,i2+1,i3   )
>              + r(i1-1,i2,   i3-1) + r(i1-1,i2,   i3+1)
      y1(i1-1) = r(i1-1,i2-1,i3-1) + r(i1-1,i2-1,i3+1)
>              + r(i1-1,i2+1,i3-1) + 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
```

Optimized stencil in ZPL

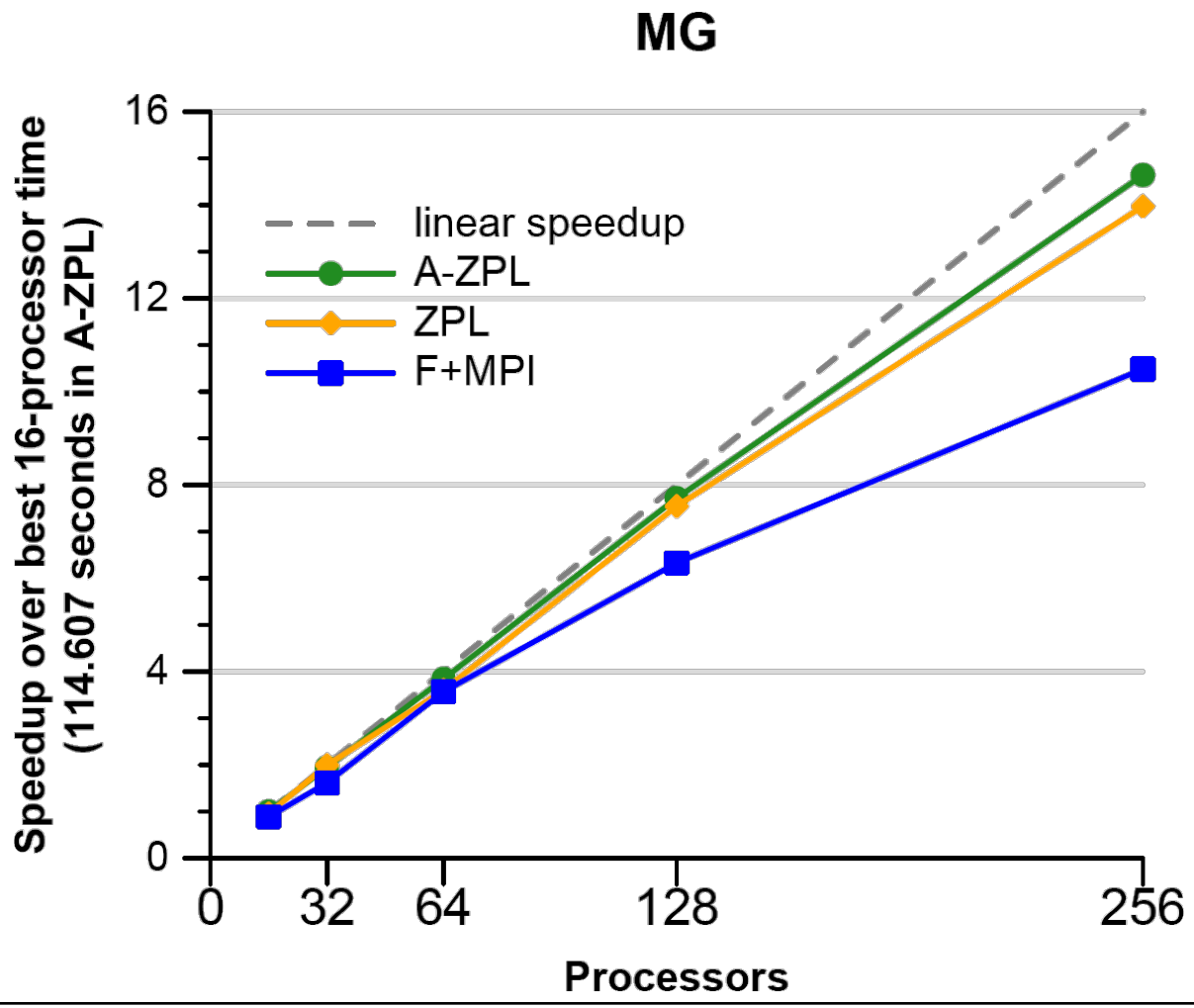
```
procedure rprj3(var S,R: [,,] double;
                d: array [] of direction);
begin
  S := 0.5000 * R +
       0.2500 * (R@^d[ 1, 0, 0] + R@^d[ 0, 1, 0] + R@^d[ 0, 0, 1] +
                 R@^d[-1, 0, 0] + R@^d[ 0,-1, 0] + R@^d[ 0, 0,-1] +
       0.1250 * (R@^d[ 1, 1, 0] + R@^d[ 1, 0, 1] + R@^d[ 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;
```

Compiler-optimized, thanks to rich semantic information about arrays (esp. relationship between hierarchical arrays and offsets)

Code Size Comparison



NAS MG Speedup: Cray T3E



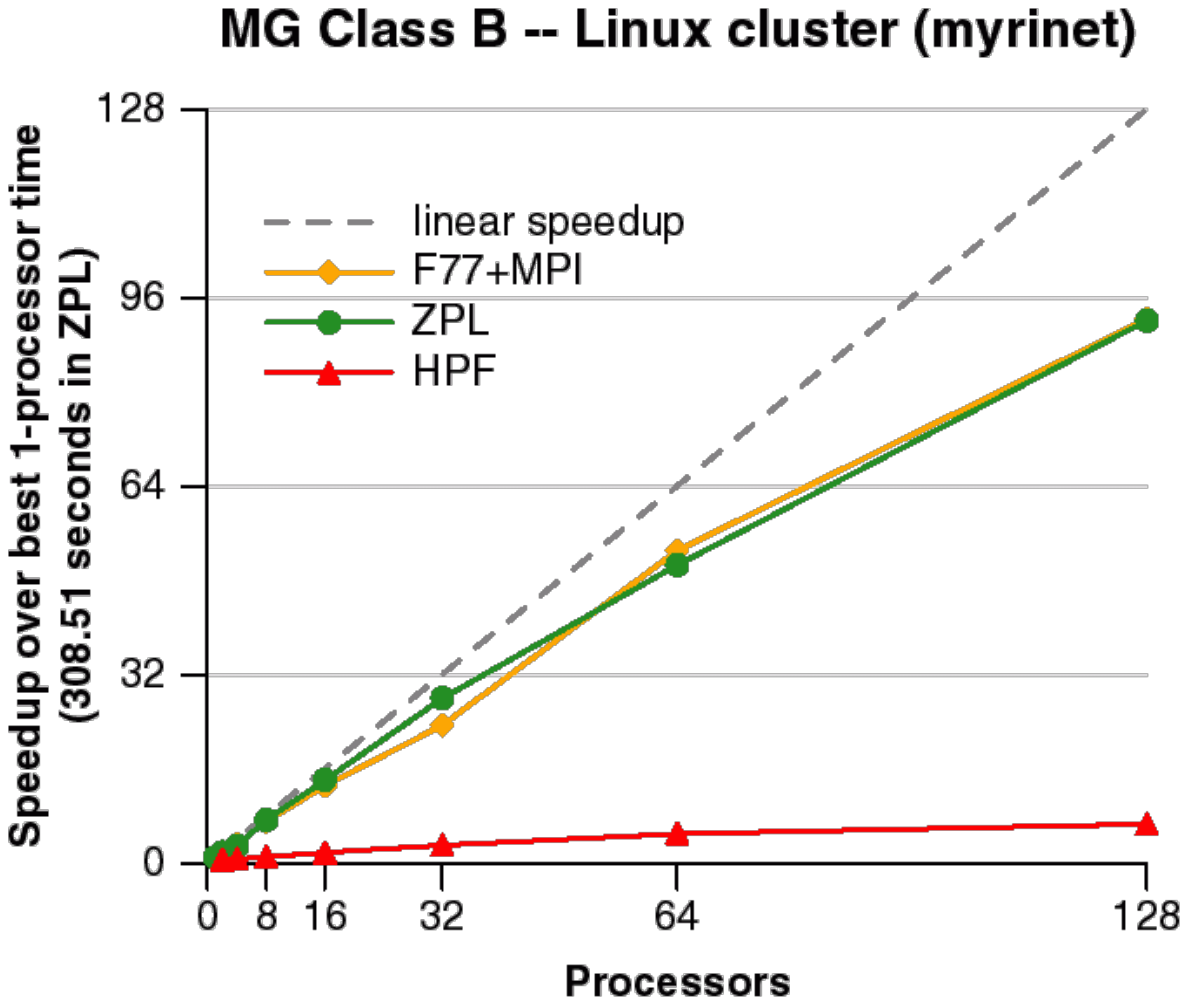
How did ZPL outperform MPI?

- MPI embeds specific communication idioms
  - non-blocking sends/receives
    - carries buffering & synchronization assumptions
- ZPL only expresses computation's intent
  - permits implementation to map to puts/gets
    - RDMA is the optimal choice on this Cray T3E
- Of course, MPI could use RDMA as well…

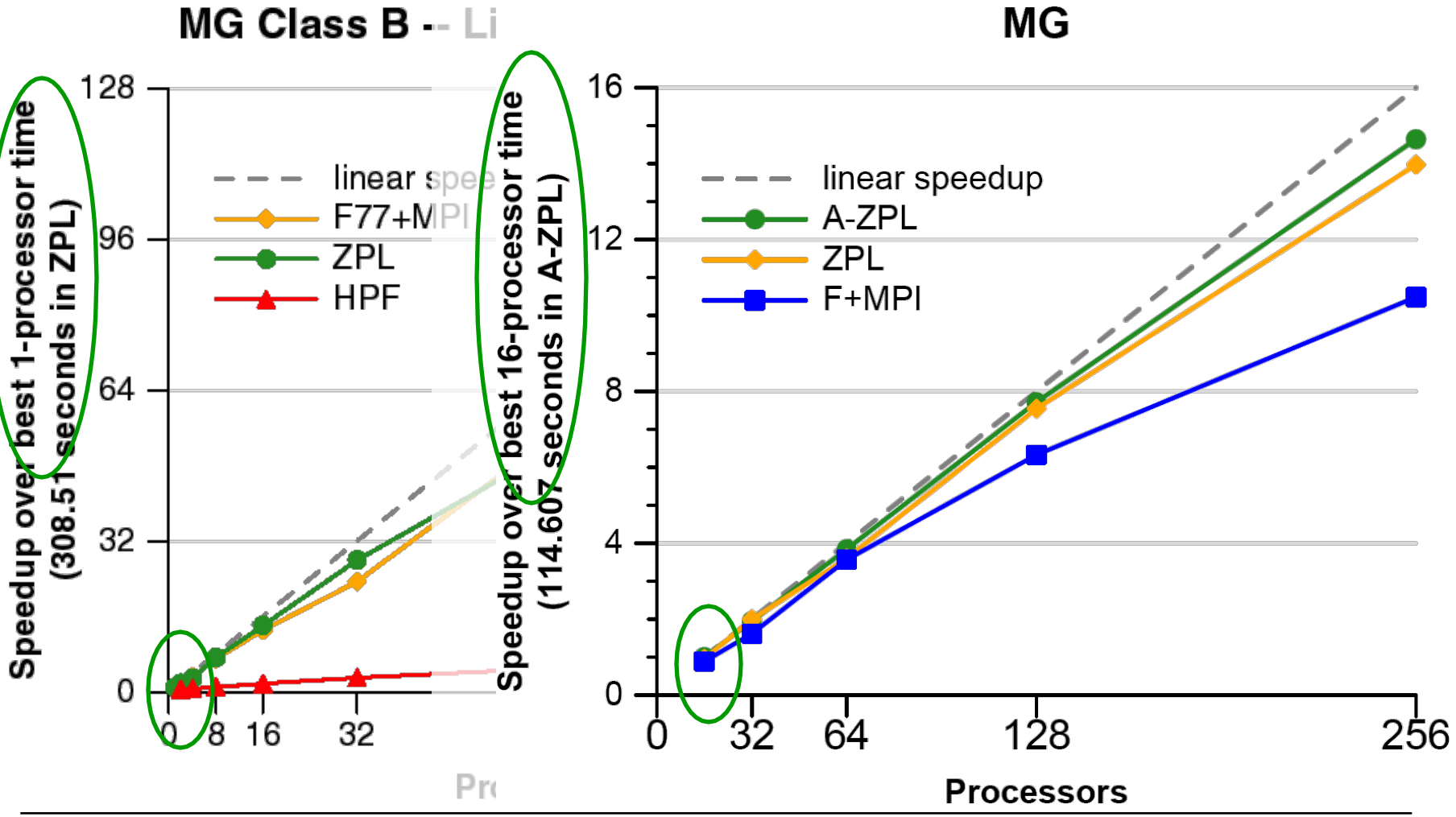
  ...but only via modifications to the code...

  ...and what would happen on non-RDMA systems?
- Meanwhile, ZPL can target MPI on clusters…

NAS MG Speedup: Myrinet cluster



ZPL was also fastest at small-scale



Code [In]Flexibility

The Fortran+MPI version…

- …only supports 2^k problem sizes
- …only supports running on 2^p compute nodes
  - moreover, both values must be specified at compile-time
- …only supports a single 3D data decomposition
- …could be rewritten to avoid these assumptions…
  - …but at what cost in terms of level of effort? code clarity?

In contrast, the ZPL version...

- …supports arbitrary problem sizes and node counts
  - and permits them to be specified at run-time
- …supports decomposing in 1, 2, or all 3 dimensions

NAS MG in ZPL: Summary

A resounding success for array programming!

- global-view of computation
- ability to reason about communication needs
- clear, concise, compact code
- optimized stencil computations
- system-specific communication idioms
- optimized communication (not described here)
- flexible parameterization (problem size, grid, …)

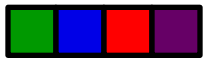
NAS MG in ZPL: Summary

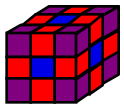
But also a significant downside:

- expression of stencil required O(*points*) code
  - OK for 27-point stencils
  - less so for, say, 216-point stencils in FMM

rprj3 stencil in Chapel

```
proc rprj3(S: [?SD], R) {
  const Stencil = {-1..1, -1..1, -1..1},

        w = (0.5, 0.25, 0.125, 0.0625),

        w3d: [Stencil] real
           = [(i,j,k) in Stencil]
               w((i!=0) + (j!=0) + (k!=0));

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

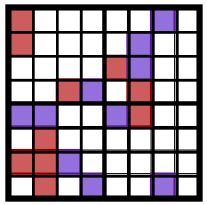




ZPL's Greatest Hits: Sparse Arrays in NAS CG

Sparse Arrays

- Good sparse arrays are a must for scientific computation
  - should be built-in
  - should be semantically similar to dense arrays



- NAS CG in a nutshell:
  - lots of sparse matrix-vector multiplications in a loop

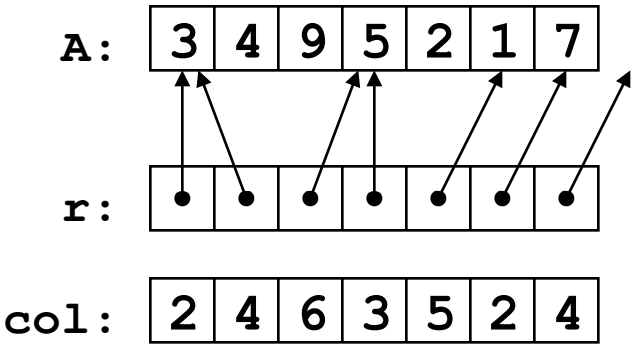
Compressed Sparse Row (CSR) Format

A:

0	0	0	0	0	0
0	3	0	4	0	9
0	0	0	0	0	0
0	0	5	0	2	0
0	1	0	0	0	0
0	0	0	7	0	0

= dense data vector

\+ structure information



Fortran Mat-Vect Multiplication

Dense Matrix-Vector Multiplication:

```
integer n
real*8 A(n,n)
real*8 t, V(n), S(n)


do i = 1,n
   t = 0.d0
   do j = 1,n
      t = t + A(i,j)*V(j)
   enddo
   S(i) = t
enddo
```

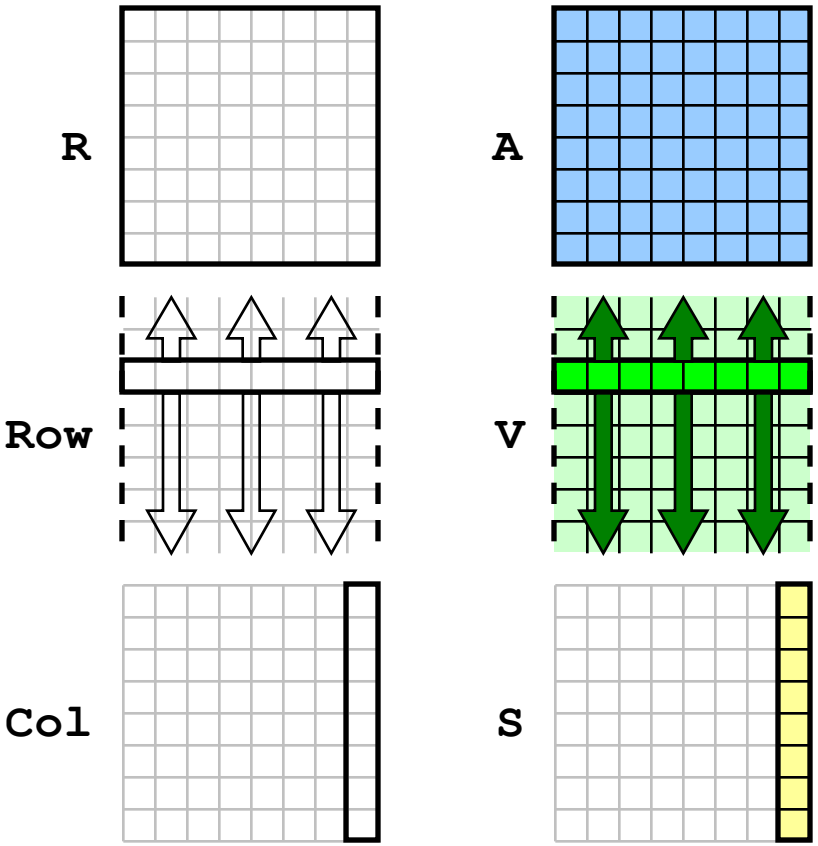
Sparse Matrix-Vector Multiplication:

```
integer n, nnz
real*8 A(nnz)
real*8 t, V(n), S(n)
integer r(n+1),col(nnz)

do i = 1,n
   t = 0.d0
   do j = r(i),r(i+1)-1
      t = t + A(j)*V(col(j))
   enddo
   S(i) = t
enddo
```

ZPL Mat-Vect Multiplication

declarations:

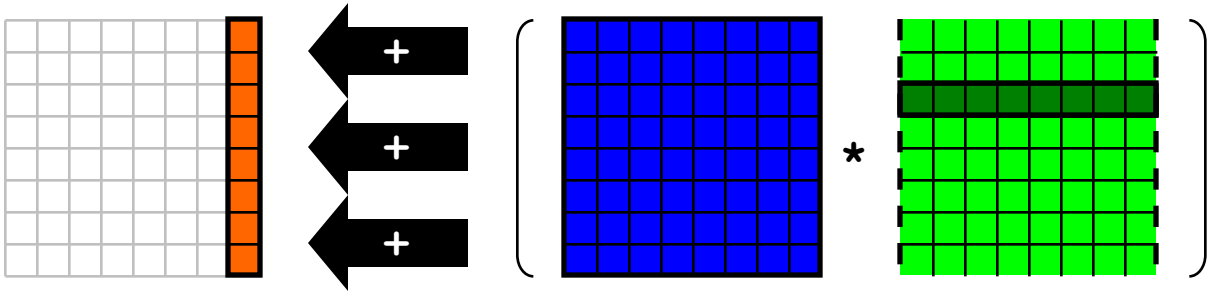
```
region R = [1..n,1..n];
       Row = [*,1..n];
       Col = [1..n,n];

var A:[R] double;
    V:[Row] double;
    S:[Col] double;
```



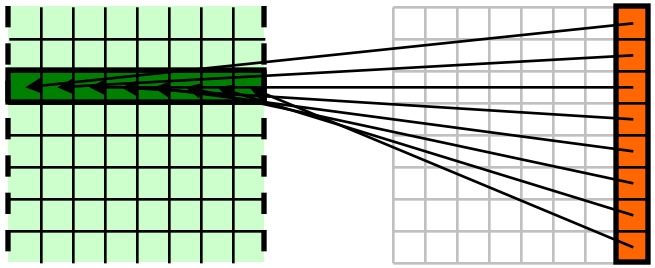
ZPL Mat-Vect Multiplication

computation:

```
[Col] S := +<<[R] (A*V);
```



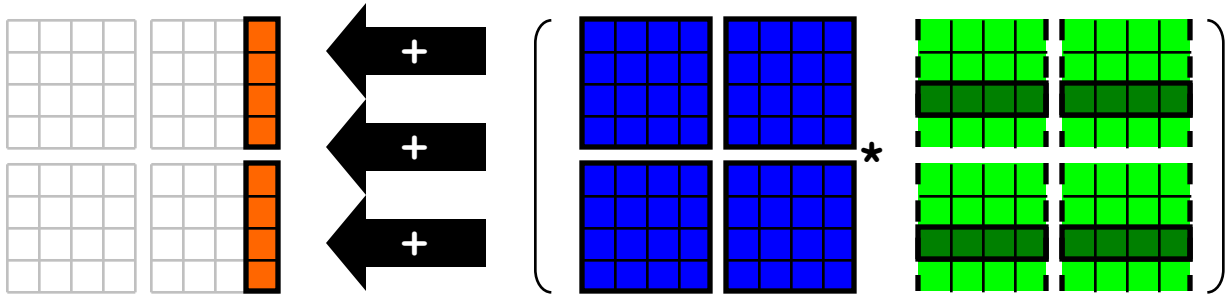

```
[Row] V := S#[n,Index1];
```



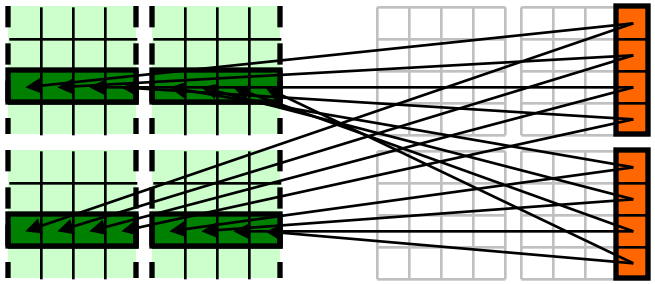
ZPL Mat-Vect Multiplication

computation (in parallel):

```
[Col] S := +<<[R] (A*V);
```




```
[Row] V := S#[n,Index1];
```



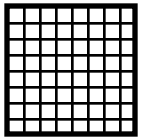
ZPL Mat-Vect Multiplication

Dense Matrix-Vector Multiplication:

```
region R = [1..n, 1..n];
       Row = [*, 1..n];
       Col = [1..n, n];


var A:[R] double;
    V:[Row] double;
    S:[Col] double;

[Col] S := +<<[R] (A*V);
```

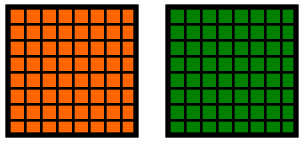
Sparse Regions and Arrays

```
region Rd = [1..n,1..n];
```

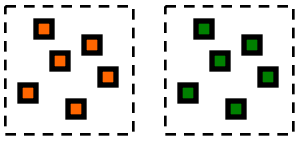


```
region Rs = Rd where <pattern>;
```

```
var Ad, Bd: [Rd] double;
```



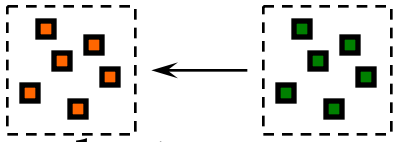
```
    As, Bs: [Rs] double;
```



Sparse Assignments

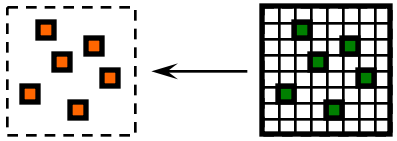
- Sparse assignment

```
[Rs] As := Bs;
```

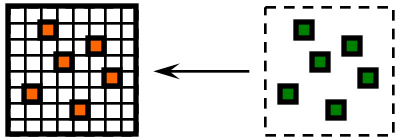


- Sparse assignment with dense array

```
[Rs] As := Bd;
```

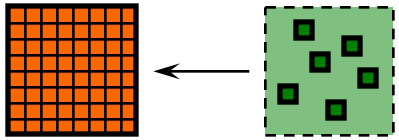


```
[Rs] Ad := Bs;
```



- Dense read of sparse array

```
[Rd] Ad := Bs;
```



- Dense assignment of sparse array – *illegal*

~~`[Rd] As := Bd;`~~

ZPL Mat-Vect Multiplication

Dense Matrix-Vector Multiplication:

```
region R = [1..n, 1..n];
       Row = [*, 1..n];
       Col = [1..n, n];


var A:[R] double;
    V:[Row] double;
    S:[Col] double;

[Col] S := +<<[R] (A*V);
```

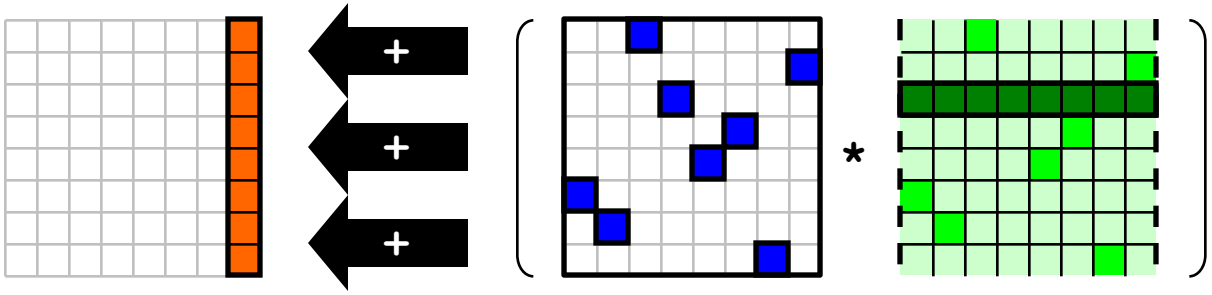
Sparse Matrix-Vector Multiplication:

```
region R = [1..n, 1..n];
       Row = [*, 1..n];
       Col = [1..n, n];
       Rs = R where …;

var A:[Rs] double;
    V:[Row] double;
    S:[Col] double;

[Col] S := +<<[Rs] (A*V);
```

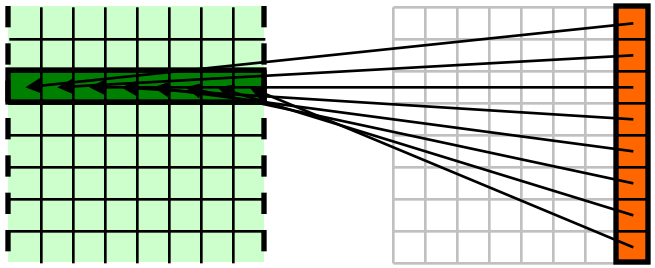
ZPL Mat-Vect Multiplication

Sparse Matrix-Vector Multiplication:

```
[Col] S := +<<[Rs] (A*V);
```




```
[Row] V := S#[n,Index1];
```



Sparse Arrays in ZPL

sparse array...

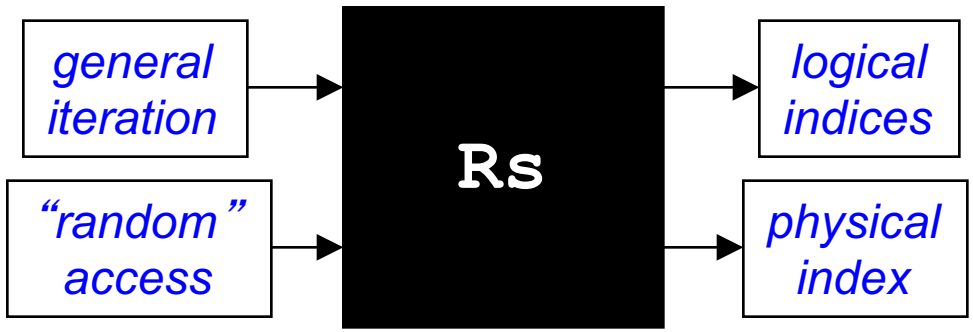
A:

0	0	0	0	0	0
0	3	0	4	0	9
0	0	0	0	0	0
0	0	5	0	2	0
0	1	0	0	0	0
0	0	0	7	0	0

= 1D dense array + sparse region

A:

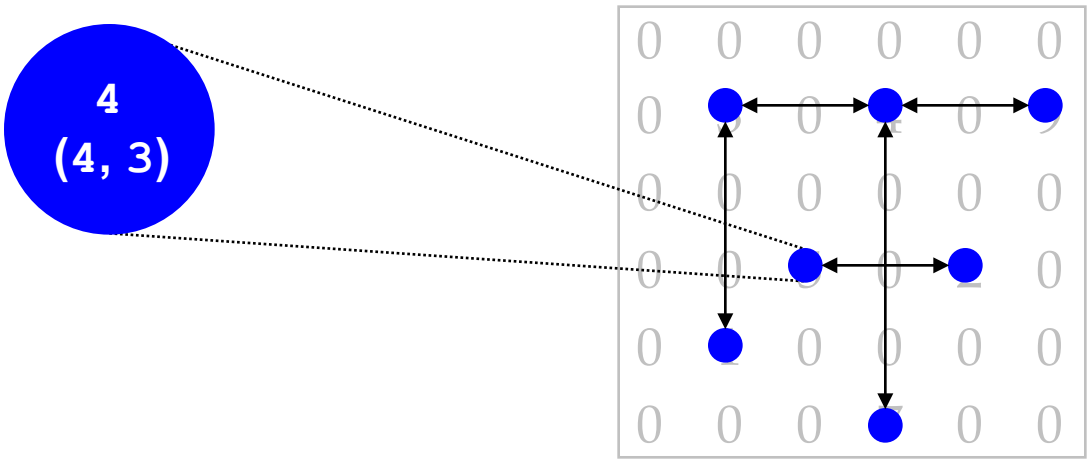
0	3	4	9	5	2	1	7



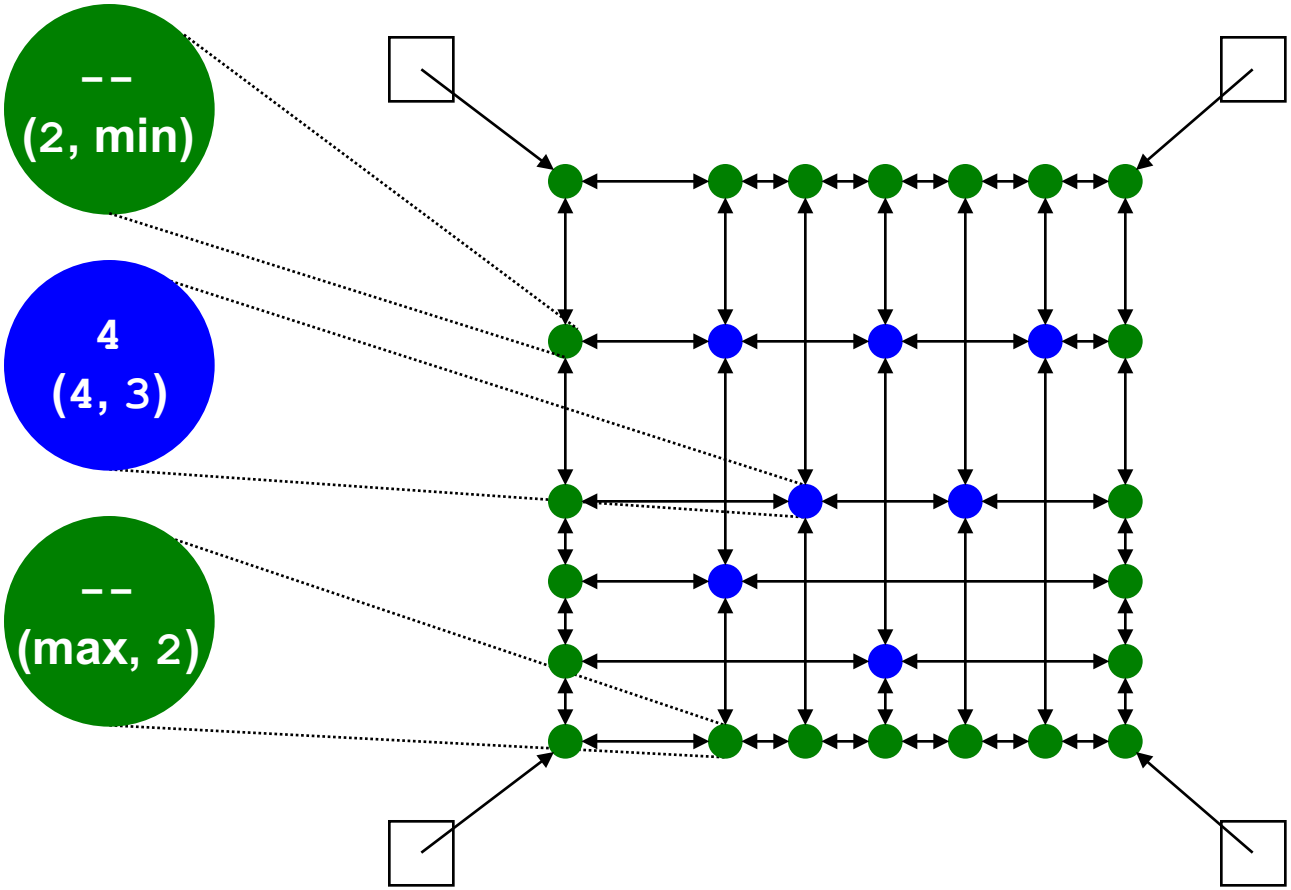
General Sparse Region Format

0	0	0	0	0	0
0	3	0	4	0	9
0	0	0	0	0	0
0	0	5	0	2	0
0	1	0	0	0	0
0	0	0	7	0	0

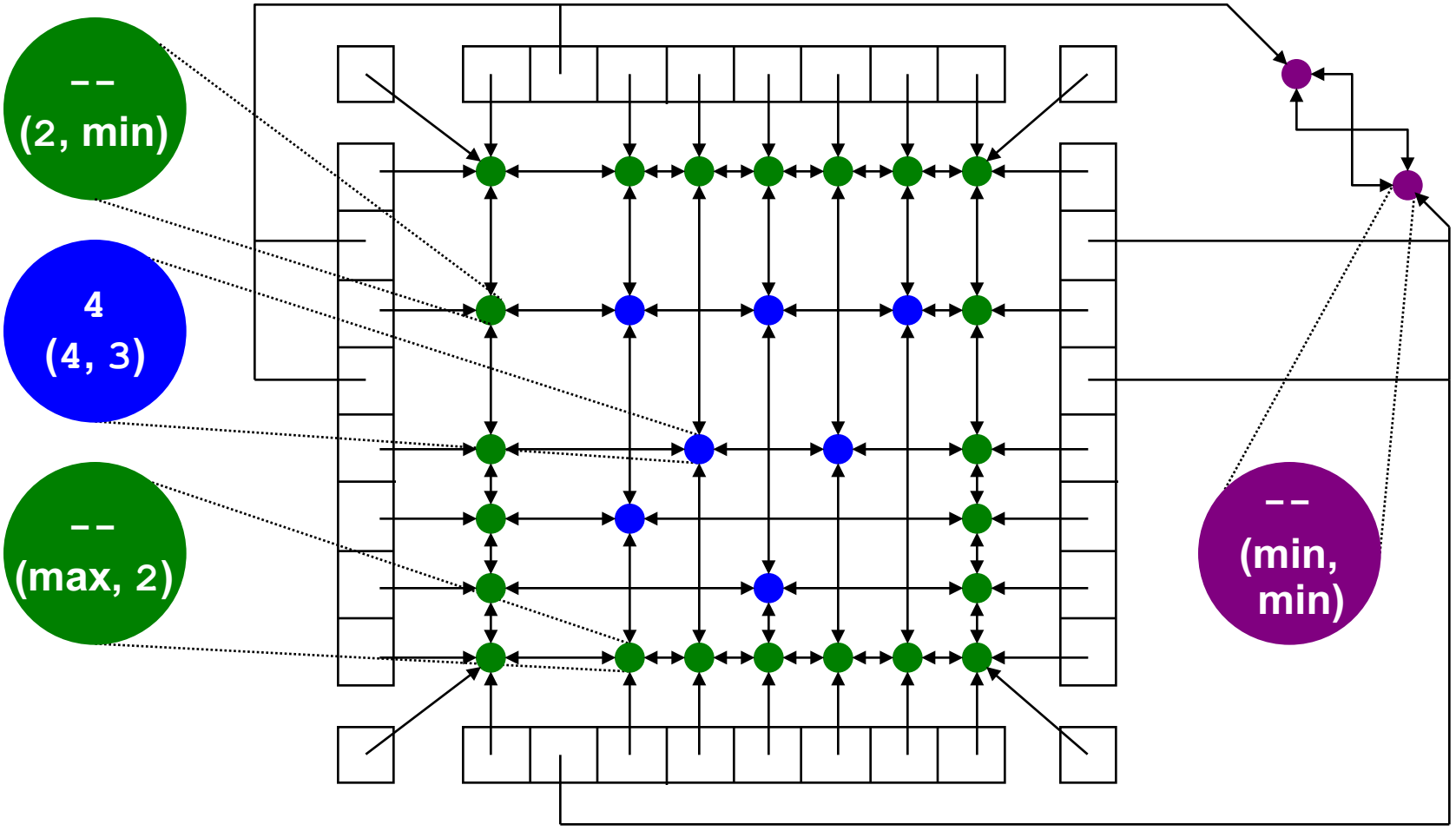
General Sparse Region Format



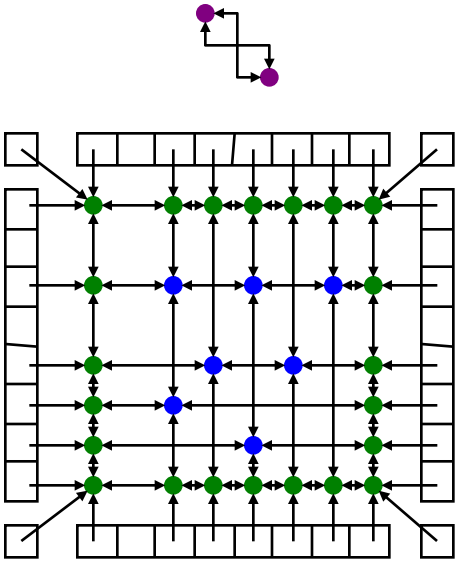
General Sparse Region Format

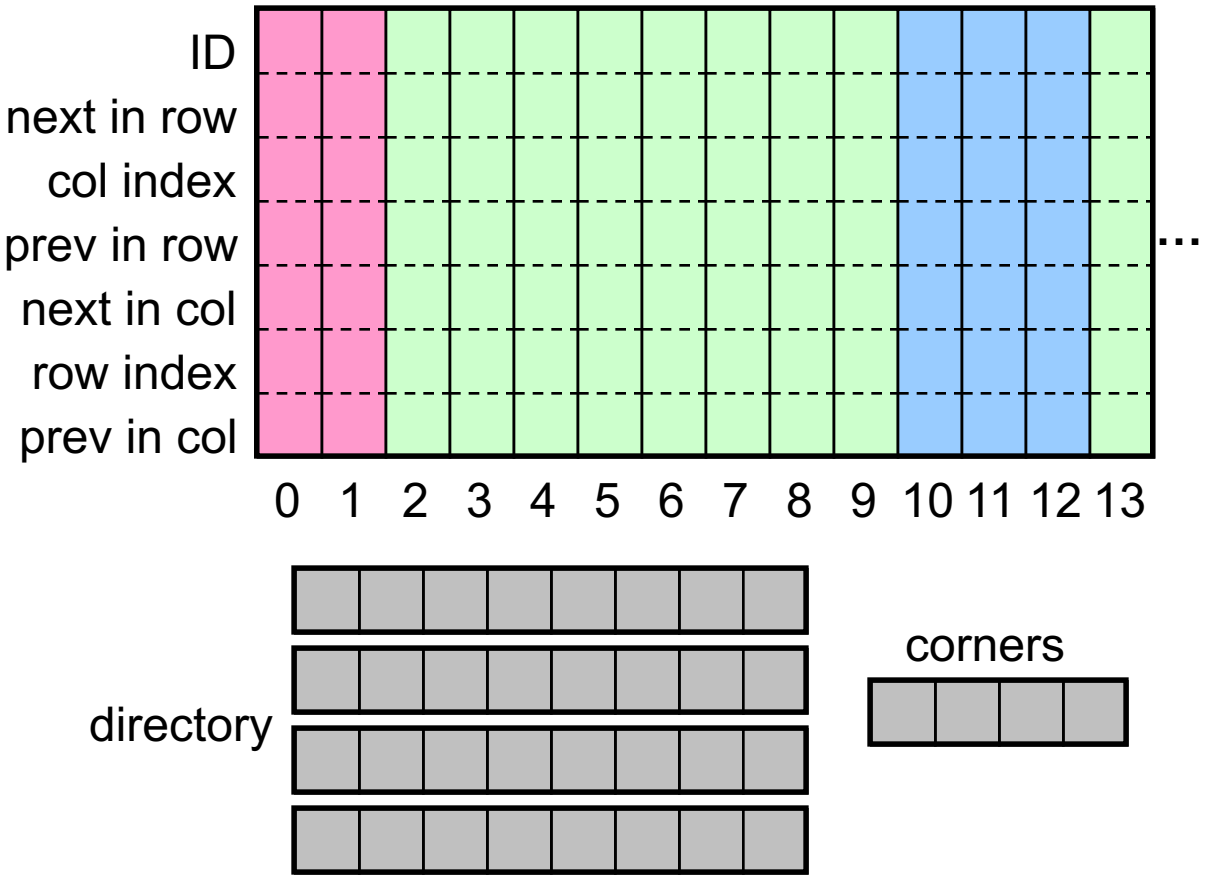


General Sparse Region Format

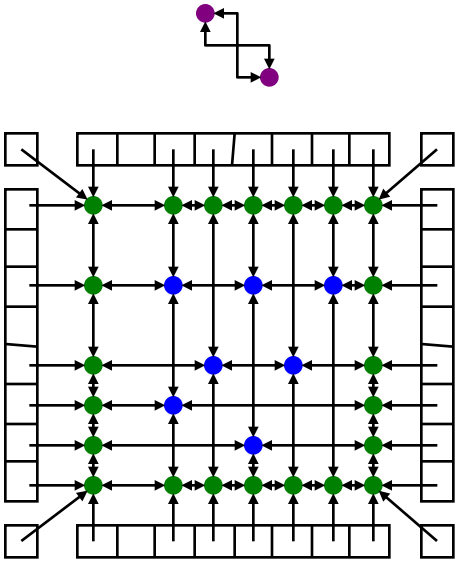


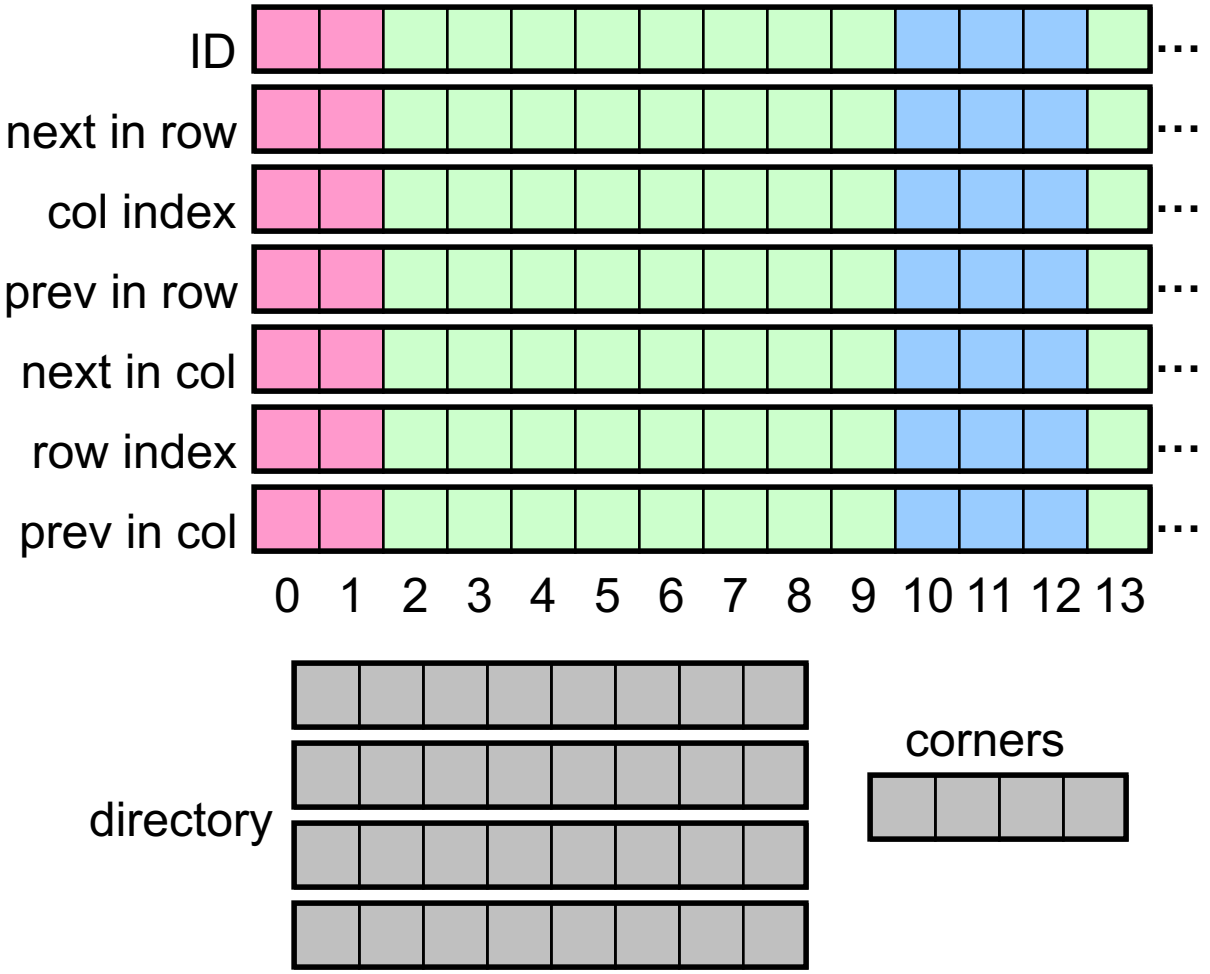
Sparse Format: Array of Records



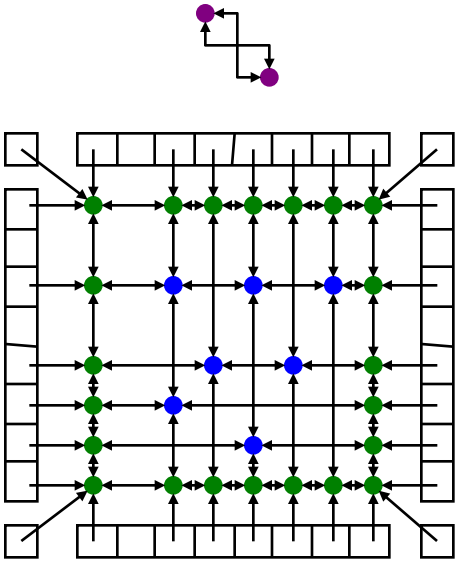


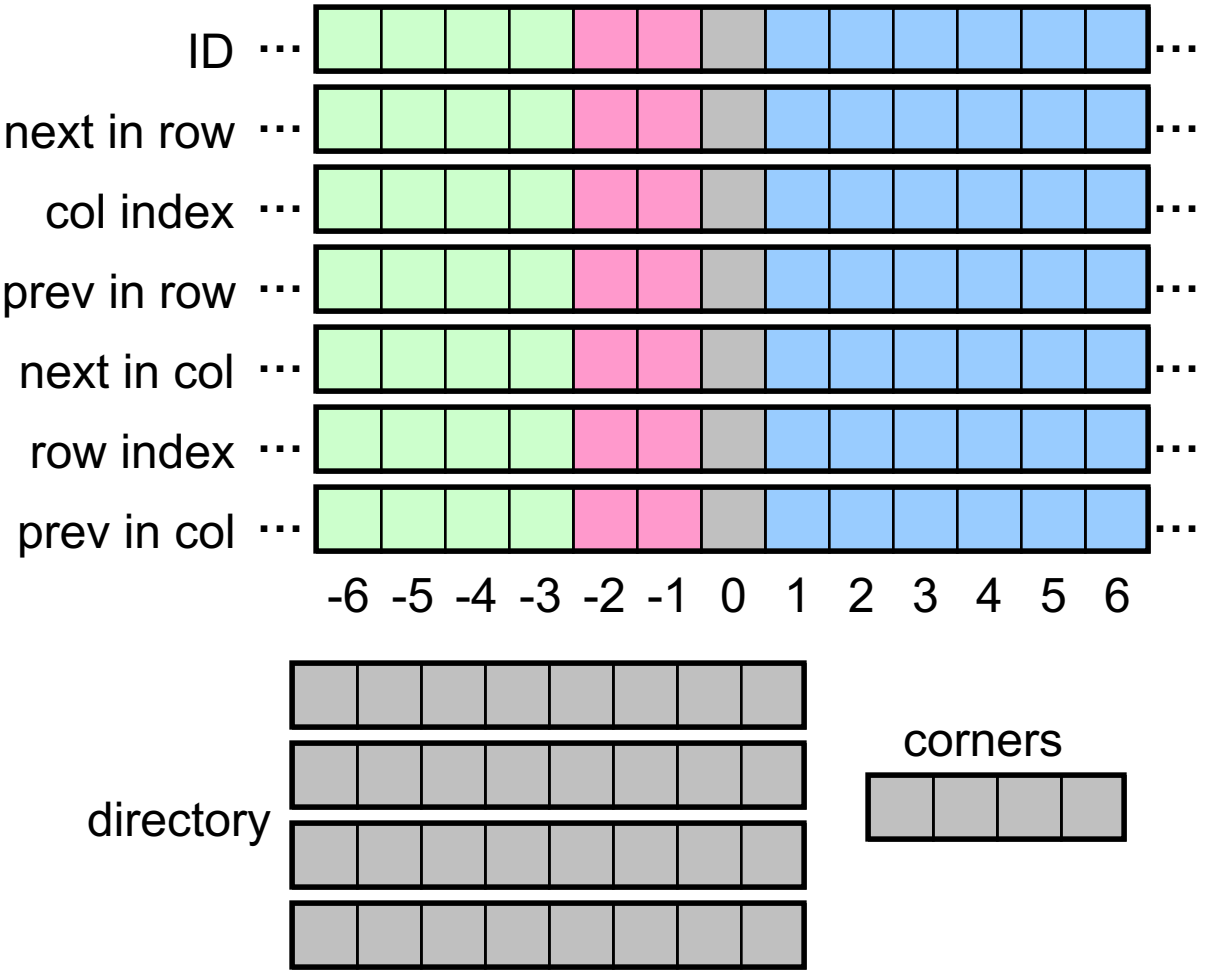
Sparse Format: Record of Arrays



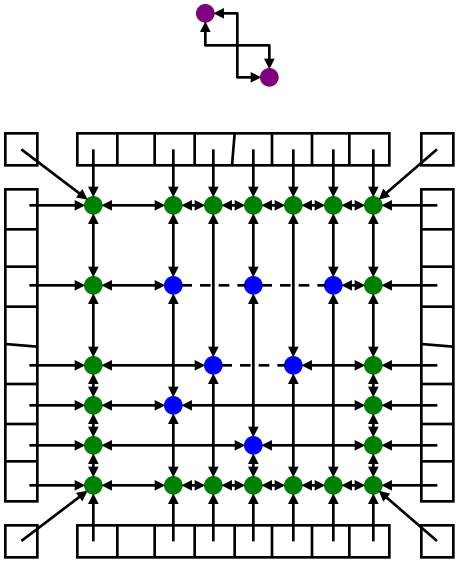


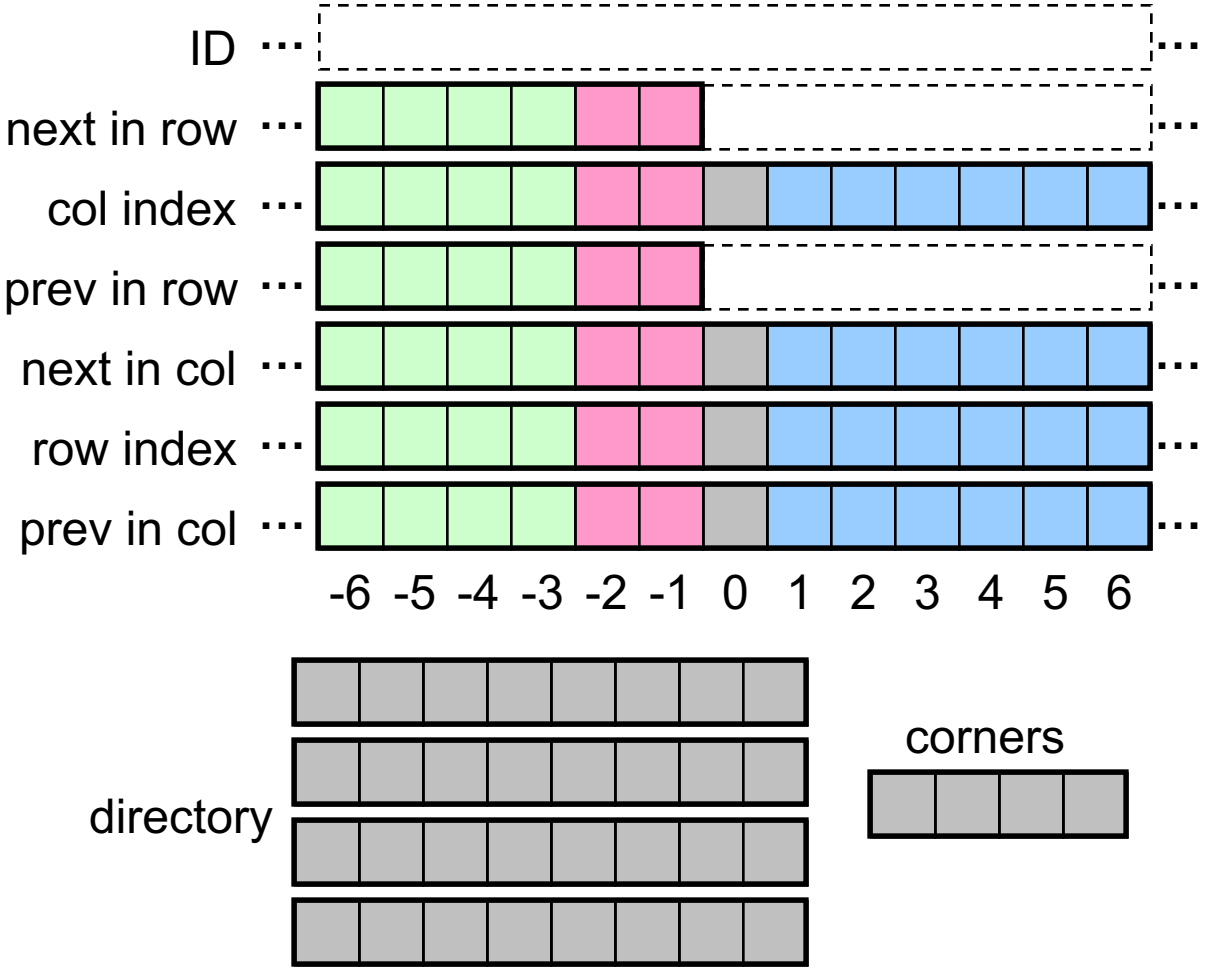
Sparse Format: Reordered



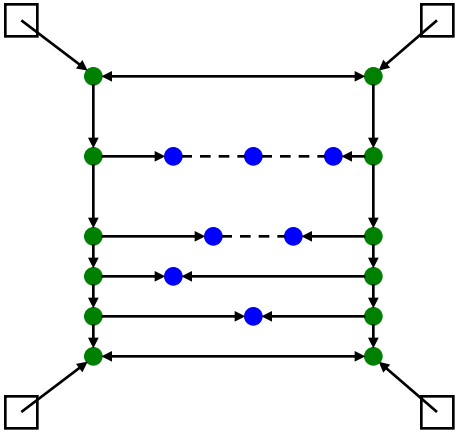


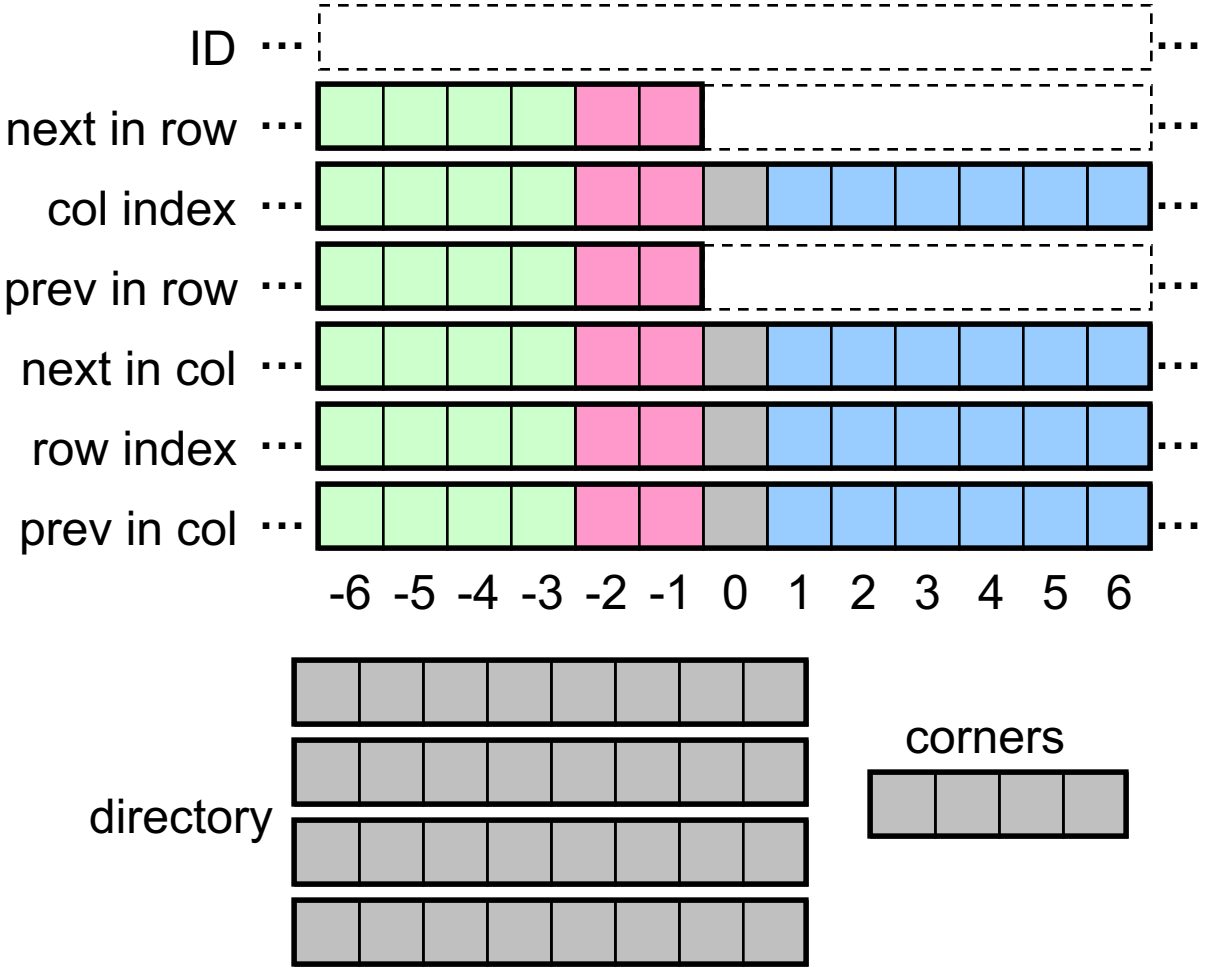
Sparse Format: Optimized



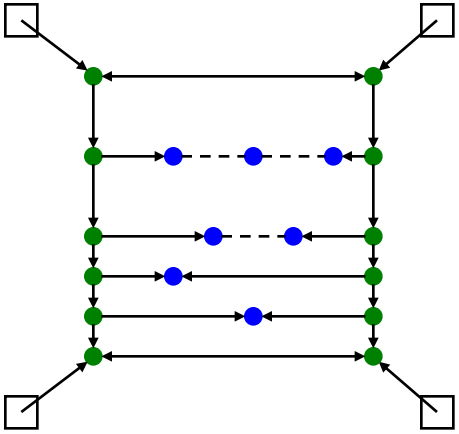


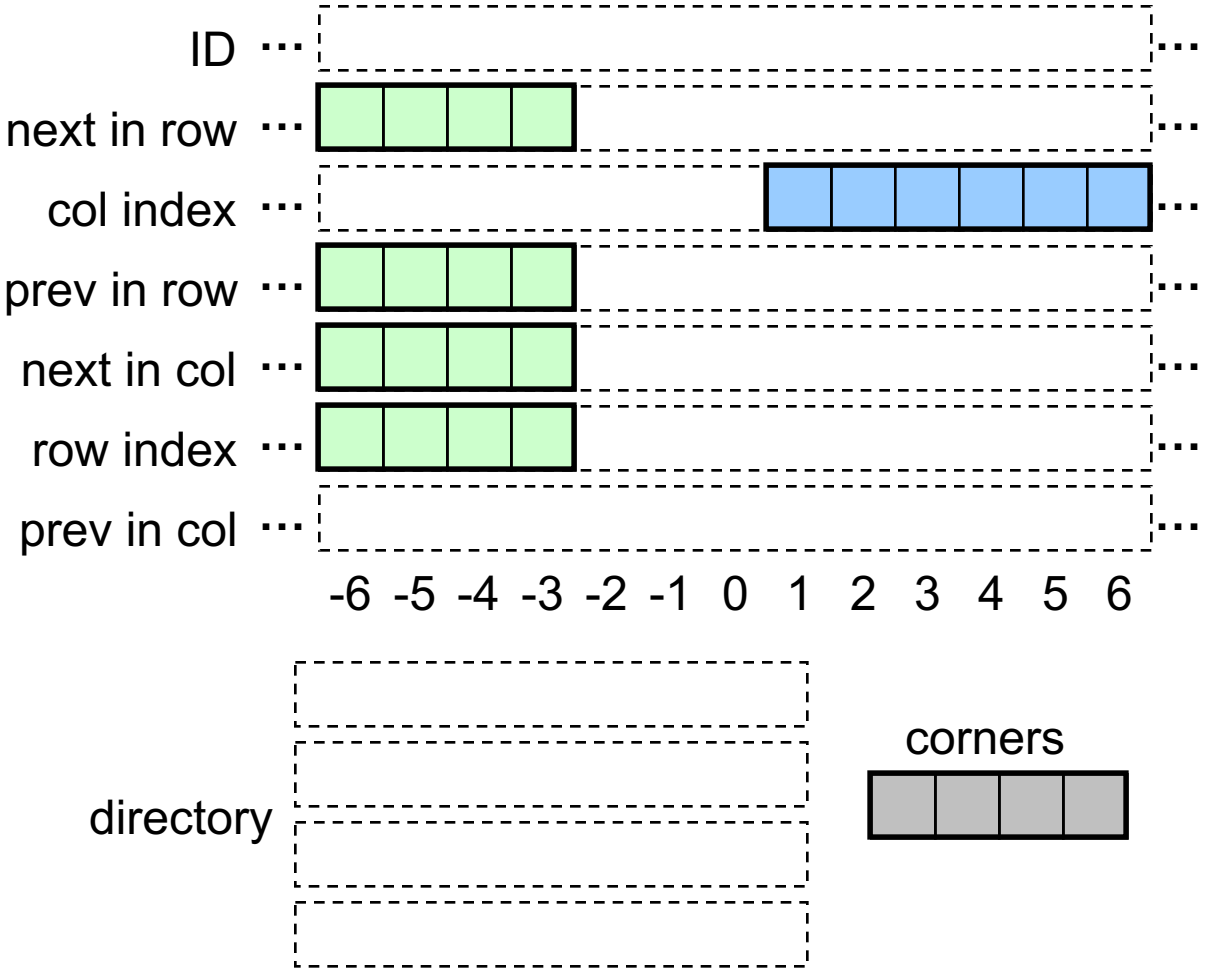
Sparse Format: True Requirements



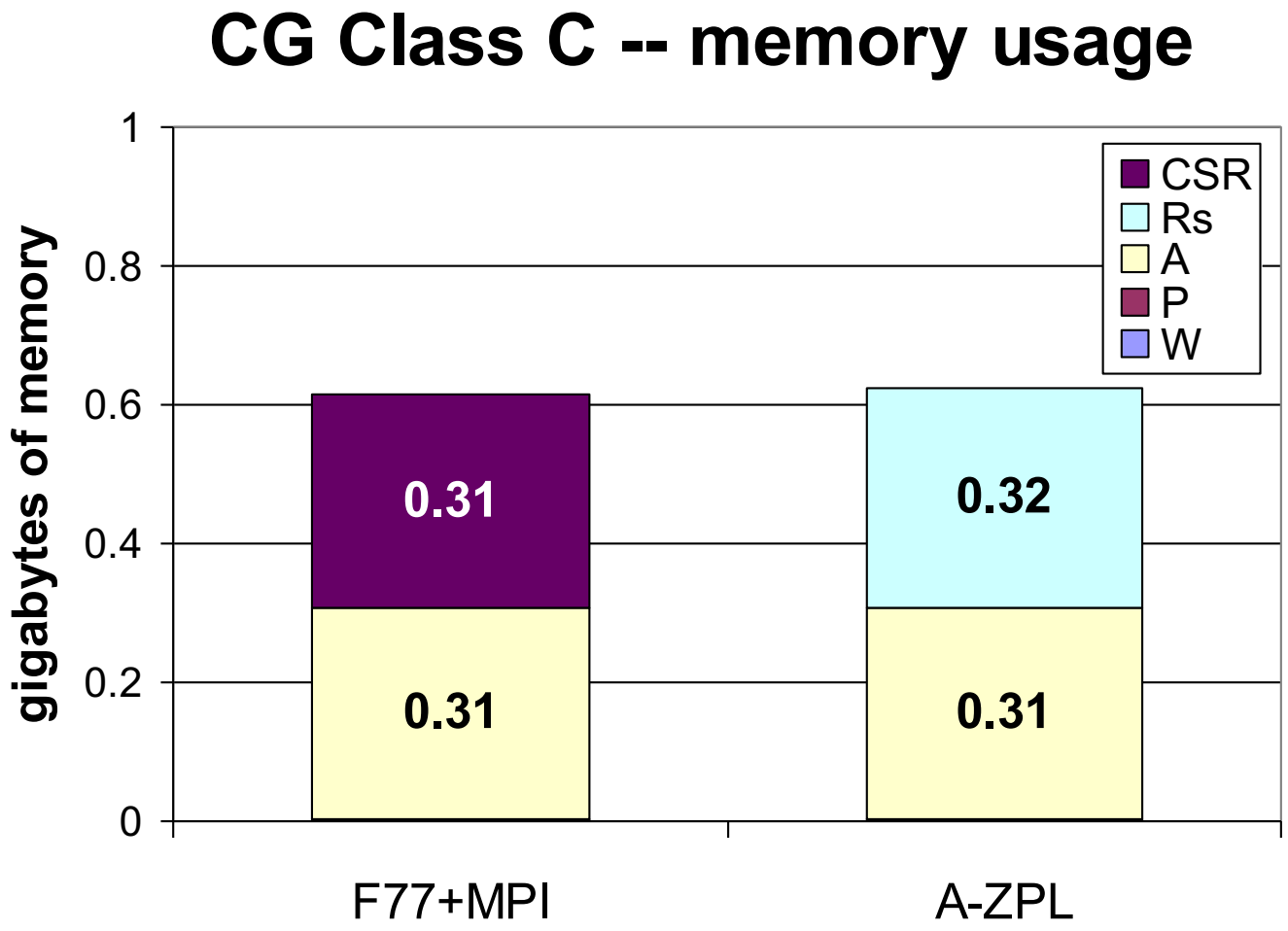


Sparse Format: Fully Optimized

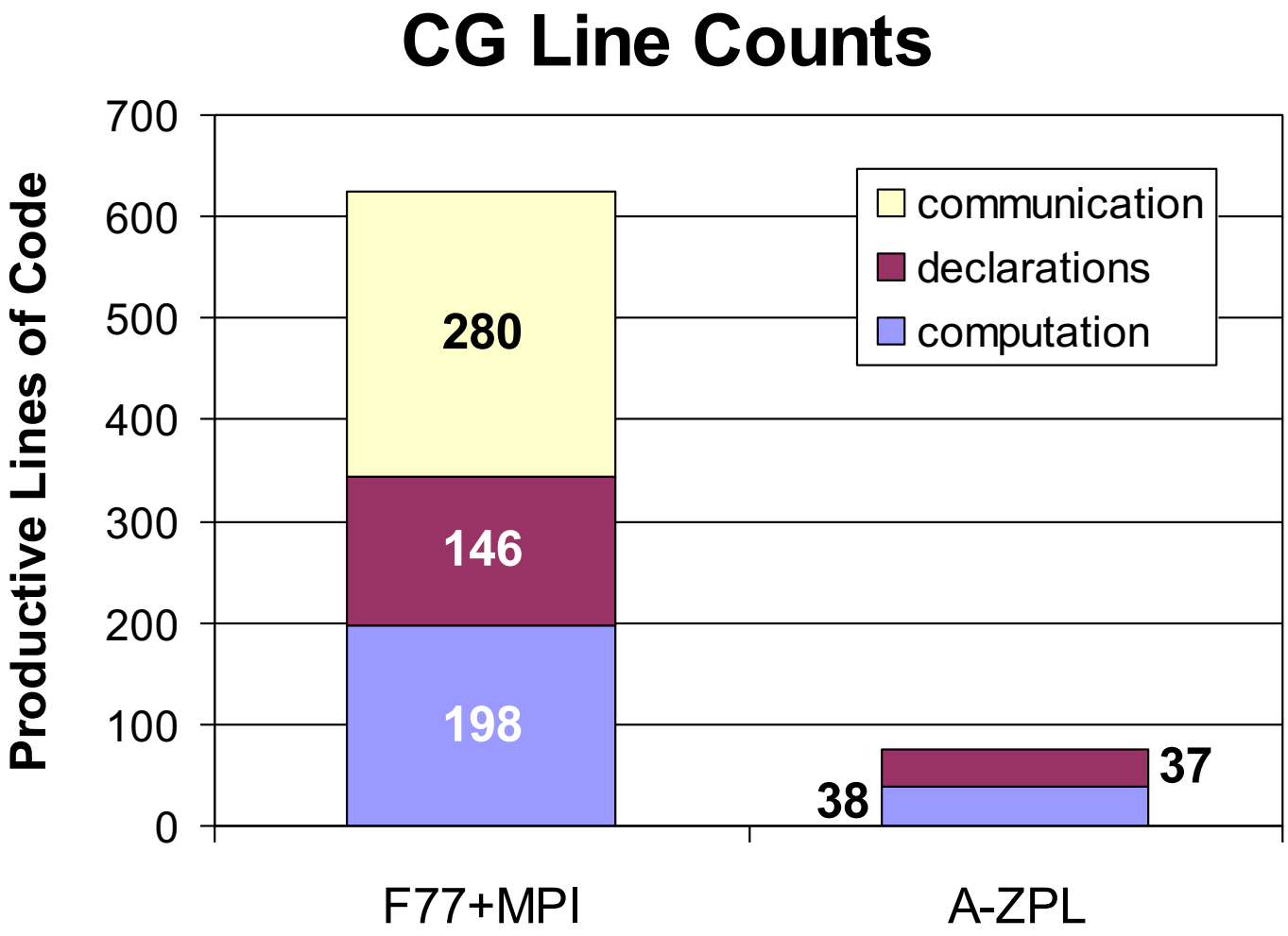


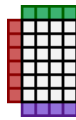


CG Memory Requirements

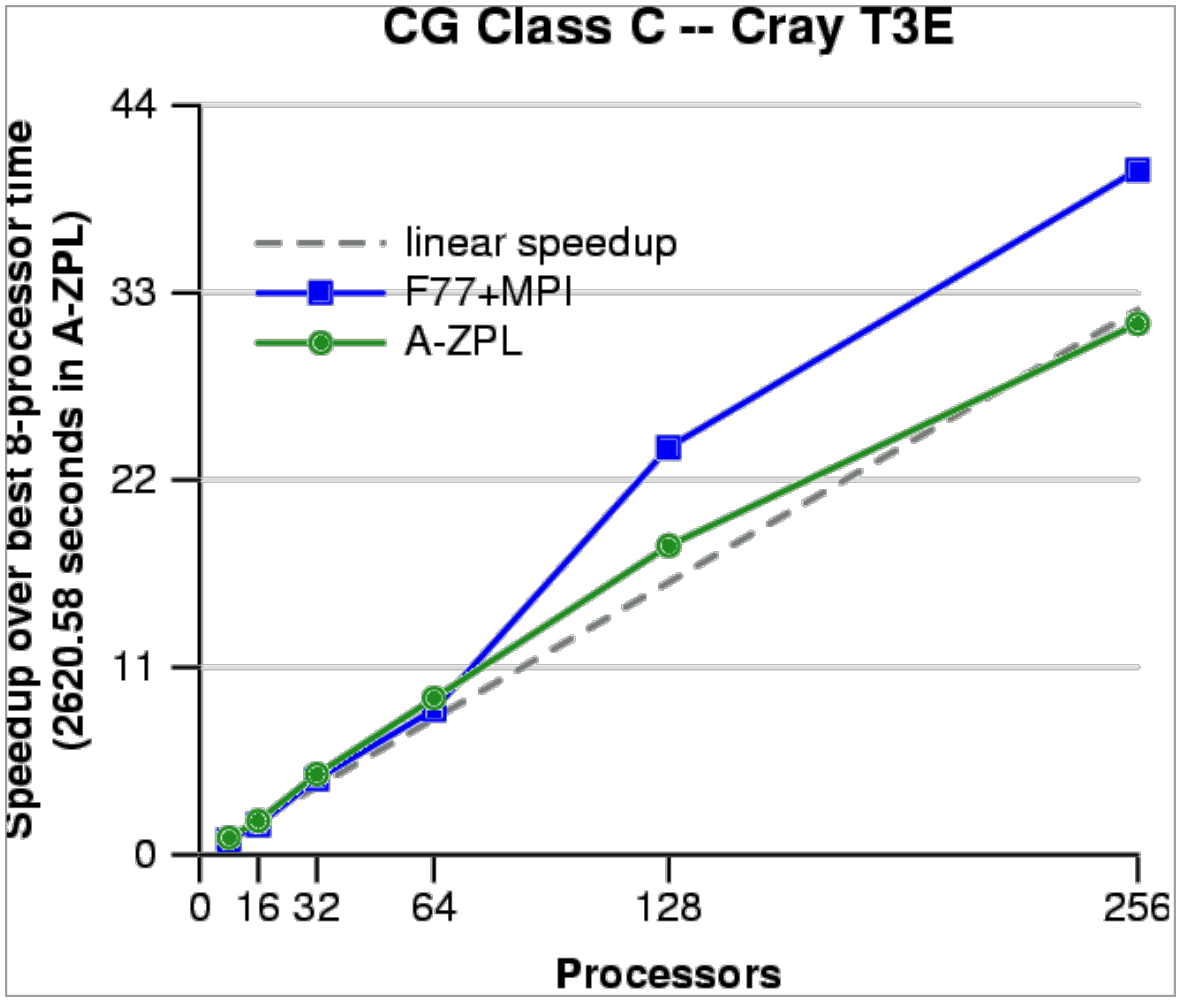


CG Code Size


CG Line Counts
Productive Lines of Code
700
600
500
400
300
200
100
0
communication
declarations
computation
280
146
198
38
37
F77+MPI
A-ZPL



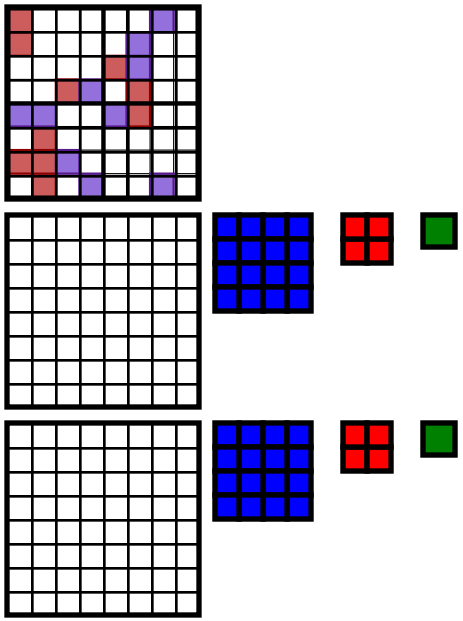
NAS CG Speedup: Cray T3E



though ZPL's sparsity pattern performed well at small node counts and scaled well, it couldn't beat hand-coded CSR at scale…

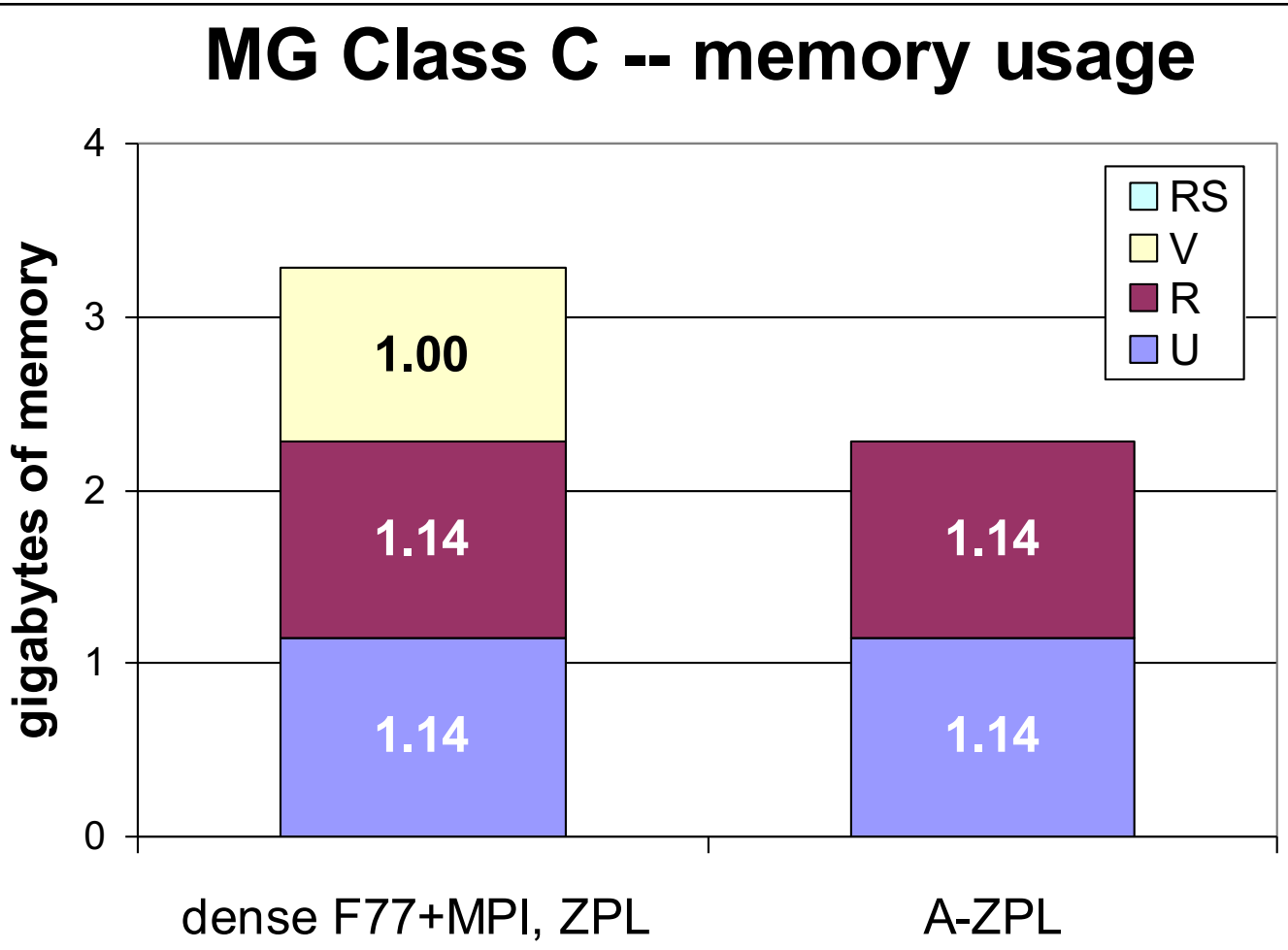
Sparse Arrays in MG

- MG's input array, *V*, has 20 non-zeroes regardless of problem size

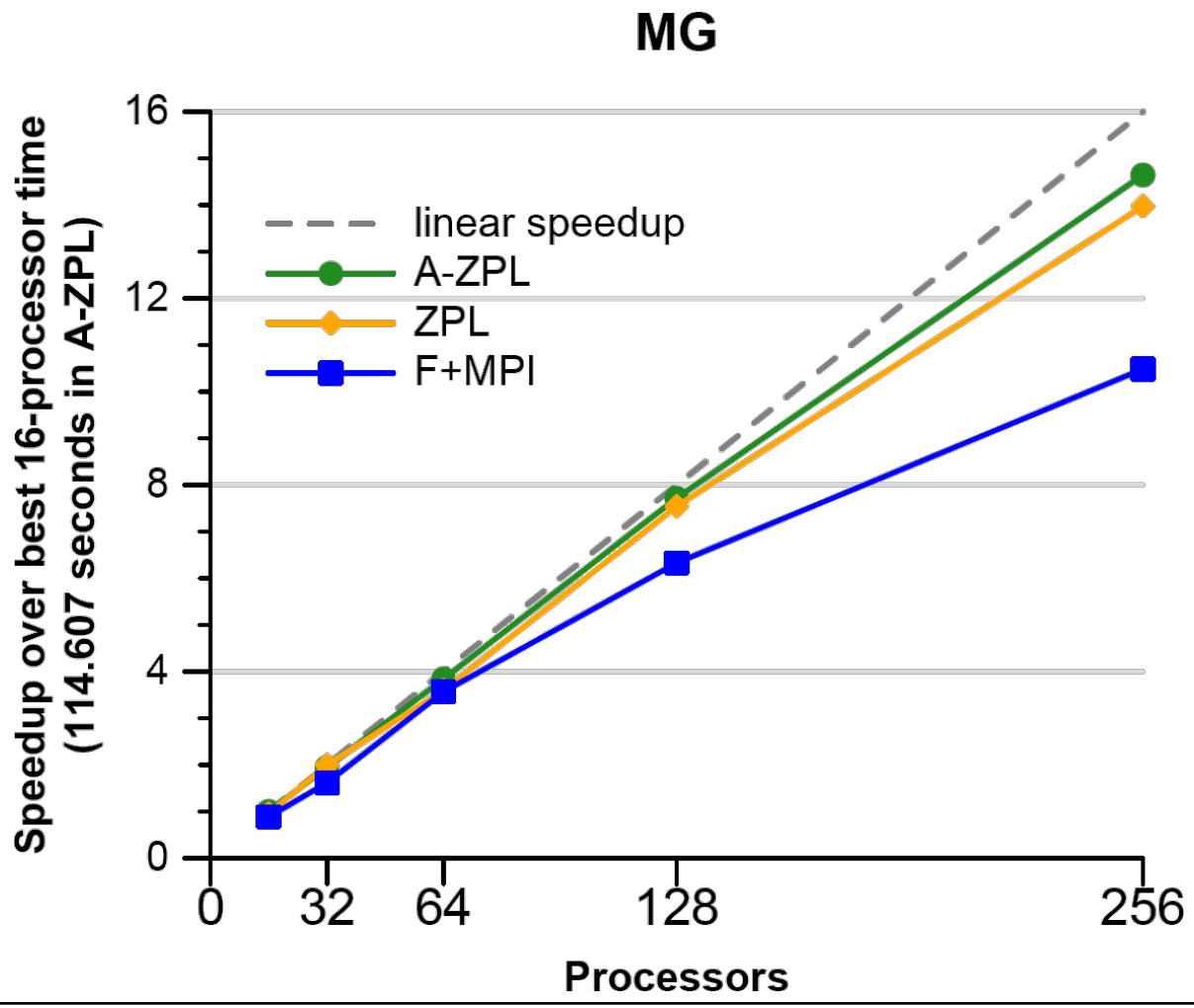


- Computes resid against *V* twice per iteration
- Wasted time and space

MG Memory Usage



NAS MG Speedup: Cray T3E



Another victory for arrays!

- Sparse computation expressed like dense
- Compact, clean, comprehensible code again
  (not seen in this presentation)
- Code expresses algorithm, not implementation
  – compiler optimizes implementation based on need

ZPL Summary

ZPL's Successes

- **First-class concept for representing index sets**
  - makes clouds of scalars in array declarations and loops concrete
  - supports global-view of data and control; improved productivity
  - useful abstraction for user and compiler

  The Design and Implementation of a Region-Based Parallel Language. Bradford L. Chamberlain. PhD thesis, University of Washington, November 2001

- **Semantics constraining alignment of interacting arrays**
  - communication requirements visible to user and compiler in syntax

  ZPL's WYSIWYG performance model. Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. In *Proceedings of the IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments*, 1998.

- **Implementation-neutral expression of communication**
  - supports implementation on each architecture using best paradigm

  A compiler abstraction for machine independent parallel communication generation. Bradford L. Chamberlain, Sung-Eun Choi, and Lawrence Snyder. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1997.

ZPL's Drawbacks

1) ZPL only supports one level of data parallelism

- No task parallelism / concurrent programming
- No nested parallelism (data or otherwise)
- ZPL's parallelism can never be very dynamic / unpredictable
  - single-threaded SPMD only

Takeaway:

Users want more general forms of parallelism (as do modern architectures and algorithms)

Challenge:

What would it take to meaningfully support general parallel programming with ZPL-like regions and arrays?

ZPL's Drawbacks

2) WYSIWYG depends on having two distinct array types:

i. parallel arrays, which you've been hearing about

ii. traditional arrays which support indexing, yet are serial / local only

```
var ThreeParArrs:     array [1..3] of [R] double;
var ParArrofTriples: [R] array [1..3] of double;

[R] TheeParArrs[2] := ParArrofTriples[2];
```

ZPL's Drawbacks

2) WYSIWYG depends on having two distinct array types:

i. parallel arrays, which you've been hearing about

ii. traditional arrays which support indexing, yet are serial / local only

```
var ThreeParArrs:     array [1..3] of [R] double;
var ParArrofTriples: [R] array [1..3] of double;
```

- this breaks code reuse and results in unfortunate code clones
  - "I need to rewrite this operation twice, once for each array type"
    - (or 2^k times for *k* array arguments?)

- there's a real tension here: WYSIWYG would be lost if…

  ...parallel arrays supported indexing

  ...serial arrays supported array operators

ZPL's Drawbacks

2) WYSIWYG depends on having two distinct array types:

i. parallel arrays, which you've been hearing about

ii. traditional arrays which support indexing, yet are serial / local only

```
var ThreeParArrs:     array [1..3] of [R] double;
var ParArrofTriples: [R] array [1..3] of double;
```

Takeaway:

Having distinct types for parallel vs. serial arrays is regrettable

Challenge:

How can ZPL's region/array benefits best be carried forward?

ZPL's Drawbacks

3) ZPL only supports Block-distributed arrays

- real algorithms may want other distributions: block-cyclic, MRB, …
- distribution baked into the compiler, not extensible
- memory layouts are also hard-coded
  - RMO for dense arrays
  - special ZPL format for sparse arrays

Takeaway:

While block is an important common case, it's not a panacea. Inability to get CSR sparse arrays hurt CG perforamnce.

Challenge:

How to support more general distributions and layouts without giving up performance?

ZPL's Drawbacks

4) Dated in several respects

- syntactically follows Modula rather than C (say)
- no support for OOP
- ...

- **a good example of academic vs. practical language design**
  - for academic work, such things may not matter (they didn't for ZPL)
  - for practical adoption, they can be show-stoppers (they were for ZPL)

Takeaway:

Users value attractive, modern, feature-rich language design.

Challenge:

How to keep the kitchen sink from dragging down a sleek design.

Main ZPL Sacrifice in moving to Chapel

Q: Any guesses?

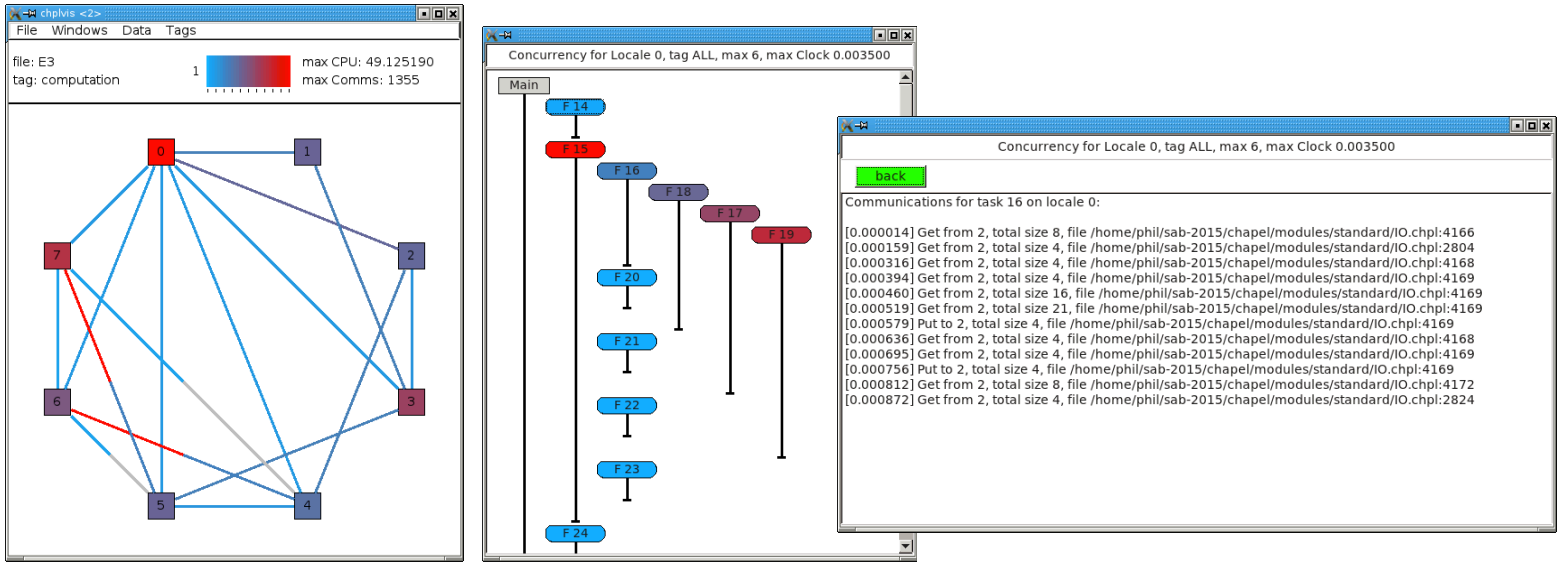
A: Chapel has no WYSIWYG model

- Chapel has one type of array for simplicity, code re-use
  - parallelism is a property of operators / loops, not arrays
  - distributed arrays support indexing
- Interacting arrays need not be distributed identically

```
var ThreeParArrs:    [1..3] [R] real;
var ParArrofTriples: [R] [1..3] real;

// ambiguous:
[1..3][i, 1..n] ThreeParArrs = ParArrofTriples;
// unless we use a tag to distinguish:
[i in 1..3] [jk in {i, 1..n}]
  ThreeParArrs[i][jk] = ParArrOfTriples[jk][i];
// which is essentially an index…
```

Life without WYSIWYG

- **Yet, Chapel users can still reason about communication**
  - semantic model is explicit about where data is placed / tasks execute
  - execution-time queries support reasoning about locality
    - `here` – where is this task running?
    - `x.locale` – where is x stored?
  - new **chplvis** tool supports visualization of communication
    - developed by Phil Nelson, WWU



ZPL vs. Chapel: Other Differences

ZPL	Chapel
data parallel only	task-, nested parallelism
built-in data distributions	user-defined data distributions
designed in an evolutionary manner "what can we do well today, tomorrow?"	designed in a blue-sky manner: "what would an ideal parallel language support?"
academic focus: pick one problem to address (array-based data parallelism)	market focus: what does a general, broad-market language require?
single-threaded, distributed-memory execution model	multithreaded, shared address space execution model
create something that matches today's architectures	create something productive; unproductive architectures may suffer

CRAY

Chapel

What is Chapel?

Chapel: An emerging parallel programming language

- portable
- open-source
- a collaborative effort
- a work-in-progress

Goals:

- Support general parallel programming
  - "any parallel algorithm on any parallel hardware"
- Make parallel programming far more productive
- Designed for practical adoption, not publications

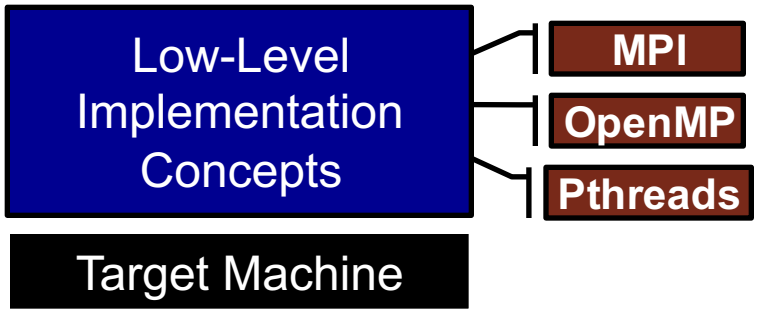
The Chapel Team at Cray (May 2016)

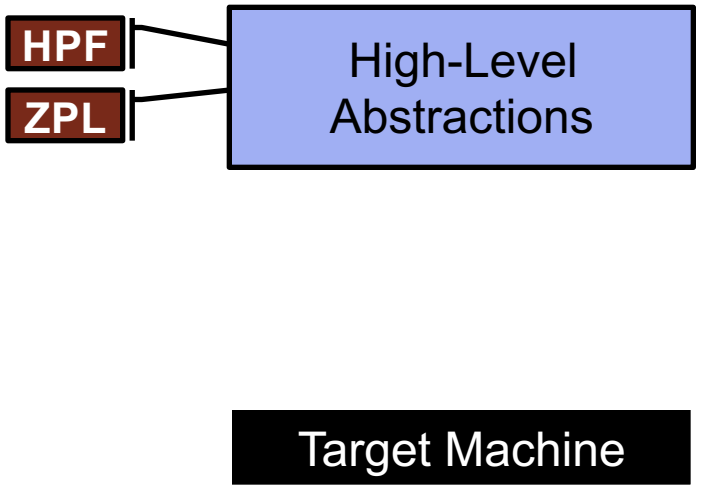
Chapel Community R&D Efforts

(and several others, some of whom you will hear from today…)

http://chapel.cray.com/collaborations.html

Chapel's Multiresolution Design: Motivation





"Why is everything so tedious/difficult?"

"Why don't my programs trivially port to new systems?"

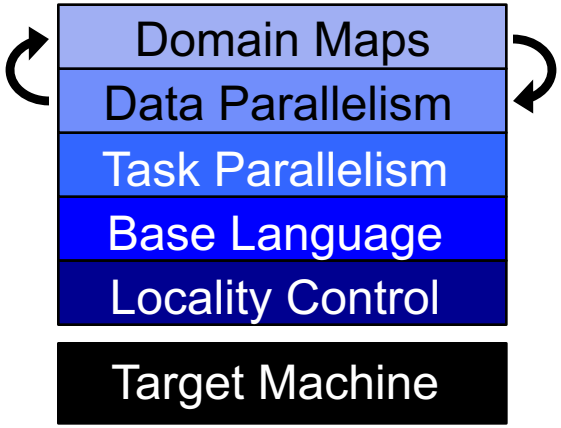
"Why don't I have more control?"

Chapel's Multiresolution Design

Multiresolution Design: **Support multiple tiers of features**

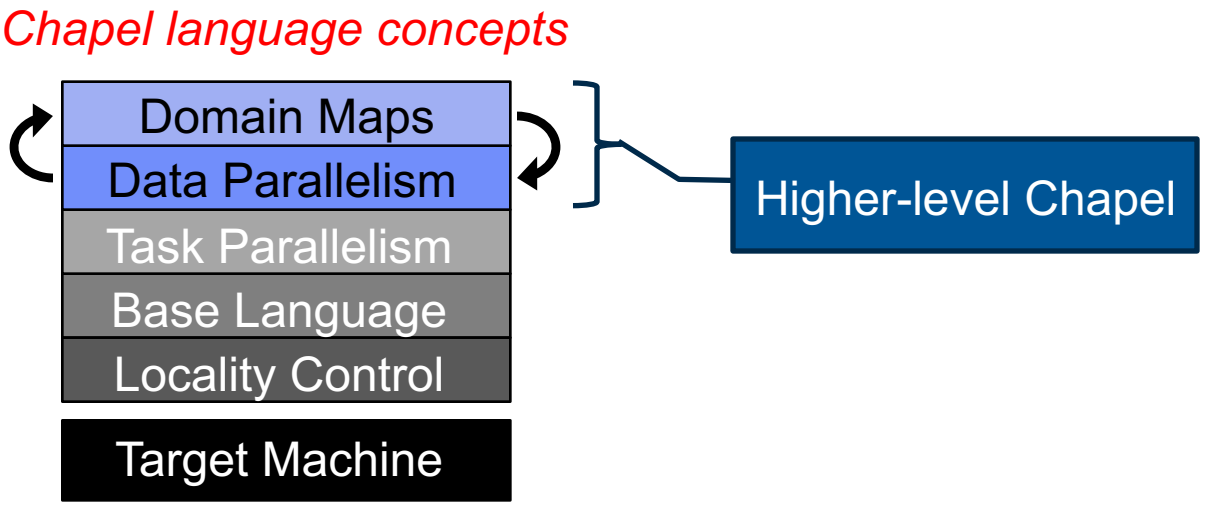
- higher levels for programmability, productivity
- lower levels for greater degrees of control

Chapel language concepts



- build the higher-level concepts in terms of the lower
- permit the user to intermix layers arbitrarily

Higher-Level Features



Data Parallelism, by example

dataParallel.chpl

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl -o dataParallel
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

Data Parallelism, by example

Domains (Chapel's *regions*)

dataParallel.chpl

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl -o dataParallel
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

Data Parallelism, by example

Arrays

dataParallel.chpl

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl -o dataParallel
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

Data Parallelism, by example

Data-Parallel Forall Loops

dataParallel.chpl

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl -o dataParallel
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

Distributed Data Parallelism, by example

Domain Maps
(Map Data Parallelism to the System)

dataParallel.chpl

```
use CyclicDist;
config const n = 1000;
var D = {1..n, 1..n}
        dmapped Cyclic(startIdx = (1,1));
var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl -o dataParallel
prompt> ./dataParallel --n=5 --numLocales=4
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

Distributed Data Parallelism, by example

dataParallel.chpl

```
use CyclicDist;
config const n = 1000;
var D = {1..n, 1..n}
        dmapped Cyclic(startIdx = (1,1));
var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl -o dataParallel
prompt> ./dataParallel --n=5 --numLocales=4
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```

Chapel Data Parallel Operations

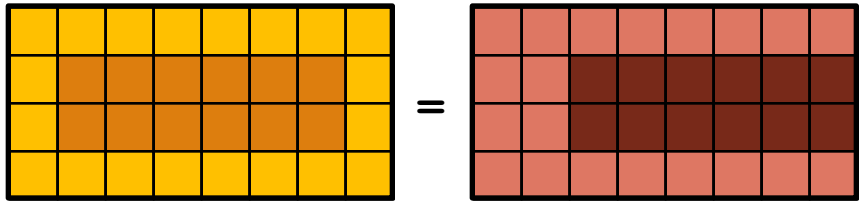
- **Data Parallel Iteration**

```
forall (i,j) in D do
  A[i,j] = i + j/10.0;
```

1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8
2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8
3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8
4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.8

- **Array Slicing; Domain Algebra**

```
A[InnerD] = B[InnerD+(0,1)];
```



- **Promotion of Scalar Functions and Operators**

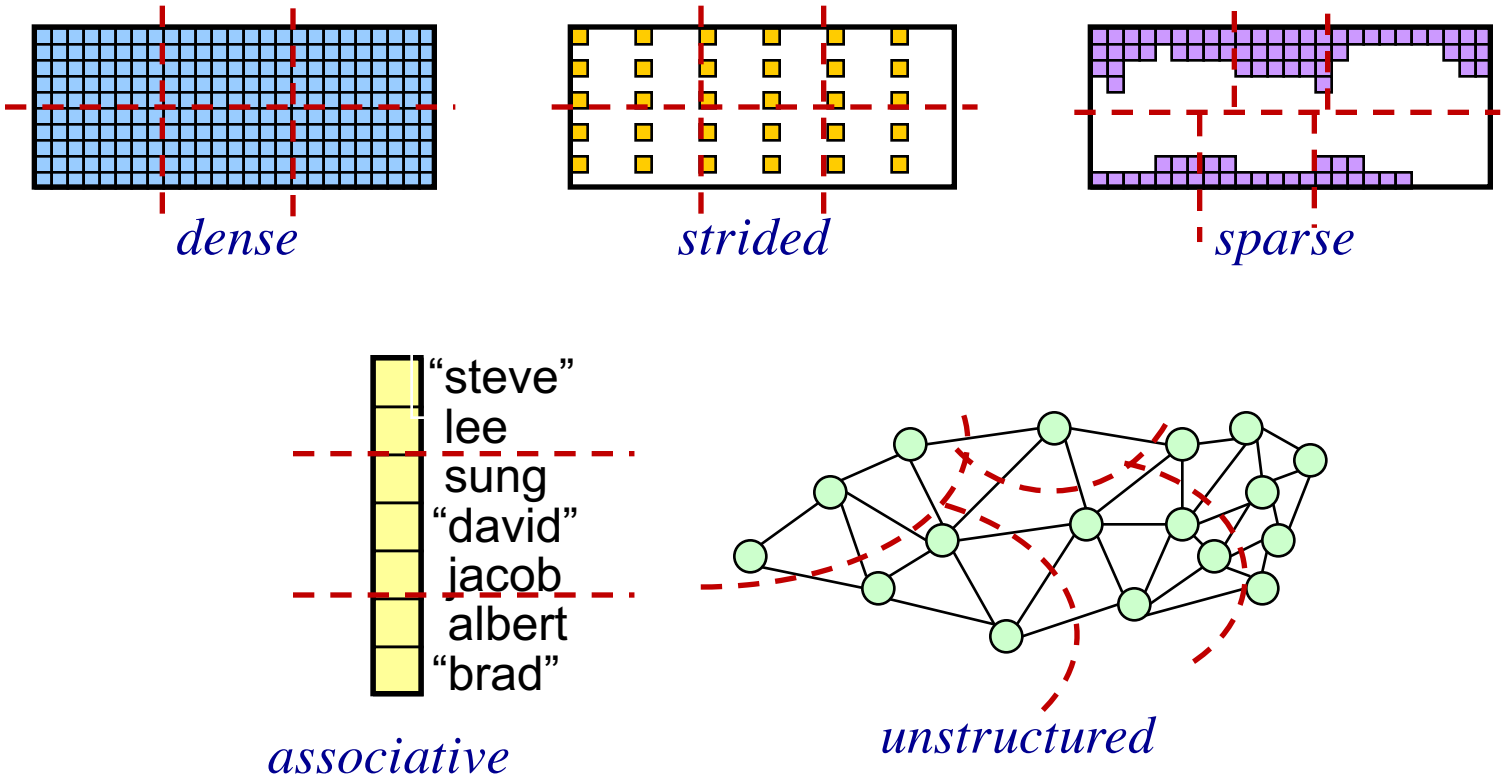
```
A = exp(B, C);
```

```
A = foo("hi", B, C);
```

```
A = B + alpha * C;
```

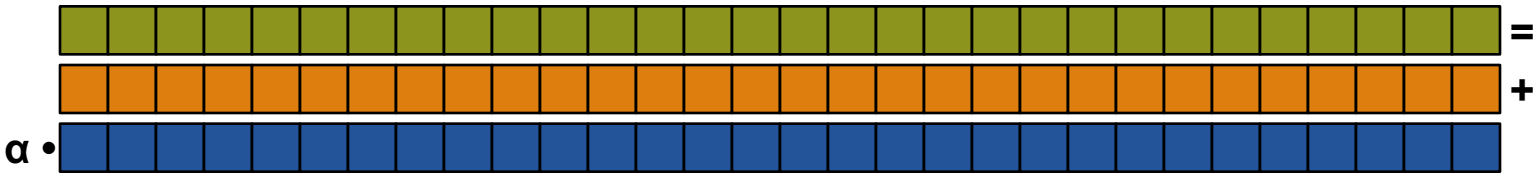
- **And many others: reductions, scans, reallocation, reshaping, remapping, set operations, aliasing, …**

Chapel has Many Types of Domains/Arrays



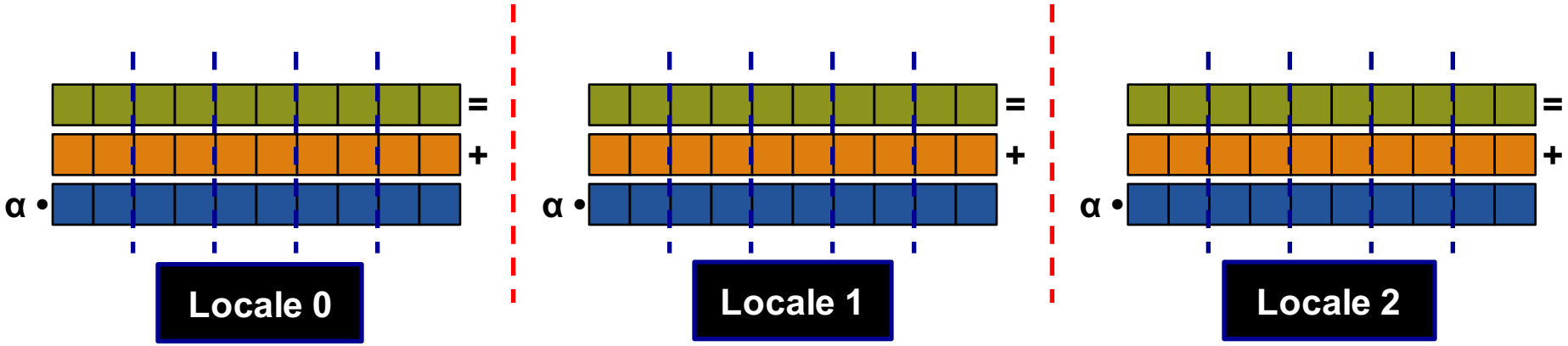
Domain Maps

Domain maps are "recipes" that instruct the compiler how to map the global view of a computation…




```
A = B + alpha * C;
```

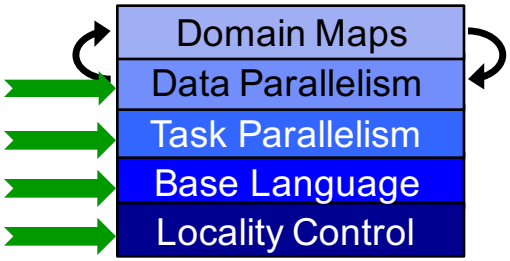
…to the target locales' memory and processors:



Chapel's Domain Map Philosophy

1. **Chapel provides a library of standard domain maps**
   - to support common array implementations effortlessly

2. **Expert users can write their own domain maps in Chapel**
   - to cope with any shortcomings in our standard library



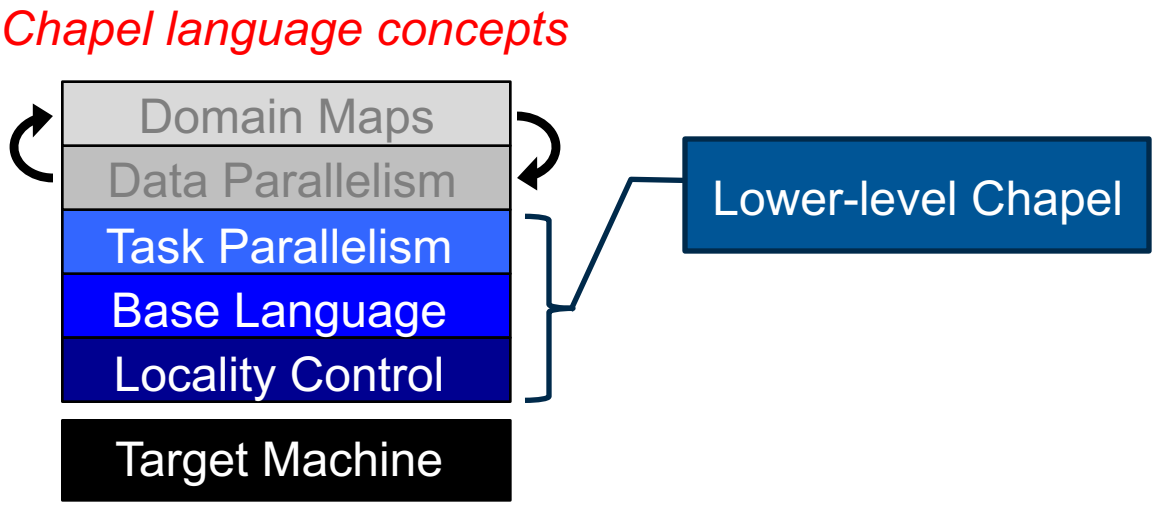
1. **Chapel's standard domain maps are written using the same end-user framework**
   - to avoid a performance cliff between "built-in" and user-defined cases

Impacts of Chapel's Domain Map Philosophy

- **Chapel supports both global- and local-view programming**
  - domain maps form a gateway between the two worldviews

- **Our approach put us at a performance disadvantage**
  - *all* Chapel arrays are written as Chapel code
    - compiler has no built-in knowledge of arrays, as in ZPL, Fortran, C
    - a distinct performance handicap compared to those languages
  - this has slowed our start, but we don't think it's inherently problematic
    - that said, we have not achieved parity with ZPL yet at scale
      - (though admittedly, our focus has been different)
    - nor have we re-implemented many key ZPL optimizations in Chapel (yet)
      - e.g., communication optimizations, stencil optimization that helped NAS MG

Lower-Level Features



Base Language Features, by example

```
iter fib(n) {
  var current = 0,
      next = 1;

  for i in 1..n {
    yield current;
    current += next;
    current <=> next;
  }
}
```

```
for (i,f) in zip(0..#n, fib(n)) do
  writeln("fib #", i, " is ", f);
```

```
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
...
```

Base Language Features, by example

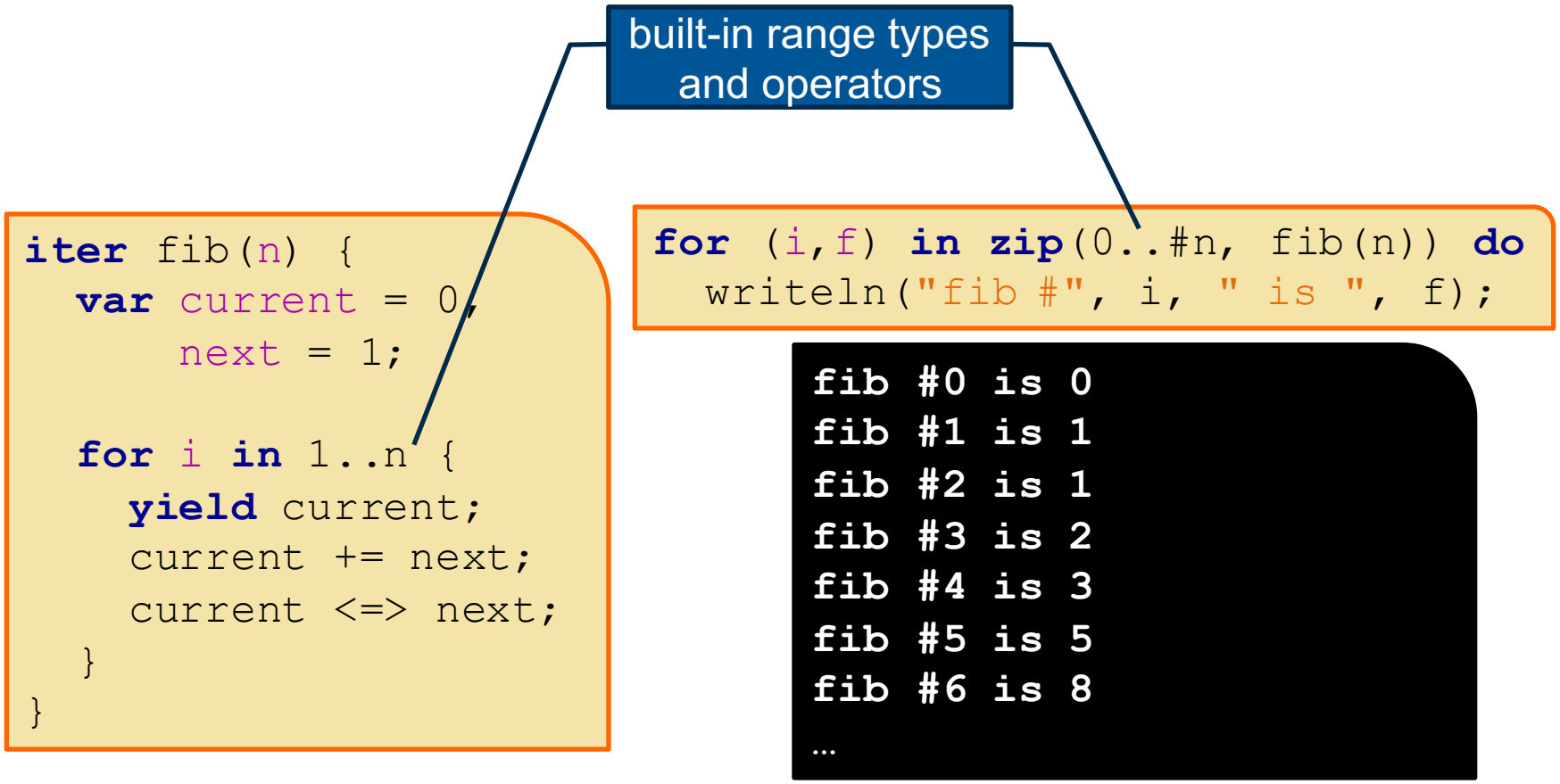
CLU-style iterators

```
iter fib(n) {
  var current = 0,
      next = 1;

  for i in 1..n {
    yield current;
    current += next;
    current <=> next;
  }
}
```

```
for (i,f) in zip(0..#n, fib(n)) do
  writeln("fib #", i, " is ", f);
```

```
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
...
```

Base Language Features, by example



Base Language Features, by example

```
iter fib(n) {
  var current = 0,
      next = 1;

  for i in 1..n {
    yield current;
    current += next;
    current <=> next;
  }
}
```

zippered iteration

```
for (i,f) in zip(0..#n, fib(n)) do
  writeln("fib #", i, " is ", f);
```

```
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
...
```

Base Language Features, by example

```
iter fib(n) {
  var current = 0,
      next = 1;

  for i in 1..n {
    yield current;
    current += next;
    current <=> next;
  }
}
```

tuples

```
for (i,f) in zip(0..#n, fib(n)) do
  writeln("fib #", i, " is ", f);
```

```
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
...
```

Base Language Features, by example

Static Type Inference for:
- arguments
- return types
- variables

```
iter fib(n) {
  var current = 0,
      next = 1;

  for i in 1..n {
    yield current;
    current += next;
    current <=> next;
  }
}
```

```
for (i,f) in zip(0..#n, fib(n)) do
  writeln("fib #", i, " is ", f);
```

```
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
...
```

Base Language Features, by example

```
iter fib(n) {
  var current = 0,
      next = 1;

  for i in 1..n {
    yield current;
    current += next;
    current <=> next;
  }
}
```

```
for (i,f) in zip(0..#n, fib(n)) do
  writeln("fib #", i, " is ", f);
```

```
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
...
```

Task Parallelism, Locality Control, by example

taskParallel.chpl

```
coforall loc in Locales do
  on loc {
    const numTasks = here.maxTaskPar;
    coforall tid in 1..numTasks do
      writef("Hello from task %n of %n "+
             "running on %s\n",
             tid, numTasks, here.name);
  }
```

```
prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```

Task Parallelism, Locality Control, by example

High-Level Task Parallelism

taskParallel.chpl

```
coforall loc in Locales do
  on loc {
    const numTasks = here.maxTaskPar;
    coforall tid in 1..numTasks do
      writef("Hello from task %n of %n "+
             "running on %s\n",
             tid, numTasks, here.name);
  }
```

```
prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```

Task Parallelism, Locality Control, by example

taskParallel.chpl

```
coforall loc in Locales do
  on loc {
    const numTasks = here.maxTaskPar;
    coforall tid in 1..numTasks do
      writef("Hello from task %n of %n "+
             "running on %s\n",
             tid, numTasks, here.name);
  }
```

```
prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```

Task Parallelism, Locality Control, by example

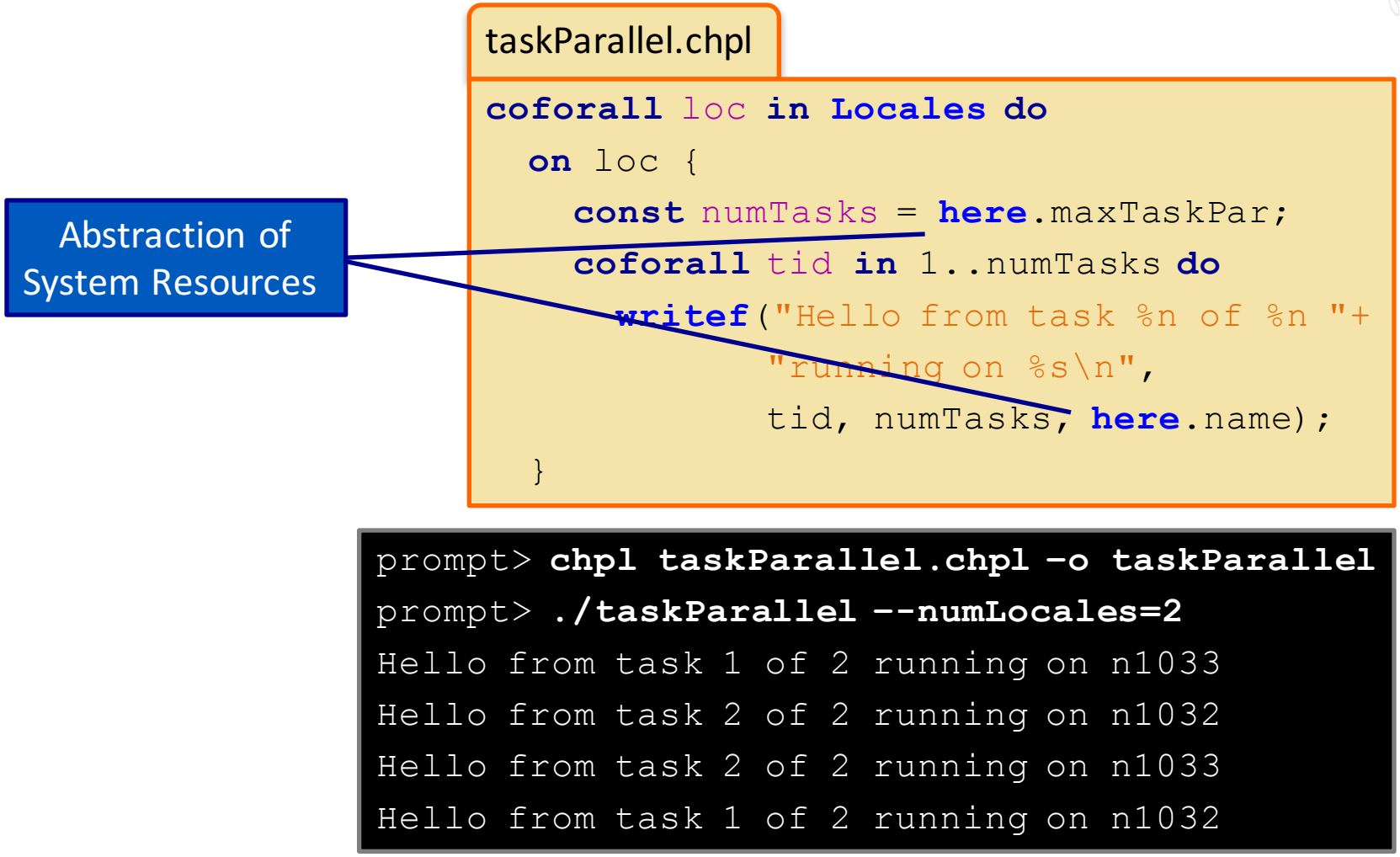
Control of Locality/Affinity

taskParallel.chpl

```
coforall loc in Locales do
  on loc {
    const numTasks = here.maxTaskPar;
    coforall tid in 1..numTasks do
      writef("Hello from task %n of %n "+
             "running on %s\n",
             tid, numTasks, here.name);
  }
```

```
prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```

Task Parallelism, Locality Control, by example



Task Parallelism, Locality Control, by example

High-Level Task Parallelism

taskParallel.chpl

```
coforall loc in Locales do
  on loc {
    const numTasks = here.maxTaskPar;
    coforall tid in 1..numTasks do
      writef("Hello from task %n of %n "+
             "running on %s\n",
             tid, numTasks, here.name);
  }
```

```
prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```

Task Parallelism, Locality Control, by example

taskParallel.chpl

```
coforall loc in Locales do
  on loc {
    const numTasks = here.maxTaskPar;
    coforall tid in 1..numTasks do
      writef("Hello from task %n of %n "+
             "running on %s\n",
             tid, numTasks, here.name);
  }
```

Not seen here:

Data-centric task coordination via atomic and full/empty vars

```
prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```

Task Parallelism, Locality Control, by example

taskParallel.chpl

```
coforall loc in Locales do
  on loc {
    const numTasks = here.maxTaskPar;
    coforall tid in 1..numTasks do
      writef("Hello from task %n of %n "+
             "running on %s\n",
             tid, numTasks, here.name);
  }
```

```
prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```

Parallelism and Locality: Orthogonal in Chapel

- This is a **parallel**, but local program:

```
coforall i in 1..msgs do
  writeln("Hello from task ", i);
```

- This is a **distributed**, but serial program:

```
writeln("Hello from locale 0!");
on Locales[1] do writeln("Hello from locale 1!");
on Locales[2] do writeln("Hello from locale 2!");
```

- This is a **distributed** **parallel** program:

```
coforall i in 1..msgs do
  on Locales[i%numLocales] do
    writeln("Hello from task ", i,
            " running on locale ", here.id);
```

ZPL and Chapel Resources

ZPL Resources

- **Project Page:**
  http://research.cs.washington.edu/zpl/home/index.html
- **Recommended Reading:**

  The Design and Development of ZPL. Lawrence Snyder. *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of Programming Languages*, 8-1–8-37, 2007.

  The Design and Implementation of a Region-Based Parallel Language. Bradford Chamberlain. *Ph.D. Thesis*, University of Washington, September 2001.

Chapel Websites

Project page: http://chapel.cray.com

- overview, papers, presentations, language spec, …

GitHub: https://github.com/chapel-lang

- download Chapel; browse source repository; contribute code

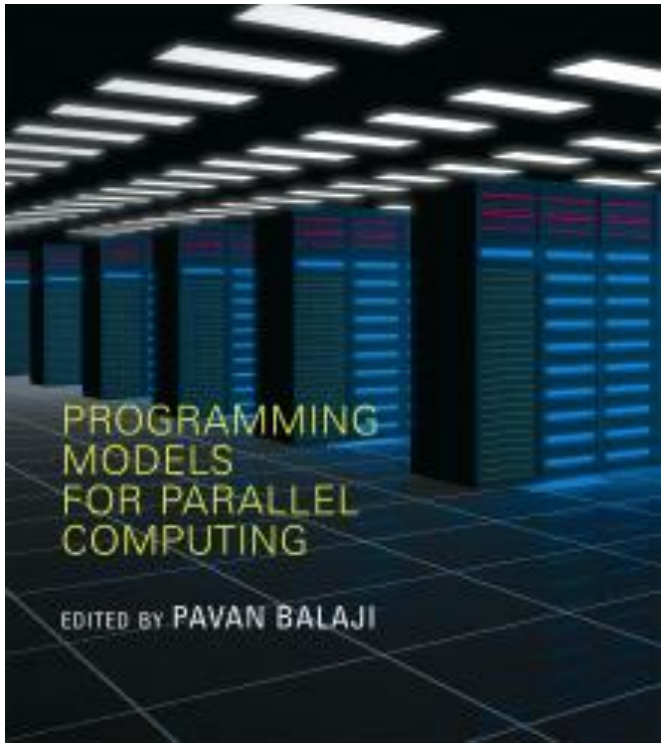
Facebook: https://www.facebook.com/ChapelLanguage

Twitter: https://twitter.com/ChapelLanguage

Suggested Reading

Chapel chapter from ***Programming Models for Parallel Computing***

- a detailed overview of Chapel's history, motivating themes, features
- edited by Pavan Balaji, published by MIT Press, November 2015
- chapter is now also available online



Other Chapel papers/publications available at **http://chapel.cray.com/papers.html**

Chapel Blog Articles

Chapel: Productive Parallel Programming, Cray Blog, May 2013.

- *a short-and-sweet introduction to Chapel*

Chapel Springs into a Summer of Code, Cray Blog, April 2016.

- *a run-down of some current events*

Six Ways to Say "Hello" in Chapel **(parts 1, 2, 3)**, Cray Blog, Sep-Oct 2015.

- *a series of articles illustrating the basics of parallelism and locality in Chapel*

Why Chapel? **(parts 1, 2, 3)**, Cray Blog, Jun-Oct 2014.

- *a series of articles answering common questions about why we are pursuing Chapel in spite of the inherent challenges*

[Ten] Myths About Scalable Programming Languages, IEEE TCSC Blog (index available on chapel.cray.com "blog articles" page), Apr-Nov 2012.

- *a series of technical opinion pieces designed to argue against standard reasons given for not developing high-level parallel languages*

Chapel Mailing Lists

low-traffic (read-only):

chapel-announce@lists.sourceforge.net: announcements about Chapel

community lists:

chapel-users@lists.sourceforge.net: user-oriented discussion list
chapel-developers@lists.sourceforge.net: developer discussions
chapel-education@lists.sourceforge.net: educator discussions
chapel-bugs@lists.sourceforge.net: public bug forum

(subscribe at SourceForge: http://sourceforge.net/p/chapel/mailman/**)**

To contact the Cray team:

chapel_info@cray.com: contact the team at Cray
chapel_bugs@cray.com: for reporting non-public bugs

Acknowledgements

Thanks to all the past members of the ZPL team for their contributions to this talk's content, particularly my advisor Larry Snyder.

Thanks also to all the past and present members of the Chapel team for their contributions, particularly Burton Smith and David Callahan for opening the gate and leading the charge.

Questions?

Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

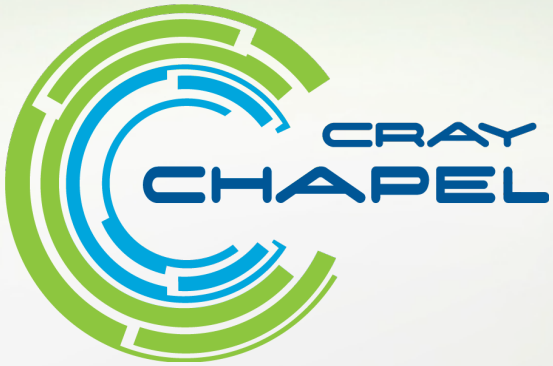
All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, and URIKA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.



CRAY

THE SUPERCOMPUTER COMPANY