Chapel
the Cascade High Productivity Language

Brad Chamberlain

UT Austin

June 3, 2008

Chapel

Chapel: a new parallel language being developed by Cray Inc.

Themes:

- **general parallel programming**
  - data-, task-, and nested parallelism
  - express general levels of software parallelism
  - target general levels of hardware parallelism
- ***global-view* abstractions**
  - stay tuned…
- ***multiresolution* design**
  - program abstractly or close to the machine as needed
- **control of locality**
  - necessary for scalability
- **reduce gap between mainstream & parallel languages**

Chapel's Setting: HPCS

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

- Goal: Raise HEC user productivity by 10× for the year 2010
- Productivity = Performance
  + Programmability
  + Portability
  + Robustness

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

      Cray: Chapel   IBM: X10   Sun: Fortress

- **Phase III**: Cray, IBM (July 2006 – 2010)
  - Implement the systems and technologies resulting from phase II
  - (Sun also continues work on Fortress, without HPCS funding)

Chapel and Productivity

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, UPC, CAF, …
- improve code robustness via improved semantics and concepts
  - eliminate common error cases altogether
  - better abstractions to help avoid other errors

Outline

- ✓ Chapel Context
- ➢ Global-view Programming Models
- ❑ Language Overview
- ❑ Example Computations
- ❑ Status, Future Work, Collaborations

Parallel Programming Model Taxonomy

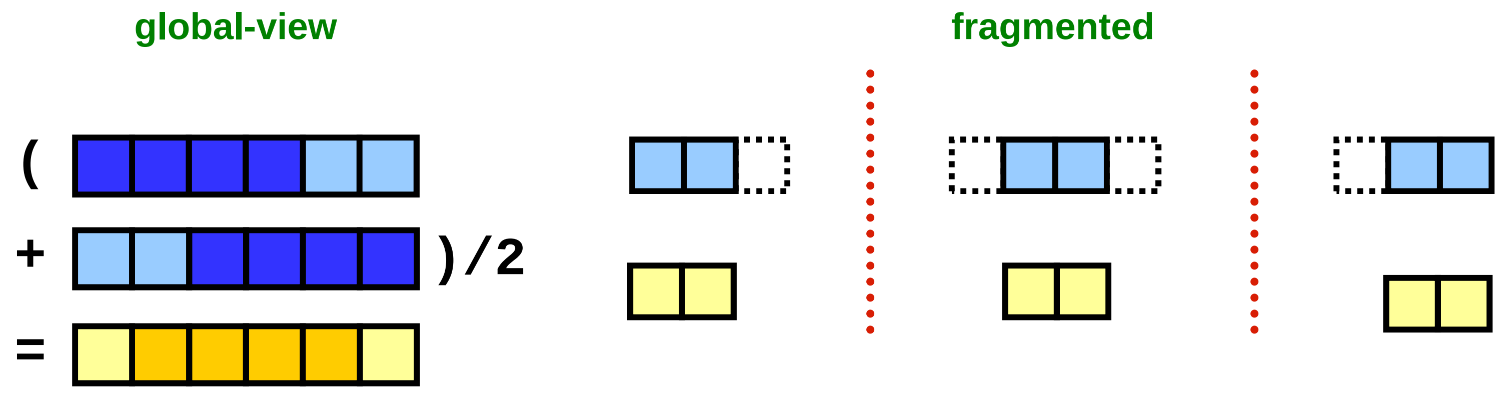
programming model: the mental model a programmer uses when coding using a language, library, or other notation

fragmented models: those in which the programmer writes code from the point-of-view of a single processor/thread

global-view models: those in which the programmer can write code that describes the computation as a whole

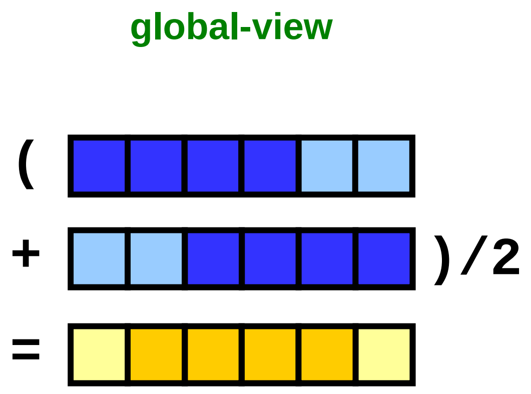
Global-view vs. Fragmented

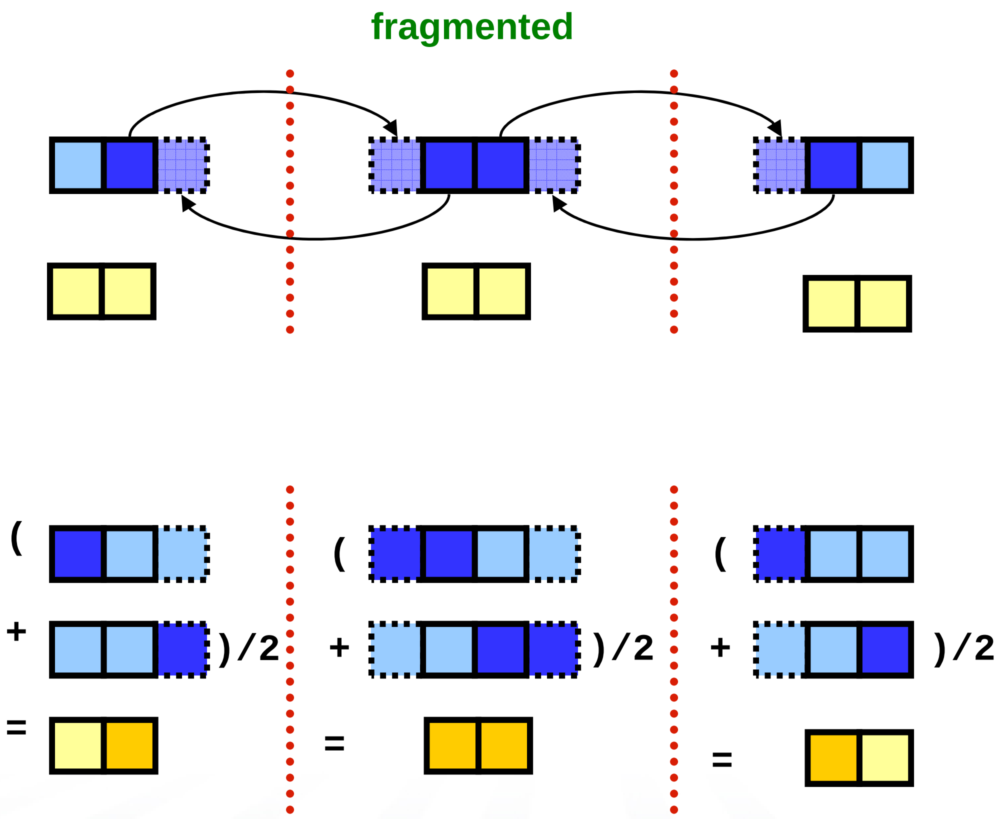
Problem: "Apply 3-pt stencil to vector"



Global-view vs. Fragmented

Problem: “Apply 3-pt stencil to vector”





Parallel Programming Model Taxonomy

programming model: the mental model a programmer uses when coding using a language, library, or other notation

fragmented models: those in which the programmer writes code from the point-of-view of a single processor/thread

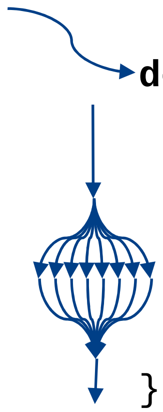
SPMD models: Single-Program, Multiple Data -- a common fragmented model in which the user writes one program & runs multiple copies of it, parameterized by a unique ID

global-view models: those in which the programmer can write code that describes the computation as a whole

Global-view vs. SPMD Code

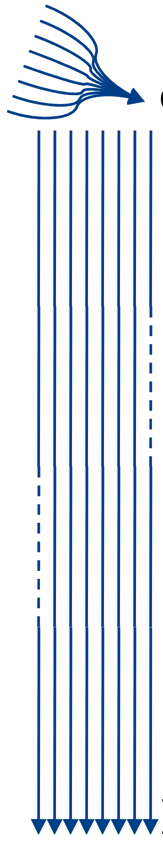
Problem: “Apply 3-pt stencil to vector”

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;
  }
}
```

SPMD



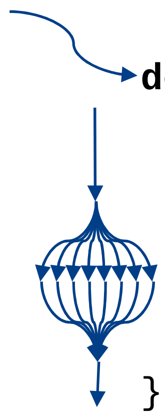
```
def main() {
  var n: int = 1000;
  var locN: int = n/numProcs;
  var a, b: [0..locN+1] real;

  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 vs. SPMD Code

Problem: "Apply 3-pt stencil to vector"

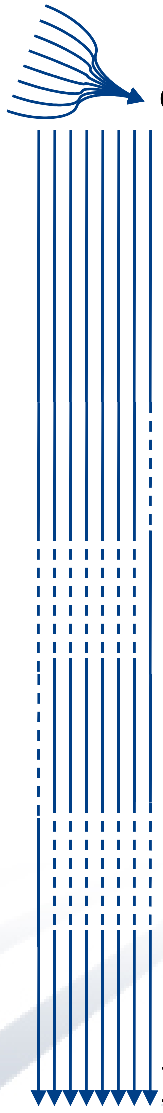
Assumes *numProcs* divides *n*; a more general version would require additional effort

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;
  }
}
```

SPMD



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

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

Current HPC Programming Notations

- **communication libraries:**

• MPI, MPI-2	(fragmented, typically SPMD)
• SHMEM, ARMCI, GASNet	(SPMD)

- **shared memory models:**

• OpenMP	(global-view, trivially)

- **PGAS languages:**

• Co-Array Fortran	(SPMD)
• UPC	(SPMD)
• Titanium	(SPMD)

MPI SPMD pseudo-code

Problem: "Apply 3-pt stencil to vector"

SPMD (pseudocode + MPI)

```
var n: int = 1000, locN: int = n/numProcs;
var a, b: [0..locN+1] real;
var innerLo: int = 1, innerHi: int = locN;
var numProcs, myPE: int;
var retval: int;
var status: MPI_Status;

MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
MPI_Comm_rank(MPI_COMM_WORLD, &myPE);
if (myPE < numProcs-1) {
  retval = MPI_Send(&(a(locN)), 1, MPI_FLOAT, myPE+1, 0, MPI_COMM_WORLD);
  if (retval != MPI_SUCCESS) { handleError(retval); }
  retval = MPI_Recv(&(a(locN+1)), 1, MPI_FLOAT, myPE+1, 1, MPI_COMM_WORLD, &status);
  if (retval != MPI_SUCCESS) { handleErrorWithStatus(retval, status); }
} else
  innerHi = locN-1;
if (myPE > 0) {
  retval = MPI_Send(&(a(1)), 1, MPI_FLOAT, myPE-1, 1, MPI_COMM_WORLD);
  if (retval != MPI_SUCCESS) { handleError(retval); }
  retval = MPI_Recv(&(a(0)), 1, MPI_FLOAT, myPE-1, 0, MPI_COMM_WORLD, &status);
  if (retval != MPI_SUCCESS) { handleErrorWithStatus(retval, status); }
} else
  innerLo = 2;
forall i in (innerLo..innerHi) {
  b(i) = (a(i-1) + a(i+1))/2;
}
```

Communication becomes geometrically more complex for higher-dimensional arrays

Fortran+MPI 3D 27-point stencil (NAS MG *rprj3*)

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

      implicit none

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

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

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


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


      implicit none

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

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

      integer i3, i2, i1, buff_len,buff_id

      buff_id = 2 + dir
      buff_len = 0

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

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

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

         else if( dir .eq. +1 ) then

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

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

         endif
      endif

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

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

         else if( dir .eq. +1 ) then

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

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

         endif
      endif

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

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

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

         else if( dir .eq. +1 ) then

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

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

         endif
      endif

      return
      end


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


      implicit none

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

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

      integer buff_id, indx

      integer i3, i2, i1

      buff_id = 3 + dir
      indx = 0

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

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

         else if( dir .eq. +1 ) then

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

         endif
      endif

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

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

         else if( dir .eq. +1 ) then

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

         endif
      endif

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

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

         else if( dir .eq. +1 ) then

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

         endif
      endif

      return
      end


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


      implicit none

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

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

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

      dir = -1

      buff_id = 3 + dir
      buff_len = nm2

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


      dir = +1

      buff_id = 3 + dir
      buff_len = nm2

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

      dir = +1

      buff_id = 2 + dir
      buff_len = 0

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

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

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

      dir = -1

      buff_id = 2 + dir
      buff_len = 0

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

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

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

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

      dir = -1

      buff_id = 3 + dir
      indx = 0

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

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

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


      dir = +1

      buff_id = 3 + dir
      indx = 0

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

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

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

      return
      end

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

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

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

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

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

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

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

Summarizing Fragmented/SPMD Models

- Advantages:
  - fairly straightforward model of execution
  - relatively easy to comprehend, learn, reason about
  - relatively easy to implement
  - reasonable performance on commodity architectures
  - portable/ubiquitous
  - lots of important scientific work has been accomplished using them

- Disadvantages:
  - blunt means of expressing parallelism: cooperating executables
  - fails to abstract away architecture / implementing mechanisms
  - obfuscates algorithms with many low-level details
    - error-prone
    - brittle code: difficult to read, maintain, modify, *experiment*
    - “MPI: the assembly language of parallel computing”

Multiresolution Language Design

Conventional Wisdom: By providing higher-level concepts in a language, programmers' hands are tied, preventing them from optimizing for performance by hand

My Belief: With appropriate design, this need not be the case

- provide high-level features and automation for convenience
  - knowledge of such features can aid in compiler optimization
- provide capabilities to drop down to lower, more manual levels
- use appropriate separation of concerns to keep this clean
  - support the 90/10 rule
- in the limit…

  …support interoperability with lower-level languages

  …support MPI interface within Chapel (?)

  …support ability to inline C/Fortran "assembly" (?)

  (I believe that such capabilities will typically not be needed)

Outline

- ✓ Chapel Context
- ✓ Global-view Programming Models
  - ZPL Detour?
- ➢ Language Overview
  - Base Language
  - Parallel Features
    - task parallel
    - data parallel
  - Locality Features
- ❑ Example Computations
- ❑ Status, Future Work, Collaborations

Base Language: Themes

- Block-structured, imperative programming
- Intentionally not an extension to an existing language
- Instead, select attractive features from others:

  ZPL, HPF: data parallelism, index sets, distributed arrays
  (see also APL, NESL, Fortran90)

  Cray MTA C/Fortran: task parallelism, lightweight synchronization

  CLU: iterators (see also Ruby, Python, C#)

  ML: latent types (see also Scala, Matlab, Perl, Python, C#)

  Java, C#: OOP, type safety

  C++: generic programming/templates (without adopting its syntax)

  C, Modula, Ada: syntax

Base Language: Overview

- Syntax
  - adopt C/C++/Java/Perl syntax whenever possible/useful
  - main departures: declarations/casts, for loops, generics
- Language Elements
  - standard scalar types, expressions, statements
  - value- and reference-based OOP (optional)
  - no pointers, restricted opportunities for aliasing
  - argument intents similar to Fortran/Ada
- My favorite base language features
  - rich compile-time language
  - latent types / simple static type inference
  - configuration variables
  - tuples
  - iterators…

Base Language: Standard Stuff

- Lexical structure and syntax based largely on C/C++

```
      { a = b + c;  foo(); }    // no surprises here
```

- Reasonably standard in terms of:
  - scalar types
  - constants, variables
  - operators, expressions, statements, functions
- Support for object-oriented programming
  - value- and reference-based classes
  - no strong requirement to use OOP
- Modules for namespace management
- Generic functions and classes

Base Language: Departures

- **Syntax:** declaration syntax differs from C/C++

  `var` `<varName> [: <definition>] [= <init>];`

  `def` `<fnName>(<argList>)[: <returnType>] { … }`

- **Types**
  - support for complex, imaginary, string types
  - sizes more explicit than in C/C++ (*e.g.*, **`int`**`(32)`, **`complex`**`(128)`)
  - richer array support than C/C++, Java, even Fortran
  - no pointers (apart from class references)

- **Operators**
  - casts via '`:`' (*e.g.*, `3.14:` **`int`**`(32)`)
  - exponentiation via '`**`' (*e.g.*, `2**n`)

- **Statements:** for loop differs from C/C++

  `for` `<indices>` **`in`** `<iterationSpace> { … }`

  e.g., **`for`** `i` **`in`** `1..n { … }`

- **Functions:** argument-passing semantics

Base Language: My Favorite Departures

- **Rich compile-time language**
  - parameter values (compile-time constants)
  - folded conditionals, unrolled for loops, expanded tuples
  - type and parameter functions – evaluated at compile-time
- **Latent types:**
  - ability to omit type specifications for convenience or reuse
  - type specifications can be omitted from…

▪ variables	(inferred from initializers)
▪ class members	(inferred from constructors)
▪ function arguments	(inferred from callsite)
▪ function return types	(inferred from return statements)

- **Configuration variables** (and parameters)

```
config const n = 100;  // override with --n=1000000
```

- **Tuples**
- **Iterators…**

Base Language: Motivation for Iterators

Given a program with a bunch of similar loops…

```
for (i=0; i<m; i++) {
  for (j=0; j<n; j++) {
    …A(i,j)…
  }
}

…

for (i=0; i<m; i++) {
  for (j=0; j<n; j++) {
    …A(i,j)…
  }
}

…
```

Consider the effort to convert them from RMO to CMO…

```
for (j=0; j<n; j++) {
  for (i=0; i<m; i++) {
    …A(i,j)…
  }
}

…

for (j=0; j<n; j++) {
  for (i=0; i<m; i++) {
    …A(i,j)…
  }
}

…
```

Or to tile the loops…

```
for (jj=0; jj<n; jj+=blocksize) {
  for (ii=0; ii<m; ii+=blocksize) {
    for (j=jj; j<min(m,jj+blocksize-1) {
      for (i=ii; i<min(n,ii+blocksize-1)
    {
        …A(i,j)…
      }
    }
  }
}

…

for (jj=0; jj<n; jj+=blocksize) {
  for (ii=0; ii<m; ii+=blocksize) {
    for (j=jj; j<min(m,jj+blocksize-1) {
      for (i=ii; i<min(n,ii+blocksize-1)
    {
        …A(i,j)…
      }
    }
  }
}
…
```

Base Language: Motivation for Iterators

Given a program with a bunch of similar loops…

```
for (i=0; i<m; i++) {
  for (j=0; j<n; j++) {
    …A(i,j)…
  }
```

…

```
for
  for (
    …A(
  }
}
```

…

Consider the effort to convert them from RMO to CMO…

```
for (j=0; j<n; j++) {
  for (i=0; i<m; i++) {
    …A(i,j)…
  }
```

Or to tile the loops…

```
for (jj=0; jj<n; jj+=blocksize) {
  for (ii=0; ii<m; ii+=blocksize) {
    for (j=jj; j<min(m,jj+blocksize-1) {
      for (i=ii; i<min(n,ii+blocksize-1)
    {
        …A(i,j)…
      }
```

Or to change the iteration order over the tiles…

Or to make them into fragmented loops for an MPI program…

Or to change the distribution of the work/arrays in that MPI program…

Or to label them as parallel for OpenMP or a vectorizing compiler…

Or to do *anything* that we do with loops all the time as a community…

We wouldn't program straight-line code this way, so why are we so tolerant of our lack of loop abstractions?

Base Language: Iterators

- like functions, but *yield* a number of elements one-by-one:

```
iterator RMO() {
  for i in 1..m do
    for j in 1..n do
      yield (i,j);
}
```

```
iterator tiled(blocksize) {
  for ii in 1..m by blocksize do
    for jj in 1..n by blocksize do
      for i in ii..min(n, ii+blocksize-1) do
        for j in jj..min(m, jj+blocksize-1) {
          yield (i,j);
}       }
```

- iterators are used to drive loops:

```
for ij in RMO() {
  …A(ij)…
}
```

```
for ij in tiled(blocksize) {
  …A(ij)…
}
```

- as with functions…

  …one iterator can be redefined to change the behavior of many loops

  …a single invocation can be altered, or its arguments can be changed

- not necessarily any more expensive than in-line loops

Task Parallelism: Task Creation

begin: creates a task for future evaluation

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

cobegin: creates a task per component statement:

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

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

coforall: creates a task per loop iteration

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

Task Parallelism: Task Coordination

sync variables: store full/empty state along with value

```
var result$: sync real;   // result is initially empty
cobegin {
  … = result$;            // block until full, leave empty
  result$ = …;            // block until empty, leave full
}
result$.readFF();         // read when full, leave full;
                          // other variations also supported
```

single-assignment variables: writable once only

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

atomic sections: support transactions against memory

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

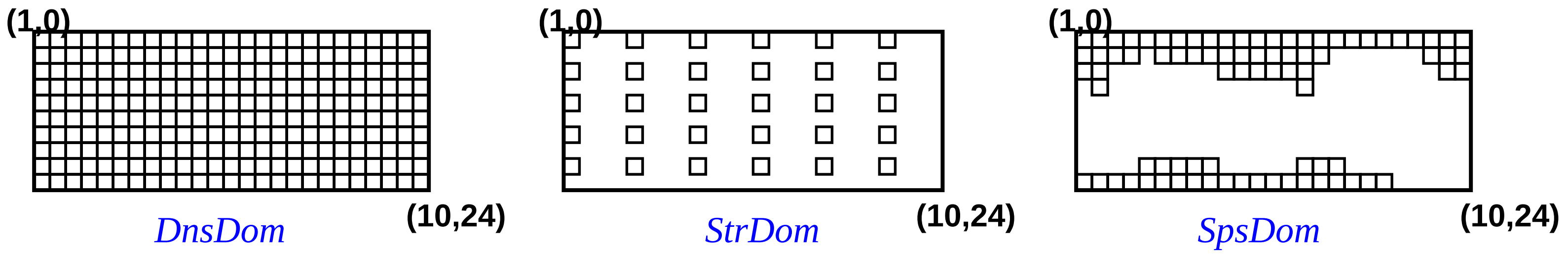
Producer/Consumer example

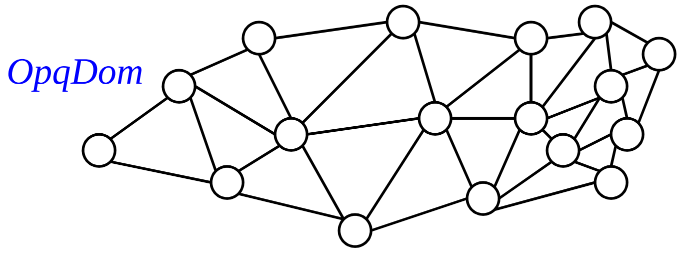
```
var buff$: [0..buffersize-1] sync int;

cobegin {
  producer();
  consumer();
}

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

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

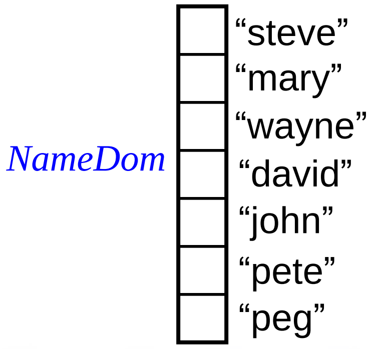
Data Parallelism: Domains

domains: first-class index sets, whose indices can be…

…integer tuples







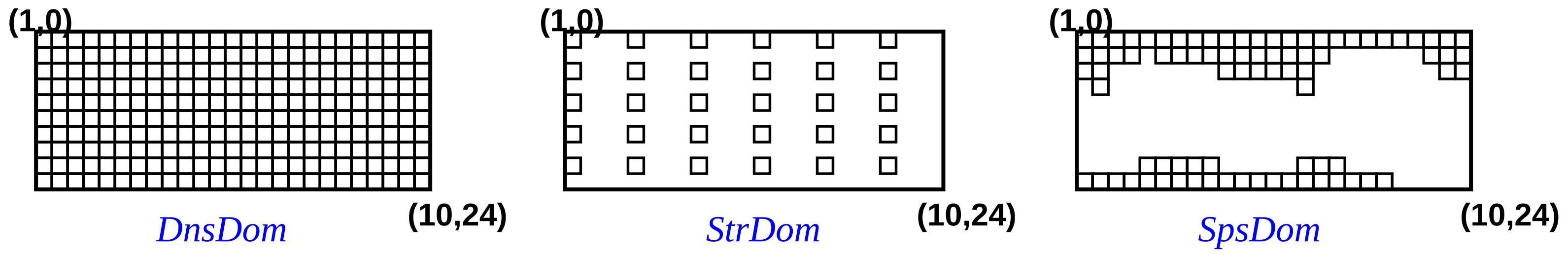
…anonymous

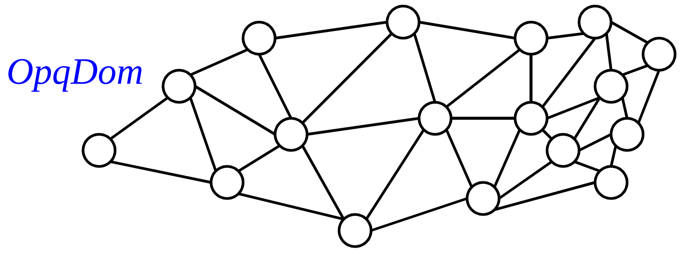
…arbitrary values

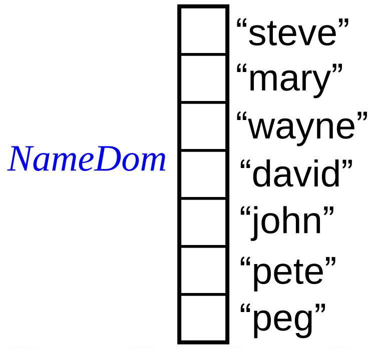
Data Parallelism: Domain Declarations

domains: first-class index sets, whose indices can be…

```
var DnsDom: domain(2) = [1..10, 0..24],
    StrDom: subdomain(DnsDom) = DnsDom by (2,4),
    SpsDom: subdomain(DnsDom) = genIndices();
```





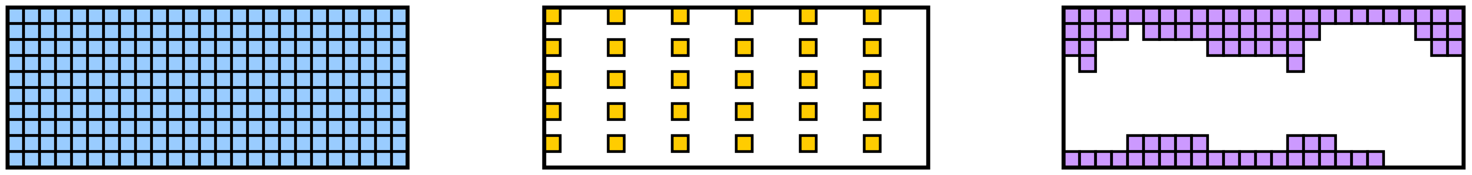


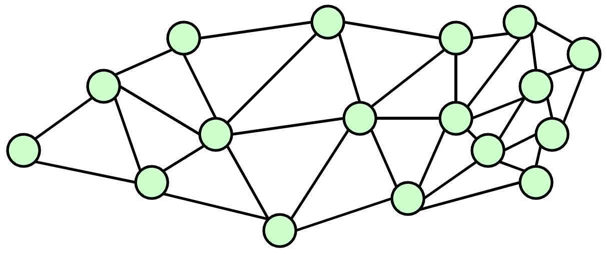
```
var OpqDom: domain(opaque),
    NameDom: domain(string) = readNames();
```

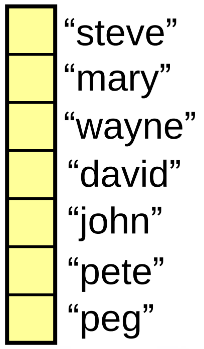
Data Parallelism: Domains and Arrays

Domains are used to declare arrays…

```
var DnsArr: [DnsDom] complex,
    SpsArr: [SpsDom] real;
…
```



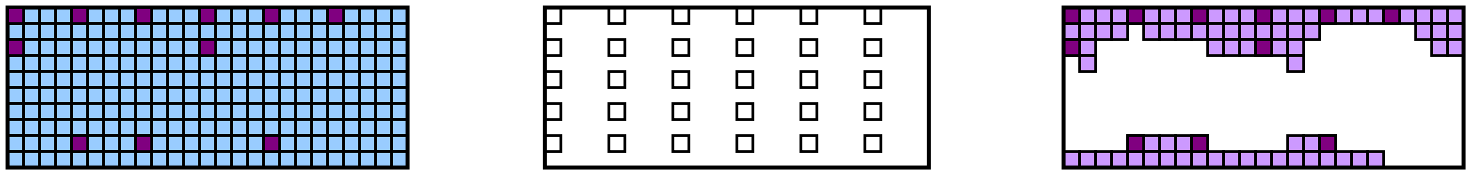


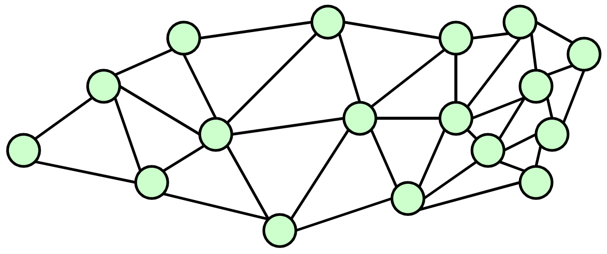


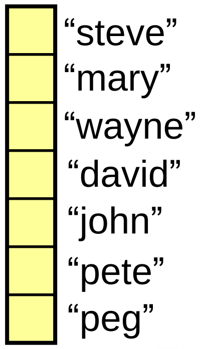
Data Parallelism: Domain Iteration

…to iterate over index spaces…

```
    forall (i,j) in StrDom {
      DnsArr(i,j) += SpsArr(i,j);
    }
```



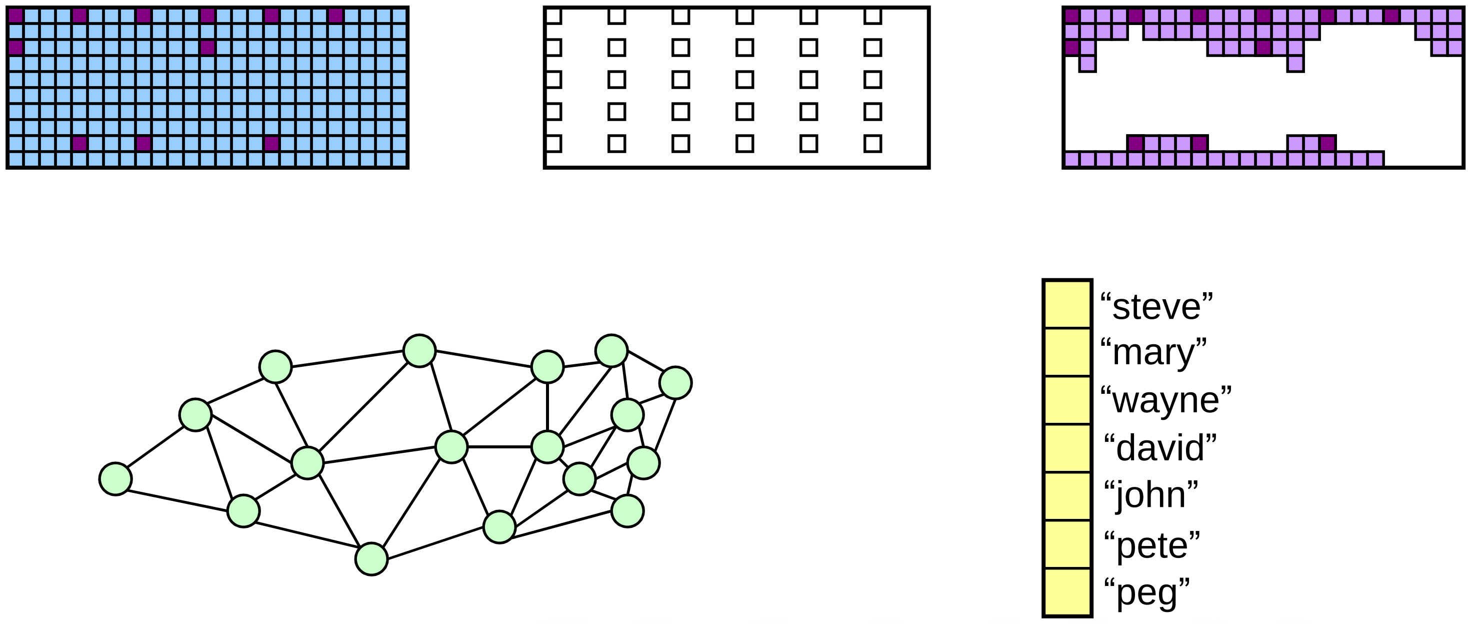




Data Parallelism: Array Slicing

…to slice arrays…

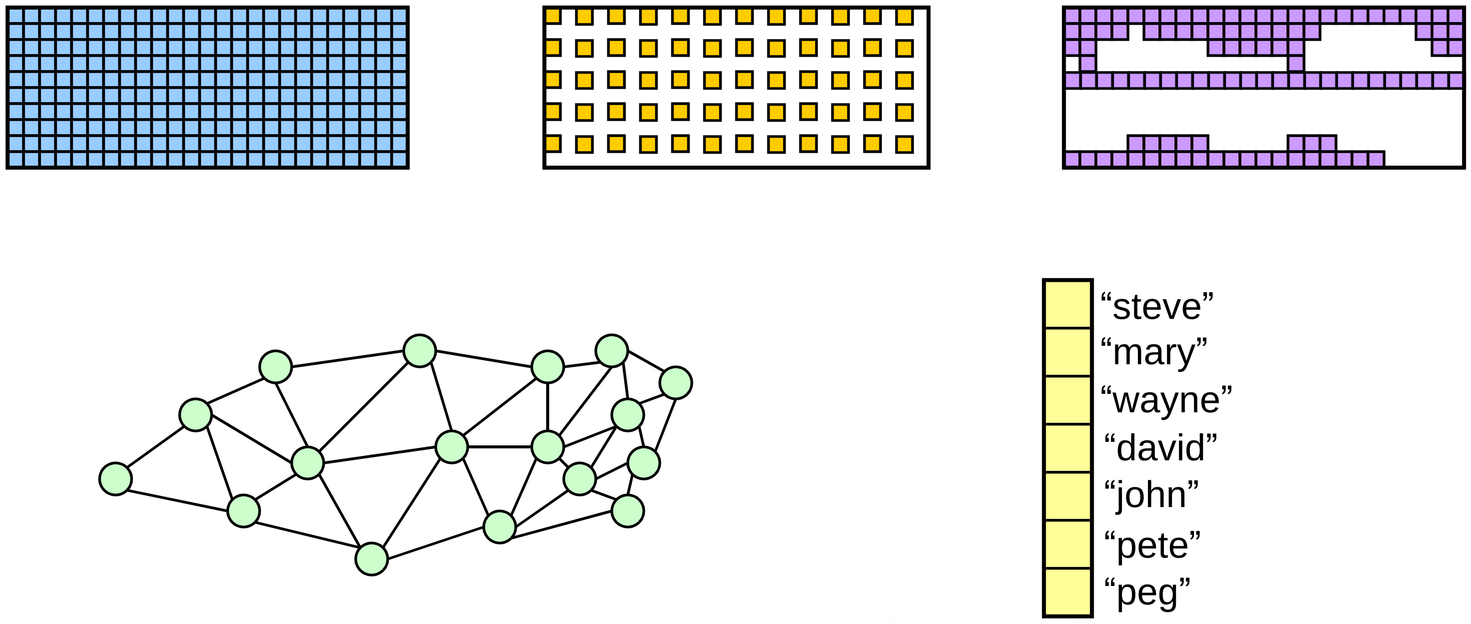
```
   DnsArr[StrDom] += SpsArr[StrDom];
```



Data Parallelism: Array Reallocation

...and to reallocate arrays

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



Locality: Locales

- *locale:* architectural unit of storage and processing
- user specifies # locales on executable command-line

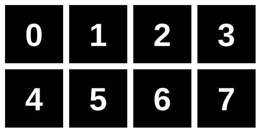
```
   prompt> myChapelProg –nl=8
```

- Chapel programs have a built-in domain/array of locales:

```
const LocaleSpace: domain(1) = [0..numLocales-1];
const Locales: [LocaleSpace] locale;
```

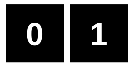
- Users can use this to create their own locale arrays:

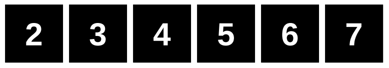
```
var CompGrid: [1..GridRows, 1..GridCols] locale = …;
```



CompGrid

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





TaskALocs *TaskBLocs*

Locality: Task Placement

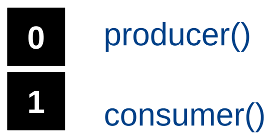
on clauses: indicate where tasks should execute

Either in a data-driven manner…

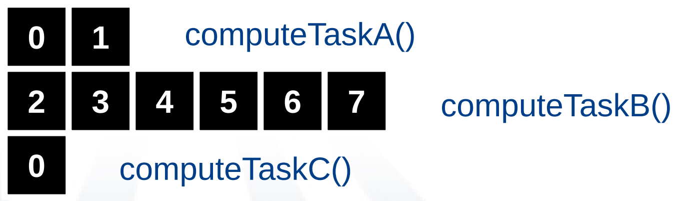
```
computePivot(lo, hi, data);
cobegin {
  on A(lo)    do Quicksort(lo, pivot, data);
  on A(pivot) do Quicksort(pivot, hi, data);
}
```

…or by naming locales explicitly

```
cobegin {
  on Locales(0) do producer();
  on Locales(1) do consumer();
}
```



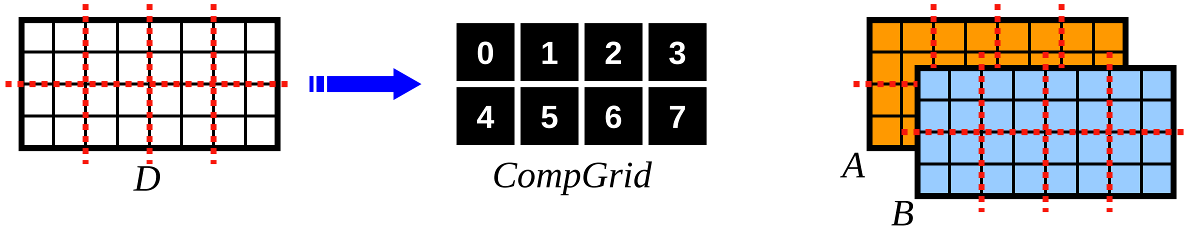
```
cobegin {
  on TaskALocs do computeTaskA(…);
  on TaskBLocs do computeTaskB(…);
  on Locales(0) do computeTaskC(…);
}
```



Locality: Domain Distribution

Domains may be distributed across locales

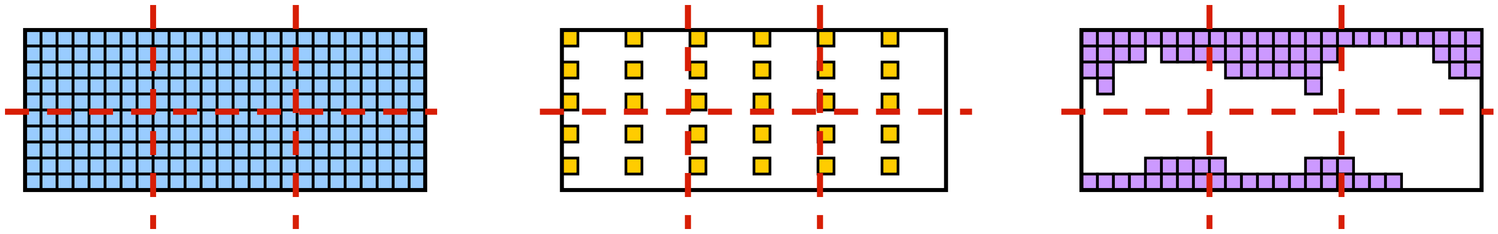
```
var D: domain(2) distributed Block on CompGrid = …;
```

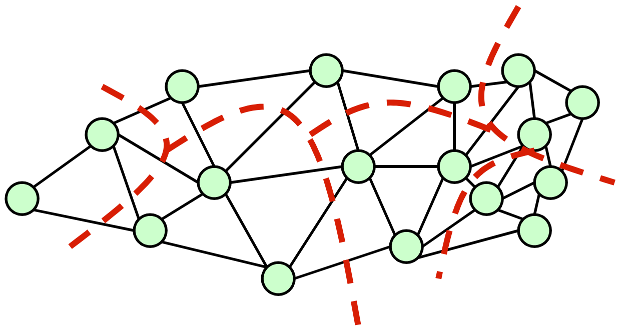


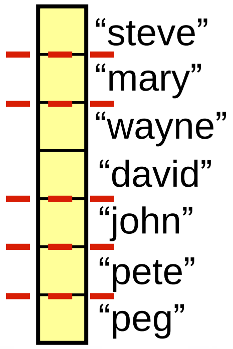
Locality: Domain Distributions

Distributions specify…

- …a mapping of indices to locales
- …per-locale storage for domain indices and array elements
- …a default work assignment for operations on domains/arrays







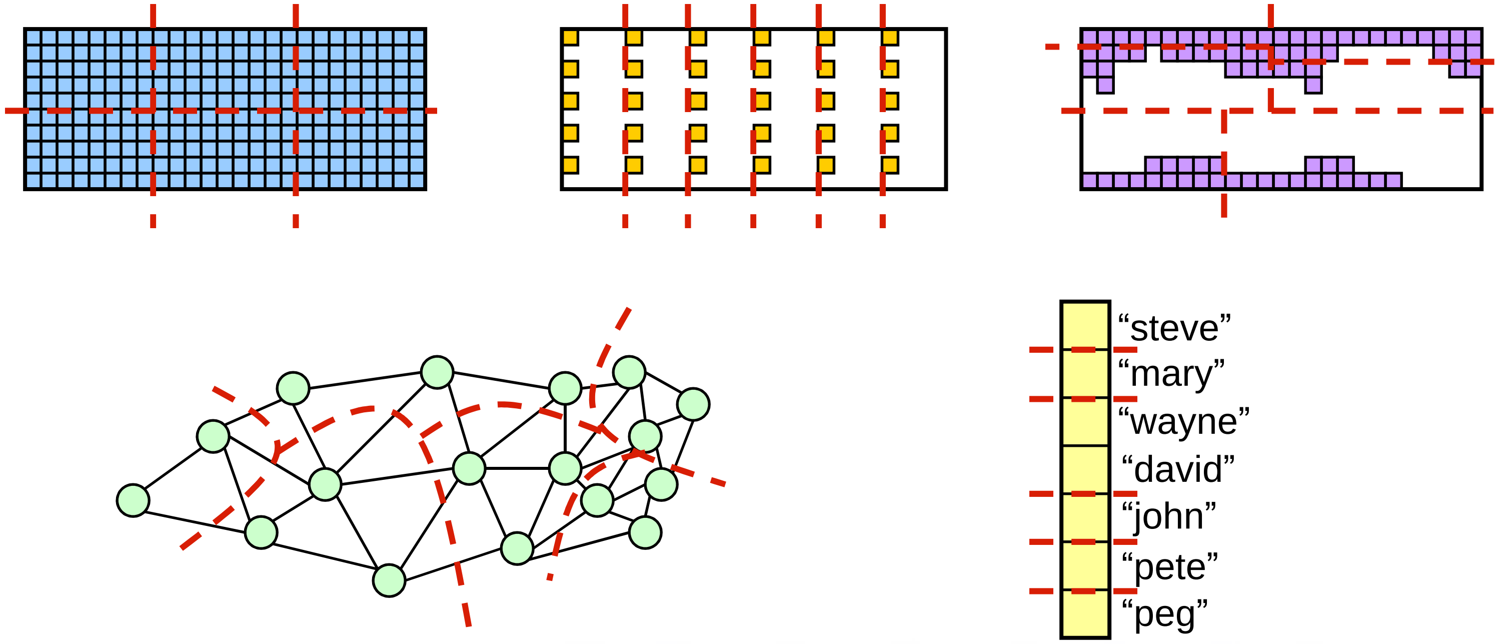
Locality: Domain Distributions

Distributions specify…

…a mapping of indices to locales

…per-locale storage for domain indices and array elements

…a default work assignment for operations on domains/arrays



Locality: Distributions Overview

Distributions: "recipes for distributed arrays"

- Intuitively, distributions implement the lowering…

  from: the user's global view of distributed data aggregates

  to: the fragmented implementation for distributed memory machines

- Users can implement custom distributions
- Author implements an interface which supports:
  - 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 provides a standard library of distributions…

  …written using the same mechanism as user-defined distributions

  …tuned for different platforms to maximize performance

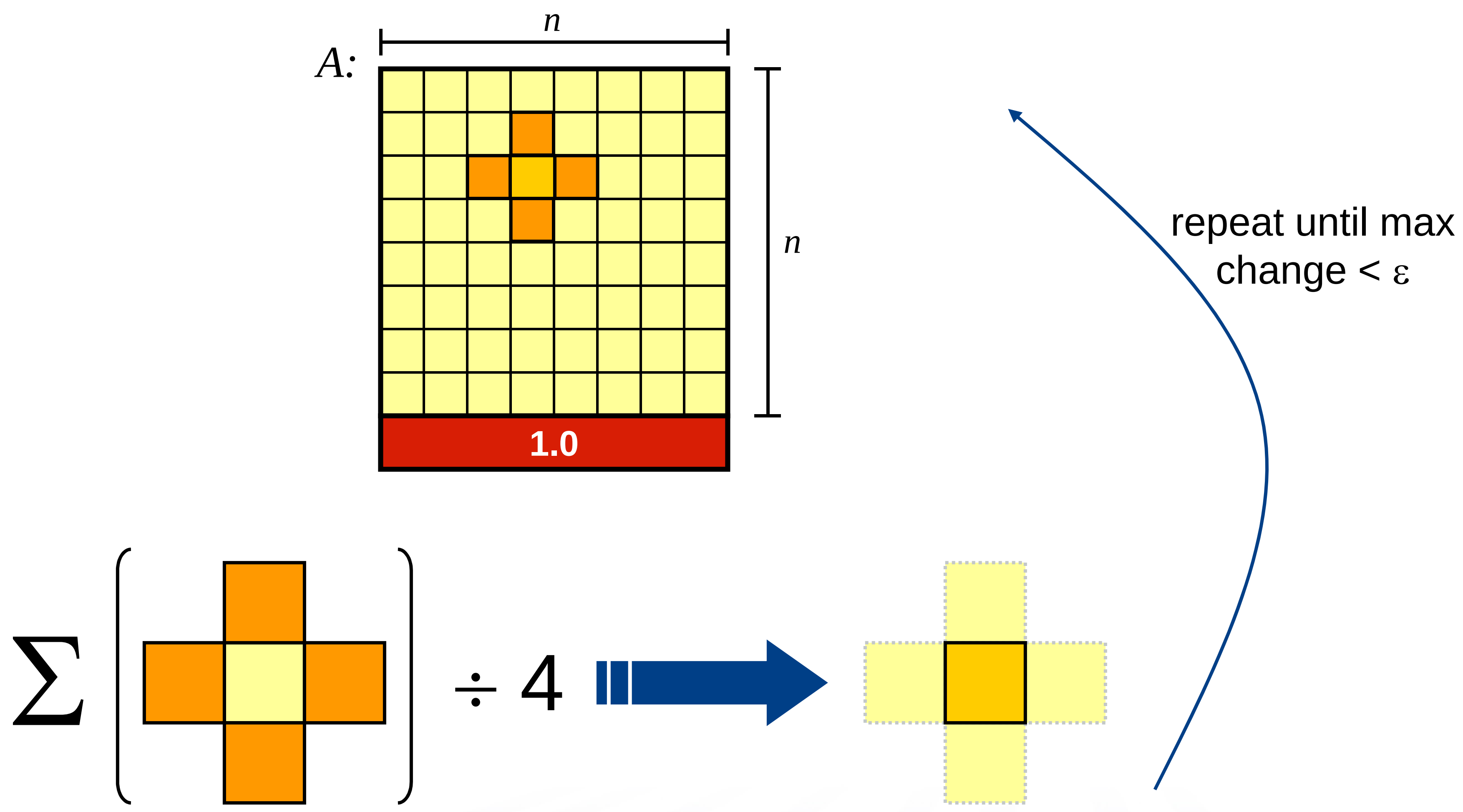
Other Features

- zippered and tensor flavors of iteration and promotion
- *subdomains* and *index types* to help reason about indices
- reductions and scans (with user-defined operators)

Outline

- ✓ Chapel Context
- ✓ Global-view Programming Models
- ✓ Language Overview
- ➢ Example Computations
- ❑ Status, Future Work, Collaborations

Example 1: Jacobi Iteration



Jacobi Iteration in Chapel

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

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

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
  [(i,j) in D] Temp(i,j) = (A(i-1,j) + A(i+1,j)
                          + A(i,j-1) + A(i,j+1)) / 4.0;

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

writeln(A);
```

Jacobi Iteration in Chapel

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

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

var A, Temp : [BigD] real;

A[Las

do {
  [(i

  var
  A(D
} whi

writeln(A);
```

Declare program parameters

const ⇒ can't change values after initialization

config ⇒ can be set on executable command-line

```
prompt> jacobi --n=10000 --epsilon=0.0001
```

note that no types are given; inferred from initializer

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

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

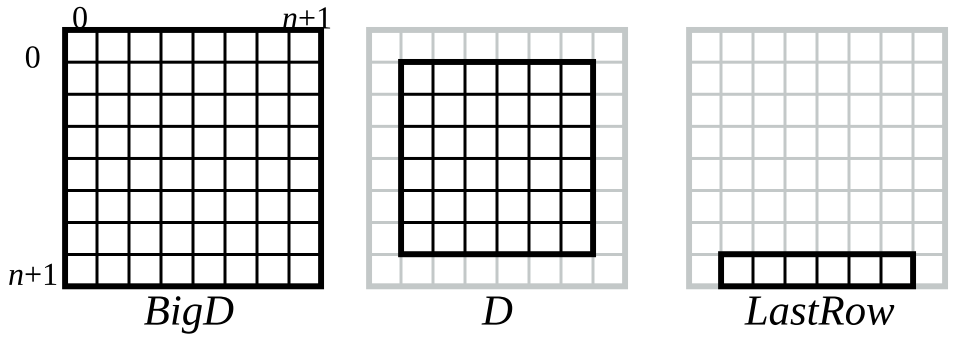
Jacobi Iteration in Chapel

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

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

Declare domains (first class index sets)

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

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



exterior ⇒ one of several built-in domain generators

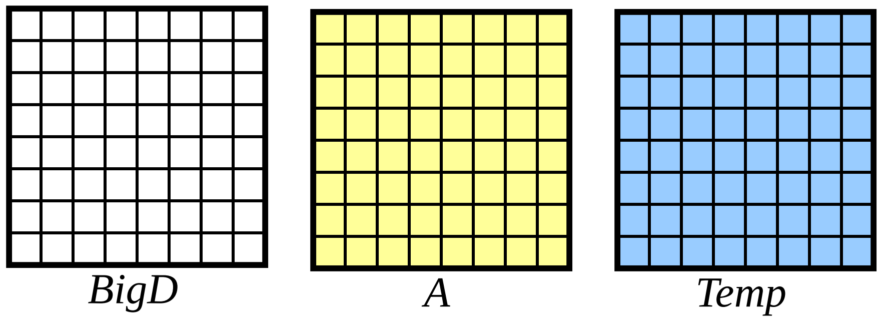
```
4.0;
```

Jacobi Iteration in Chapel

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

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

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

Declare arrays

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



4.0;

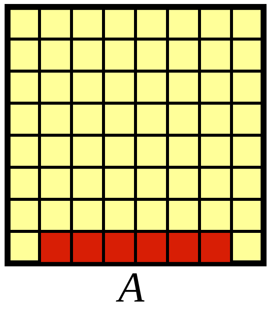
Jacobi Iteration in Chapel

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

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

var A, Temp : [BigD] real;

A[LastRow] = 1.0;
```

4.0;

Set Explicit Boundary Condition

indexing by domain ⇒ slicing mechanism
array expressions ⇒ parallel evaluation

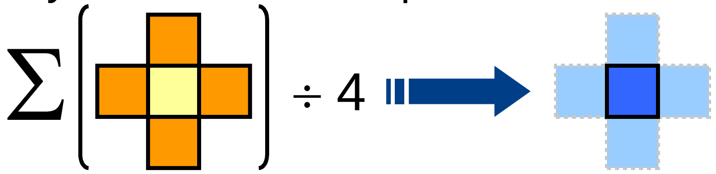


Jacobi Iteration in Chapel

Compute 5-point stencil

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

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



```
    [(i,j) in D] Temp(i,j) = (A(i-1,j) + A(i+1,j)
                            + A(i,j-1) + A(i,j+1)) / 4.0;

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

  writeln(A);
```

Jacobi Iteration in Chapel

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

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

Compute maximum change

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

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

```
do {
  [(i,j) in D] Temp(i,j) = (A(i-1,j) + A(i+1,j)
                          + A(i,j-1) + A(i,j+1)) / 4.0;

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

writeln(A);
```

Jacobi Iteration in Chapel

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

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

var

A[La

do {
  [(i,j) in D] Temp(i,j) = (A(i-1,j) + A(i+1,j)
                          + A(i,j-1) + A(i,j+1)) / 4.0;

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

writeln(A);
```

Copy data back & Repeat until done

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

Jacobi Iteration in Chapel

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

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

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
  [(                                            j)
                                                1)) / 4.0;

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

writeln(A);
```

Write array to console

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

Jacobi Iteration in Chapel

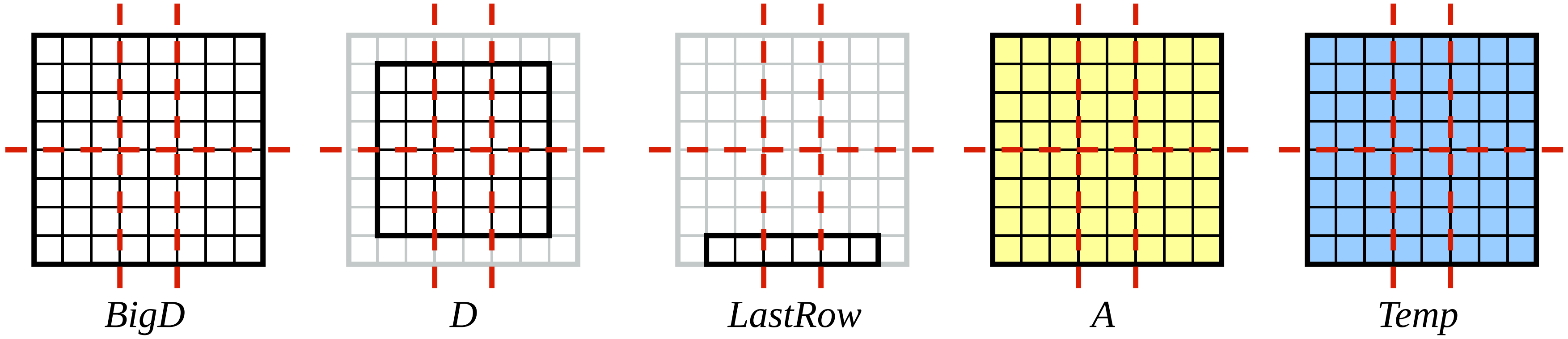
```
config const n = 6,
             epsilon = 1.0e-5;

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

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

With this change, same code runs in a distributed manner

Domain distribution maps indices to *locales*

⇒ decomposition of arrays & default location of iterations over locales

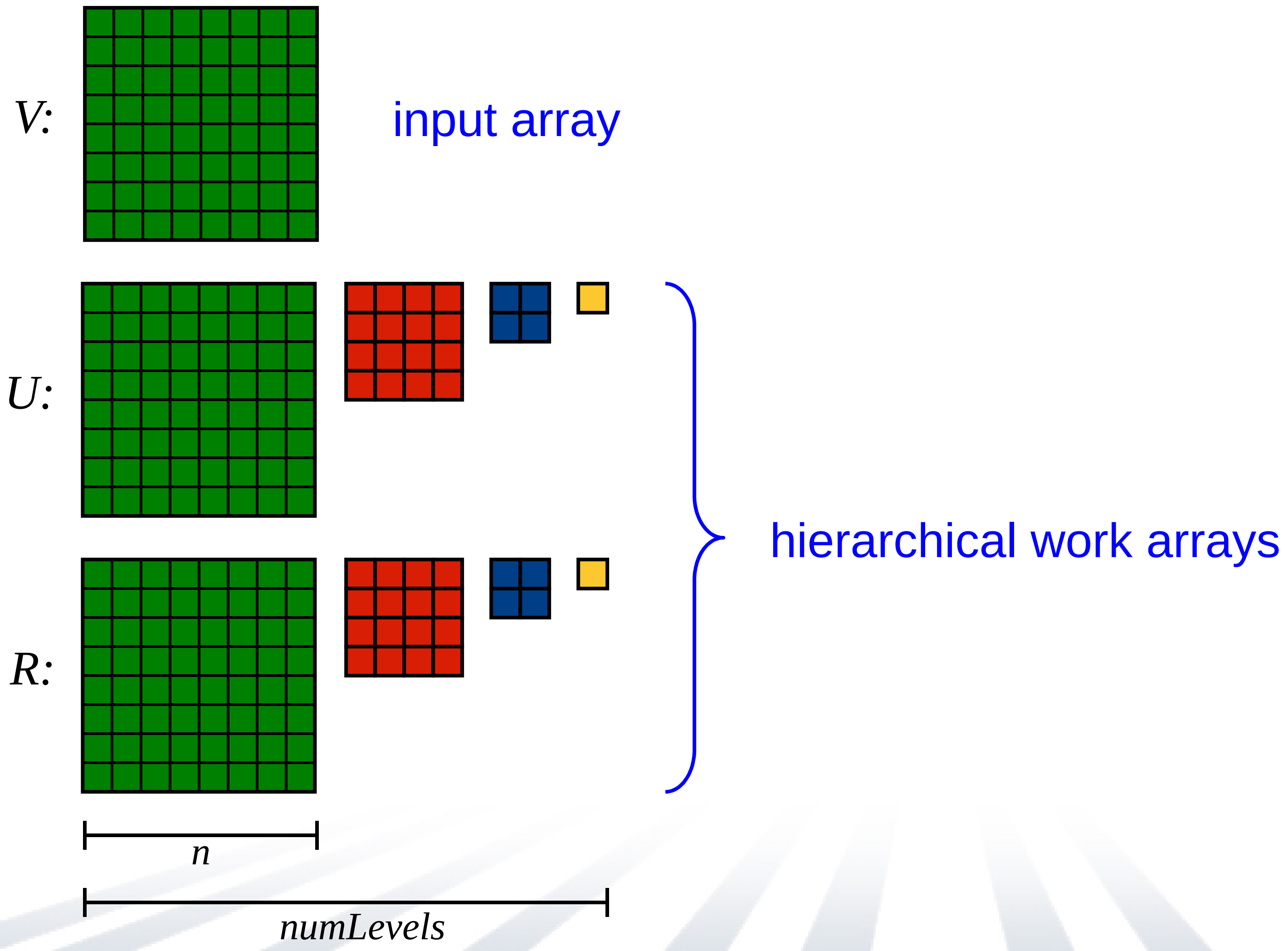
Subdomains inherit parent domain's distribution



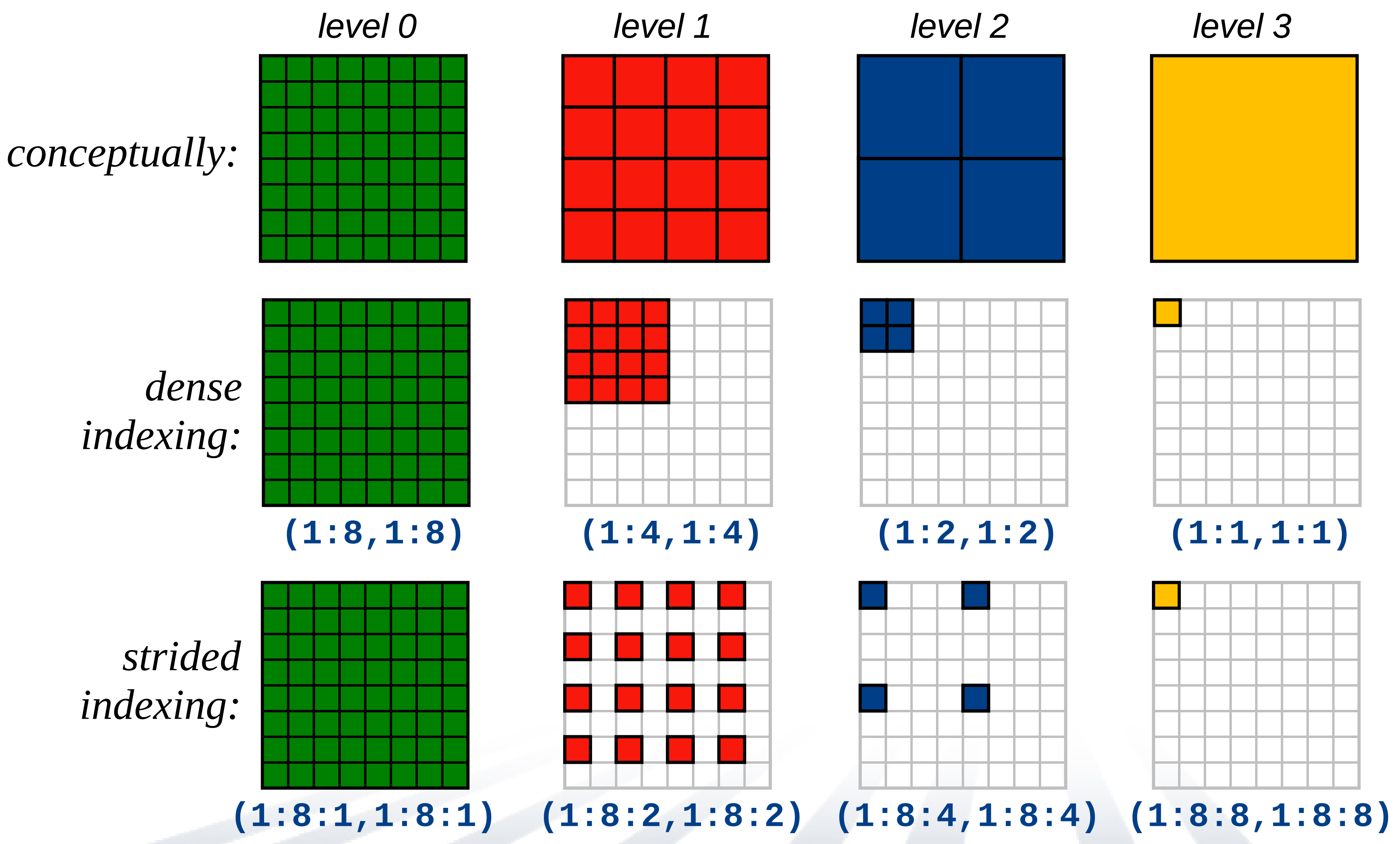
Jacobi Iteration in Chapel

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

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

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
  [(i,j) in D] Temp(i,j) = (A(i-1,j) + A(i+1,j)
                          + A(i,j-1) + A(i,j+1)) / 4.0;

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

writeln(A);
```

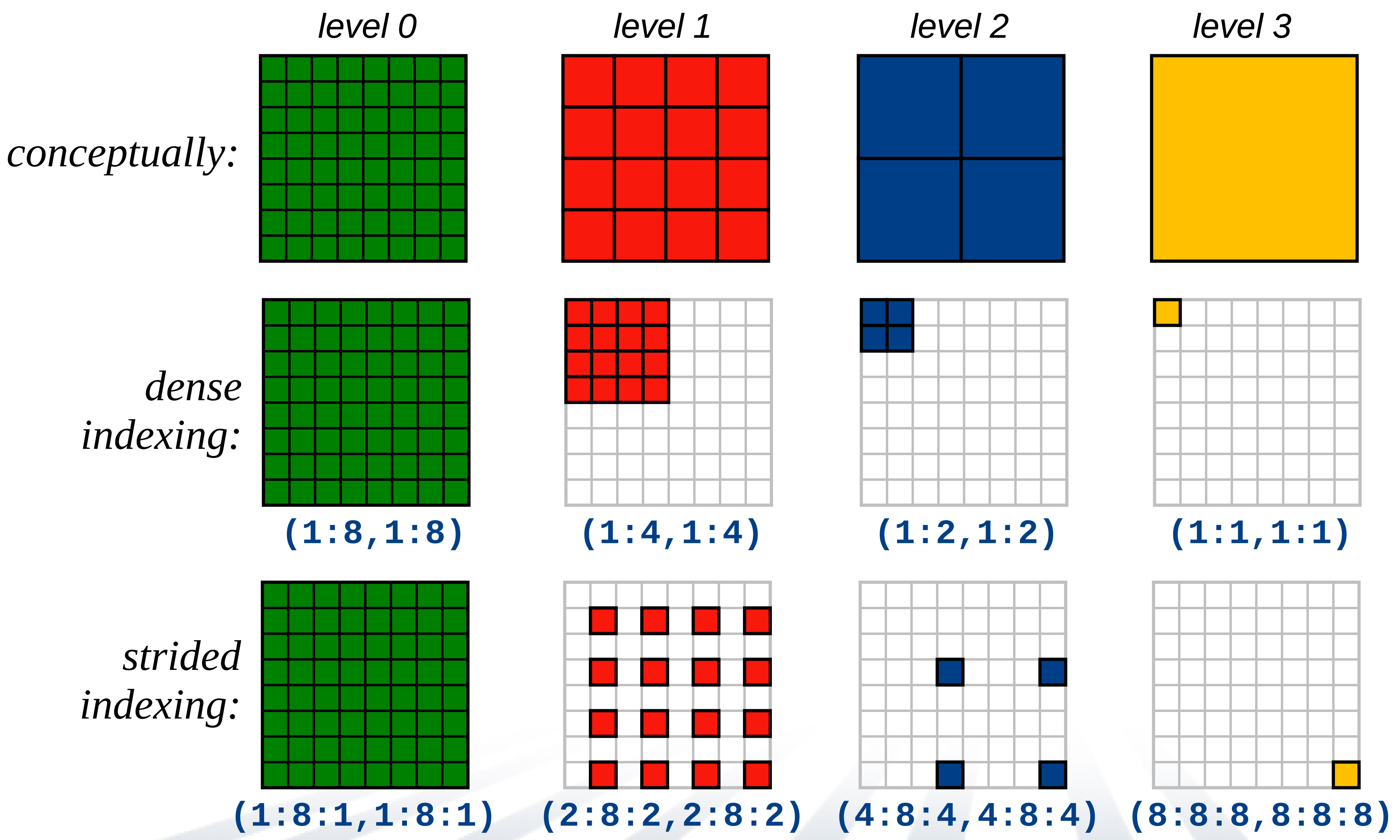
Example 2: Multigrid



Hierarchical Arrays



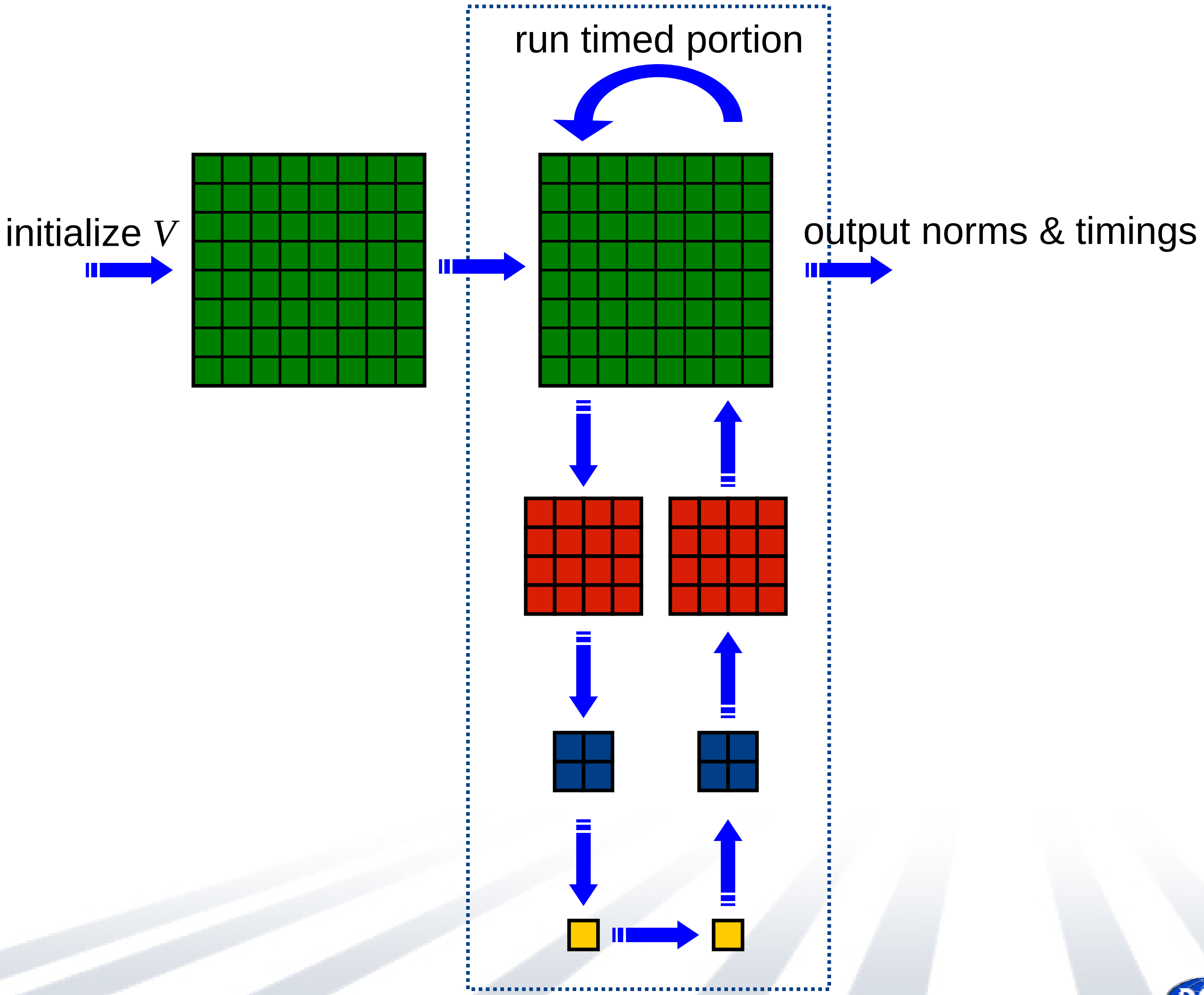
Hierarchical Arrays



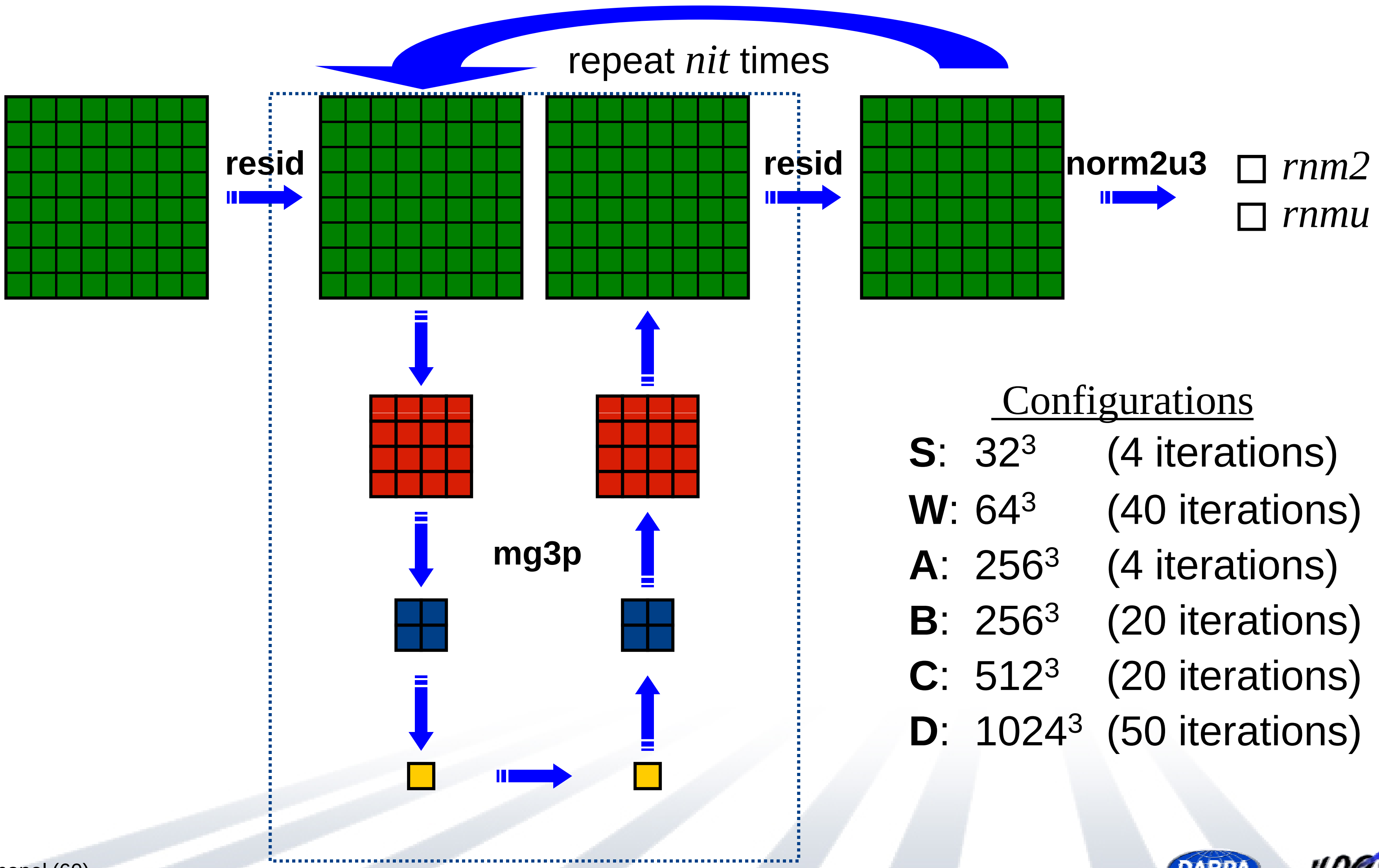
Hierarchical Array Declarations in Chapel

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

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

var V: [ProblemSpace] real;

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

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

Overview of NAS MG



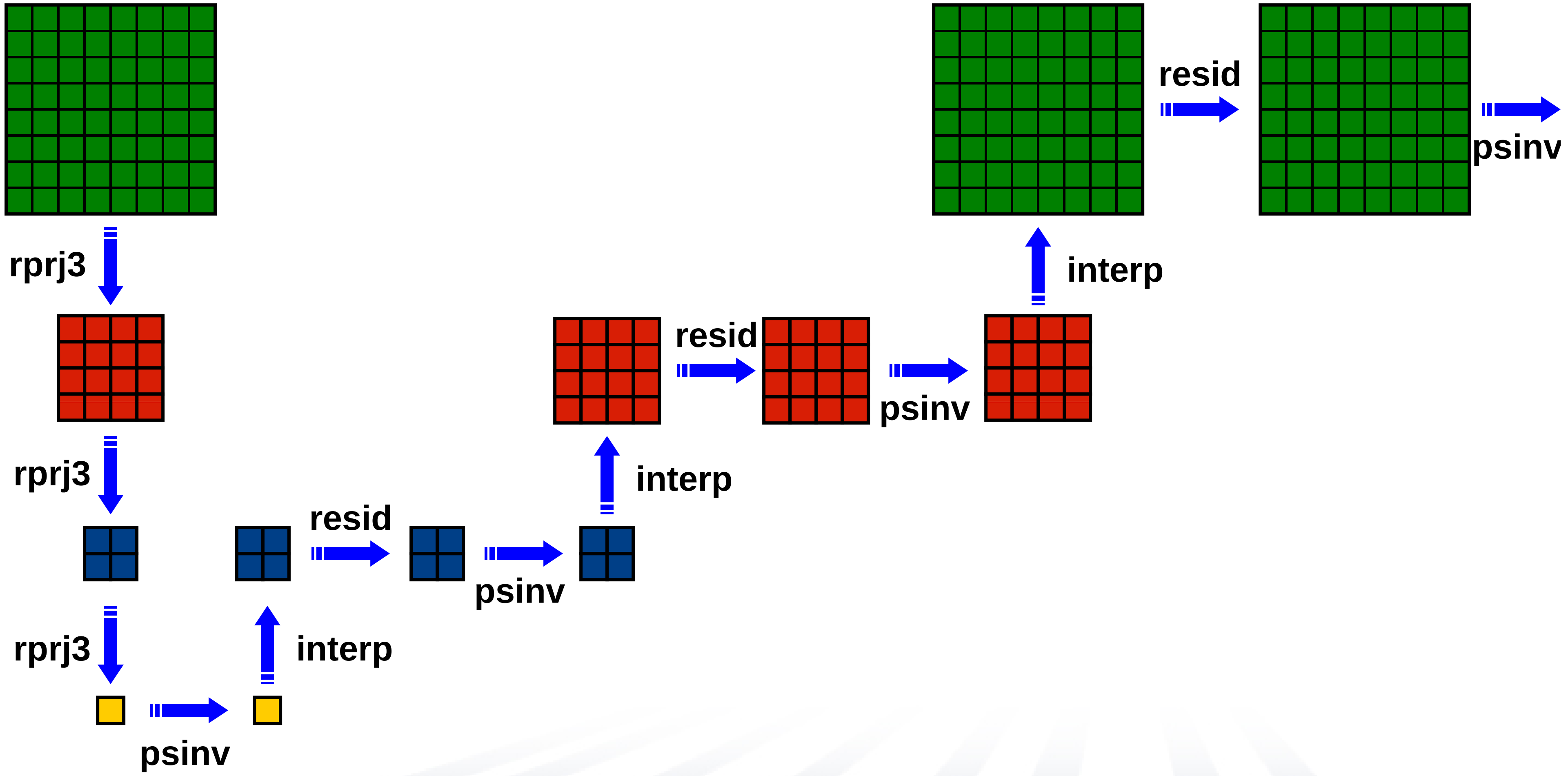
MG's Timed Portion



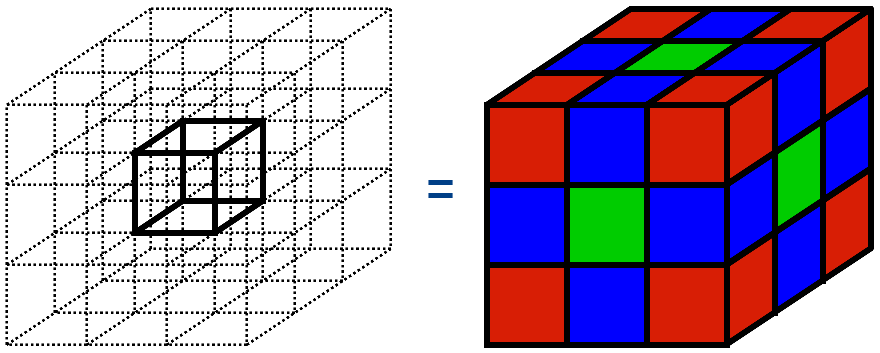
Configurations

S:	32^3	(4 iterations)
W:	64^3	(40 iterations)
A:	256^3	(4 iterations)
B:	256^3	(20 iterations)
C:	512^3	(20 iterations)
D:	1024^3	(50 iterations)

MG’s projection/interpolation cycle



NAS MG: *rprj3* stencil

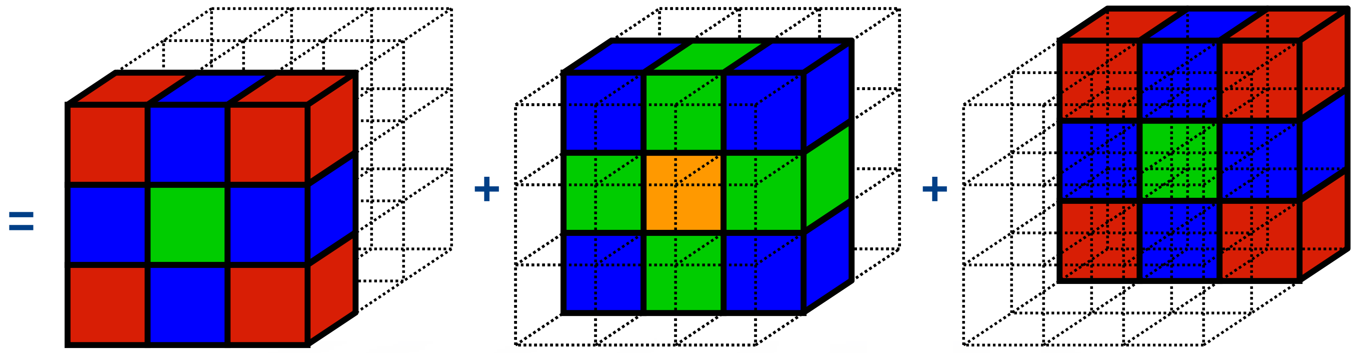


🟧 = w_0

🟩 = w_1

🟦 = w_2

🟥 = w_3



Multigrid: Stencils in Chapel

- Can write them out explicitly, as in Jacobi…

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

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

Multigrid: Stencils in Chapel

- …or, note that a stencil is simply a reduction over a small subarray expression
- Thus, stencils can be written in a “syntactically scalable” way using reductions:

```
def rprj3(S, R) {
  const Stencil: domain(3) = [-1..1, -1..1, -1..1], // 27-points
        w: [0..3] real = (0.5, 0.25, 0.125, 0.0625), // 4 wgts
        w3d = [(i,j,k) in Stencil] w((i!=0) + (j!=0) + (k!=0));

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

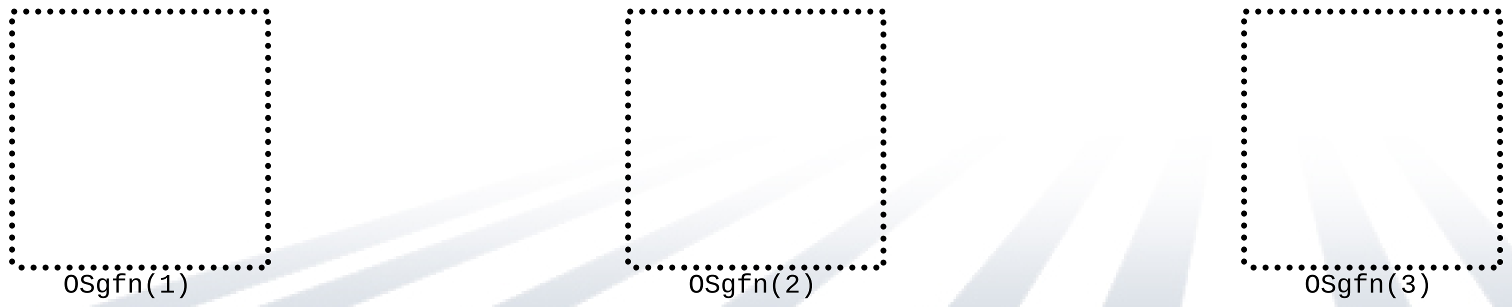
NAS MG *rprj3* stencil in Fortran+MPI

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

 implicit none

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

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

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


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


 implicit none

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

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

 integer i3, i2, i1, buff_len,buff_id

 buff_id = 2 + dir
 buff_len = 0

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

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

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

    else if( dir .eq. +1 ) then

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

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

    endif
 endif

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

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

    else if( dir .eq. +1 ) then

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

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

    endif
 endif

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

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

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

    else if( dir .eq. +1 ) then

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

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

    endif
 endif

 return
 end


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


 implicit none

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

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

 integer buff_id, indx

 integer i3, i2, i1

 buff_id = 3 + dir
 indx = 0

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

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

    else if( dir .eq. +1 ) then

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

    endif
 endif

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

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

    else if( dir .eq. +1 ) then

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

    endif
 endif

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

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

    else if( dir .eq. +1 ) then

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

    endif
 endif

 return
 end


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


 implicit none

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

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

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

 dir = -1

 buff_id = 3 + dir
 buff_len = nm2

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


 dir = +1

 buff_id = 3 + dir
 buff_len = nm2

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

 dir = +1

 buff_id = 2 + dir
 buff_len = 0

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

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

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

 dir = -1

 buff_id = 2 + dir
 buff_len = 0

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

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

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

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

 dir = -1

 buff_id = 3 + dir
 indx = 0

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

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

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


 dir = +1

 buff_id = 3 + dir
 indx = 0

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

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

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

 return
 end

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

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

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

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

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

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

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

Example 3: Fast Multipole Method (FMM)

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

1D array over levels of the hierarchy



Example 3: Fast Multipole Method (FMM)

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

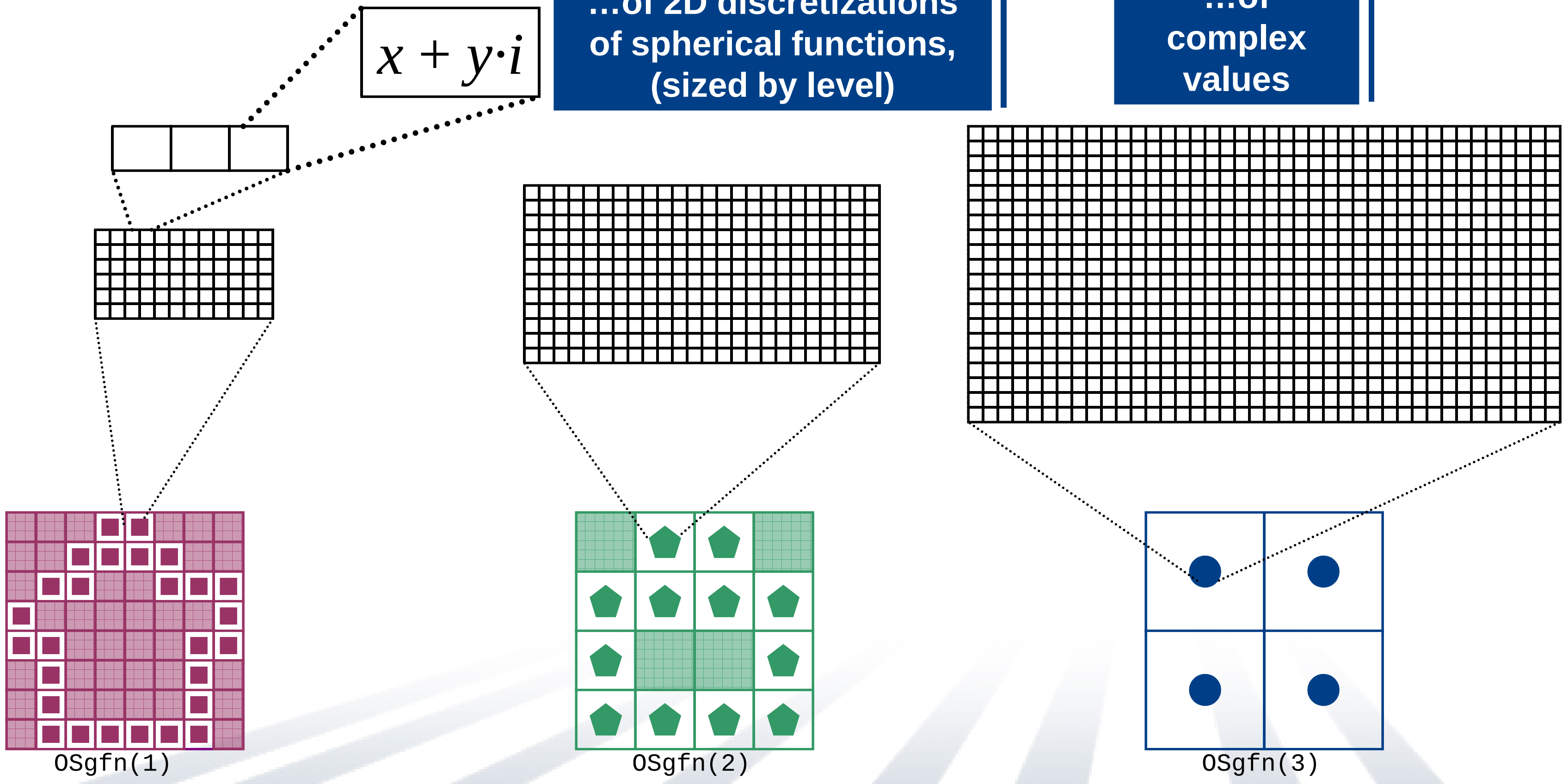
1D array over levels of the hierarchy

…of 3D sparse arrays of cubes (per level)

…of 1D vectors

…of 2D discretizations of spherical functions, (sized by level)

…of complex values

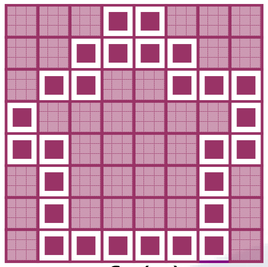


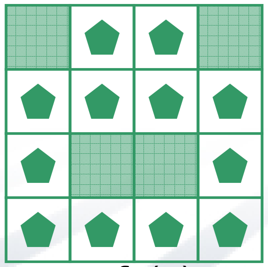
FMM: Supporting Declarations

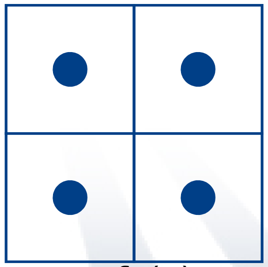
```
var OSgfn, ISgfn: [lvl in Levels] [SpsCubes(lvl)] [Sgfns(lvl)] [1..3] complex;
```

previous definitions:

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

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

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

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

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


OSgfn(1)


OSgfn(2)

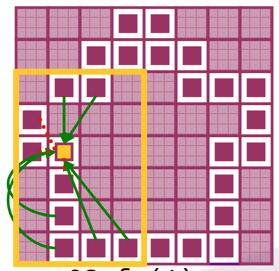

OSgfn(3)

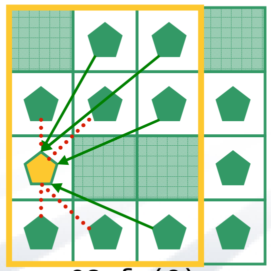
FMM: Computation

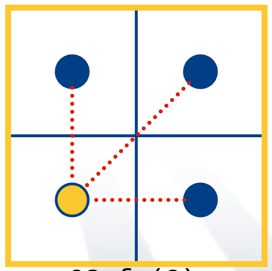
```
var OSgfn, ISgfn: [lvl in Levels] [SpsCubes(lvl)] [Sgfns(lvl)] [1..3] complex;
```

outer-to-inner translation:

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

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


OSgfn(1)


OSgfn(2)


OSgfn(3)

Fast Multipole Method: Summary

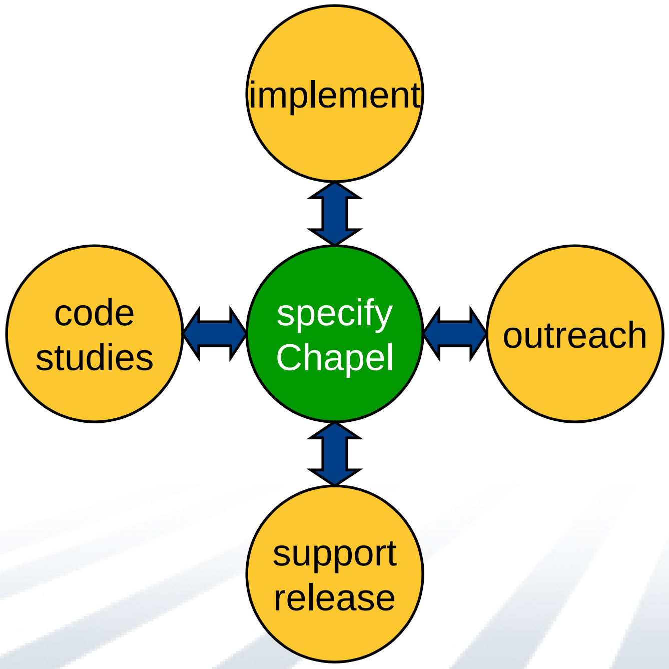
- Chapel code captures structure of data and computation far better than sequential Fortran/C versions (let alone MPI versions of them)
  - cleaner, more succinct, more informative
  - rich domain/array support plays a big role in this
- Code very clear to Boeing engineer familiar with FMM, unfamiliar with Chapel
- Parallelism shifts at different levels of hierarchy
  - Global view and syntactic separation of concerns helps here
  - Imagine writing in a fragmented language
- Yet, I've elided some non-trivial code (data distribution)

Outline

- ✓ Chapel Context
- ✓ Global-view Programming Models
- ✓ Language Overview
- ✓ Example Computations
- ➢ Status, Future Work, Collaborations

Chapel Work

- Chapel Team's Focus:
  - specify Chapel syntax and semantics
  - implement prototype compiler for Chapel
  - code studies of benchmarks, applications, and libraries in Chapel
  - community outreach to inform and learn from users/researchers
  - support users of preliminary releases
  - refine language based on all these activities



Prototype Implementation

- Approach:
  - source-to-source compiler for portability (Chapel-to-C)
  - link against runtime libraries to hide machine details
    - threading layer currently implemented using pthreads
    - communication currently implemented using Berkeley's GASNet
- Status:
  - **base language:** solid, usable (a few gaps remain)
  - **task parallel:** multiple threads, multiple locales
  - **data parallel:** single-threaded, single-locale
  - **performance:** has received little effort (but much planning)
- Current Focus:
  - multi-threaded implementation of data parallel features
  - distributed domains and arrays
  - performance optimizations
  - hope to unveil first performance results at SC08 in Austin this fall
- Early releases to ~40 users at ~20 sites (academic, gov't, industry)

Research Challenges

- Near-term:
  - user-defined distributions
  - zippered parallel iteration
  - index/subdomain optimizations
- Medium-term:
  - memory management policies/mechanisms
  - task scheduling policies
  - tuning for multicore processors
  - unstructured/graph-based codes
  - compiling/optimizing atomic sections (STM)
  - language interoperability
  - parallel I/O
- Longer-term:
  - checkpoint/resiliency mechanisms
  - exotic architectures (GPUs, FPGAs?)
  - hierarchical/heterogeneous notion of locales
  - increased static safety via type system

Chapel Design Philosophies

- A research project…

  …but intentionally broader than an academic project would tend to be
    - due to emphasis on general parallel programming
    - due to the belief that success requires a broad feature set
    - to create a platform for broad community involvement

- Nurture within Cray, then turn over to community
  - currently releasing to small set of “friendly” users
  - hope to do public release in late 2008
  - turn over to community when it can stand on its own

Collaborations

UIUC (Vikram Adve and Rob Bocchino): Software Transactional Memory (STM) over distributed memory (PPoPP `08)

ORNL (David Bernholdt *et al.*): Chapel code studies – Fock matrix computations, MADNESS, Sweep3D, … (HIPS `08)

PNNL (Jarek Nieplocha *et al.*): ARMCI port of comm. layer

CMU (Franz Franchetti): Chapel as portable parallel back-end language for SPIRAL

EPCC (Michele Weiland, Thom Haddow): performance study of single-locale task parallelism

(Your name here?)

Possible Collaboration Areas

- any of the previously-mentioned research topics…
- task parallel concepts
  - implementation using alternate threading packages
  - work-stealing task implementation
- application/benchmark studies
- different back-ends (LLVM? MS CLR?)
- visualizations, algorithm animations
- library support
- tools
  - correctness debugging
  - performance debugging
  - IDE support
- runtime compilation
- (your ideas here…)

Chapel Team

Steve Deitz, Brad Chamberlain

David Iten, Samuel Figueroa, ~~Mary Beth Hribar~~

Questions?

bradc@cray.com

chapel_info@cray.com

http://chapel.cs.washington.edu

ZPL Sidebar

ZPL

ZPL: an array-based data parallel language

Developed by: University of Washington

Timeframe: 1991 – 2003 (can still download today)

Target machines: 1990's HPC parallel platforms

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

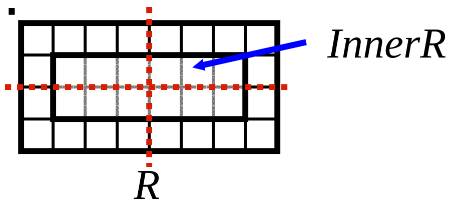
Main concepts:

- abstract machine model: CTA
- regions: first-class index sets
- WYSIWYG performance model

ZPL Concepts: Regions

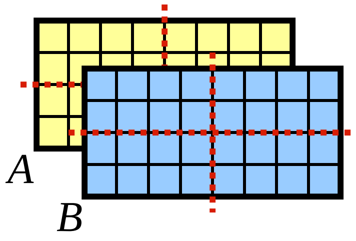
regions: first-class distributed index sets…

```
region R      = [1..m, 1..n];
       InnerR = [2..m-1, 2..n-1];
```



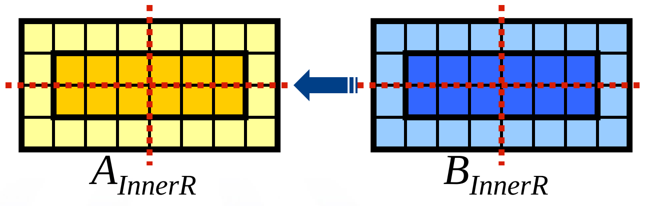
…used to declare distributed arrays…

```
var A, B: [R] double;
```



…and computation over distributed arrays

```
[InnerR] A = B;
```

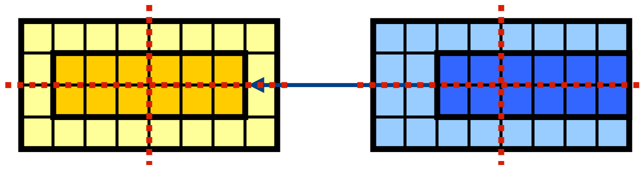


ZPL Concepts: Array Operators

array operators: describe nontrivial array indexing

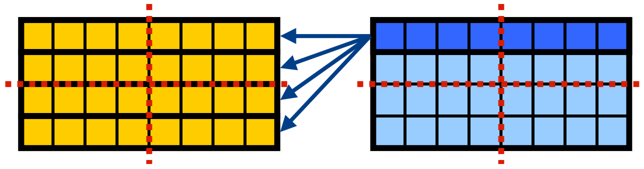
- *at operator* (@): translation

```
[InnerR] A = B@[0,1];
```



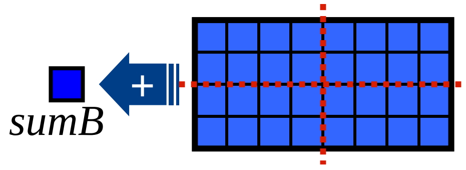
- *flood operator* (>>): replication

```
[R] A = >>[1, 1..n] B;
```



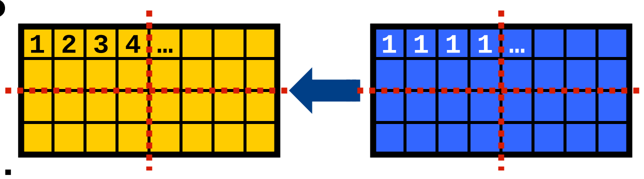
- *reduction operator* (*op*<<): reductions

```
[R] sumB = +<< B;
```



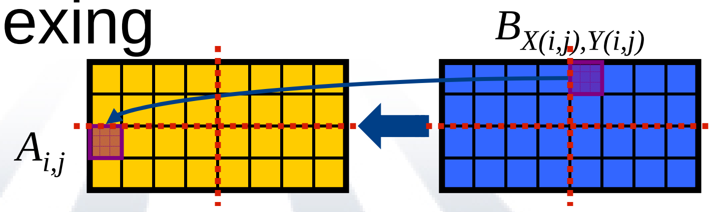
- *scan operator* (*op*||): parallel prefixes

```
[R] A = +|| B;
```



- *remap operator* (#): whole-array indexing

```
[R] A = B#[X,Y];
```



ZPL Concepts: Syntactic Performance Model

```
[InnerR] A = B;
```

No Array Operators ⇒ No Communication

```
[InnerR] A = B@[0,1];
```

At Operator ⇒ Point-to-Point Communication

```
[R] A = >>[1, 1..n] B;
```

Flood Operator ⇒ Broadcast (log-tree) Communication

```
[R] sumB = +<< B;
```

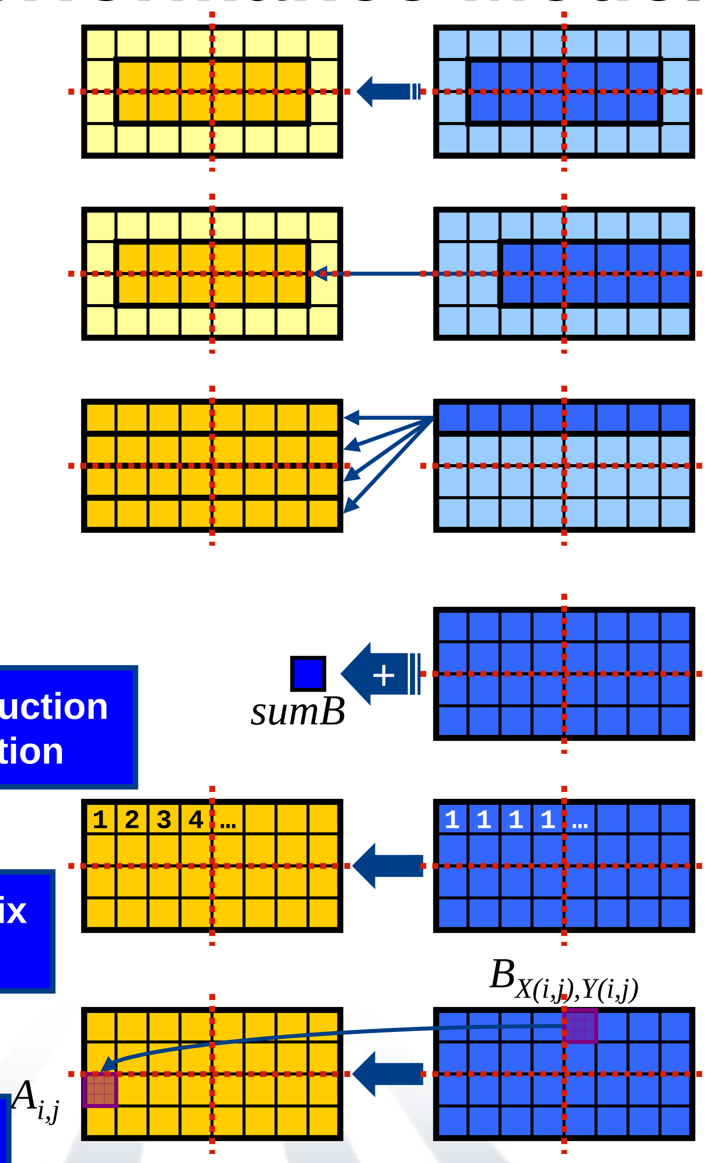
Reduce Operator ⇒ Reduction (log-tree) Communication

```
[R] A = +|| B;
```

Scan Operator ⇒ Parallel-Prefix (log-tree) Communication

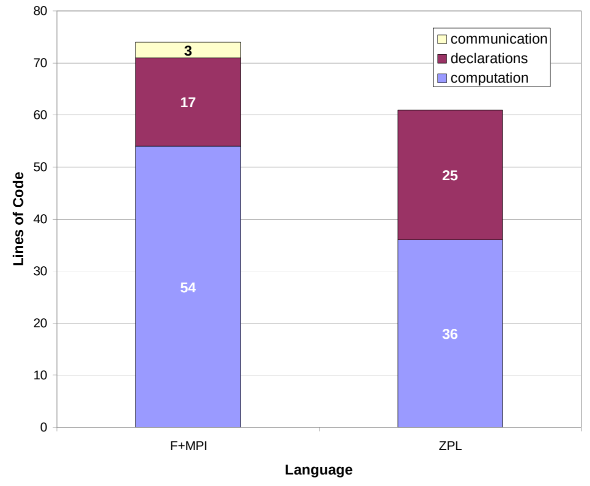
```
[R] A = B#[X,Y];
```

Remap Operator ⇒ Arbitrary (all-to-all) Communication

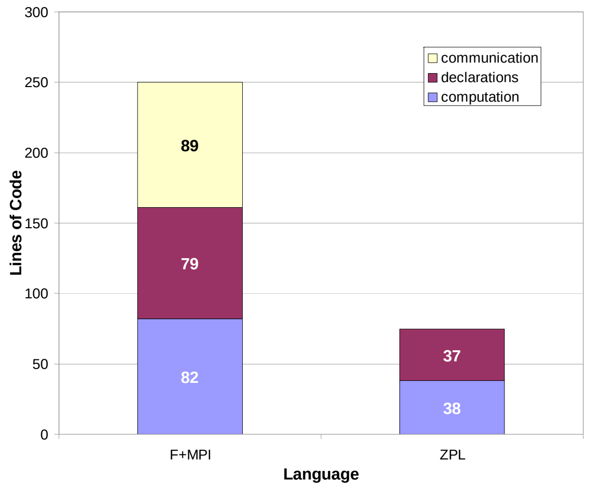


NPB: MPI vs. ZPL Code Size (timed kernels)

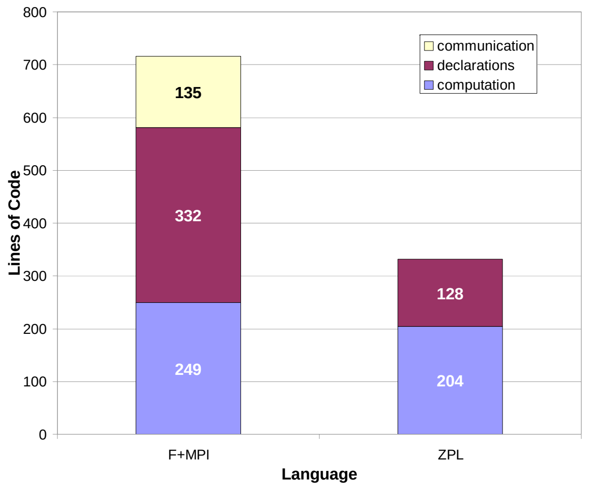
EP



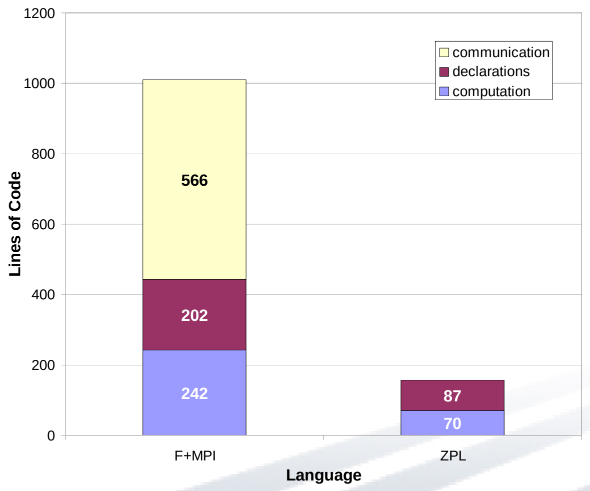
CG



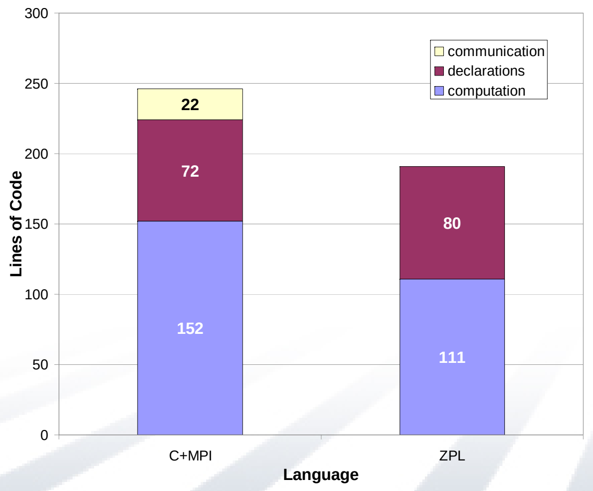
FT



MG

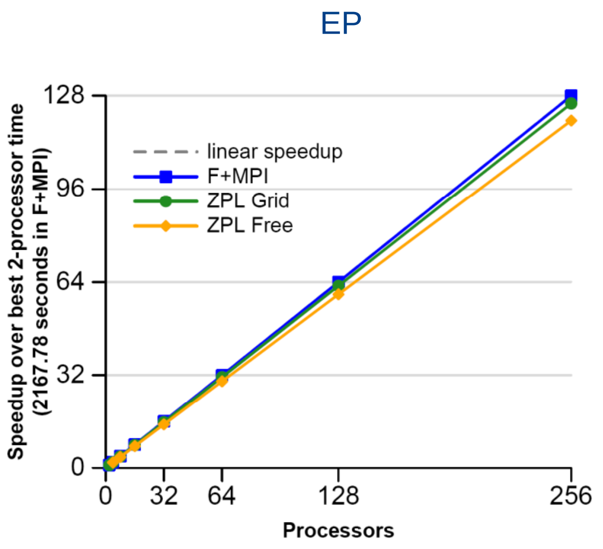


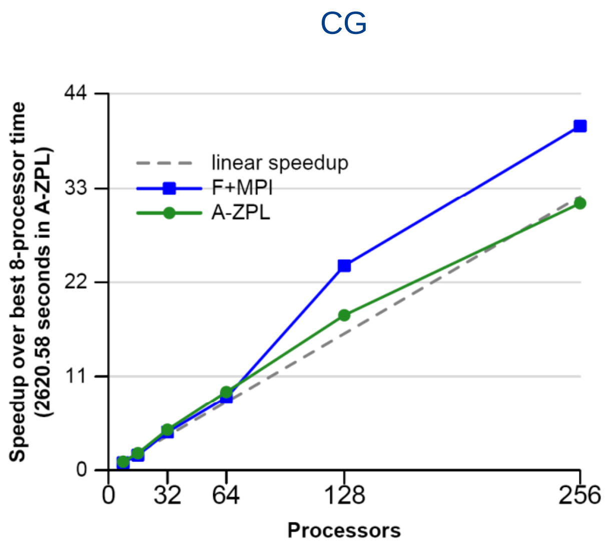
IS

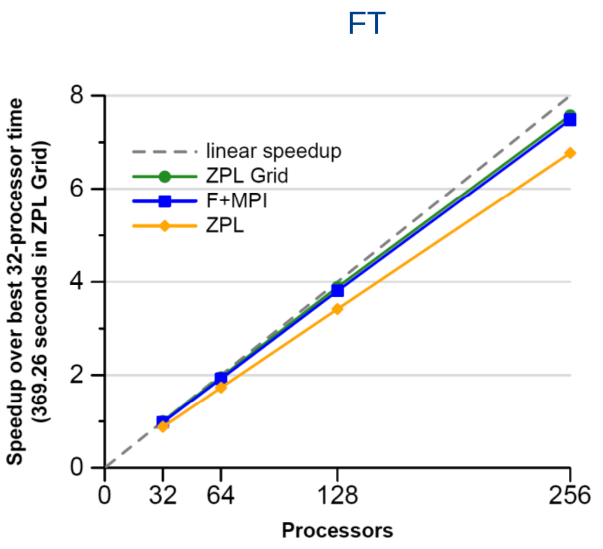


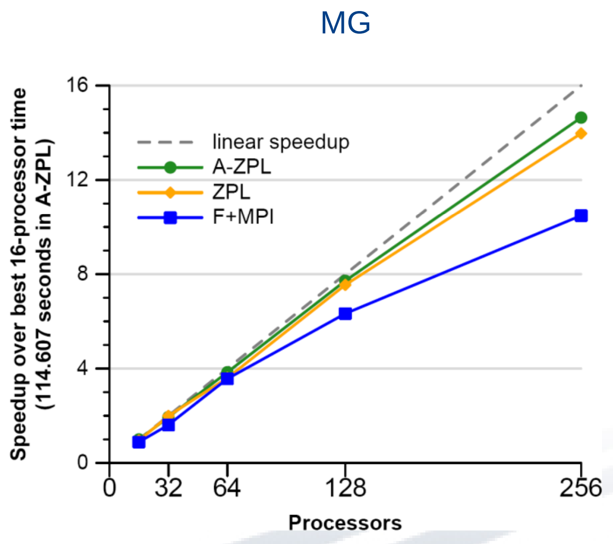
▢	communication
🟪	declarations
🟦	computation

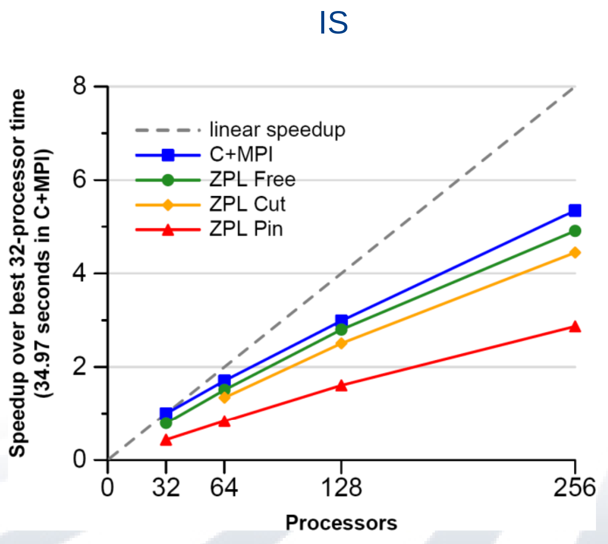
NPB: MPI vs. ZPL Performance











■	C/Fortran + MPI
●	ZPL versions
◆	
▲	

ZPL Summary

+ Global-view programming with syntactic performance model
  - good for the compiler
  - good for the performance-oriented user
+ concise/clean compared to MPI, w/ competitive performance

– only supports a single level of data parallelism
– only supports a small set of distributions
– distinct concepts for sequential and parallel arrays

For more information:

http://cs.washington.edu/research/zpl

zpl-info@cs.washington.edu

HoPL’07 paper about ZPL

ZPL

ZPL strengths

+ syntactic performance model (*e.g.*, communication visible in source)
  - helps user reason about program's parallel implementation
  - helps compiler implement and optimize it
+ global view of data and computation
  - programmer need not think in SPMD
+ implementation-neutral expression of communication
  - permits mapping to best mechanisms for given architecture/level

ZPL weaknesses

– only supports one level of data parallelism; no true task parallelism
  - a consequence of its use of an SPMD execution model

– distinct concepts for parallel and serial arrays

– only supports a small number of built-in distributions

But let's take the lessons from ZPL that we can and keep striving forward… (and from other "failed" 1990's parallel languages as well)

(RETURN)

NAS MG *rprj3* stencil in ZPL

```
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)
```

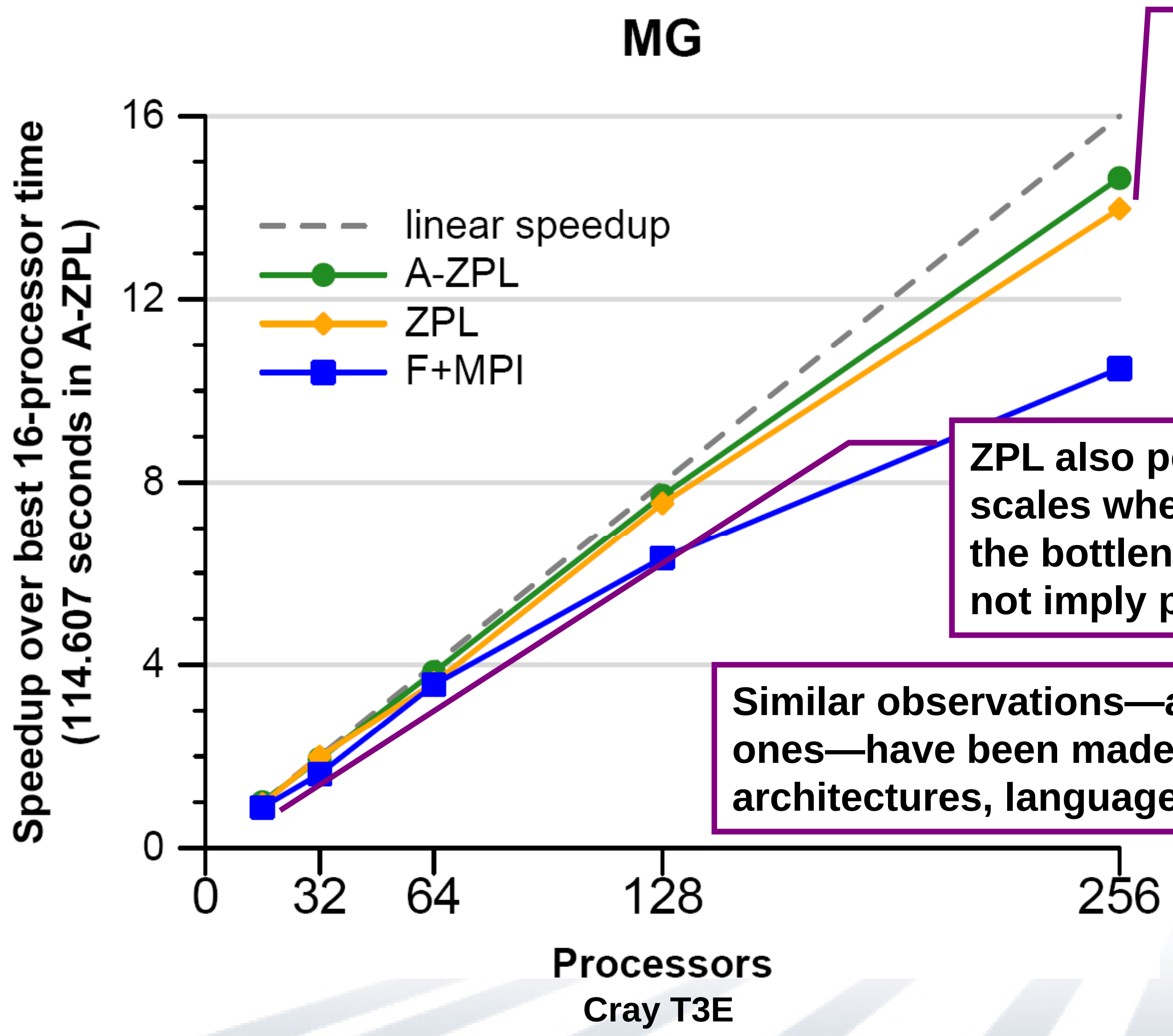
```
 else if( dir .eq. +1 ) then

       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
```

Performance Notes

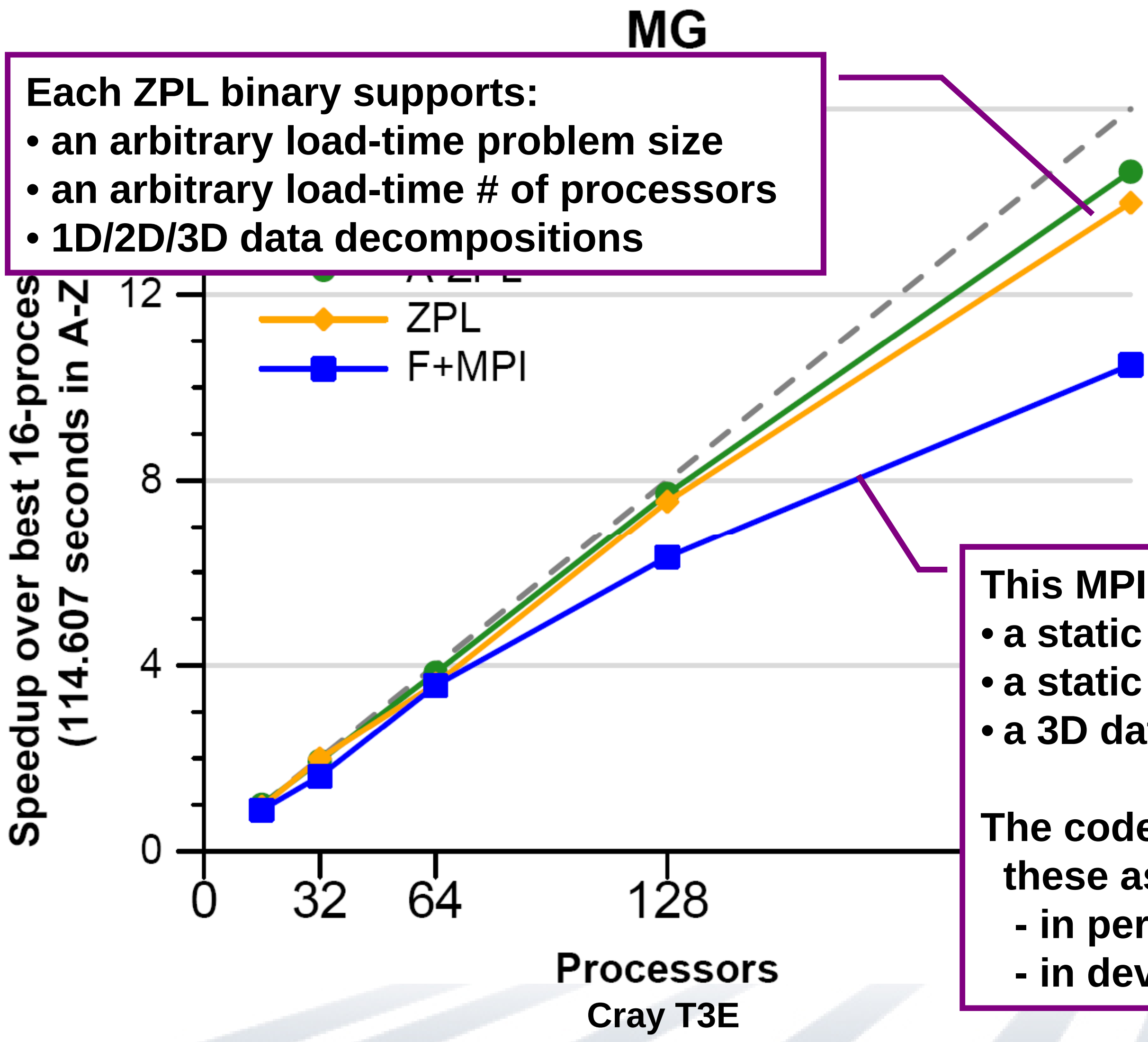


ZPL scales better than MPI since its communication is expressed in an implementation-neutral way; this permits the compiler to use SHMEM on this Cray T3E but MPI on a commodity cluster

ZPL also performs better at smaller scales where communication is not the bottleneck ⇒ new languages need not imply performance sacrifices

Similar observations—and more dramatic ones—have been made using more recent architectures, languages, and benchmarks

Generality Notes

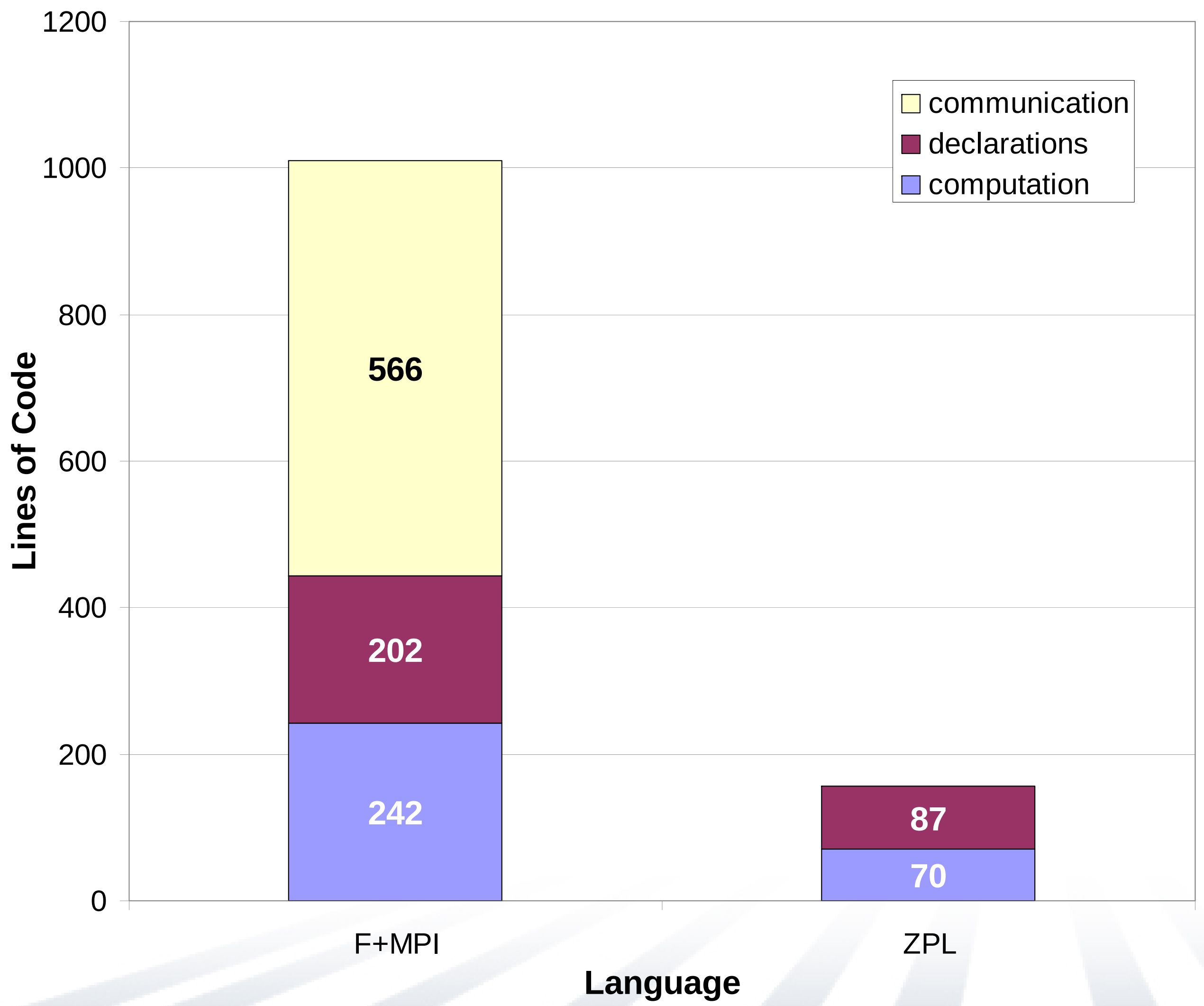


Each ZPL binary supports:
- **an arbitrary load-time problem size**
- **an arbitrary load-time # of processors**
- **1D/2D/3D data decompositions**

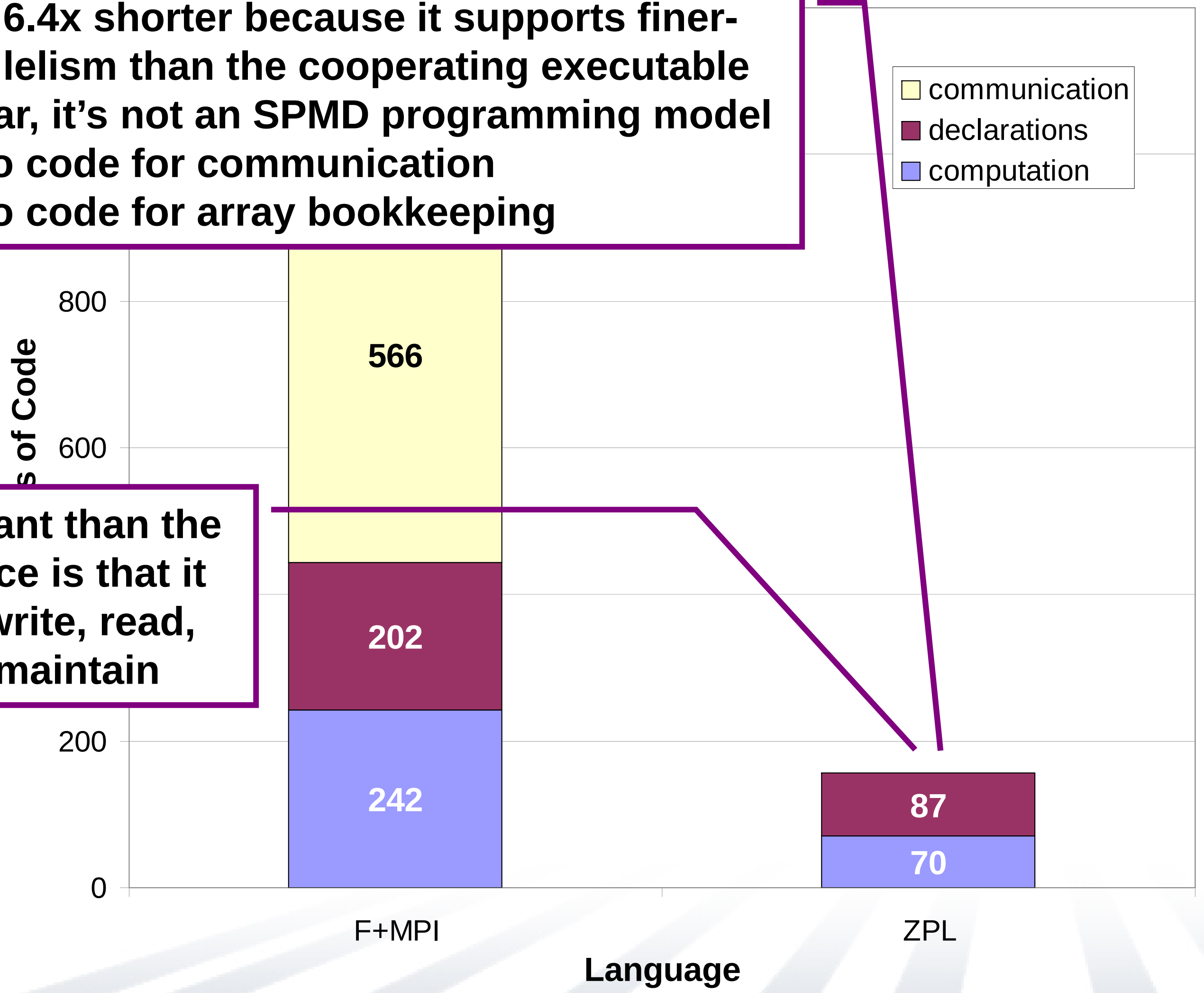
This MPI binary only supports:
- **a static 2*****k*** **problem size**
- **a static 2*****j*** **# of processors**
- **a 3D data decomposition**

The code could be rewritten to relax these assumptions, but at what cost?
- **in performance?**
- **in development effort?**

Code Size Notes



Code Size Notes



- the ZPL is 6.4x shorter because it supports finer-grain parallelism than the cooperating executable
- in particular, it's not an SPMD programming model
  - ⇒ little/no code for communication
  - ⇒ little/no code for array bookkeeping

More important than the size difference is that it is easier to write, read, modify, and maintain

(RETURN)