Introduction to Chapel
A Next-Generation HPC Language

Steve Deitz

Cray Inc.

Download Chapel v0.9 (v1.0 soon)
http://sourceforge.net/projects/chapel/
Compatible with Linux/Unix, Mac OS X, Cygwin

What is Chapel?

- A new parallel language
  - Under development at Cray Inc.
  - Supported through the DARPA HPCS program
- Goals
  - **Improve programmer productivity**
  - Improve the programmability of parallel computers
  - Match or improve performance of MPI/UPC/CAF
  - Provide better portability than MPI/UPC/CAF
  - Improve robustness of parallel codes
  - Support multi-core and multi-node systems

The Chapel Team

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

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

Goals For This Morning

- Introduce you to Chapel with a focus on
  - Task parallelism
  - Data parallelism
  - Multi-locale parallelism
- Answer questions about Chapel Version 0.9 (and 1.0)
- Get your feedback on Chapel
- Point you towards resources to use after today
- Look for collaboration opportunities

Rough Outline

Download Chapel v0.9 (v1.0 soon)
http://sourceforge.net/projects/chapel/
Compatible with Linux/Unix, Mac OS X, Cygwin

1:00 – Welcome
1:15 – Chapel Background
1:35 – Language Basics
2:00 – Task Parallelism
2:30 – *Break*
2:40 – Data Parallelism
3:10 – Locality and Affinity
3:30 – Implementation Overview and Wrap Up

Chapel: Background

Steve Deitz

Cray Inc.

Chapel Settings

- **HPCS**: **H**igh **P**roductivity **C**omputing **S**ystems (DARPA)
  - Goal: Raise HEC user productivity by 10x by 2010

    Productivity = Performance + Programmability + Portability + Robustness
- Phase II: Cray, IBM, Sun (July 2003 – June 2006)
  - Evaluated entire system architecture
  - Three new languages (Chapel, X10, Fortess)
- Phase III: Cray, IBM (July 2006 – 2010)
  - Implement phase II systems
  - Work continues on all three languages

Chapel Productivity Goals

- Improve programmability over current languages
  - Writing parallel codes
  - Reading, changing, porting, tuning, maintaining, ...
- Support performance at least as good as MPI
  - Competitive with MPI on generic clusters
  - Better than MPI on more capable architectures
- Improve portability over current languages
  - As ubiquitous as MPI
  - More portable than OpenMP, UPC, CAF, ...
- Improve robustness via improved semantics
  - Eliminate common error cases
  - Provide better abstractions to help avoid other errors

Chapel Design Concepts

- Support general parallel programming
  - Data, task, and nested parallelism
  - Express all levels of software parallelism
  - Target all levels of hardware parallelism
- *Support global-view abstractions*
- *Support multiple levels of design*
- Allow for control of locality
- Bring mainstream features to parallel languages

Outline

- Concepts, Settings and Goals
- Chapel's Programming Model
  - Fragmented vs. **Global-View**
  - Low-Level vs. High-Level vs. **Multiple Levels**

Fragmented vs. Global-View: Definitions

- Programming model

  The mental model of a programmer

- Fragmented models

  Programmers take point-of-view of a single processor/thread

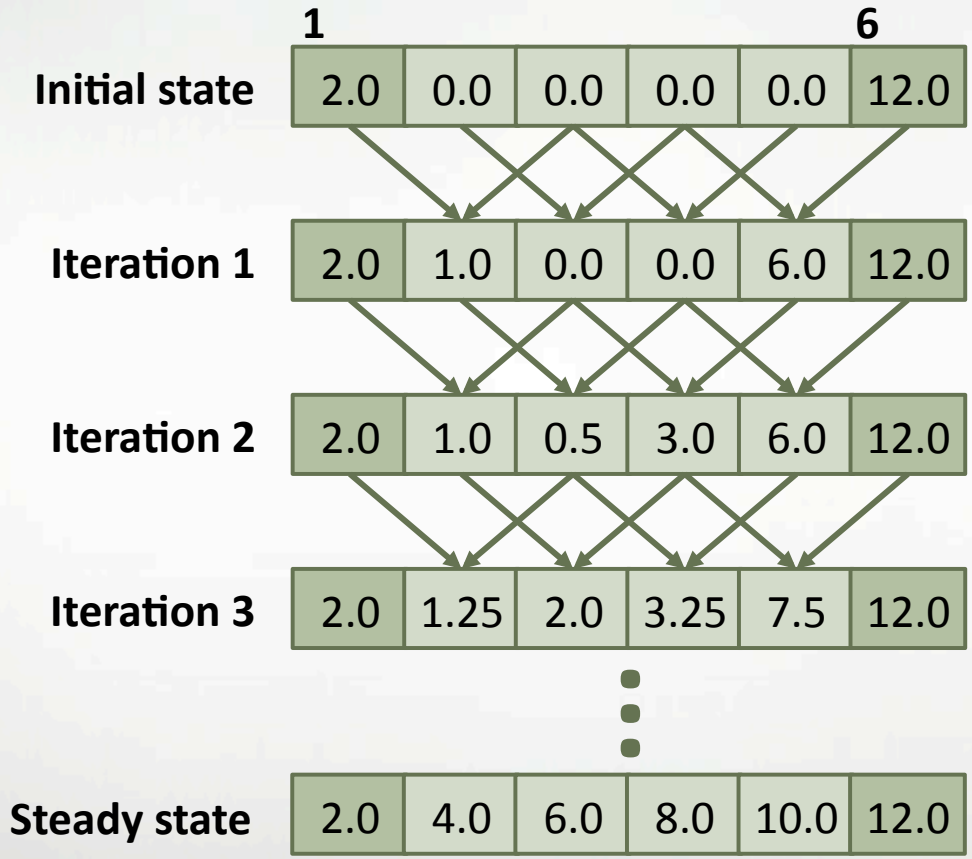
- SPMD models (Single Program, Multiple Data)

  Fragmented models with multiple copies of one program

- Global-view models

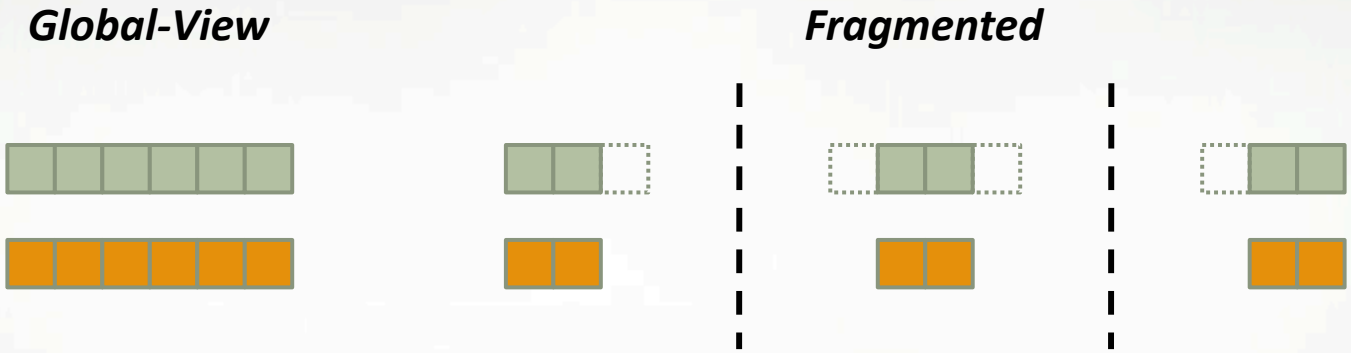
  Programmers write code to describe computation as a whole

3-Point Stencil Example (n=6)



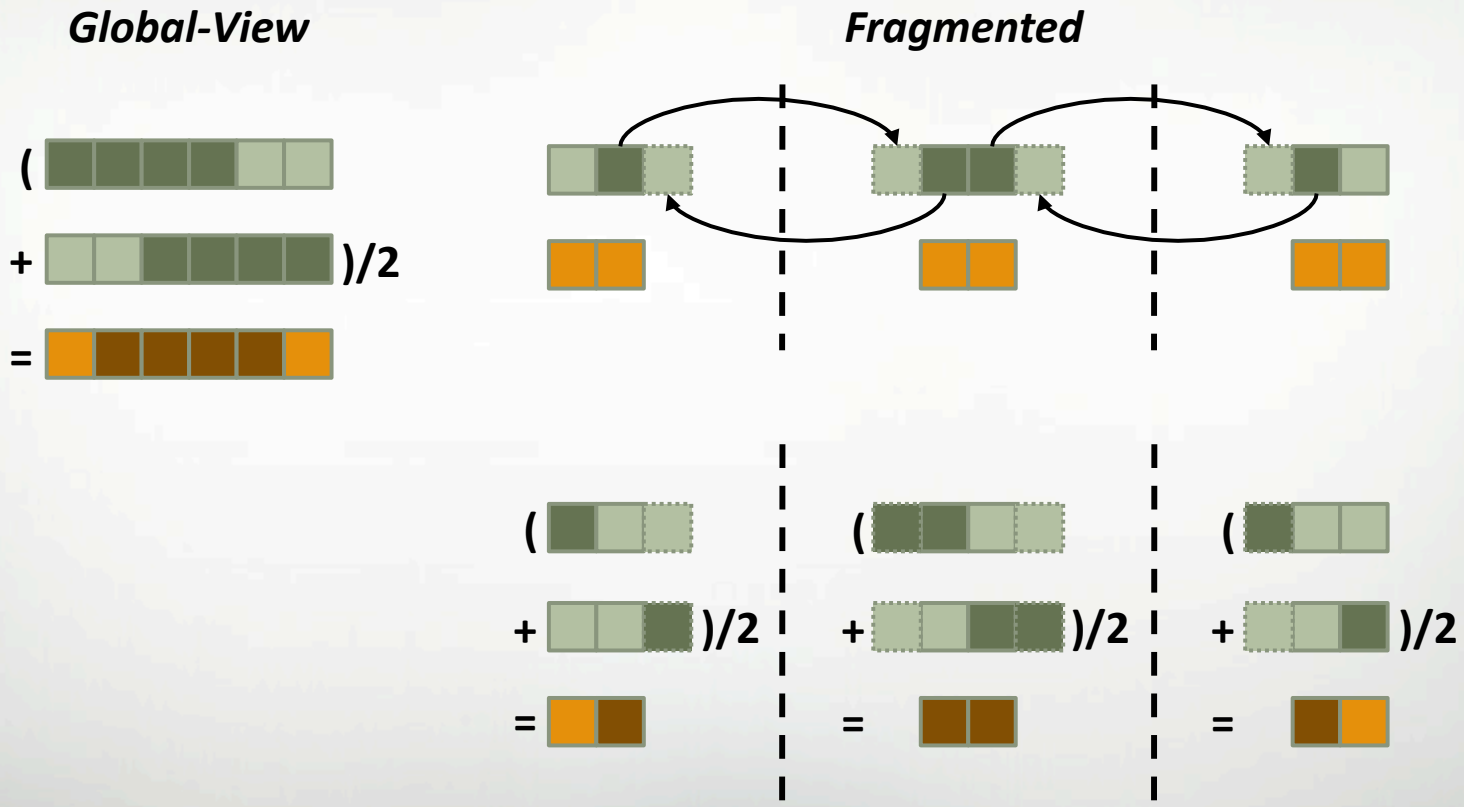
3-Point Stencil Example: Data

Global-View vs. Fragmented Data Declarations



3-Point Stencil Example: Computation

Global-View vs. Fragmented Computation

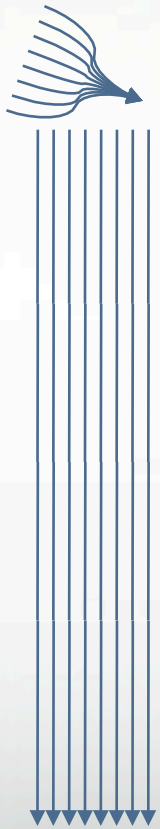


3-Point Stencil Example: Code

Global-View vs. Fragmented Code

Global-View

```
def main() {
  var n = 1000;
  var A, B: [1..n] real;

  forall i in 2..n-1 do
    B(i) = (A(i-1)+A(i+1))/2;
}
```

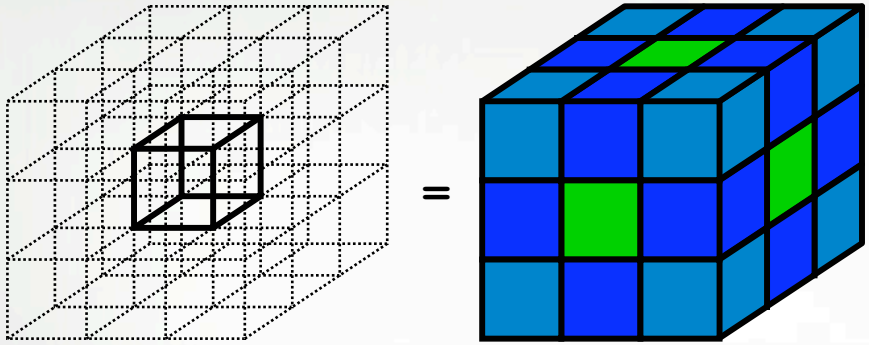


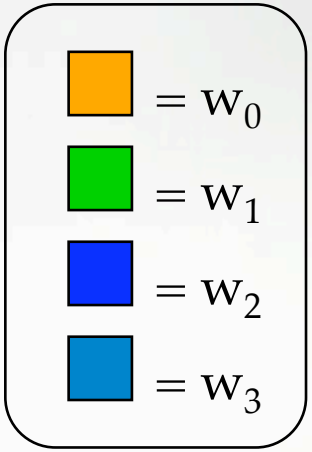
Fragmented

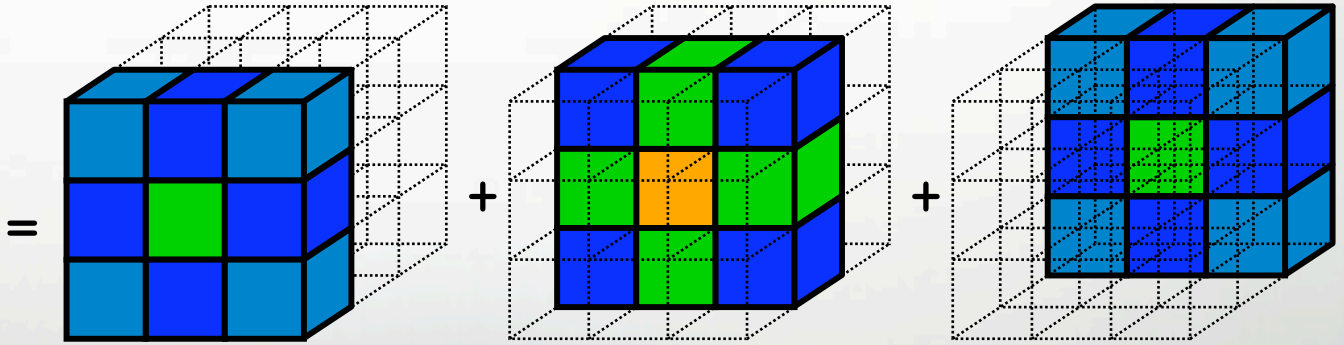
Assumes p divides n

```
def main() {
  var n = 1000;
  var me = commRank(), p = commSize(),
    myN = n/p, myLo = 1, myHi = myN;
  var A, B: [0..myN+1] real;

  if me < p {
    send(me+1, A(myN));
    recv(me+1, A(myN+1));
  } else myHi = myN-1;
  if me > 1 {
    send(me-1, A(1));
    recv(me-1, A(0));
  } else myLo = 2;
  for i in myLo..myHi do
    B(i) = (A(i-1)+A(i+1))/2;
}
```

NAS MG Stencil







NAS MG Stencil in Fortran + MPI

```
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'  subroutine
     comm3(u,n1,n2,n3,kk)

 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  i2=2,n2-1
             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

    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( i1,n2-1,i3)
       enddo
    enddo

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

 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
             indx = indx + 1
             u(i1,n2,i3) = buff(indx,
     buff_id )
          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,n3-1)
          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 = 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

    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

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

    else 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

    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 = 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

 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( 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

 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

 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

 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

 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

 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

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

 endif

 return
 end


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

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

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

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

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

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

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

NAS MG Stencil in Chapel

```
def rprj3(S, R) {
  const Stencil = [-1..1, -1..1, -1..1],
        W: [0..3] real = (0.5, 0.25, 0.125, 0.0625),
        W3D = [(i,j,k) in Stencil] W((i!=0)+(j!=0)+(k!=0));

  forall inds in S.domain do
    S(inds) =
      + reduce [offset in Stencil] (W3D(offset) *
                                    R(inds + offset*R.stride));
}
```

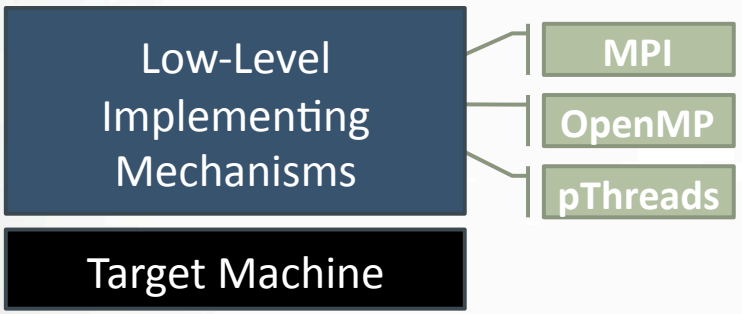
Previous work shows performance is still possible:

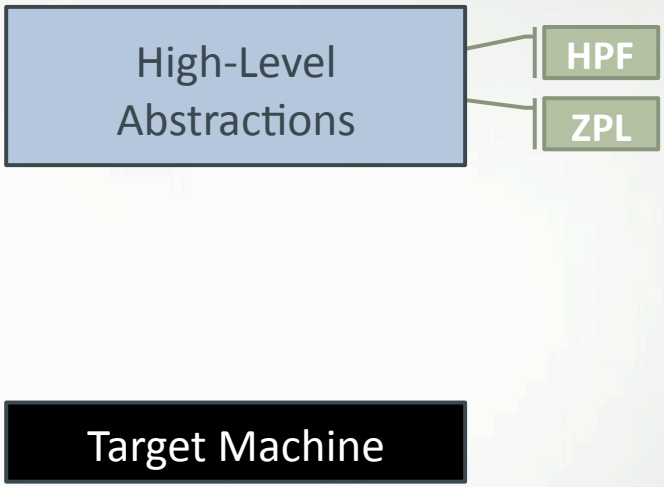
B. L. Chamberlain, S. J. Deitz, and L. Snyder. *A comparative study of the NAS MG benchmark across parallel languages and architectures*. In Proceedings of the ACM Conference on Supercomputing, 2000.

Summary of Current Programming Systems

	System	Data Model	Compute Model
Communication Libraries	MPI/MPI-2	Fragmented	Fragmented
	SHMEM	Fragmented	Fragmented
	ARMCI	Fragmented	Fragmented
	GASNet	Fragmented	Fragmented
Shared Memory	OpenMP, pThreads	Global-View (trivially)	Global-View (trivially)
PGAS Languages	Co-Array Fortran	Fragmented	Fragmented
	UPC	Global-View	Fragmented
	Titanium	Fragmented	Fragmented
HPCS Languages	Chapel	Global-View	Global-View
	X10 (IBM)	Global-View	Global-View
	Fortress (Sun)	Global-View	Global-View

Low-Level vs. High-Level Programming

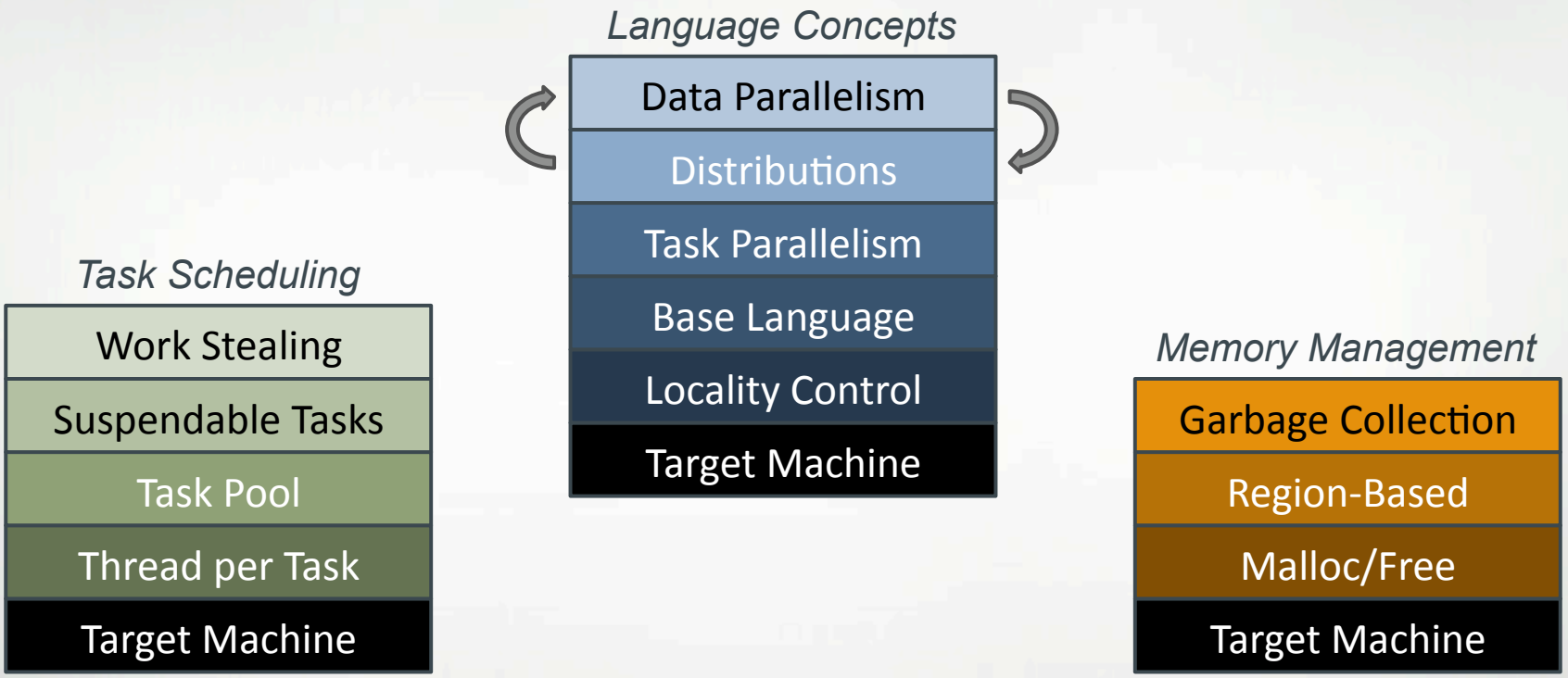




"Why is everything so difficult?"

"Why can't I optimize this?"

Multiple Levels of Design



Questions?

- Concepts, Settings and Goals
- Chapel's Programming Model
  - Fragmented vs. **Global-View**
  - Low-Level vs. High-Level vs. **Multiple Levels**

Chapel: Language Basics

Steve Deitz

Cray Inc.

The HelloWorld Program

- Fast Prototyping

```
writeln("hello, world");
```

- Structured Programming

```
def main() {
  writeln("hello, world");
}
```

- Production-Level

```
module HelloWorld {
  def main() {
    writeln("hello, world");
  }
}
```

Chapel Stereotypes and Generalizations

- Syntax
  - Basics from C, C#, C++, Java, Ada, Perl, ...
  - Specifics from many other languages
- Semantics
  - Imperative, block-structured, array paradigms
  - Optional object-oriented programming (OOOP)
  - Static typing for performance and safety
  - Elided types for convenience and generic coding
- Features
  - No pointers and few references
  - No compiler-inserted array temporaries

Chapel Influences

ZPL, HPF: data parallelism, index sets, distributed arrays

CRAY MTA C/Fortran: task parallelism, synchronization

CLU, Ruby, Python: iterators

ML, Scala, Matlab, Perl, Python, C#: latent types

Java, C#: OOP, type safety

C++: generic programming/templates

Outline

- High-Level Comments
- Elementary Concepts
  - Lexical structure
  - Types, variables, and constants
  - Input and output
- Data Structures and Control
- Miscellaneous

Lexical Structure

- Comments

```
/* standard
   C-style */
// standard C++ style
```

- Identifiers
  - Composed of A-Z, a-z, 0-9, _, and $
  - Starting with A-Z, a-z, and _
- Case-sensitive
- Whitespace-aware
  - Composed of spaces, tabs, and linefeeds
  - Separates tokens and ends //-comments

Primitive Types

Type	Description	Default	Bit Width	Supported Bit Widths
bool	logical value	false	impl-dep	8, 16, 32, 64
int	signed integer	0	32	8, 16, 32, 64
uint	unsigned integer	0	32	8, 16, 32, 64
real	real floating point	0.0	64	32, 64
imag	imaginary floating point	0.0i	64	32, 64
complex	complex floating points	0.0 + 0.0i	128	64, 128
string	character string	""	NA	NA

- Syntax

```
primitive-type:
  type-name [( bit-width )]
```

- Examples

```
int(64)   // 64-bit int
real(32)  // 32-bit real
uint      // 32-bit uint
```

Variables, Constants, and Parameters

- Syntax

```
declaration:
  var identifier [: type] [= init-expr]
  const identifier [: type] [= init-expr]
  param identifier [: type] [= init-expr]
```

- Semantics
  - Const-ness: not, at runtime, at compile-time
  - Omitted *type*, type is inferred from *init-expr*
  - Omitted *init-expr*, value is assigned default for type
- Examples

```
var count: int;
const pi: real = 3.14159;
param debug = true;
```

Config Declarations

- Syntax

```
config-declaration:
  config declaration
```

- Semantics
  - Supports command-line overrides
  - Requires global-scope declaration
- Examples

```
config param intSize = 32;
config const start: int(intSize) = 1;
config var epsilon = 0.01;
```

```
chpl -sintSize=16 -o a.out myProgram.chpl
a.out --start=2 --epsilon=0.001;
```

Input and Output

- Input
  - read(expr-list): reads values into the arguments
  - read(type-list): returns values read of given types
  - readln variant: also reads through new line
- Output
  - write(expr-list): writes arguments
  - writeln variant: also writes new line
- Support for arbitrary types (including user-defined)
- File and string I/O via method variants of the above

Outline

- High-Level Comments
- Elementary Concepts
- Data Structures and Control
  - Ranges
  - Arrays
  - For loops
  - Traditional constructs
- Miscellaneous

Range Values

- Syntax

```
range-expr:
  [low] .. [high] [by stride]
```

- Semantics
  - Regular sequence of integers

    stride > 0: *low*, *low*+*stride*, *low*+2**stride*, ... ≤ *high*

    stride < 0: *high*, *high*+*stride*, *high*+2**stride*, ... ≥ *low*
  - Default *stride* = 1, default *low* or *high* is unbounded
- Examples

```
1..6 by 2       // 1, 3, 5
1..6 by -1      // 6, 5, 4, 3, 2, 1
3.. by 3        // 3, 6, 9, 12, ...
```

Array Types

- Syntax

```
array-type:
  [ index-set-expr ] type
```

- Semantics
  - Stores an element of *type* for each index in set
- Examples

```
var A: [1..3] int,          // 3-element array of ints
    B: [1..3, 1..5] real, // 2D array of reals
    C: [1..3][1..5] real; // array of arrays of reals
```

Much more on arrays in data parallelism part

For Loops

- Syntax

```
for-loop:
  for index-expr in iterator-expr { stmt-list }
```

- Semantics
  - Executes loop body once per loop iteration
  - Indices in index-expr are new variables
- Examples

```
var A: [1..3] string = ("DO", "RE", "MI");

for i in 1..3 do write(A(i));        // DOREMI
for a in A { a += "LA"; write(a); } // DOLARELAMILA
```

Zipper "()" and Tensor "[]" Iteration

- Syntax

```
tensor-for-loop:
  for index-expr in [ iterator-expr-list ] { stmt-list }

zipper-for-loop:
  for index-expr in ( iterator-expr-list ) { stmt-list }
```

- Semantics
  - Tensor iteration is over all pairs of yielded indices
  - Zipper iteration is over all yielded indices pair-wise
- Examples

```
for i in [1..2, 1..2] do  // (1,1), (1,2), (2,1), (2,2)

for i in (1..2, 1..2) do  // (1,1), (2,2)
```

Traditional Control

- Conditional statements

```
if cond then computeA() else computeB();
```

- While loops

```
while cond {
  compute();
}
```

- Select statements

```
select key {
  when value1 do compute1();
  when value2 do compute2();
  otherwise compute3();
}
```

Outline

- High-Level Comments
- Elementary Concepts
- Data Structures and Control
- Miscellaneous
  - Functions and iterators
  - Records and classes
  - Generics

Function Examples

- Example to compute the area of a circle

```
def area(radius: real)
  return 3.14 * radius**2;

writeln(area(2.0));   // 12.56
```

- Example of function arguments

```
def writeCoord(x: real = 0.0, y: real = 0.0) {
  writeln("(", x, ", ", y, ")");
}

writeCoord(2.0);      // (2.0, 0.0)
writeCoord(y=2.0);    // (0.0, 2.0)
```

What is an Iterator?

- An abstraction for loop control
  - Yields (returns) indices for each iteration
  - Otherwise, like a function
- Example

```
def string_chars(s: string) {
  var i = 1, limit = length(s);
  while i <= limit {
    yield s.substring(i);
    i += 1;
  }
}

for c in string_chars(s) do ...
```

Iterator Advantages

- Separation of concerns
  - Loop logic is abstracted from computation
- Efficient implementations
  - When the values cannot be pre-computed
    - Memory is insufficient
    - Infinite or cyclic
    - Side effects
  - When not all of the values need to be used

Records

- User-defined data structures
  - Contain variable definitions (fields)
  - Contain function definitions (methods)
  - Value-semantics (assignment copies fields)
  - Similar to C++ classes
- Example

```
record circle { var x, y, radius: real; }
var c1, c2: circle;
c1.x = 1.0; c1.y = 1.0; c1.radius = 2.0;
c2 = c1; // copy of value
```

Classes

- Reference-based records
  - Reference-semantics (assignment aliases)
  - Dynamic allocation
  - OOP-capable
  - Similar to Java classes
- Example

```
class circle { var x, y, radius: real; }
var c1, c2: circle;
c1 = new circle(x=1.0, y=1.0, radius=2.0);
c2 = c1; // c2 is an alias of c1
```

Method Examples

Methods are functions associated to data.

```
def circle.area()
  return 3.14 * this.radius**2;

writeln(c1.area());
```

note: this is implicit

Methods can be defined for any type.

```
def int.square
  return this**2;

writeln(5.square);
```

(parentheses optional)

Generic Functions

Generic functions are replicated for each unique call site. They can be defined by explicit type and param arguments:

```
def foo(type t, x: t) { ...

def bar(param bitWidth, x: int(bitWidth)) { ...
```

Or simply by eliding an argument type (or type part):

```
def goo(x, y) { ...

def sort(A: []) { ...
```

Generic Types

Generic types are replicated for each unique instantiation. They can be defined by explicit type and param fields:

```
class Table { param numFields: int; ...

class Matrix { type eltType; ...
```

Or simply by eliding a field type (or type part):

```
record Triple { var x, y, z; }
```

Questions?

- High-Level Comments
- Elementary Concepts
  - Lexical structure
  - Types, variables, and constants
  - Input and output
- Data Structures and Control
  - Ranges
  - Arrays
  - For loops
  - Traditional constructs
- Miscellaneous
  - Functions and iterators
  - Records and classes
  - Generics

Chapel: Task Parallelism

Steve Deitz

Cray Inc.

Outline

- Primitive Task-Parallel Constructs
  - The **begin** statement
  - The **sync** types
- Structured Task-Parallel Constructs
- Atomic Transactions and Memory Consistency
- Implementation Notes and Examples

Unstructured Task Creation: Begin

- Syntax

```
begin-stmt:
  begin stmt
```

- Semantics
  - Creates a concurrent task to execute *stmt*
  - Control continues immediately (no join)
- Example

```
begin writeln("hello world");
writeln("good bye");
```

- Possible output

```
hello world
good bye
```

```
good bye
hello world
```

Synchronization via Sync-Types

- Syntax

```
sync-type:
  sync type
```

- Semantics
  - Default read blocks until written (until "full")
  - Default write blocks until read (until "empty")
- Examples: Critical sections and futures

```
var lock$: sync bool;

lock$ = true;
critical();
lock$;
```

```
var future$: sync real;

begin future$ = compute();
computeSomethingElse();
useComputeResults(future$);
```

Sync-Type Methods

- `readFE():t` wait until full, leave empty, return value
- `readFF():t` wait until full, leave full, return value
- `readXX():t` non-blocking, return value
- `writeEF(v:t)` wait until empty, leave full, set value to `v`
- `writeFF(v:t)` wait until full, leave full, set value to `v`
- `writeXF(v:t)` non-blocking, leave full, set value to `v`
- `reset()` non-blocking, leave empty, reset value
- `isFull: bool` non-blocking, return true if full else false

- Defaults – read: `readFE`, write: `writeEF`

Outline

- Primitive Task-Parallel Constructs
- Structured Task-Parallel Constructs
  - The **cobegin** statement
  - The **coforall** loop
  - The **sync** statement
  - The **serial** statement
- Atomic Transactions and Memory Consistency
- Implementation Notes and Examples

Structured Task Invocation: Cobegin

- Syntax

```
cobegin-stmt:
  cobegin { stmt-list }
```

- Semantics
  - Invokes a concurrent task for each listed *stmt*
  - Control waits to continue – implicit join
- Example

```
cobegin {
  consumer(1);
  consumer(2);
  producer();
}
```

Cobegin is Unnecessary

Any cobegin statement

```
cobegin {
  stmt1();
  stmt2();
  stmt3();
}
```

can be rewritten in terms of begin statements

```
var s1$, s2$, s3$: sync bool;
begin { stmt1(); s1$ = true; }
begin { stmt2(); s2$ = true; }
begin { stmt3(); s3$ = true; }
s1$; s2$; s3$;
```

but the compiler may miss out on optimizations.

A "Cobegin" Loop: Coforall

- Syntax

```
coforall-loop:
  coforall index-expr in iterator-expr { stmt }
```

- Semantics
  - Loops over *iterator-expr* invoking concurrent tasks
  - Control waits to continue – implicit join
- Example

```
begin producer();
coforall i in 1..numConsumers {
  consumer(i);
}
```

- Note: Like cobegin, coforall is unnecessary

Usage of Begin, Cobegin, and Coforall

- Use begin when
  - Creating tasks with unbounded lifetimes
  - Load balancing needs to be dynamic
  - Cobegin is insufficient to structure the tasks
- Use cobegin when
  - Invoking a fixed # of tasks
  - The tasks have bounded lifetimes
  - Load balancing is not an issue
- Use coforall when
  - Cobegin applies but the # of tasks is dynamic

Structuring Sub-Tasks: Sync-Statements

- Syntax

```
sync-statement:
  sync stmt
```

- Semantics
  - Executes *stmt*
  - Waits on all *dynamically-encountered* begins
- Example

```
sync {
  for i in 1..numConsumers {
    begin consumer(i);
  }
  producer();
}
```

Program Termination and Sync-Statements

Where the cobegin statement is static,

```
cobegin {
  call1();
  call2();
}
```

the sync statement is dynamic.

```
sync {
  begin call1();
  begin call2();
}
```

Program termination is defined by an implicit sync.

```
sync main();
```

Limiting Concurrency: Serial

- Syntax

```
serial-statement:
  serial expr { stmt }
```

- Semantics
  - Evaluates *expr* and then executes *stmt*
  - Squelches dynamically-encountered concurrency
- Example

```
def search(i: int) {
  // search node i
  serial i > 8 do cobegin {
    search(i*2);
    search(i*2+1);
  }
}
```

Outline

- Primitive Task-Parallel Constructs
- Structured Task-Parallel Constructs
- Atomic Transactions and Memory Consistency
  - The **atomic** statement
  - Races and memory consistency
- Implementation Notes and Examples

Atomic Transactions (Unimplemented)

- Syntax

```
atomic-statement:
  atomic stmt
```

- Semantics
  - Executes stmt so it appears as a single operation
  - No other task sees a partial result
- Example

```
atomic A(i) = A(i) + 1;
```

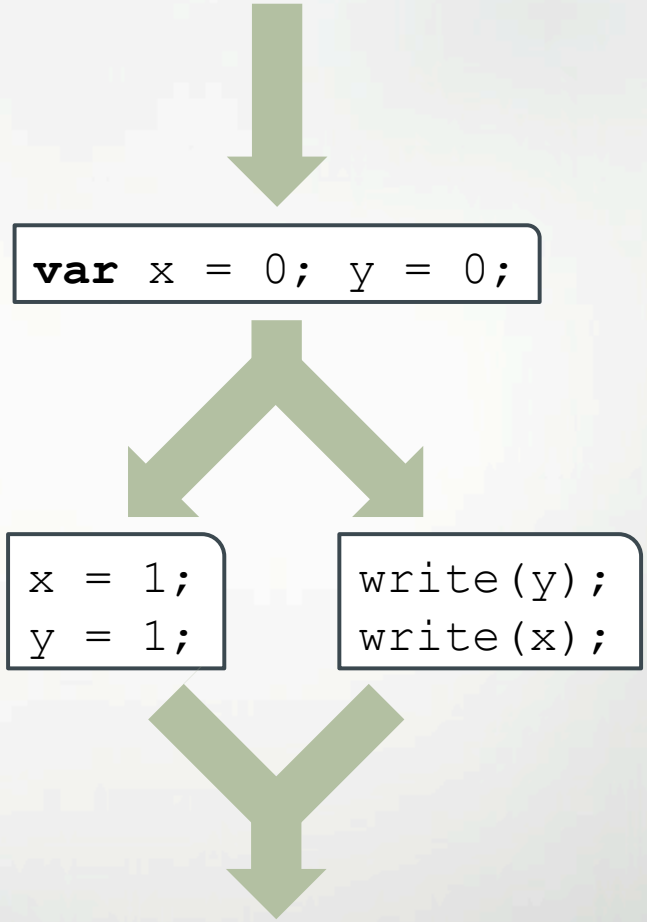
```
atomic {
  newNode.next = node;
  newNode.prev = node.prev;
  node.prev.next = newNode;
  node.prev = newNode;
}
```

Races and Memory Consistency

- Example

```
var x = 0, y = 0;
cobegin {
  { x = 1; y = 1; }
  { write(y); write(x); }
}
```

- Expected Outputs
  - 11
  - 01
  - 00
- What about 10?



Data-Race-Free Programs (The Small Print)

A program without races is sequentially consistent.

> A multi-processing system has sequential consistency if "*the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*" – Leslie Lamport

The behavior of a program with races is undefined.

Synchronization is achieved in two ways:

- By reading or writing sync (or single) variables
- By executing atomic statements

Outline

- Primitive Task-Parallel Constructs
- Structured Task-Parallel Constructs
- Atomic Transactions and Memory Consistency
- Implementation Notes and Examples
  - Using pThreads
  - Quick sort example

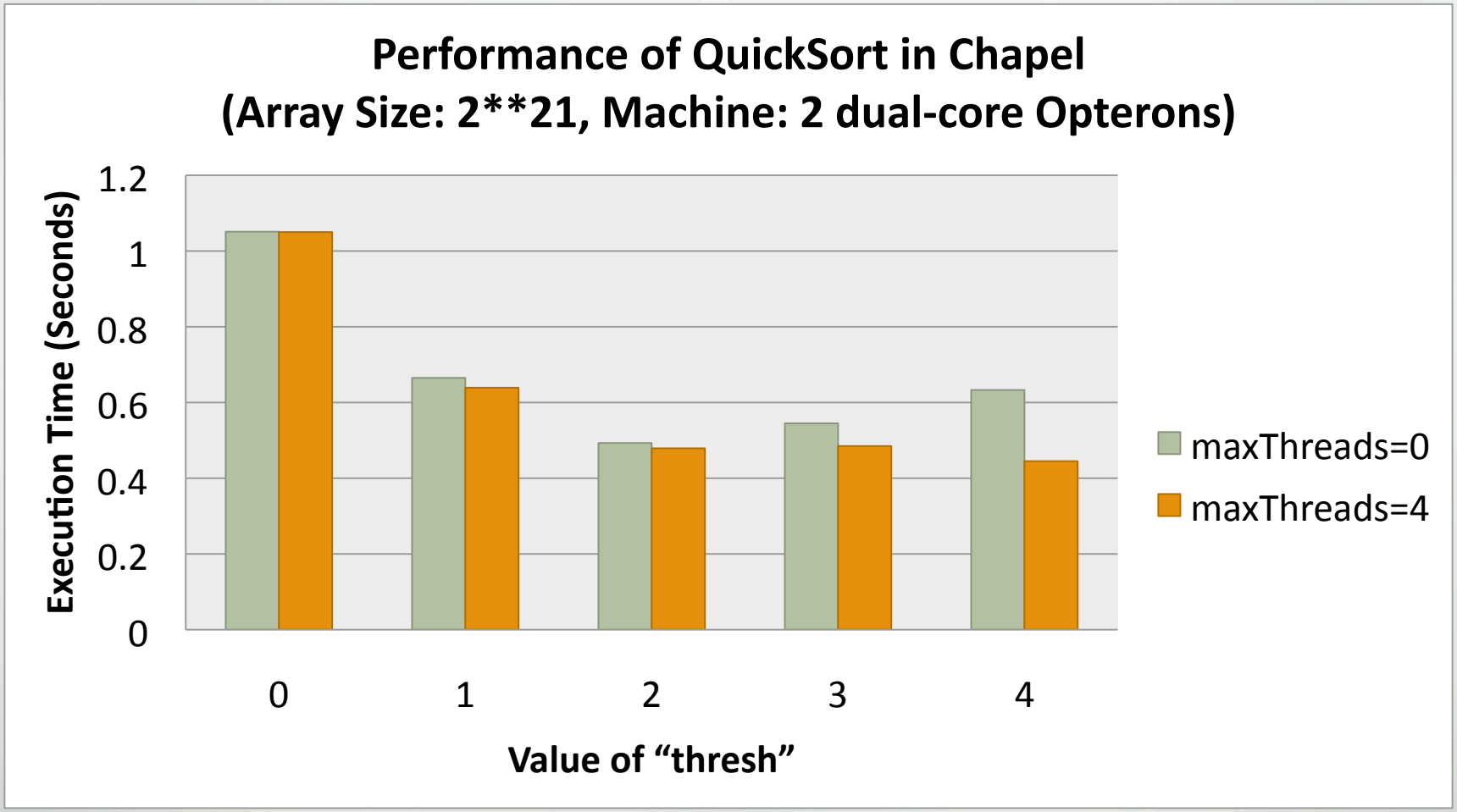
Using Chapel Version 0.9

- **CHPL_THREADS**: Environment variable for threading
  - Default for most platforms is **pthreads**
  - Current alternatives include **none** and **mta**
- **maxThreads**: Config variable for limiting concurrency
  - Use **--maxThreads=#** to use at most # threads
  - Use **--maxThreads=0** to use a system limit
- Current task-to-thread scheduling policy
  - Once a thread gets a task, it runs to completion
  - If an execution runs out of thread, it may deadlock
  - In the future, blocked threads will run other tasks

Quick Sort in Chapel

```
def quickSort(arr: [],
              thresh: int,
              low: int = arr.domain.low,
              high: int = arr.domain.high) {
  if high - low < 8 {
    bubbleSort(arr, low, high);
  } else {
    const pivotVal = findPivot(arr, low, high);
    const pivotLoc = partition(arr, low, high, pivotVal);
    serial thresh == 0 do cobegin {
      quickSort(arr, thresh-1, low, pivotLoc-1);
      quickSort(arr, thresh-1, pivotLoc+1, high);
    }
  }
}
```

Performance of Multi-Threaded Chapel



Questions?

- Primitive Task-Parallel Constructs
  - The **begin** statement
  - The **sync** types
- Structured Task-Parallel Constructs
  - The **cobegin** statement
  - The **coforall** loop
  - The **sync** statement
  - The **serial** statement
- Atomic Transactions and Memory Consistency
  - The **atomic** statement
  - Races and memory consistency
- Implementation Notes and Examples
  - Using pThreads
  - Quick sort example

Chapel: Data Parallelism

Steve Deitz

Cray Inc.

Outline

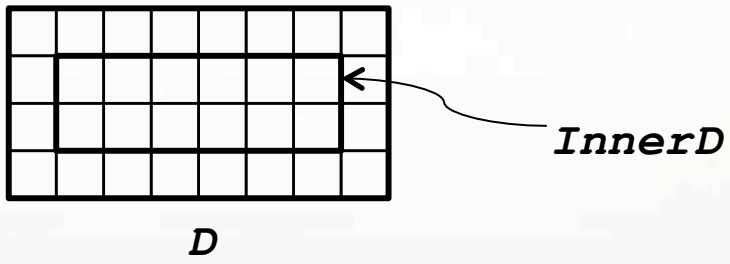
- Domains and Arrays
  - Overview
  - Arithmetic
- Other Domain Types
- Data Parallel Operations
- User Example

Domains

- A first-class index set
  - Specifies size and shape of arrays
  - Supports iteration, array operations
  - Potentially distributed across machines
- Three main classes
  - Arithmetic—indices are Cartesian tuples
  - Associative—indices are hash keys
  - Opaque—indices are anonymous
- Fundamental Chapel concept for data parallelism
- A generalization of ZPL's region concept

Sample Arithmetic Domains

```
config const m = 4, n = 8;

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

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



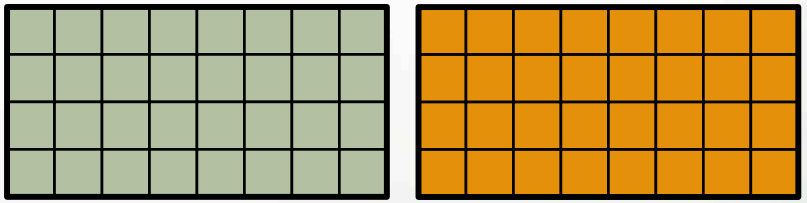
Domains Define Arrays

- Syntax

```
array-type:
  [ domain-expr ] type
```

- Semantics
  - Associates data with each index in *domain-expr*
- Example

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



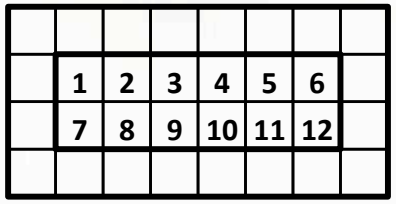
- Revisited example

```
var A: [1..3] int; // creates anonymous domain [1..3]
```

Domain Iteration

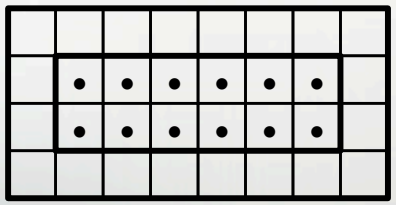
- For loops (discussed already)
  - Executes loop body once per loop iteration
  - Order is serial

```
for i in InnerD do ...
```



- Forall loops
  - Executes loop body once per loop iteration
  - Order is parallel (must be *serializable*)

```
forall i in InnerD do ...
```



Other Forall Loops

Forall loops also support...

- A symbolic shorthand:

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

- An expression-based form:

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

- A sugar for array initialization:

```
var A: [(i,j) in D] real = 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

A

Usage of For, Forall, and Coforall

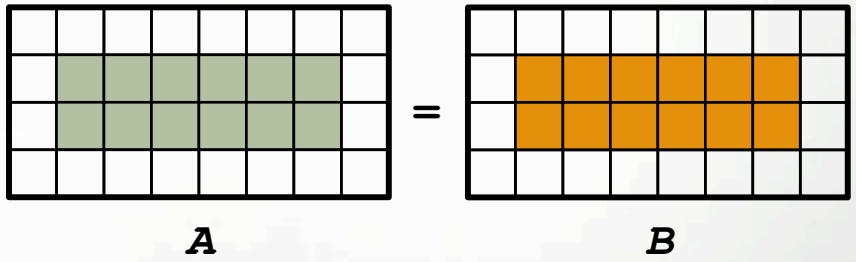
- Use for when
  - A loop must be executed serially
  - One task is sufficient for performance
- Use forall when
  - The loop can be executed in parallel
  - The loop can be executed serially
- Use coforall when
  - The loop must be executed in parallel

    (And not just for performance reasons!)

Other Domain Functionality

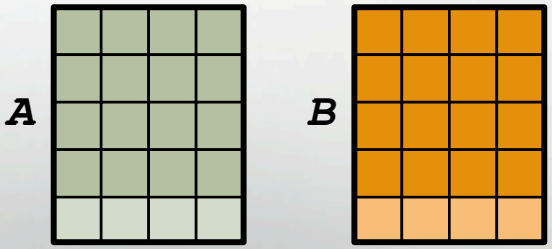
- Domain methods (exterior, interior, translate, ...)
- Domain slicing (intersection)
- Array slicing (sub-array references)

```
A(InnerD) = B(InnerD);
```



- Array reallocation
  - Reassign domain → change array
  - Values are preserved (new elements initialized)

```
D = [1..m+1, 1..m];
```

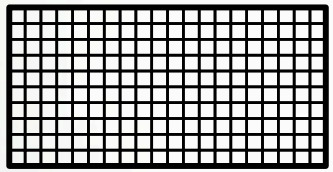


Outline

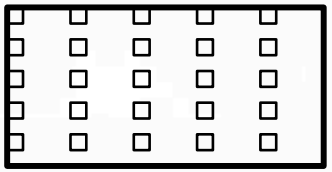
- Domains and Arrays
- Other Domain Types
  - Strided
  - Sparse
  - Associative
  - Opaque
- Data Parallel Operations
- User Example

The Varied Kinds of Domains

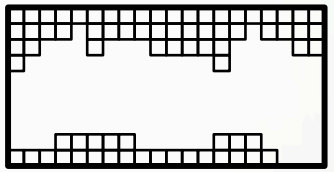
```
var Dense: domain(2) = [1..10, 1..20],
    Strided: domain(2) = Dense by (2, 4),
    Sparse: subdomain(Dense) = genIndices(),
    Associative: domain(string) = readNames(),
    Opaque: domain(opaque);
```



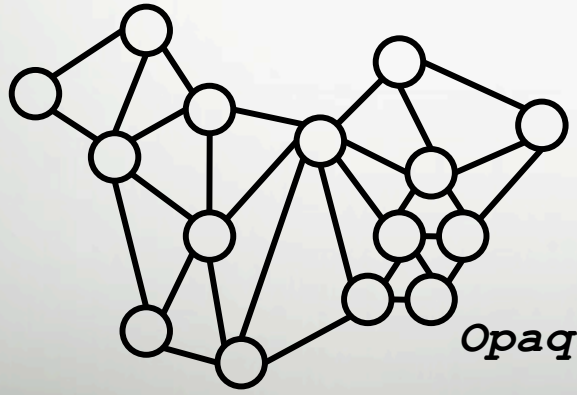
Dense



Strided



Sparse

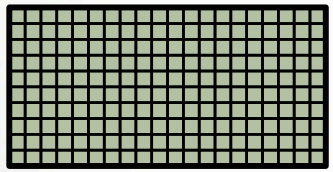


Opaque

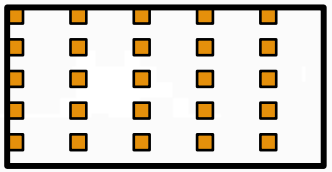
Associative

The Varied Kinds of Arrays

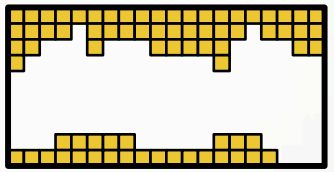
```
var DenseArr: [Dense] real,
    StridedArr: [Strided] real,
    SparseArr: [Sparse] real,
    AssociativeArr: [Associative] real,
    OpaqueArr: [Opaque] real;
```



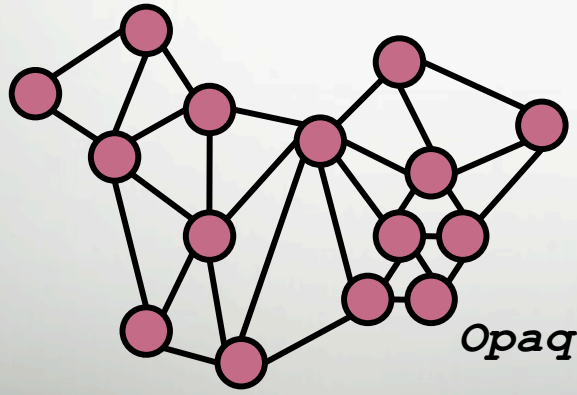
DenseArr



StridedArr



SparseArr

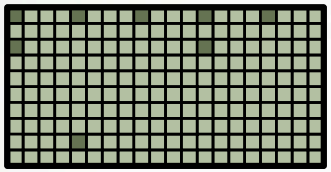


OpaqueArr

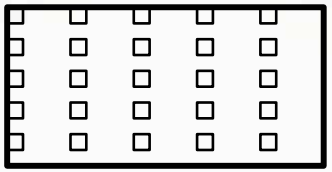
AssociativeArr

All Domains Support Iteration

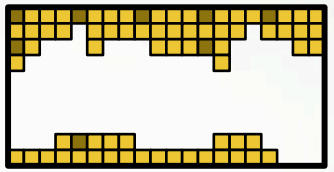
```
forall (i,j) in Strided {
  DenseArr(i,j) += SparseArr(i,j);
}
```



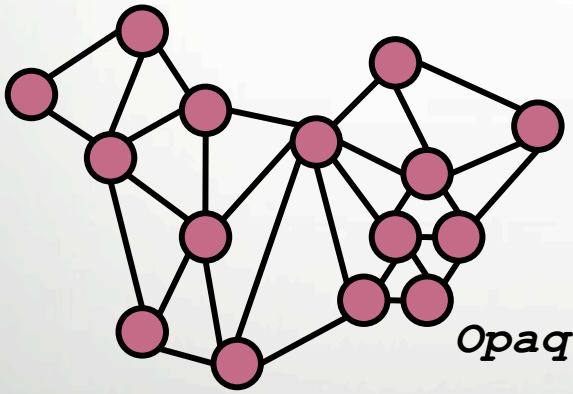
DenseArr



Strided



SparseArr



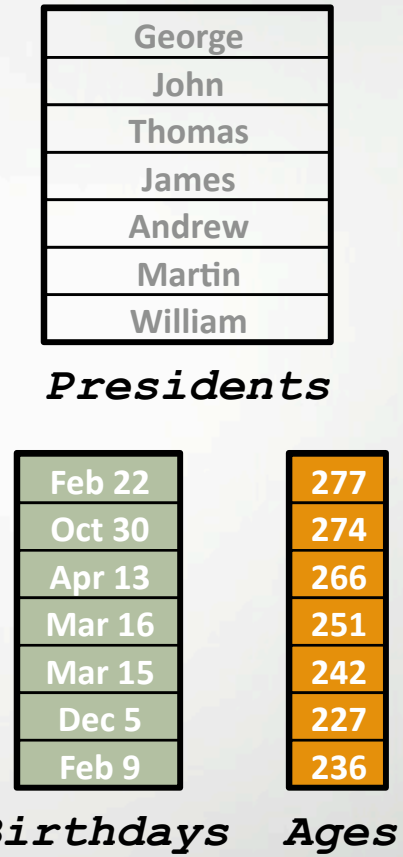
OpaqueArr

AssociativeArr

(Also, all domains support slicing, reallocation, ...)

Associative Domains and Arrays by Example

```
var Presidents: domain(string) =
      ("George", "John", "Thomas",
       "James", "Andrew", "Martin");

Presidents += "William";


var Ages: [Presidents] int,
    Birthdays: [Presidents] string;

Birthdays("George") = "Feb 22";


forall president in Presidents do
  if Birthdays(president) == today then
    Ages(president) += 1;
```



Outline

- Domains and Arrays
- Other Domain Types
- Data Parallel Operations
  - Promotion
  - Reductions and scans
- User Example

Data Parallel Promotion

Functions/operators expecting scalars can also take...

- Arrays, causing each element to be passed

```
...sin(A)...
...2*A...
```

≈

```
...[a in A] sin(a)...
...[a in A] 2*a...
```

- Domains, causing each index to be passed

```
foo(Sparse); // calls foo for all indices in Sparse
```

Multiple arguments can promote using either...

- Zipper promotion

```
...pow(A, B)...
```

≈

```
...[(a,b) in (A,B)] pow(a,b)...
```

- Tensor promotion

```
...pow[A, B]...
```

≈

```
...[(a,b) in [A,B]] pow(a,b)...
```

Reductions

- Syntax

```
reduce-expr:
  reduce-op reduce iterator-expr
```

- Semantics
  - Combines iterated elements with *reduce-op*
  - *Reduce-op* may be built-in or user-defined
- Examples

```
total = + reduce A;
bigDiff = max reduce [i in InnerD] abs(A(i)-B(i));
```

Scans

- Syntax

```
scan-expr:
  scan-op scan iterator-expr
```

- Semantics
  - Computes parallel prefix of *scan-op* over elements
  - *Scan-op* may be any *reduce-op*
- Examples

```
var A, B, C: [1..5] int;
A = 1;                          // A:  1  1  1  1  1
B = + scan A;                   // B:  1  2  3  4  5
B(3) = -B(3);                   // B:  1  2 -3  4  5
C = min scan B;                 // C:  1  1 -3 -3 -3
```

Reduction and Scan Operators

- Built-in
  - +, *, &&, ||, &, |, ^, min, max
  - minloc, maxloc

    (Generate a tuple of the min/max and its index)
- User-defined
  - Defined via a class that supplies a set of methods
  - Compiler generates code that calls these methods
  - More information:

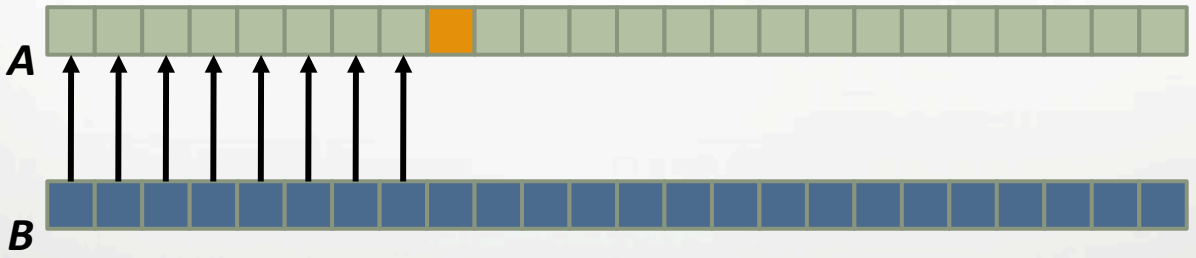
    S. J. Deitz, D. Callahan, B. L. Chamberlain, and L. Snyder. *Global-view abstractions for user-defined reductions and scans*. In Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, 2006.

Outline

- Domains and Arrays
- Other Domain Types
- Data Parallel Operations
- User Example

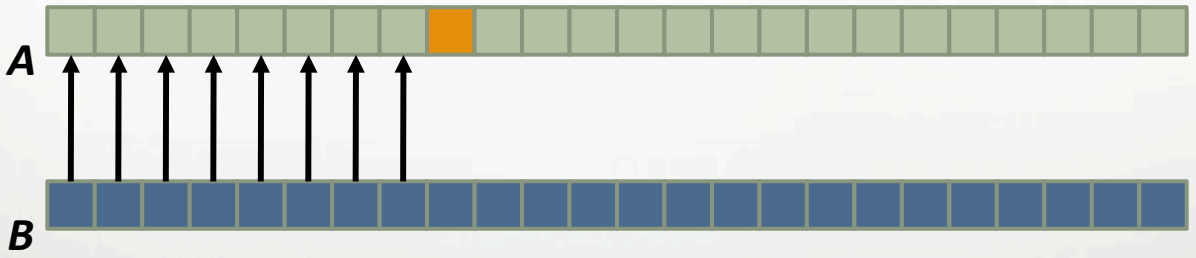
Serial Array Copy Before Key

```
var A, B: [1..n] real, key: real;

for i in 1..n {
  if A(i) == key then
    break;
  A(i) = B(i);
}
```



Data-Parallel Array Copy Before Key

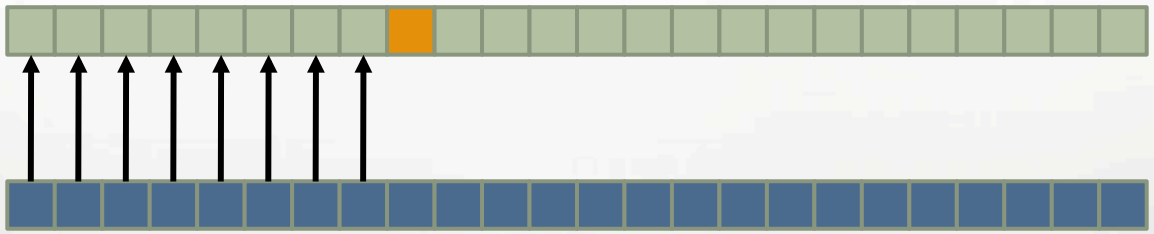
```
var A, B: [1..n] real, key: real;

forall i in 1..n {
  if A(i) == key then
    break;
  A(i) = B(i);
}
```




```
error: break is not allowed in forall statement
```

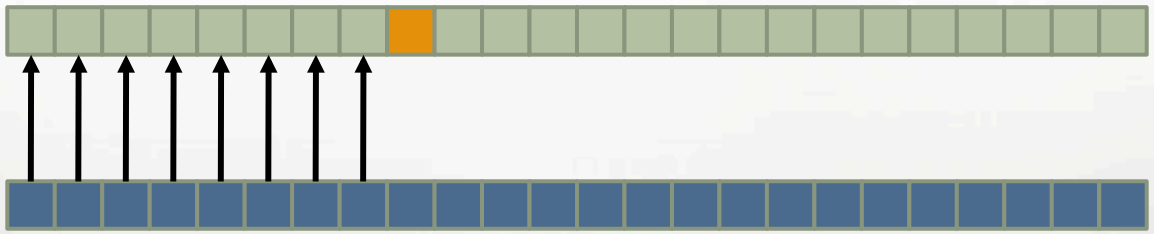
Pre-Compute Data-Parallel Region

```
var A, B: [1..n] real, key: real;

var loc: int = n;
for i in 1..n do
  if A(i) == key then
    loc = min(loc, i)
forall i in 1..loc do
  A(i) = B(i);
```



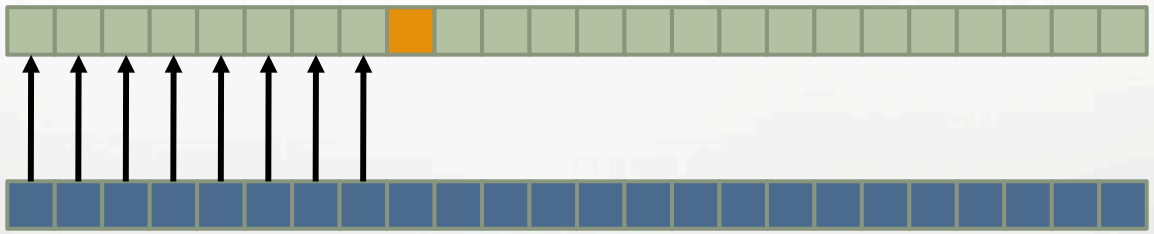
Data-Parallel Pre-Computation

```
var A, B: [1..n] real, key: real;

var loc: sync int = n;
forall i in 1..n do
  if A(i) == key then
    loc = min(loc, i)
forall i in 1..loc do
  A(i) = B(i);
```



The maxloc Reduction

```
var A, B: [1..n] real, key: real;

var (val, loc) =
  maxloc reduce (A==key, 1..n);
if val == false then
  loc = n;
forall i in 1..loc do
  A(i) = B(i);
```



Questions?

- Domains and Arrays
  - Overview
  - Arithmetic
- Other Domain Types
  - Strided
  - Sparse
  - Associative
  - Opaque
- Data Parallel Operations
  - Promotion
  - Reductions and scans
- User Example

Chapel: Locality and Affinity

Steve Deitz

Cray Inc.

Outline

- Multi-Locale Basics
  - Locales
  - On, here, and communication
- Distributed Domains and Arrays
- Heat Transfer Example

The Locale Type

- Definition
  - Abstract unit of target architecture
  - Capacity for processing and storage
  - Supports reasoning about locality
- Properties
  - Locale's tasks have uniform access to local memory
  - Other locale's memory is accessible, but at a price
- Examples
  - A multi-core processor
  - An SMP node

Program Startup

- Execution Context

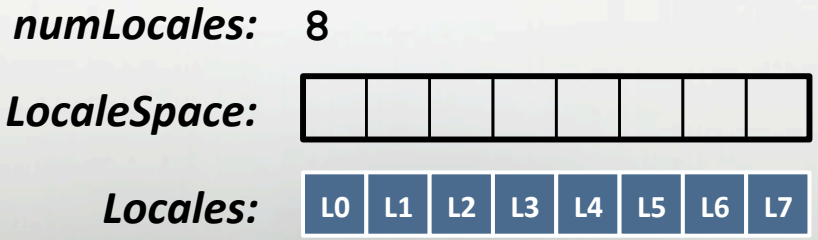
```
config const numLocales: int;
const LocaleSpace: domain(1) = [0..numLocales-1];
const Locales: [LocaleSpace] locale;
```

- Specify # of locales when running executable

```
prompt> a.out --numLocales=8
```

Alternatively,

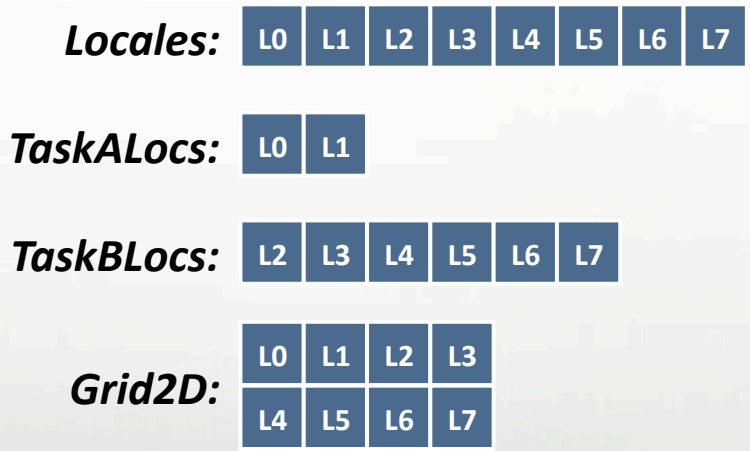
```
prompt> a.out -nl 8
```



Rearranging Locales

Create locale views with standard array operations:

```
var TaskALocs = Locales[0..1];
var TaskBLocs = Locales[2..numLocales-1];

var Grid2D = Locales.reshape([1..2, 1..4]);
```



Locale Methods

- `def locale.id: int { ... }`

  Returns index in LocaleSpace

- `def locale.name: string { ... }`

  Returns name of locale (like uname -a)

- `def locale.numCores: int { ... }`

  Returns number of cores available to locale

- `def locale.physicalMemory(...) { ... }`

  Returns physical memory available to locale

  Example

```
const totalSystemMemory =
  + reduce Locales.physicalMemory();
```

The On Statement

- Syntax

```
on-stmt:
  on expr { stmt }
```

- Semantics
  - Executes *stmt* on the locale specified by *expr*
  - Does not introduce concurrency
- Example

```
var A: [LocaleSpace] int;
coforall loc in Locales do on loc do
  A(loc.id) = compute(loc.id);
```

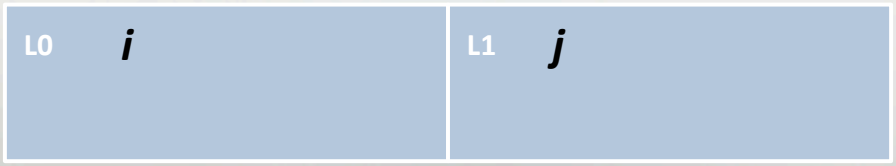
Querying a Variable's Locale

- Syntax

```
locale-query-expr:
  expr . locale
```

- Semantics
  - Returns the locale on which *expr* is allocated
- Example

```
var i: int;
on Locales(1) {
  var j: int;
  writeln(i.locale.id, j.locale.id); // outputs 01
}
```



Here

- Built-in locale

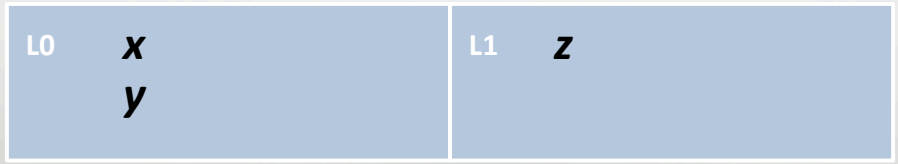
```
const here: locale;
```

- Semantics
  - Refers to the locale on which the task is executing
- Example

```
writeln(here.id);     // outputs 0
on Locales(1) do
  writeln(here.id);   // outputs 1
```

Serial Example with Implicit Communication

```
var x, y: real;       // x and y allocated on locale 0

on Locales(1) {       // migrate task to locale 1
  var z: real;        // z allocated on locale 1

  z = x + y;          // remote reads of x and y

  on Locales(0) do    // migrate back to locale 0
    z = x + y;        // remote write to z
                      // migrate back to locale 1
  on x do             // data-driven migration to locale 0
    z = x + y;        // remote write to z
                      // migrate back to locale 1
}                     // migrate back to locale 0
```



The Fragmented Model in Chapel

```
def main() {
  coforall loc in Locales do on loc {
    myFragmentedMain();
  }
}

def myFragmentedMain() {
  const size = numLocales, rank = here.id;
  ...
```

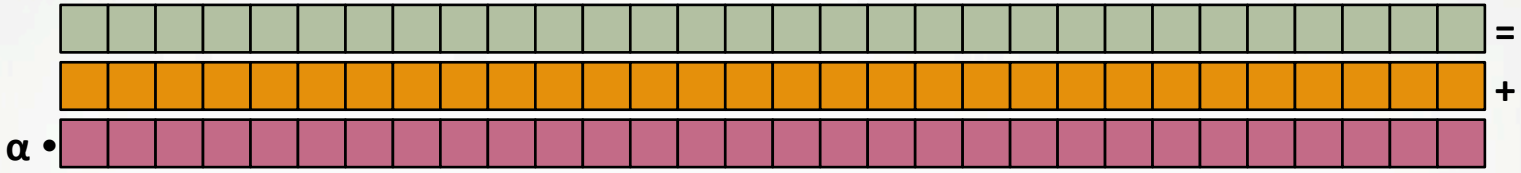
Outline

- Multi-Locale Basics
- Distributed Domains and Arrays
- Heat Transfer Example

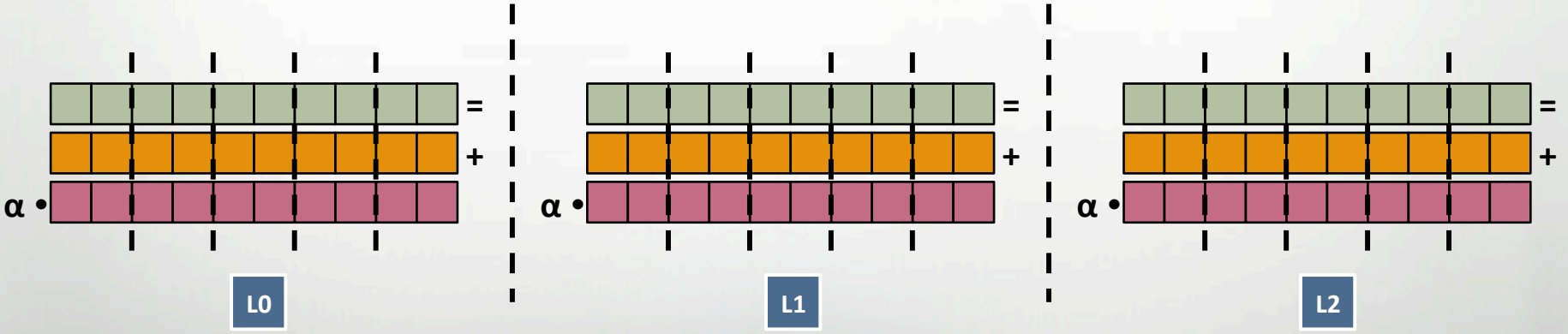
What is a Distribution?

A "recipe" for distributed arrays that...

Instructs the compiler how to Map the global view...



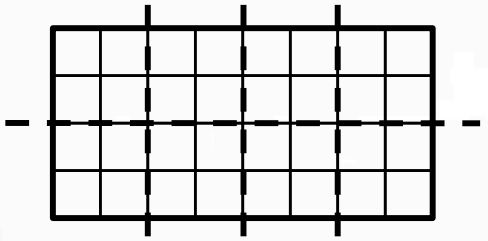
...to a fragmented, per-processor implementation



Distributing a Domain

Domains are associated to a distribution

```
const Dist = new Block(rank=2, bbox=[1..4, 1..8]);

var Dom: domain(2) distributed Dist = [1..4, 1..8];
```



distributed over

The distribution defines:

- Ownership of domain indices and array elements
- Default distribution of work (task-to-locale map)

  E.g., forall loops over distributed domains/arrays

Authoring Distributions

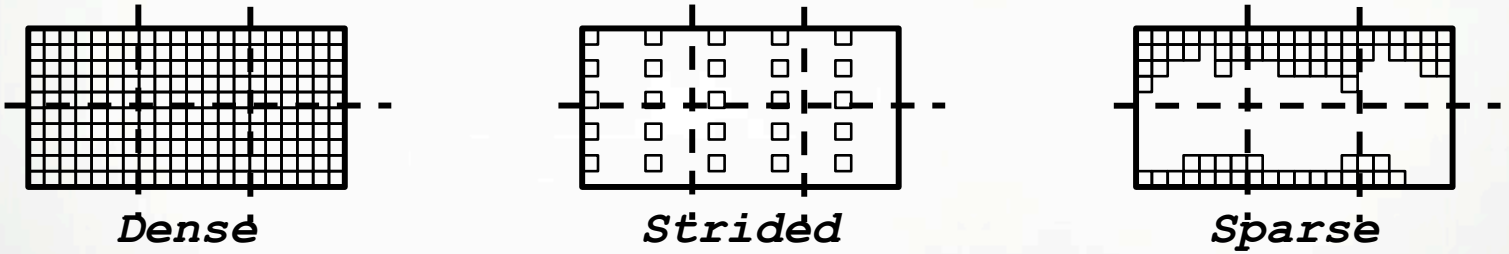
- (Advanced) programmers can write distributions
- Built-in library of distributions
  - No extra compiler support for built-in distributions
  - Compiler uses structural interface:
    - Create domains and arrays
    - Map indices to locales
    - Access array elements
    - Iterate over indices/elements sequentially, in parallel, zippered
    - …
- Distributions are built using language-level concepts
  - On for data and task locality
  - Begin, cobegin, and coforall for data parallelism

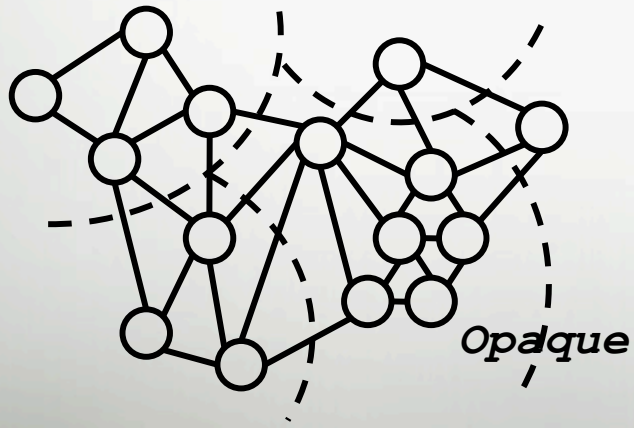
Distributing Domains

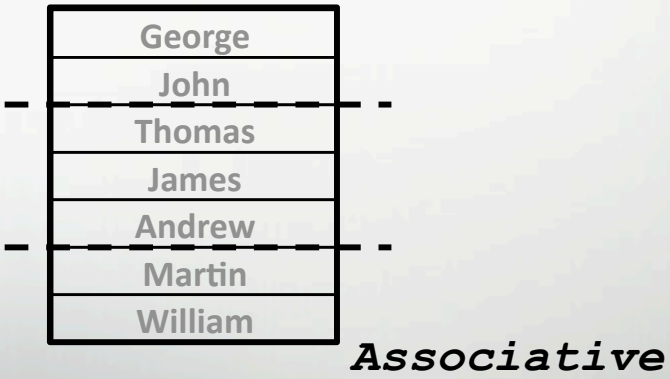
All domain types can be distributed.

Semantics are independent of distribution.

(Though performance and parallelism will vary...)



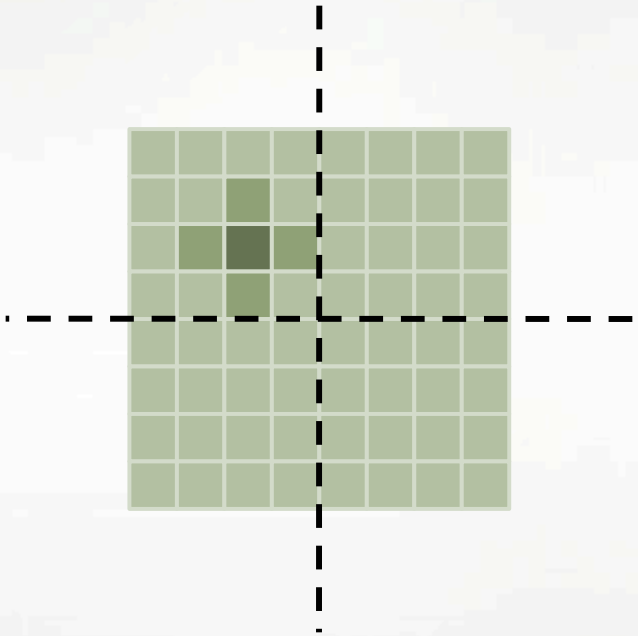




Outline

- Multi-Locale Basics
- Distributed Domains and Arrays
- Heat Transfer Example

Heat Transfer Example – Illustration



Heat Transfer Example – Declarations

```
use BlockDist;

config var n : int = 5;
config var epsilon : real = 0.00001;
config var verbose : bool = false;

const Dist =
  distributionValue(new Block(2,int,[1..n,1..n]));

var BigR : domain(2) distributed Dist = [0..n+1, 0..n+1];
var R : domain(2) distributed Dist = [1..n, 1..n];
var South : domain(2) distributed Dist = [n+1..n+1, 1..n];

var A : [BigR] real;
var Temp : [R] real;
```

Heat Transfer Example – Initialization

```
forall (i,j) in BigR {
  A(i,j) = 0.0;
}

forall (i,j) in South {
  A(i,j) = 1.0;
}
```

Alternative 1

```
A = 0.0;
A(South) = 1.0;
```

Alternative 2

```
forall a in A do
  a = 0.0;
forall a in A(South) do
  a = 1.0;
```

Heat Transfer Example – Main Loop

```
var iteration : int = 0;
var delta : real = 1.0;

while (delta > epsilon) {
  forall (i,j) in R do
    Temp(i,j) = (A(i-1,j)+A(i+1,j)+
                 A(i,j-1)+A(i,j+1))/4.0;
  delta = + reduce [(i,j) in R] Temp(i,j)-A(i,j);
  forall (i,j) in R {
    A(i,j) = Temp(i,j);
  }
  iteration += 1;
}
```

Questions?

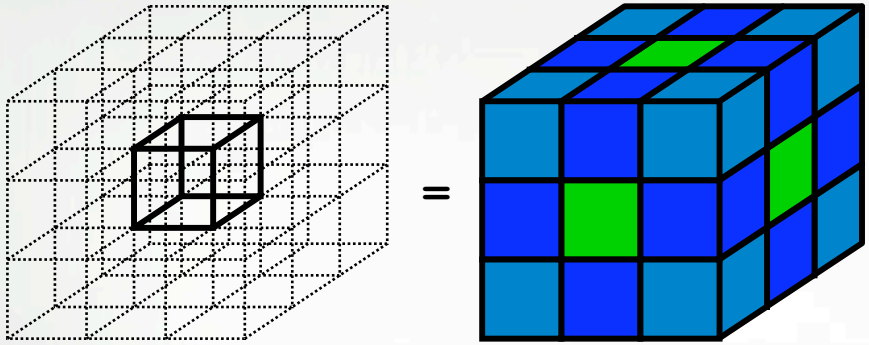
- Multi-Locale Basics
  - Locales
  - On, here, and communication
- Domain and Array Distributions
- Heat Transfer Example

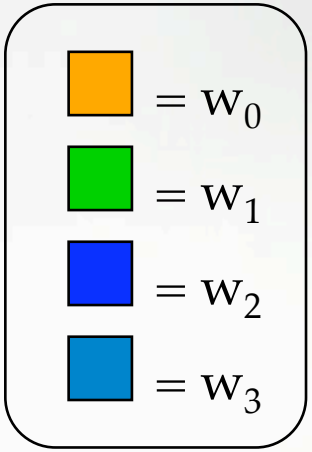
Chapel: Wrap Up

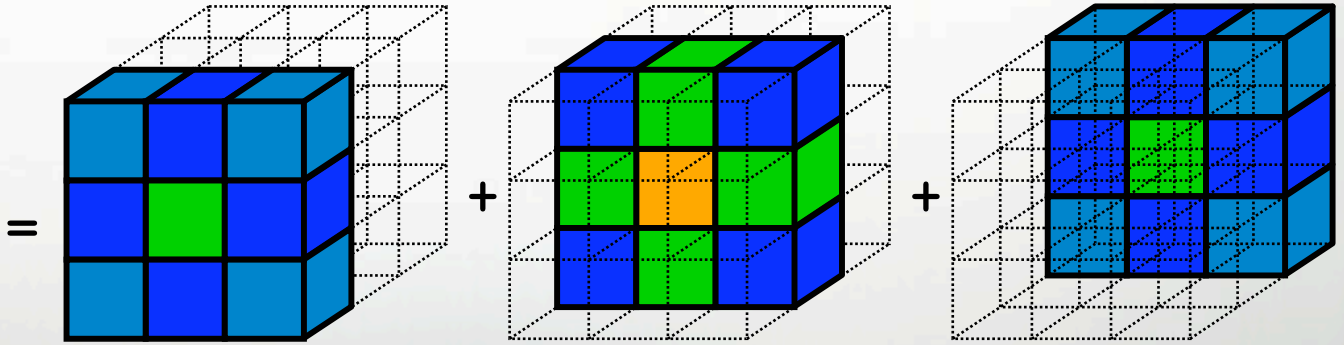
Steve Deitz

Cray Inc.

NAS MG Stencil Revisited







NAS MG Stencil in Chapel Revisited

```
def rprj3(S, R) {
  const Stencil = [-1..1, -1..1, -1..1],
        W: [0..3] real = (0.5, 0.25, 0.125, 0.0625),
        W3D = [(i,j,k) in Stencil] W((i!=0)+(j!=0)+(k!=0));

  forall inds in S.domain do
    S(inds) =
      + reduce [offset in Stencil] (W3D(offset) *
                                    R(inds + offset*R.stride));
}
```

Outline

- NAS MG Stencil Revisited
- Chapel Compiler System Overview
- Version 0.9 Release and Status

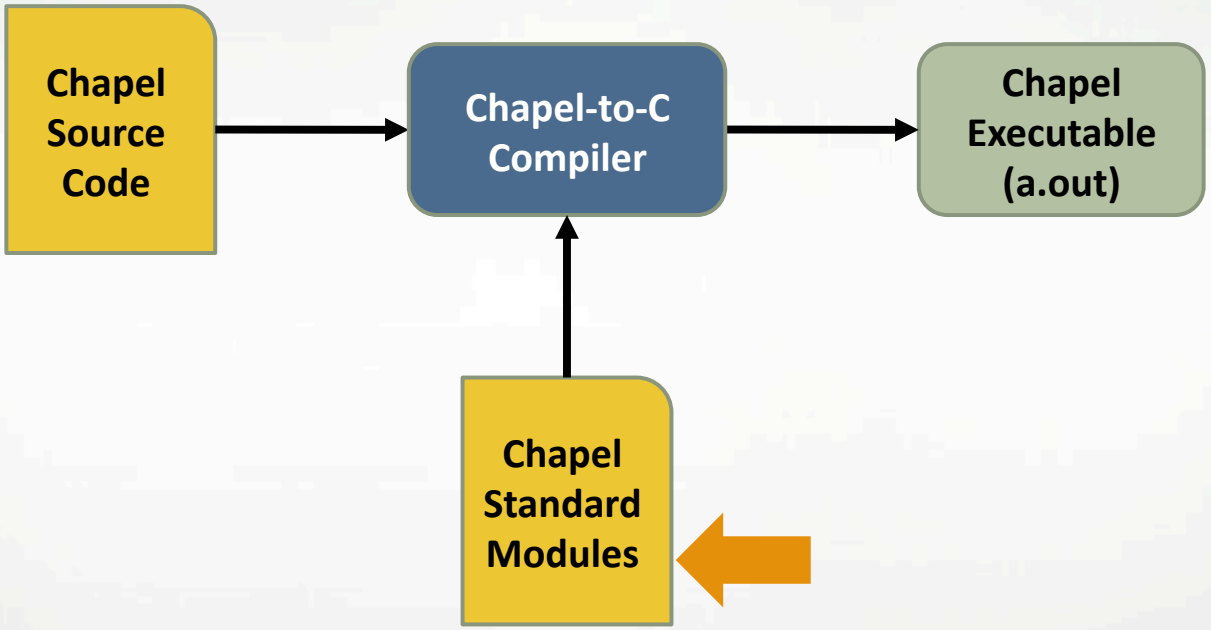
Prototype Compiler Development Strategy

- Start development within Cray under HPCS
- Initial releases to select users
- First public release November 2008
- Second public release April 2009
  - Migrated to SourceForge
  - Major step in opening development
- Turn over to community when ready

Prototype Compilation Approach

- Chapel-to-C compiler for portability
  - C++ compiler generates strict C code
  - Tested against GCC and several vendor's compilers
- Link against threading and communication libraries
  - Default threading layer on most platforms: pThreads
  - Default communication layer on most platforms: GASNet
- Use many standard and internal Chapel modules

Compiler Schematic



Chapel Standard Modules

Standard modules implement standard library routines.

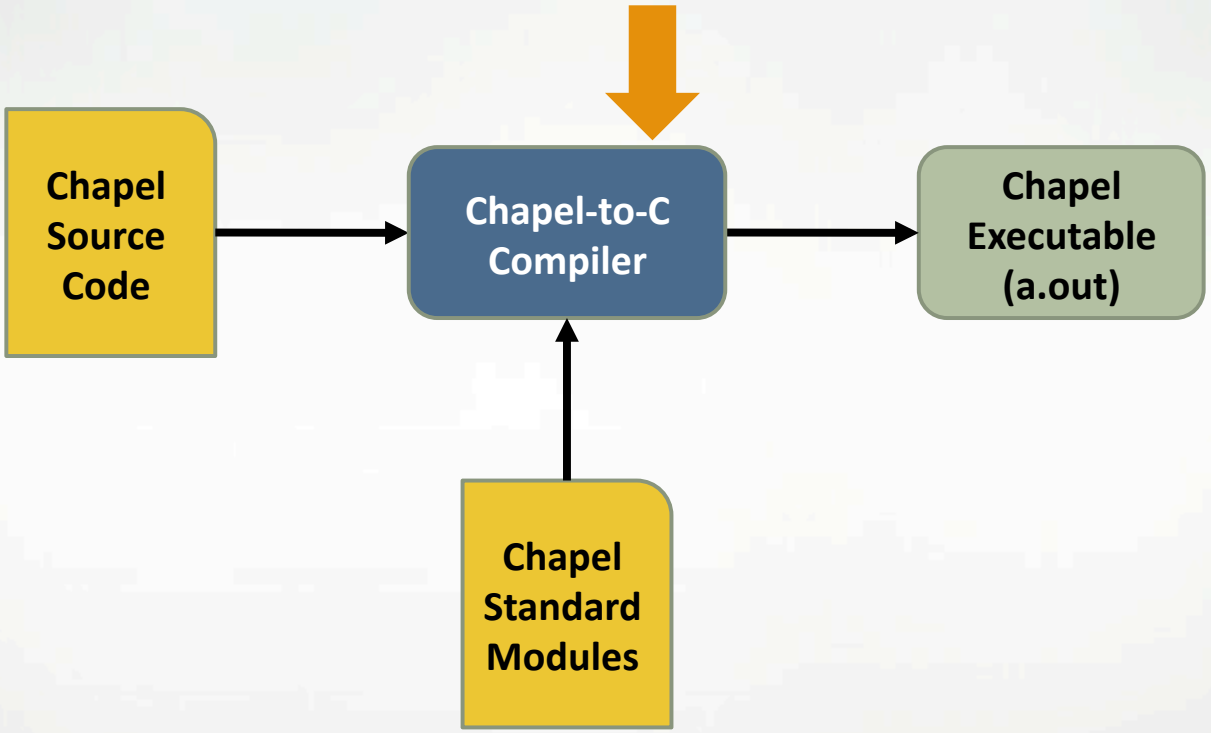
- **BlockDist**: Definition of Block distribution
- **BitOps**: Specialized bit manipulation
- **Random**: Random number generation
- **Time**: Timer and time-of-day support

...

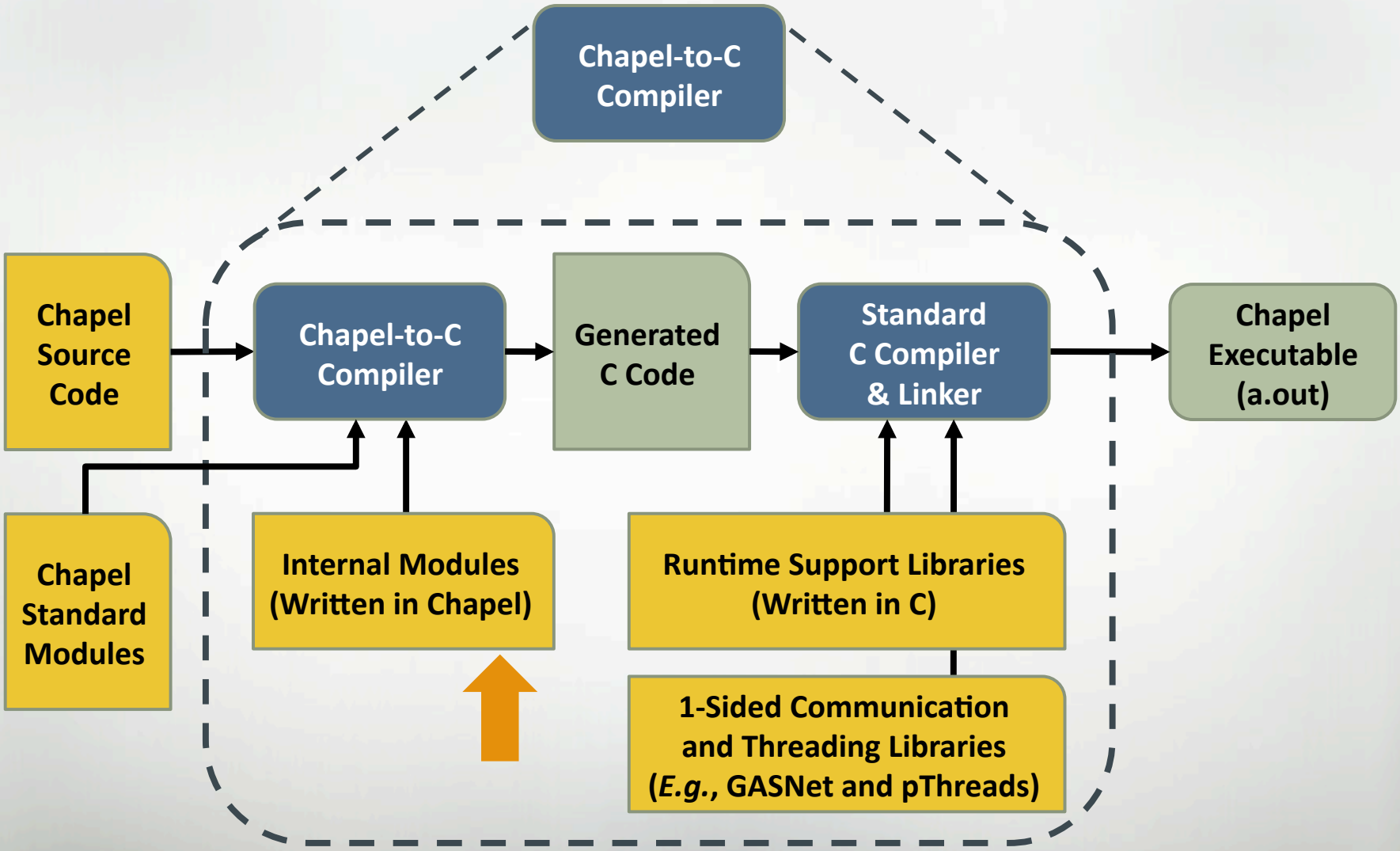
Standard modules must be explicitly used

E.g., `use BlockDist;`

Compiler Schematic



Detailed Compiler Schematic



Chapel Internal Modules

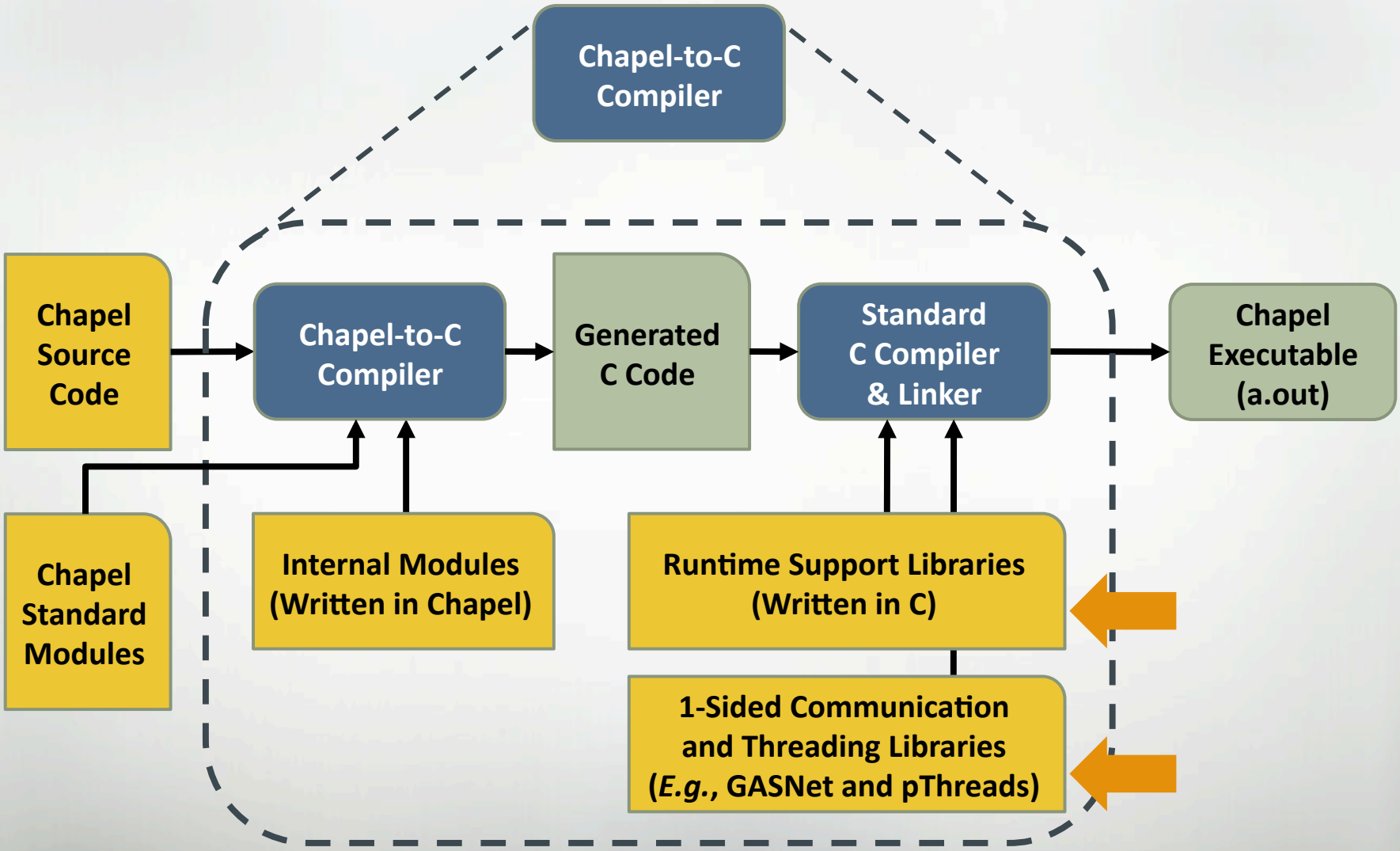
Internal modules implement basic Chapel features.

- Standard operators
- Standard math routines
- User-level I/O routines
- User-level assertions and halts
- Tuples, ranges, domains, and arrays
- Synchronization variables

Essential to development

- Improves robustness by using the language
- Makes development easier because Chapel is productive

Detailed Compiler Schematic



Runtime Support Libraries

Runtime support libraries bootstrap Chapel.

- Command-line argument passing
- Console and file I/O primitives
- Error handling
- Memory management
- Type conversions
- Time primitives
- Thread creation and management
- Inter-process communication and coordination

This functionality has been migrating to Chapel.

Outline

- NAS MG Stencil Revisited
- Chapel Compiler System Overview
- Version 0.9 Release and Status

Chapel Version 0.9

- Available on SourceForge
  http://sourceforge.net/projects/chapel/
- Distributed BSD Open Source license
- Systems: Linux/Unix, Mac, Cygwin
- Contents
  - Compiler and standard modules
  - Runtime and third-party libraries (e.g., GASNet)
  - Top-level README for quick start
  - Language spec, quick reference, HPCC tutorial
  - Examples (tutorials, programs, and HPCC benchmarks)
  - Portability scripts

Implementation Status

- Base language and task parallelism
  - Complete with minor gaps (*e.g.*, multiple inheritance)
- Data parallelism
  - Serial reference implementation
  - Initial support for concurrency via distributions
- Distributed memory
  - Task parallelism across locales
  - Initial support for distributed arrays and domains
- Performance
  - Focus on a small set of language features

Unimplemented Features Seen Today

- Base language
  - Constness is not checked for domains, arrays, fields
- Task parallelism
  - Atomic statements are not atomic
- Data parallelism
  - Promoted functions/operators do not preserve shape
  - Reductions and scans cannot be user-defined or partial
  - Arrays of arrays require inner arrays to use a single domain
- Locality and affinity
  - User-defined distributions are not yet specified

Where to Learn More

- Full day tutorials
  - Upcoming joint tutorial with X10 and UPC at SC '09
- Download the release
  - http://sourceforge.net/projects/chapel/
- Contact us
  - Send us mail at chapel_info@cray.com
  - Visit our web page at http://chapel.cs.washington.edu/
  - View archives of chapel-users@lists.sourceforge.net