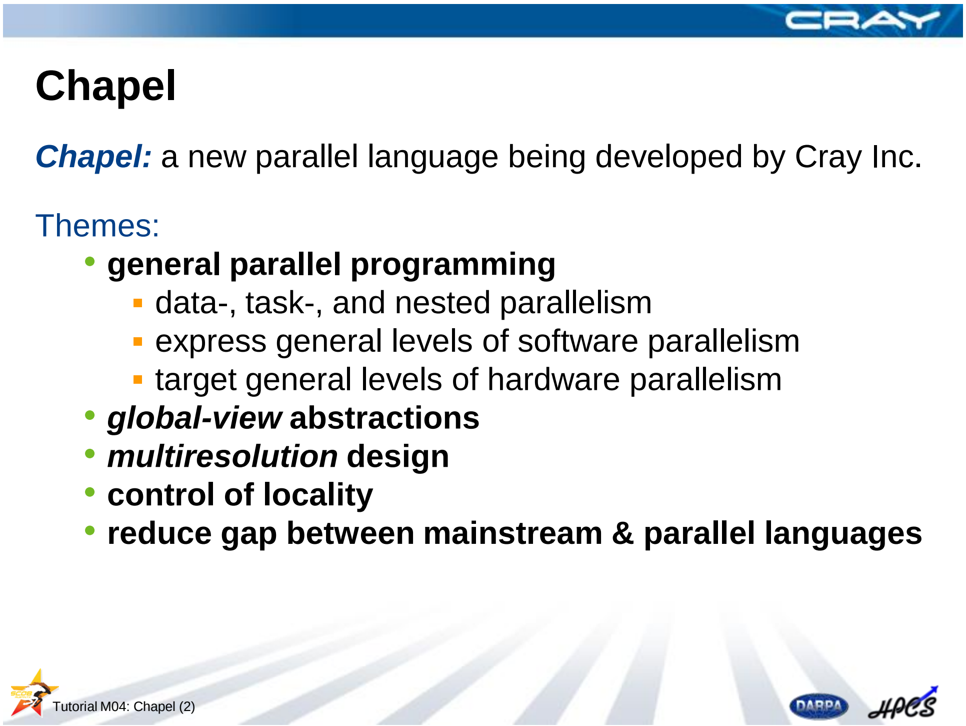
Chapel

the Cascade High Productivity Language

Brad Chamberlain
Cray Inc.

SC08: Tutorial M04 – 11/17/08

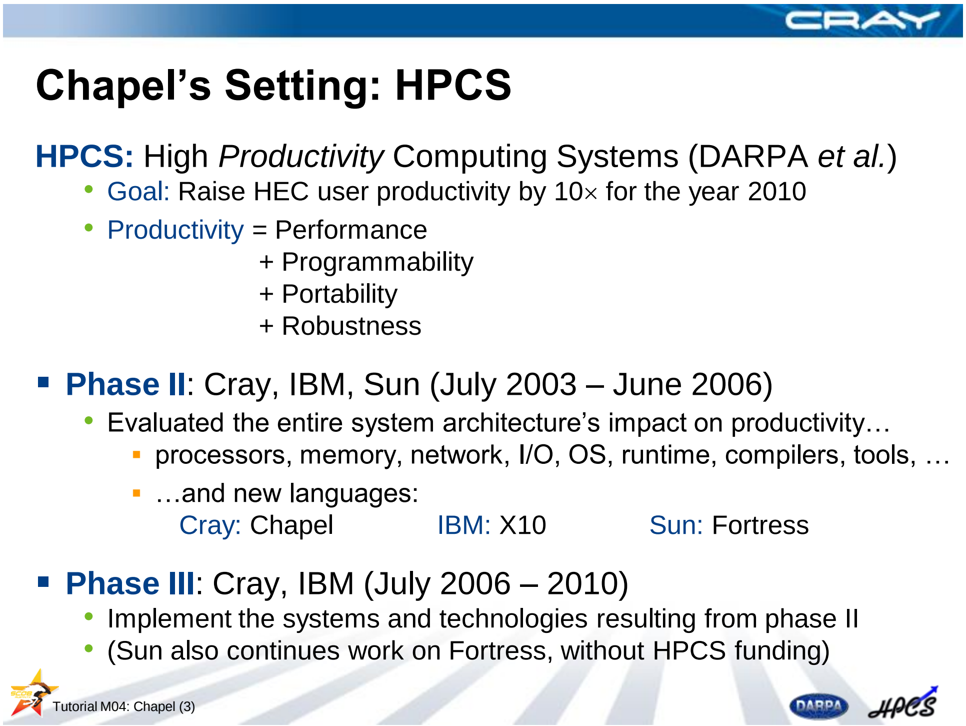


Chapel

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**
- ***multiresolution* design**
- **control of locality**
- **reduce gap between mainstream & parallel languages**

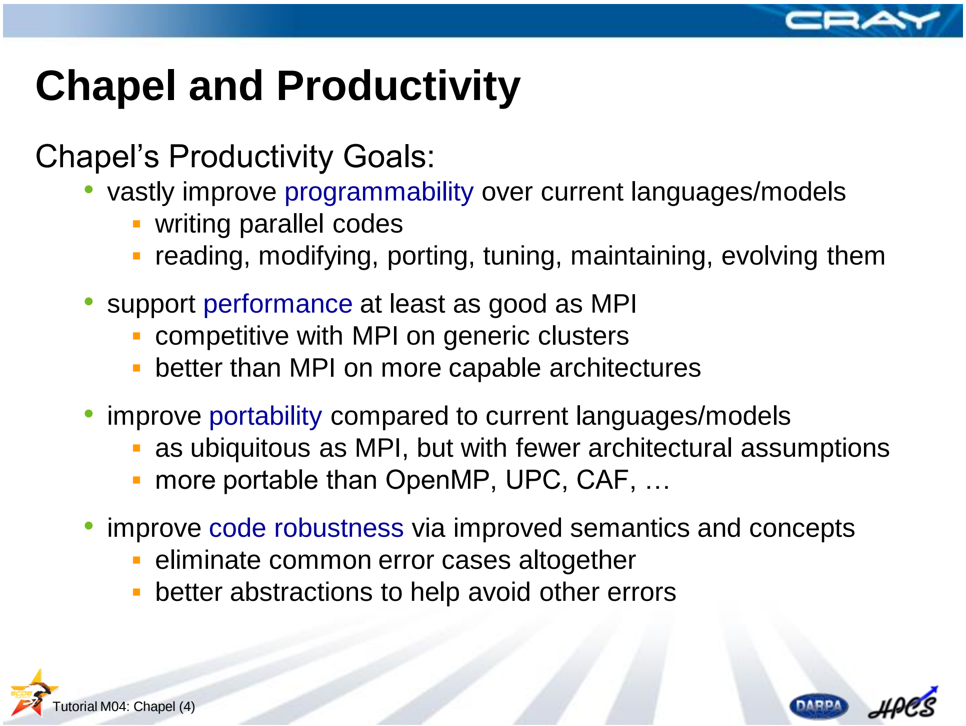


Chapel's Setting: HPCS

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

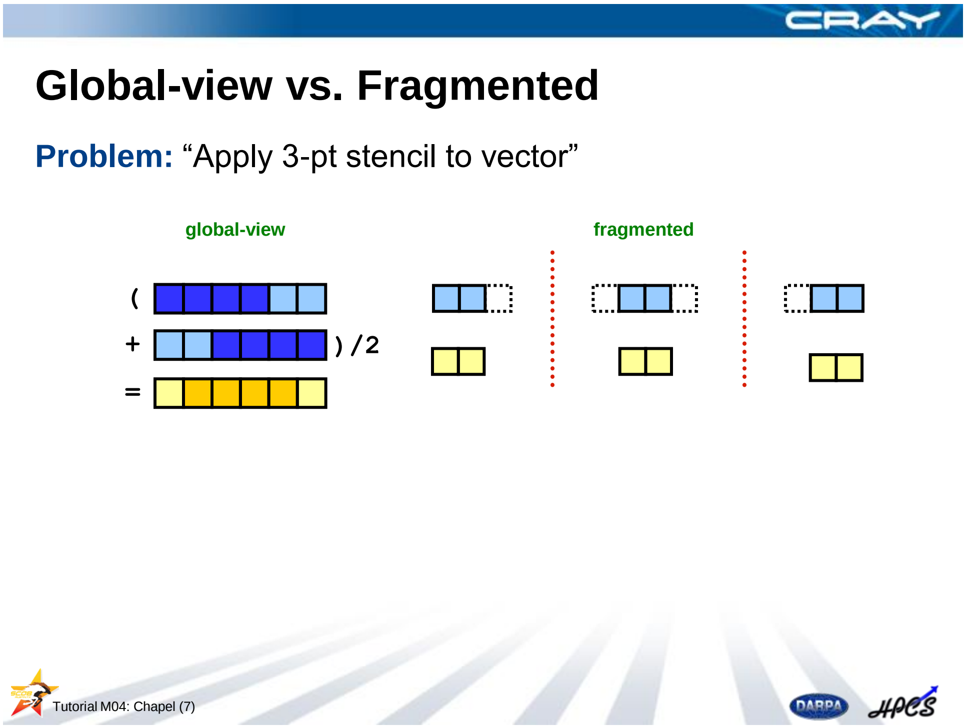
✓ Chapel Context
➢ Terminology: Global-view & Multiresolution Prog. Models
☐ Language Overview
☐ Status, Future Work, Collaborations

Parallel Programming Model Taxonomy

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

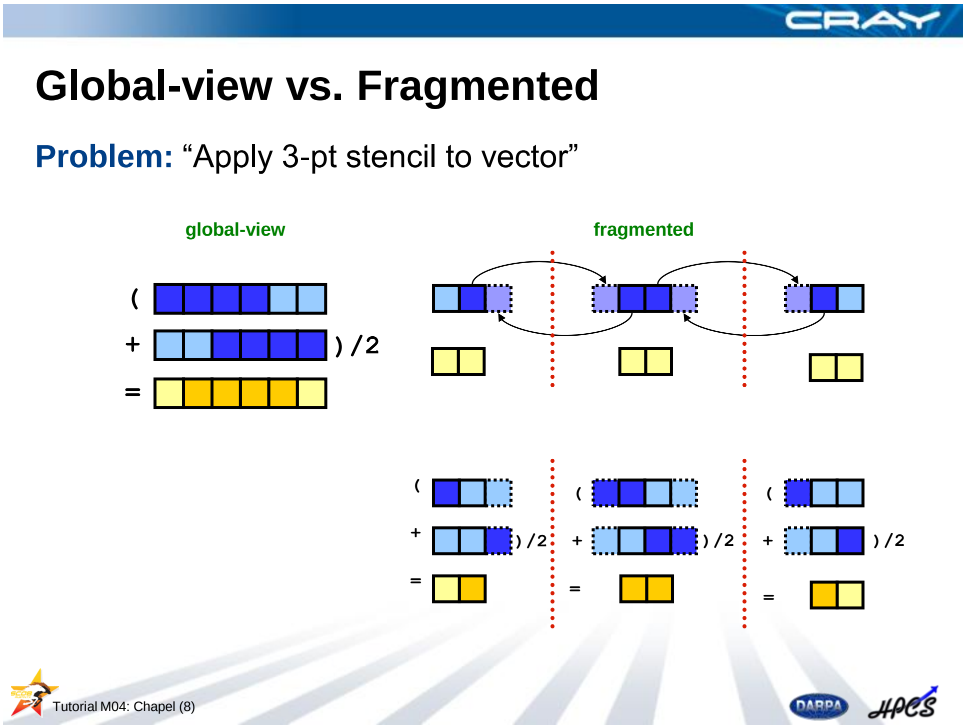
fragmented models: those in which the programmer writes code from the point-of-view of a single processor/thread

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

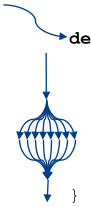
SPMD models: Single-Program, Multiple Data -- a common fragmented model in which the user writes one program & runs multiple copies of it, parameterized by a unique ID

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

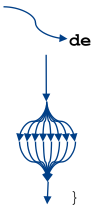
```
def main() {
  var n: int = 1000;
  var locN: int = n/numProcs;
  var a, b: [0..locN+1] real;

  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;
  }
}
```

MPI SPMD pseudo-code

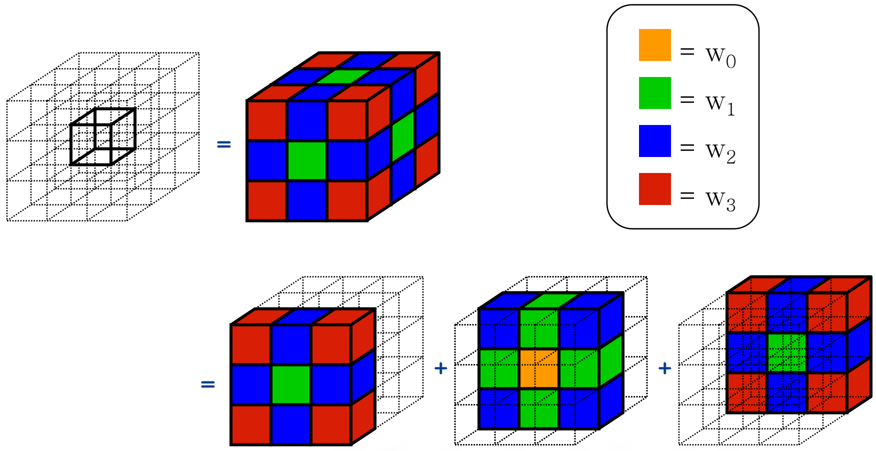
Problem: “Apply 3-pt stencil to vector”

SPMD (pseudocode + MPI)

Communication becomes geometrically more complex for higher-dimensional arrays

```
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;
}
```

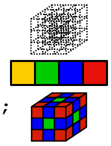
rprj3 stencil from NAS MG



NAS MG *rprj3* stencil in Fortran + MPI

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

implicit none

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

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

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


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


implicit none

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

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

integer i3, i2, i1, buff_len,buff_id

buff_id = 2 + dir
buff_len = 0

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

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

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

   else if( dir .eq. +1 ) then

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

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

   endif
endif

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

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

   else if( dir .eq. +1 ) then

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

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

   endif
endif

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

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

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

   else if( dir .eq. +1 ) then

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

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

   endif
endif

return
end


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


implicit none

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

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

integer buff_id, indx

integer i3, i2, i1

buff_id = 3 + dir
indx = 0

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

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

   else if( dir .eq. +1 ) then

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

   endif
endif

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

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

   else if( dir .eq. +1 ) then

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

   endif
endif

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

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

   else if( dir .eq. +1 ) then

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

   endif
endif

return
end

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


implicit none

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

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

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

dir = -1

buff_id = 3 + dir
buff_len = nm2

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


dir = +1

buff_id = 3 + dir
buff_len = nm2

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

dir = +1

buff_id = 2 + dir
buff_len = 0

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

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

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

dir = -1

buff_id = 2 + dir
buff_len = 0

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

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

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

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

dir = -1

buff_id = 3 + dir
indx = 0

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

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

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


dir = +1

buff_id = 3 + dir
indx = 0

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

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

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

return
end

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

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

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

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

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

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

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

NAS MG *rprj3* stencil in Chapel



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

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

Our previous work in ZPL showed that compact, global-view codes like these can result in performance that matches or beats hand-coded Fortran+MPI

Summarizing Fragmented/SPMD Models

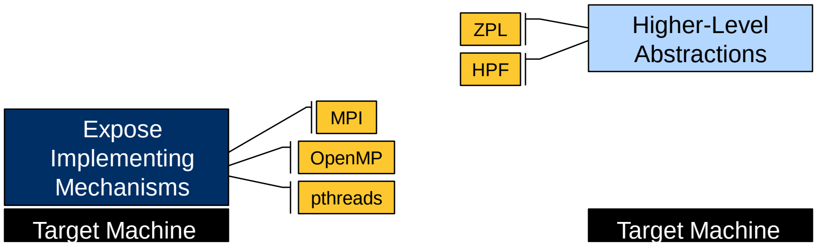
- Advantages:
  - fairly straightforward model of execution
  - relatively easy to implement
  - reasonable performance on commodity architectures
  - portable/ubiquitous
  - lots of important scientific work has been accomplished with them

- 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”

Current HPC Programming Notations

	data / control
communication libraries:	
MPI, MPI-2	fragmented / fragmented/SPMD
SHMEM, ARMCI, GASNet	fragmented / SPMD
shared memory models:	
OpenMP, pthreads	global-view / global-view (trivially)
PGAS languages:	
Co-Array Fortran	fragmented / SPMD
UPC	global-view / SPMD
Titanium	fragmented / SPMD
HPCS languages:	
Chapel	global-view / global-view
X10 (IBM)	global-view / global-view
Fortress (Sun)	global-view / global-view

Parallel Programming Models: Two Camps



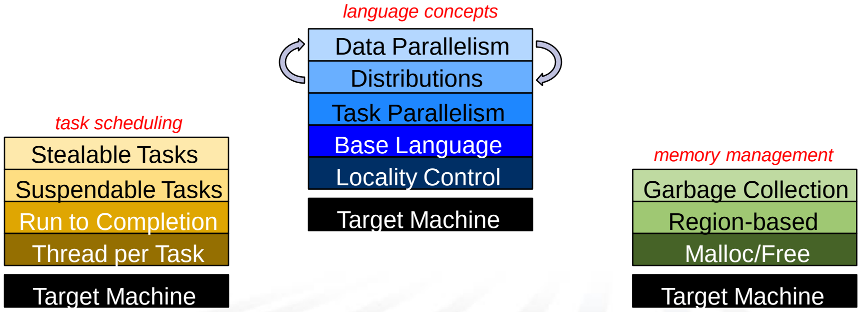
"Why is everything so painful?"

"Why do my hands feel tied?"

Multiresolution Language Design

Our Approach: Permit the language to be utilized at multiple levels, as required by the problem/programmer

- provide high-level features and automation for convenience
- provide the ability to drop down to lower, more manual levels
- use appropriate separation of concerns to keep these layers clean



language concepts

Data Parallelism
Distributions
Task Parallelism
Base Language
Locality Control

Target Machine

task scheduling

Stealable Tasks
Suspendable Tasks
Run to Completion
Thread per Task

Target Machine

memory management

Garbage Collection
Region-based
Malloc/Free

Target Machine

Outline

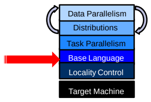
✓ Chapel Context

✓ Terminology: Global-view & Multiresolution Prog. Models

➢ Language Overview
  - Base Language
  - Parallel Features
    - task parallel
    - data parallel
  - Locality Features

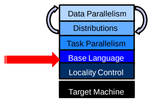
☐ Status, Future Work, Collaborations

Base Language: Design



- 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: Standard Stuff

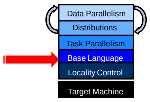


- Lexical structure and syntax based largely on C/C++

```
{ 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++

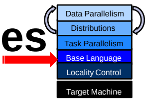
  `var` <varName> `[:` <definition>`]` `[=` <init>`];`

  `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> `{` … `}`

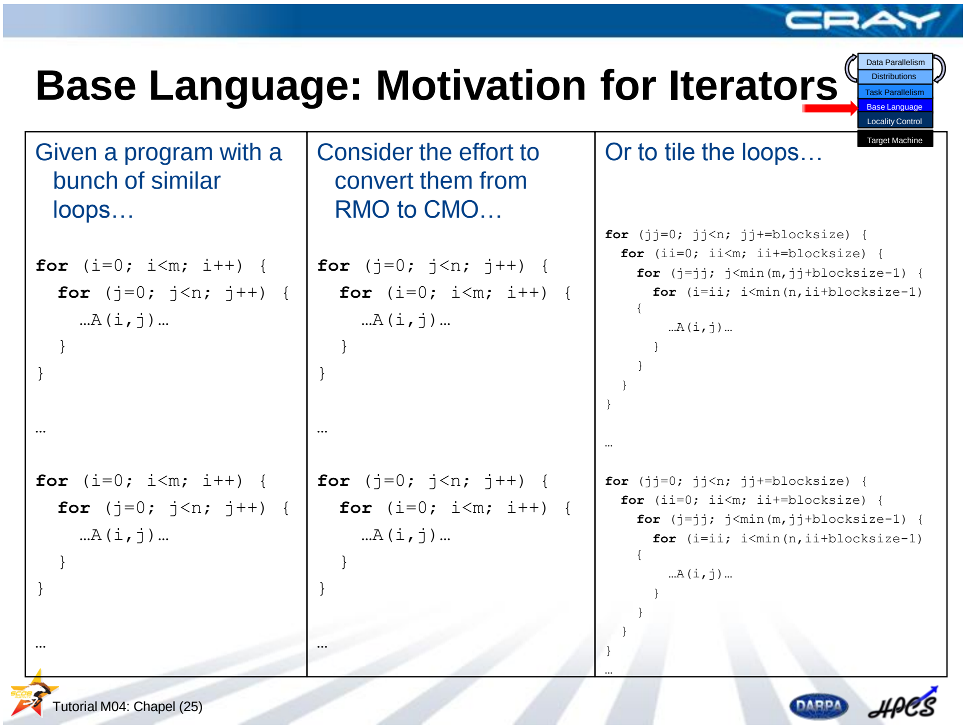
  e.g., **`for`** `i` **`in`** `1..n { … }`
- **Functions:** argument-passing semantics

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…

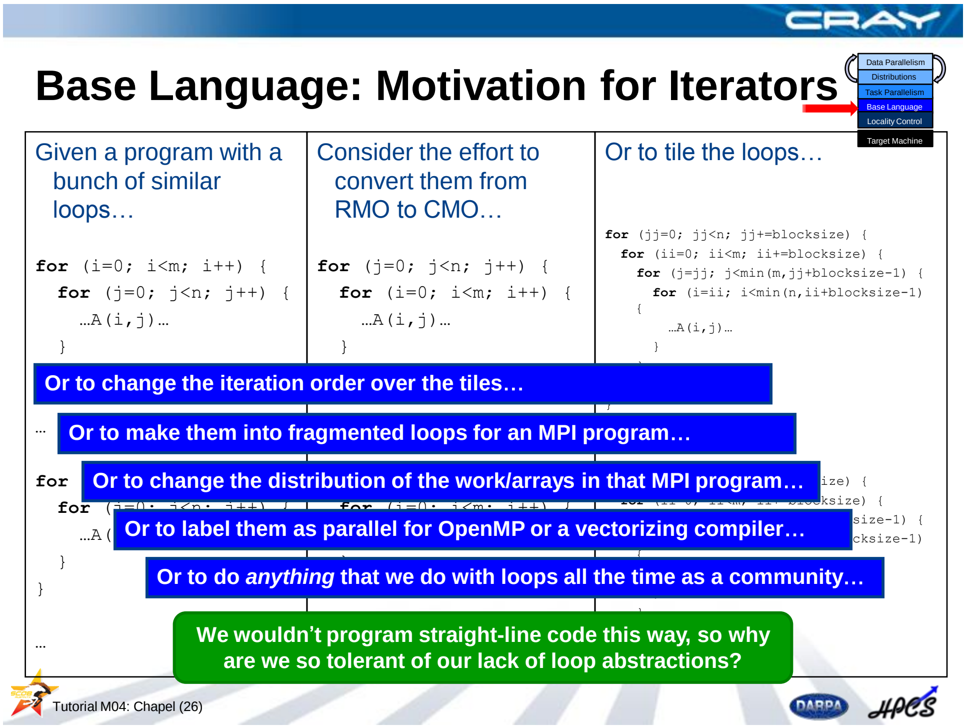
```
for (i=0; i<m; i++) {
  for (j=0; j<n; j++) {
    …A(i,j)…
  }
}

…

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)…
      }
                                    ize) {
                                  ksize) {
                                   size-1) {
                                   cksize-1)
```

Or to change the iteration order over the tiles…

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

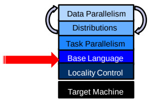
Or to change the distribution of the work/arrays in that MPI program…

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);
}       }
```

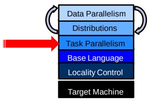
- iterators are used to drive loops:

```
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

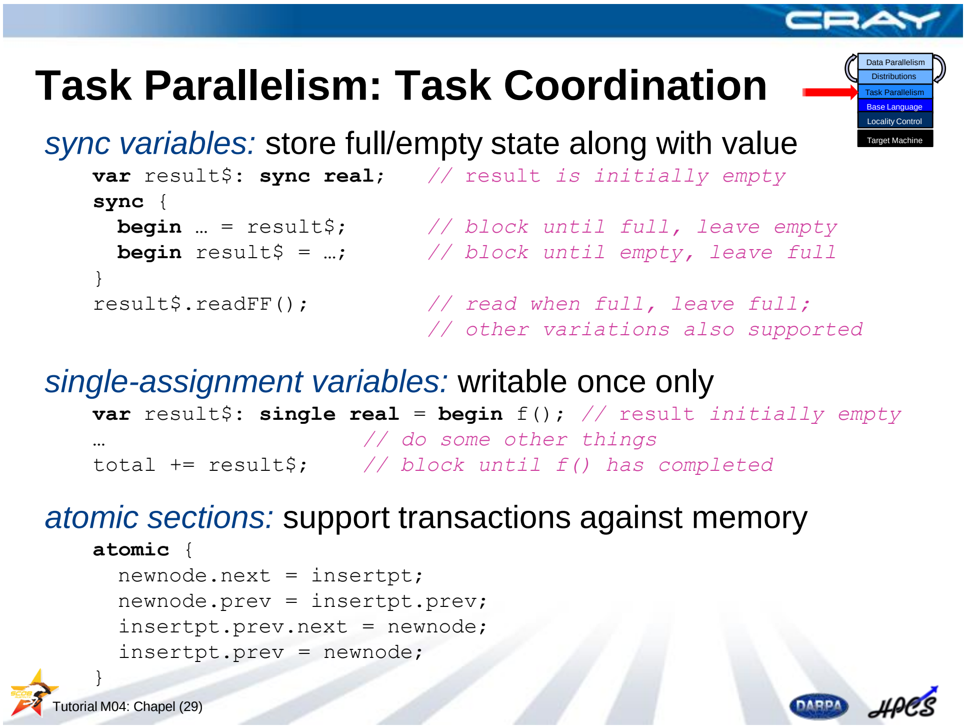


begin: creates a task for future evaluation

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

sync: waits on all begins created within a dynamic scope

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



Task Parallelism: Task Coordination

sync variables: store full/empty state along with value

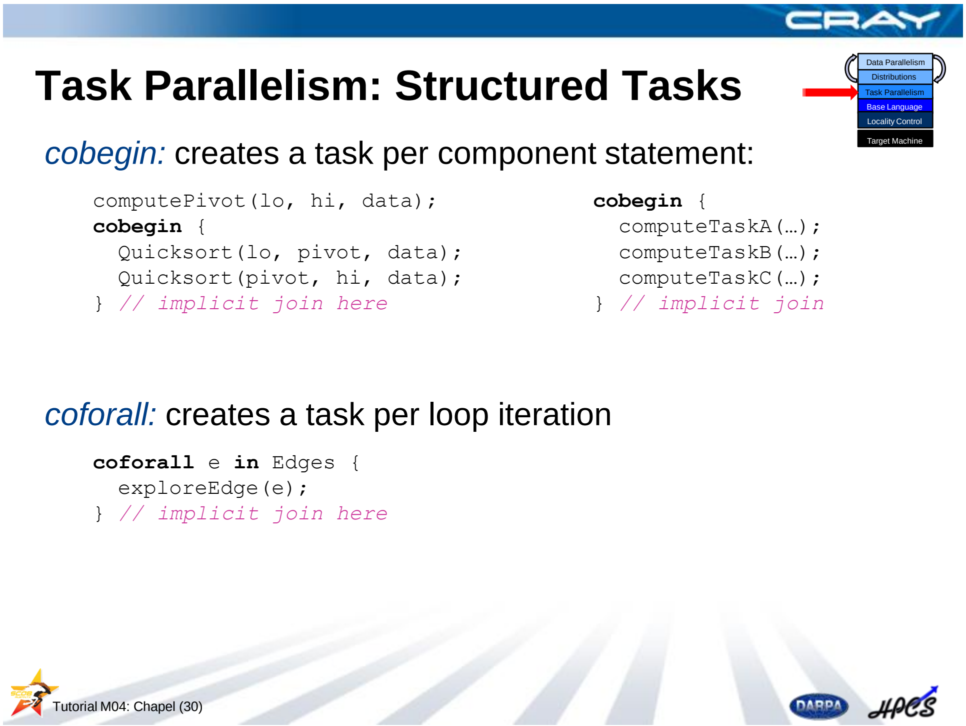
```
var result$: sync real;    // result is initially empty
sync {
  begin … = result$;       // block until full, leave empty
  begin result$ = …;       // block until empty, leave full
}
result$.readFF();          // read when full, leave full;
                           // other variations also supported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

single-assignment variables: writable once only

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

atomic sections: support transactions against memory

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



Task Parallelism: Structured Tasks

cobegin: creates a task per component statement:

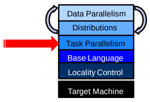
```
computePivot(lo, hi, data);
cobegin {
  Quicksort(lo, pivot, data);
  Quicksort(pivot, hi, data);
} // implicit join here
```

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

coforall: creates a task per loop iteration

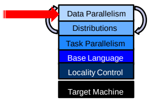
```
coforall e in Edges {
  exploreEdge(e);
} // implicit join here
```

Producer/Consumer example



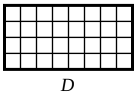
```
var buff$: [0..buffersize-1] sync int;

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)…;
  }
}
```

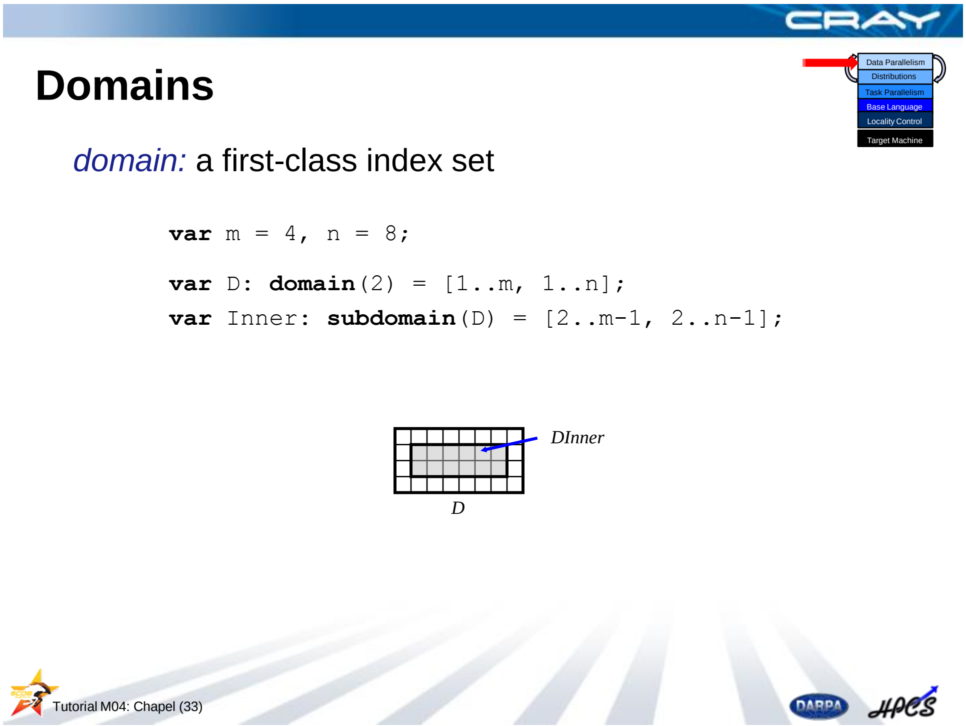
Domains



domain: a first-class index set

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

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

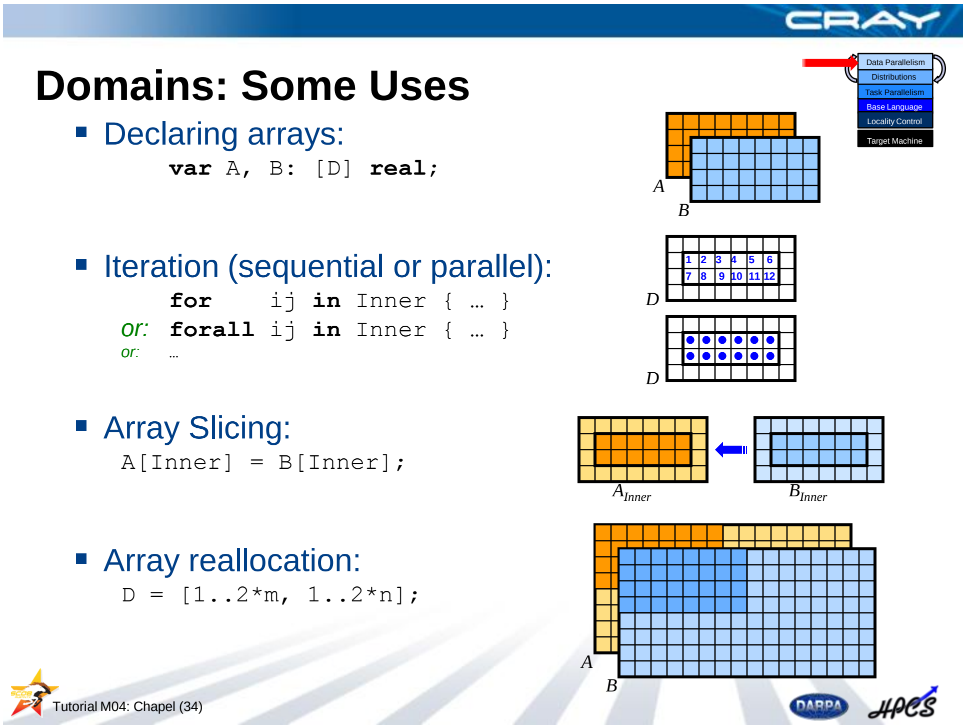




Domains

domain: a first-class index set

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

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



Domains: Some Uses

- Declaring arrays:

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

- Iteration (sequential or parallel):

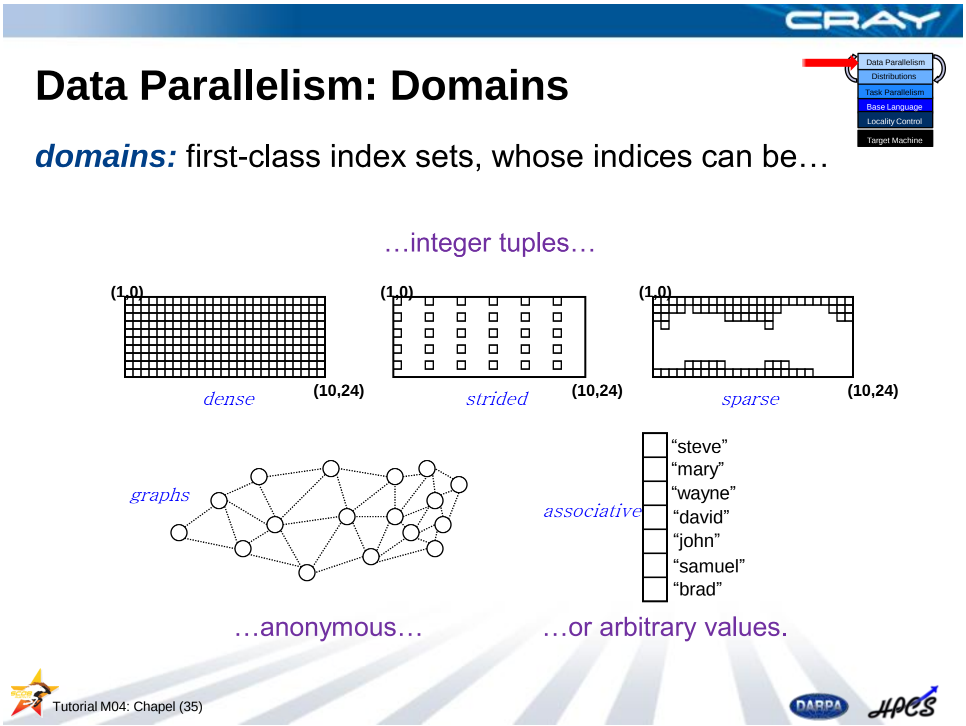
```
    for    ij in Inner { … }
or: forall ij in Inner { … }
or: …
```

- Array Slicing:

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

- Array reallocation:

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



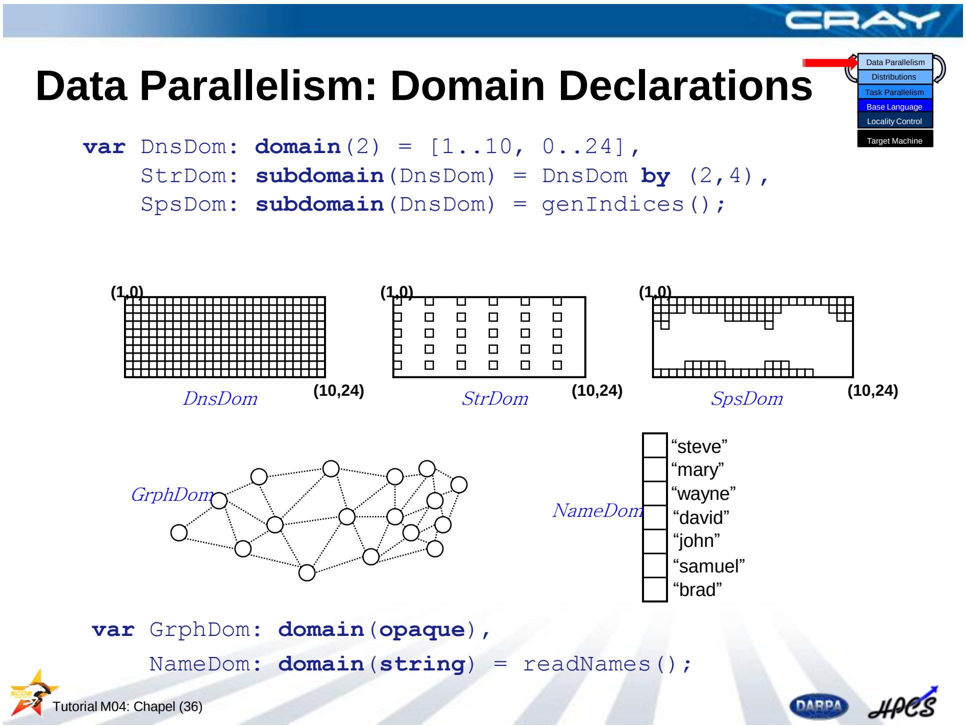
Data Parallelism: Domains

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

…integer tuples…

…anonymous…

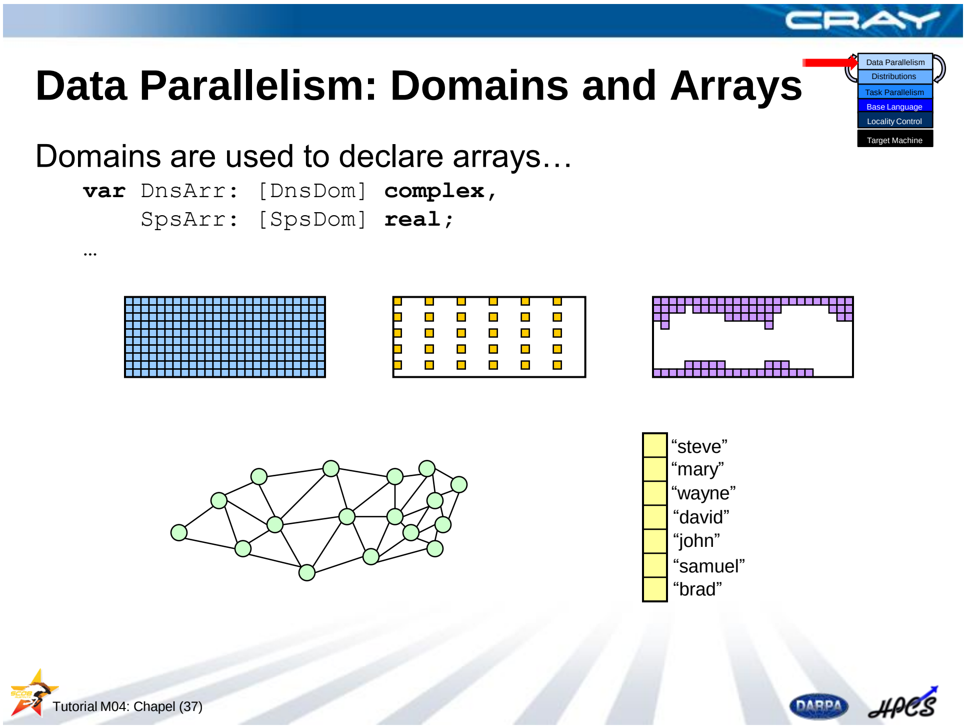
…or arbitrary values.



Data Parallelism: Domain Declarations

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

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

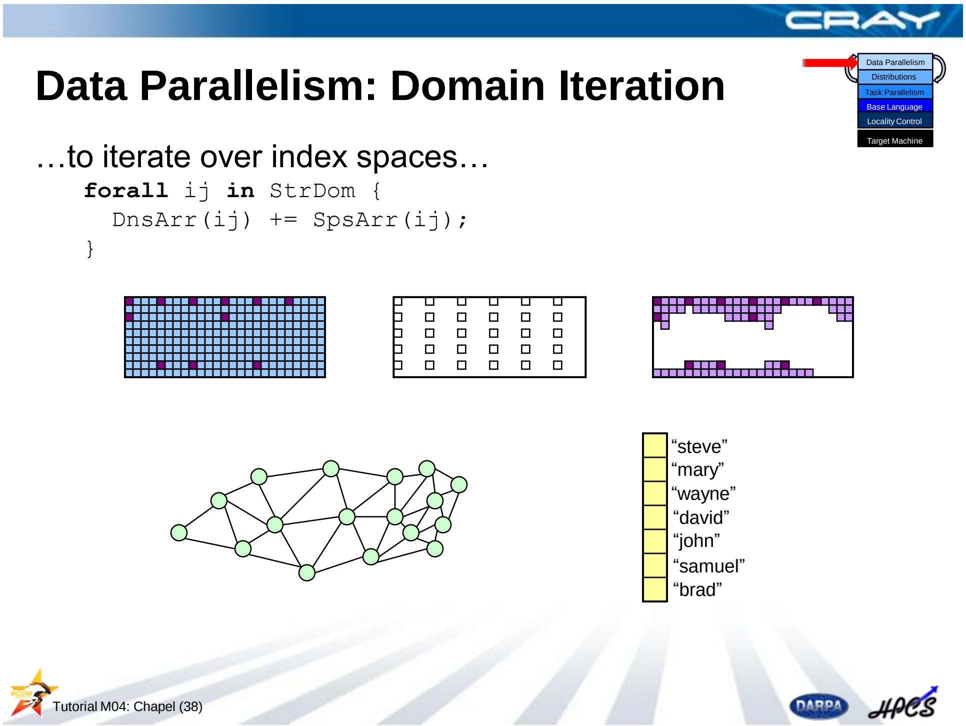


Data Parallelism: Domains and Arrays

Domains are used to declare arrays…

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

Tutorial M04: Chapel (37)

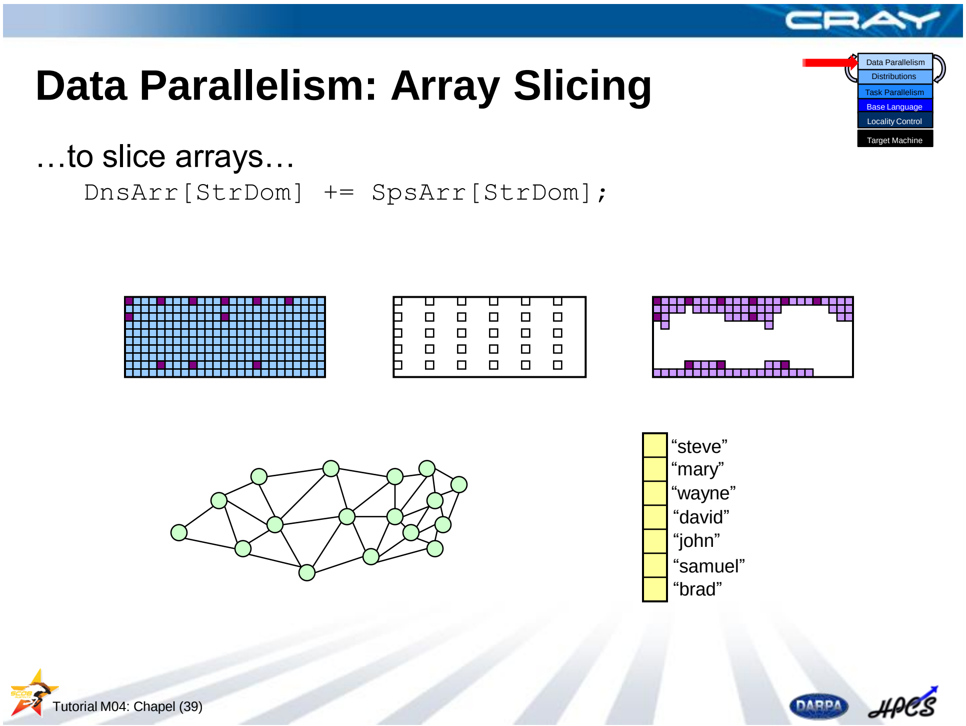


Data Parallelism: Domain Iteration

…to iterate over index spaces…

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

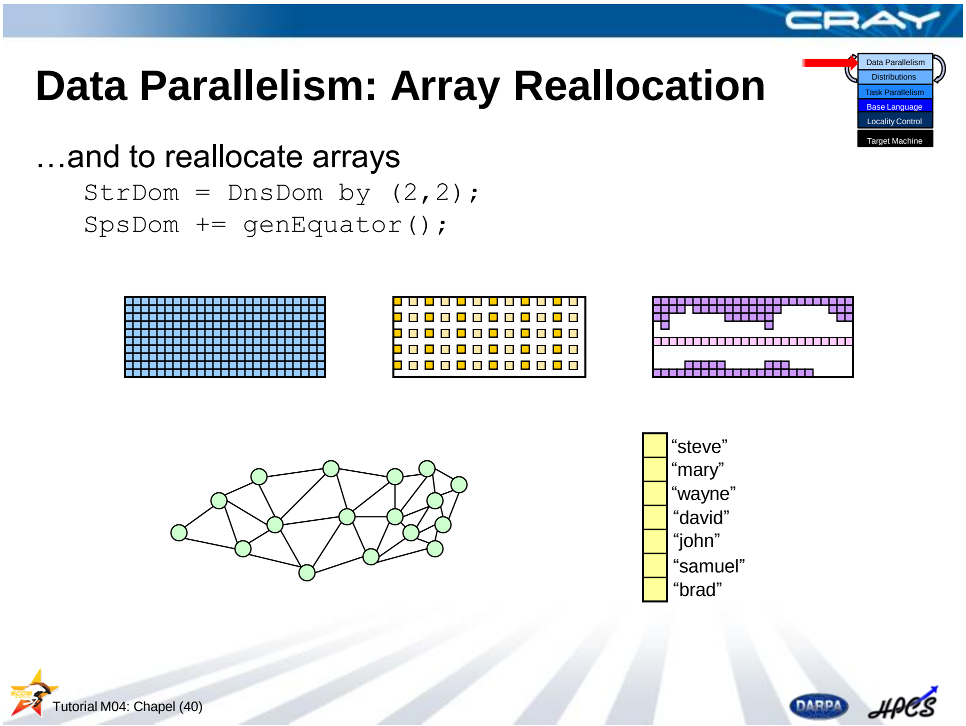
Tutorial M04: Chapel (38)



Data Parallelism: Array Slicing

…to slice arrays…

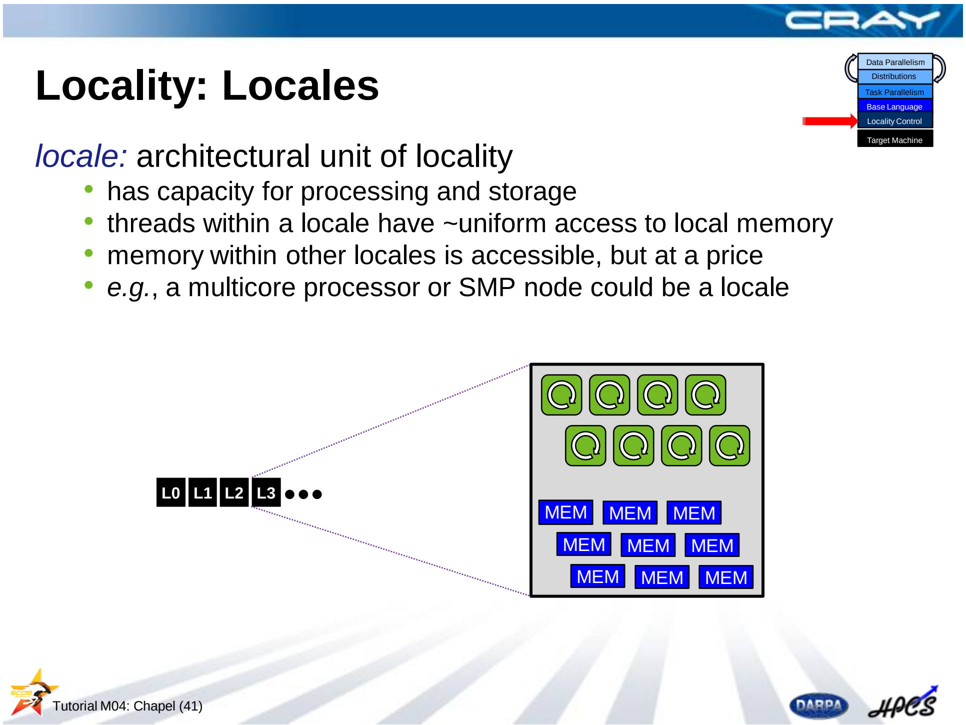
```
DnsArr[StrDom] += SpsArr[StrDom];
```



Data Parallelism: Array Reallocation

…and to reallocate arrays

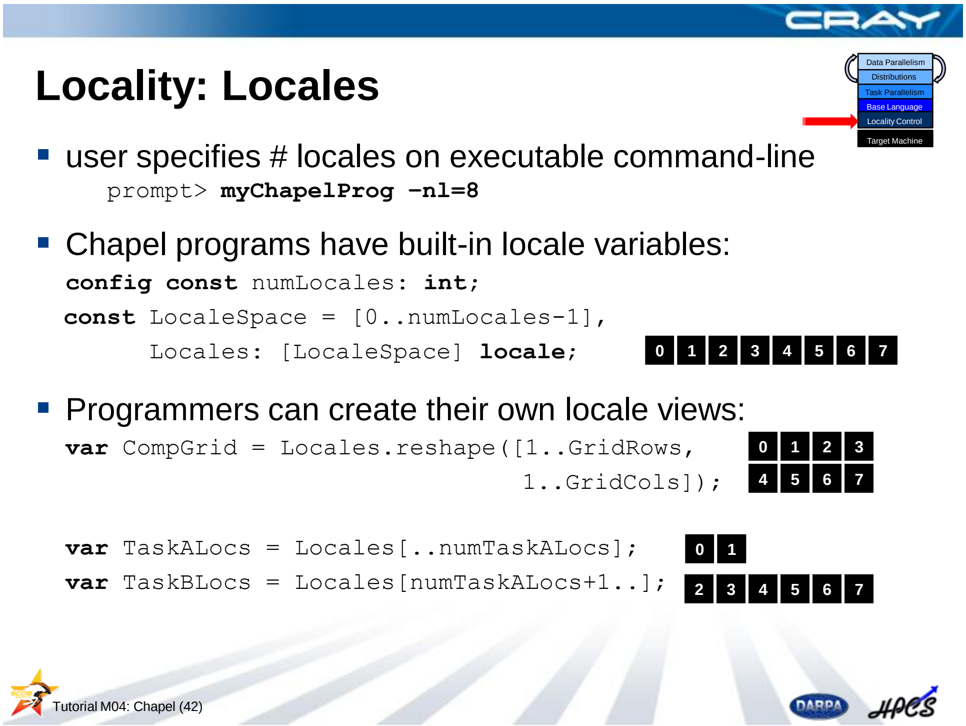
```
StrDom = DnsDom by (2,2);
SpsDom += genEquator();
```



Locality: Locales

locale: architectural unit of locality

- has capacity for processing and storage
- threads within a locale have ~uniform access to local memory
- memory within other locales is accessible, but at a price
- *e.g.*, a multicore processor or SMP node could be a locale



Locality: Locales

- user specifies # locales on executable command-line

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

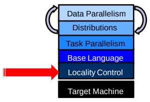
- Chapel programs have built-in locale variables:

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

- Programmers can create their own locale views:

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

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

Locality: Task Placement



on clauses: indicate where tasks should execute

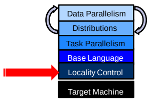
Either in a data-driven manner…

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

…or by naming locales explicitly

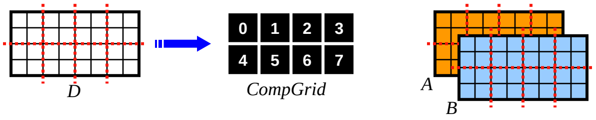
```
cobegin {
  on TaskALocs do computeTaskA(…);
  on TaskBLocs do computeTaskB(…);
  on Locales(0) do computeTaskC(…);
}
```

Locality: Domain Distribution



Domains may be distributed across locales

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



A distribution implies…

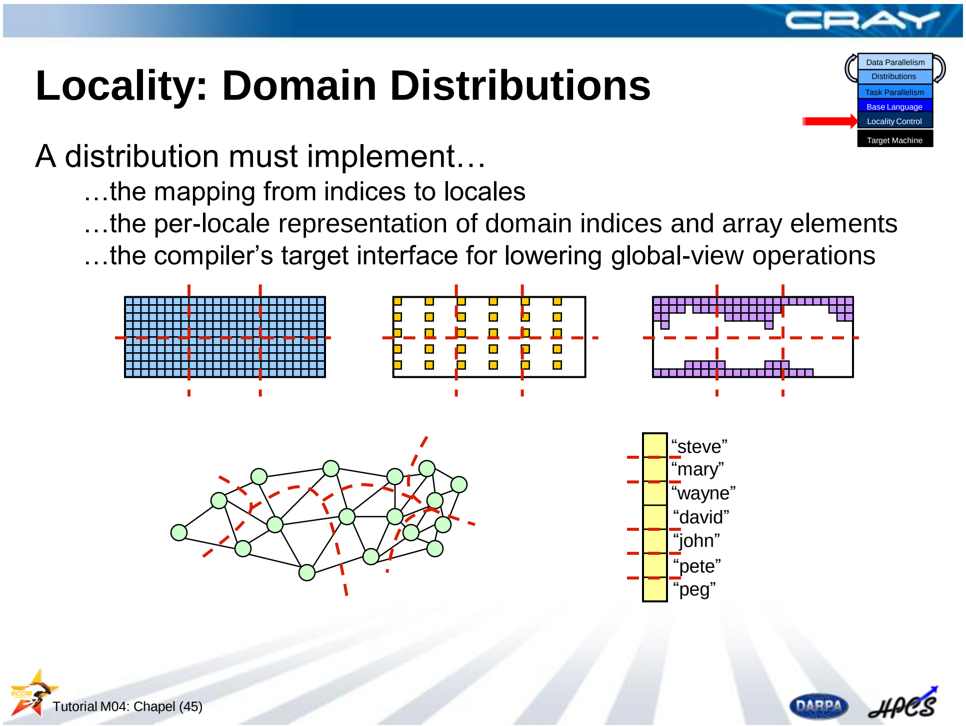
…ownership of the domain's indices (and its arrays' elements)

…the default work ownership for operations on the domains/arrays

Chapel provides…

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

…the means for advanced users to author their own distributions



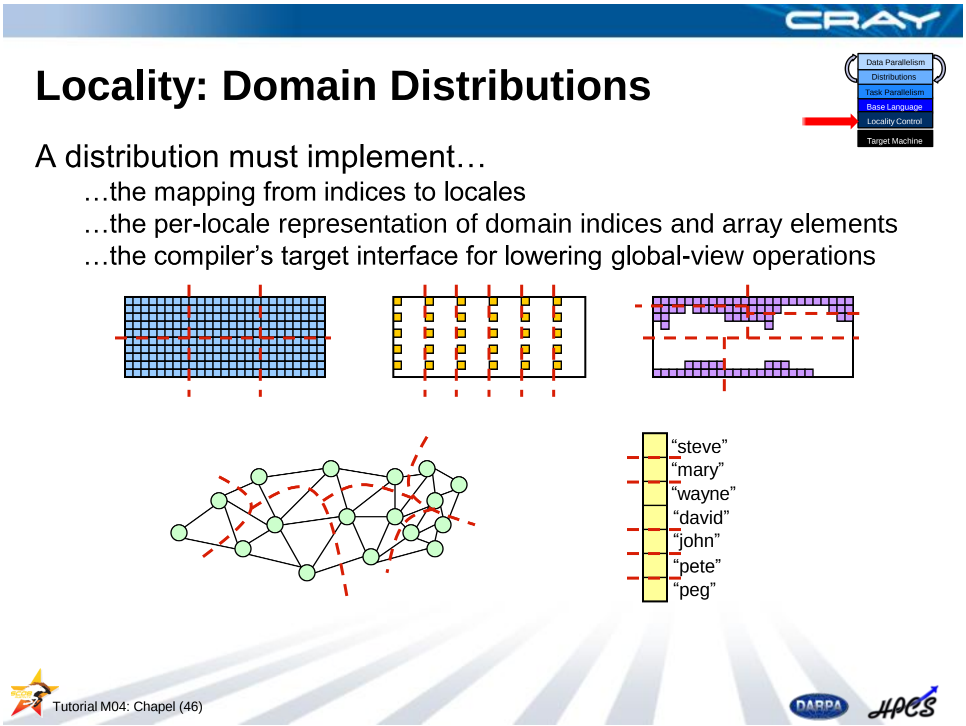
Locality: Domain Distributions

Data Parallelism
Distributions
Task Parallelism
Base Language
Locality Control
Target Machine

A distribution must implement…

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

Tutorial M04: Chapel (45)



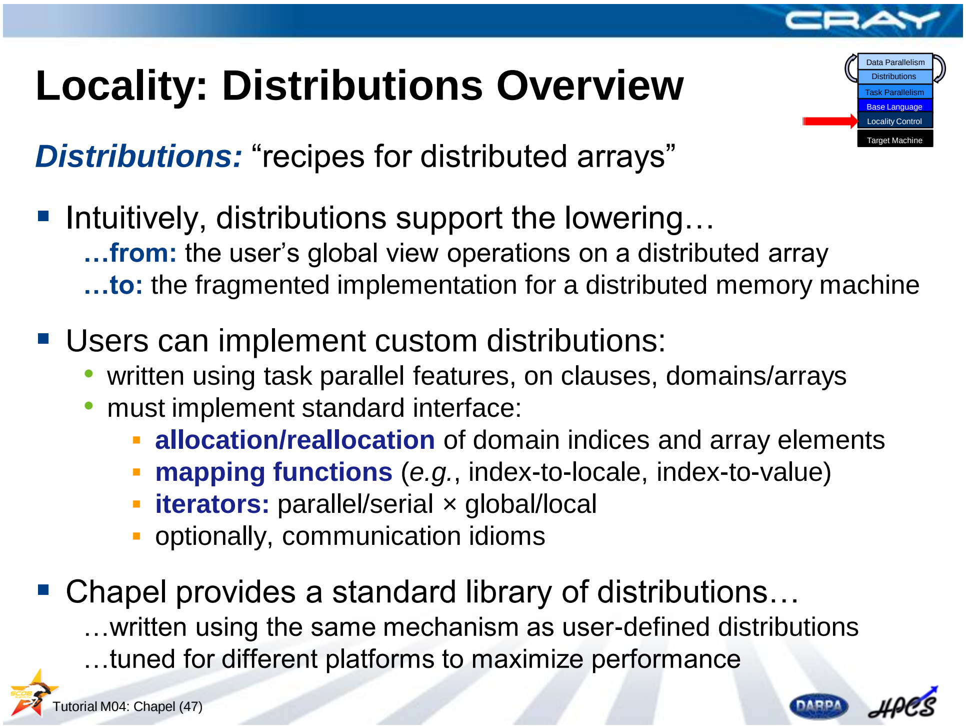
Locality: Domain Distributions

Data Parallelism
Distributions
Task Parallelism
Base Language
Locality Control
Target Machine

A distribution must implement…

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

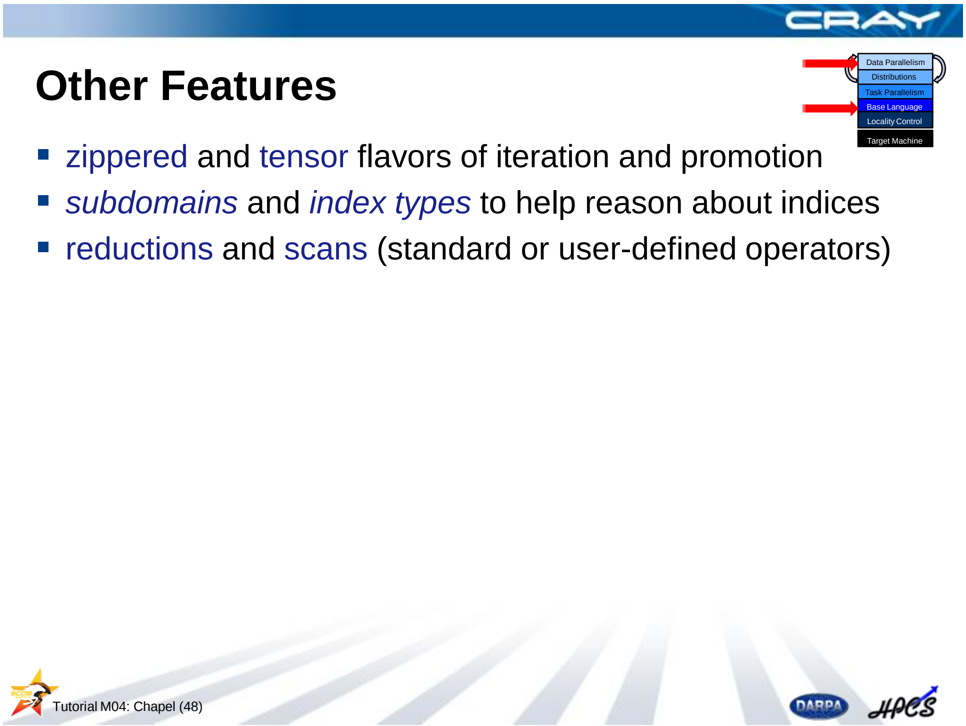
Tutorial M04: Chapel (46)



Locality: Distributions Overview

Distributions: “recipes for distributed arrays”

- Intuitively, distributions support the lowering…
  …from: the user’s global view operations on a distributed array
  …to: the fragmented implementation for a distributed memory machine
- Users can implement custom distributions:
  - written using task parallel features, on clauses, domains/arrays
  - must implement standard interface:
    - **allocation/reallocation** of domain indices and array elements
    - **mapping functions** (*e.g.*, index-to-locale, index-to-value)
    - **iterators:** parallel/serial × global/local
    - optionally, communication idioms
- Chapel provides a standard library of distributions…
  …written using the same mechanism as user-defined distributions
  …tuned for different platforms to maximize performance



Other Features

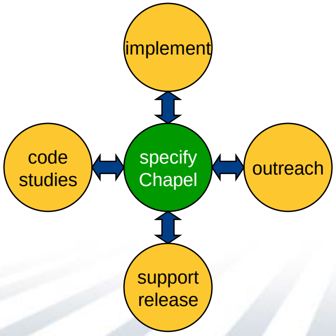
- zippered and tensor flavors of iteration and promotion
- *subdomains* and *index types* to help reason about indices
- reductions and scans (standard or user-defined operators)

Outline

✓ Chapel Context
✓ Global-view Programming Models
✓ Language Overview
☐ Status, Future Work, Collaborations

Chapel Work

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



implement

code studies

specify Chapel

outreach

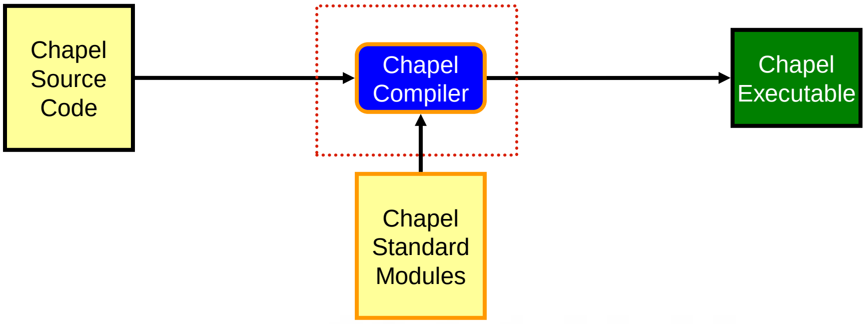
support release

Prototype Compiler Development

- Development Strategy:
  - start by developing and nurturing within Cray under HPCS
  - initial releases to small sets of “friendly” users for past few years
  - public release scheduled for SC08
  - turn over to community when it’s ready to stand on its own
- Compilation 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

---

Compiling Chapel



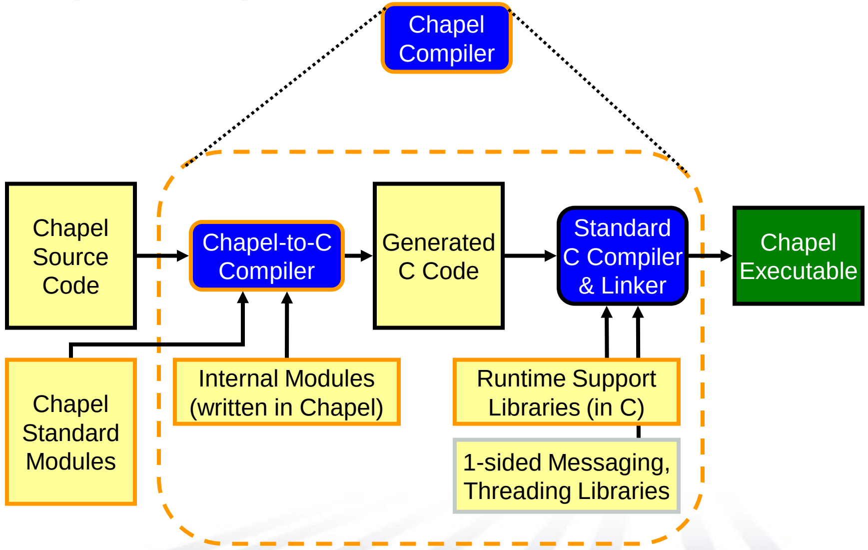
Chapel Source Code

Chapel Compiler

Chapel Executable

Chapel Standard Modules

Chapel Compiler Architecture



Chapel Compiler

Chapel Source Code

Chapel-to-C Compiler

Generated C Code

Standard C Compiler & Linker

Chapel Executable

Chapel Standard Modules

Internal Modules (written in Chapel)

Runtime Support Libraries (in C)

1-sided Messaging, Threading Libraries

Implementation Status

- **Base language:** stable (a few gaps and bugs remain)
- **Task parallel:** stable, multithreaded
- **Data parallel:**
  - stable serial reference implementation
  - initial support for multi-threaded implementation
- **Locality:**
  - stable locale types and arrays
  - stable task parallelism across multiple locales
  - initial support for distributed arrays across multiple locales
- **Performance:**
  - has received much attention in designing the language
  - yet very little implementation effort thus far

Chapel and Research

- Chapel contains a number of research challenges
  - the broadest: “solve the parallel programming problem”
- We intentionally bit off more than an academic project would
  - due to our emphasis on general parallel programming
  - due to the belief that adoption requires a broad feature set
  - to create a platform for broad community involvement
- Most Chapel features are taken from previous work
  - though we mix and match heavily which brings new challenges
- Others represent research of interest to us/the community

Some Research Challenges

- Near-term:
  - user-defined distributions
  - zippered parallel iteration
  - index/subdomain optimizations
  - heterogeneous locale types
  - language interoperability
- Medium-term:
  - memory management policies/mechanisms
  - task scheduling policies
  - performance tuning for multicore processors
  - unstructured/graph-based codes
  - compiling/optimizing atomic sections (STM)
  - parallel I/O
- Longer-term:
  - checkpoint/resiliency mechanisms
  - mapping to accelerator technologies (GP-GPUs, FPGAs?)
  - hierarchical locales

Chapel and Community

- Our philosophy:
  - Help the community understand what we are doing
  - Make our code available to the community
  - Encourage external collaborations

- Goals:
  - to get feedback that will help make the language more useful
  - to support collaborative research efforts
  - to accelerate the implementation
  - to aid with adoption

Current Collaborations

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

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

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

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

(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…)

Next Steps

- Continue to improve performance
- Continue to add missing features
- Expand the set of codes that we are currently studying
- Expand the set of architectures that we are targeting
- Support the public release
- Continue to support collaborations and seek out new ones

Summary

Chapel strives to solve the Parallel Programming Problem

through its support for…

…general parallel programming
…global-view abstractions
…control over locality
…multiresolution features
…modern language concepts and themes

Chapel Team

- Current Team
  - Brad Chamberlain 
  - Samuel Figueroa 
  - Steve Deitz 
  - David Iten 
- Interns
  - Robert Bocchino (`06 – UIUC)
  - James Dinan (`07 – Ohio State)
  - Mackale Joyner (`05 – Rice)
  - Andy Stone (`08 – Colorado St)
- Alumni
  - David Callahan
  - Roxana Diaconescu
  - Shannon Hoffswell
  - Mary Beth Hribar
  - Mark James
  - John Plevyak
  - Wayne Wong
  - Hans Zima

For More Information

chapel_info@cray.com

http://chapel.cs.washington.edu

Parallel Programmability and the Chapel Language; Chamberlain, Callahan, Zima; International Journal of High Performance Computing Applications, August 2007, 21(3):291-312.

Questions?

SC08: Tutorial M04 – 11/17/08