# Chapel
## the Cascade High Productivity Language

Brad Chamberlain
Cray Inc.

**Bridging Multicore's Programmability Gap**

**SC08: November 17, 2008**

# Multicore Systems and HPC

- Multicore is here, apparently to stay awhile
  - for the mainstream programmer and the HPC programmer alike
- For the HPC programmer, is the sky falling? Or not?
  - Perhaps multicore can be effectively harnessed with MPI + OpenMP?
  - Or, perhaps it can be effectively harnessed with MPI alone?

    *(Many will argue that this was the case for clusters of SMPs)*
- Or…
  - Perhaps MPI + OpenMP were already causing a programmability gap on single core systems and we've just become numb to it as a community?

# MPI (Message Passing Interface)

## MPI strengths

- \+ people are able to accomplish real work with it
- \+ it runs on most parallel platforms
- \+ it is relatively easy to implement (or, that's the conventional wisdom)
- \+ for many architectures, it can result in near-optimal performance
- \+ it serves as a strong foundation for higher-level technologies

## MPI weaknesses

- – encodes too much about "how" data should be transferred rather than simply "what data" (and possibly "when")
  - can mismatch architectures with different data transfer capabilities
- – only supports parallelism at the "cooperating executable" level
  - applications and architectures contain parallelism at many levels
  - doesn't reflect how one abstractly thinks about parallel algorithms

# What problems are poorly served by MPI?

**My response:** What problems are *well-served* by MPI?

*“well-served”:* MPI is a natural (productive?) form for expressing them

- **embarrassingly parallel:** arguably
- **data parallel:** not particularly, due to cooperating executable issues
  - communication, synchronization, data replication
  - bookkeeping details related to manual data decomposition
  - local vs. global indexing issues
  - code can be obfuscated/brittle due to these issues
- **task parallel:** even less so
  - *e.g.*, write a divide-and-conquer algorithm in MPI…
    …without MPI-2 dynamic process creation – yucky
    …with it, your unit of parallelism is the executable – weighty

# What might one desire in an alternative?

## General programming models with broad applicability

- any parallel program you want to write should be expressible
- should map well to arbitrary parallel architectures
- in particular, we should break away from SPMD prog./exec. models
  - should be a case worth optimizing for, not the only tool in the box

## Ones that separate concerns appropriately

- *e.g.*, separate expression of parallelism/locality from implementing mechanisms

## Ones that admit optimization

- by a compiler
- by a sufficiently motivated programmer

## Ones that interoperate with existing programming models

- to preserve legacy codes and flexibility

# 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
- ***multiresolution* design**
- ***global-view* abstractions**
- **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:

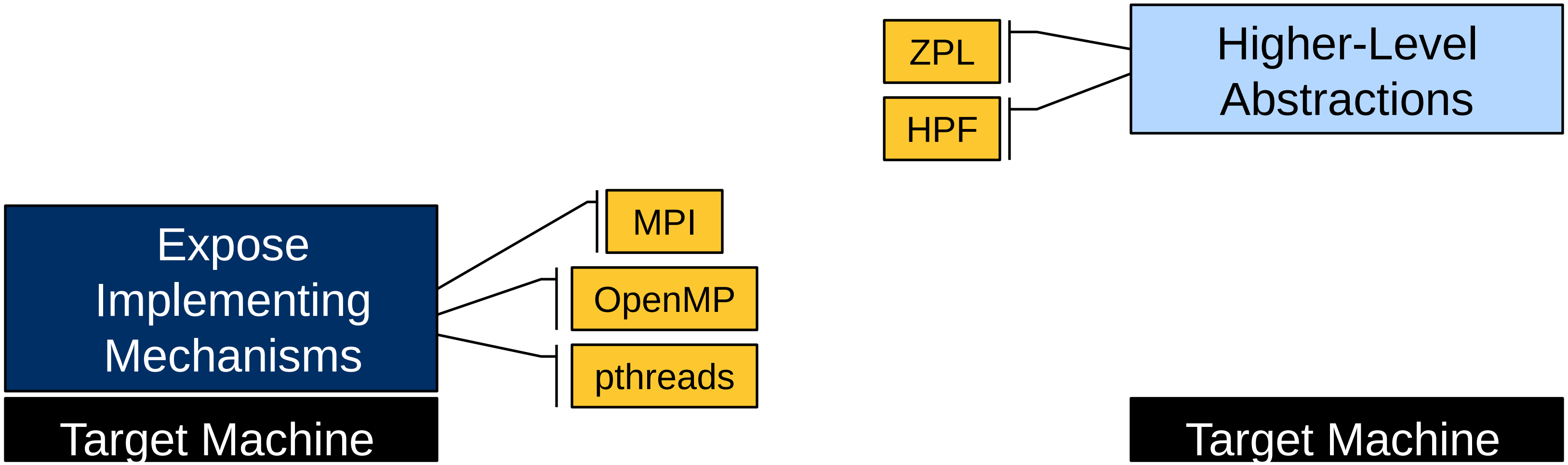
      Cray: Chapel    IBM: X10    Sun: Fortress

- **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)

# Outline

- ✓ Chapel Context
- ➢ Terminology: Multiresolution & Global-view Programming Models
- ❑ Language Overview
- ❑ Chapel and Mainstream Multicore
- ❑ Status, Future Work, Collaborations

# Parallel Programming Models: Two Camps



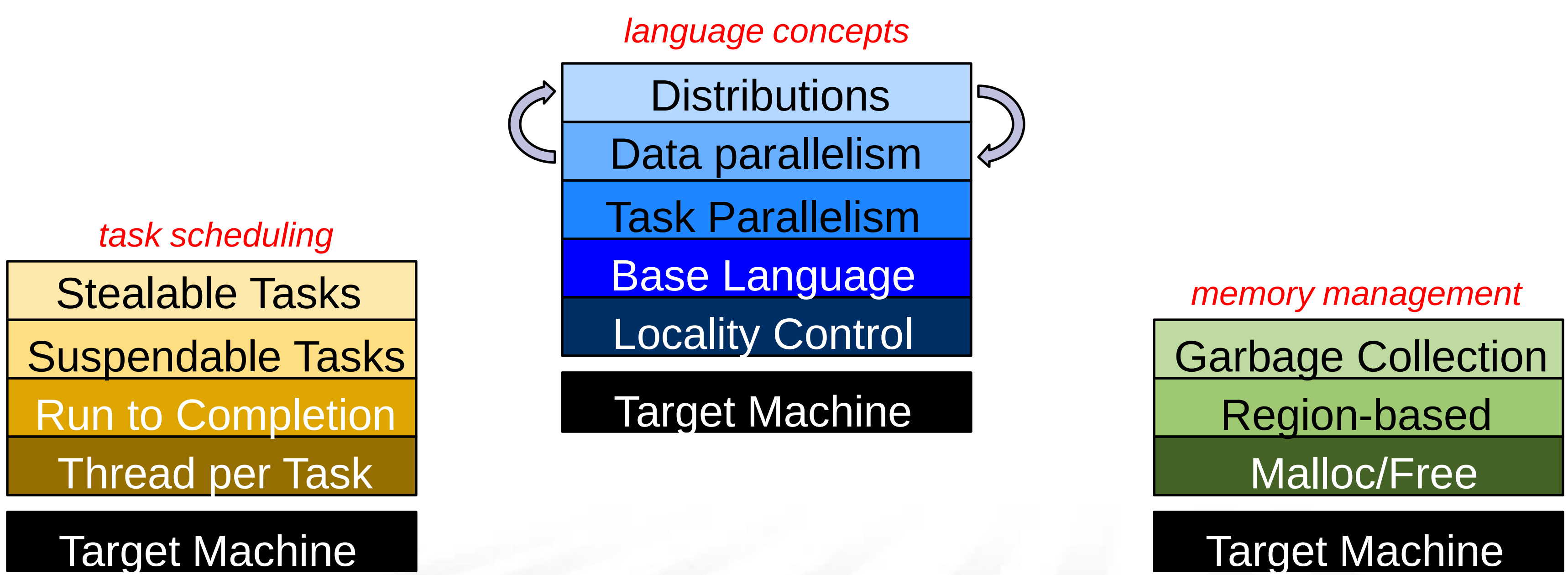

"Why is everything so painful?"

"Why do my hands feel tied?"

# Multiresolution Language Design

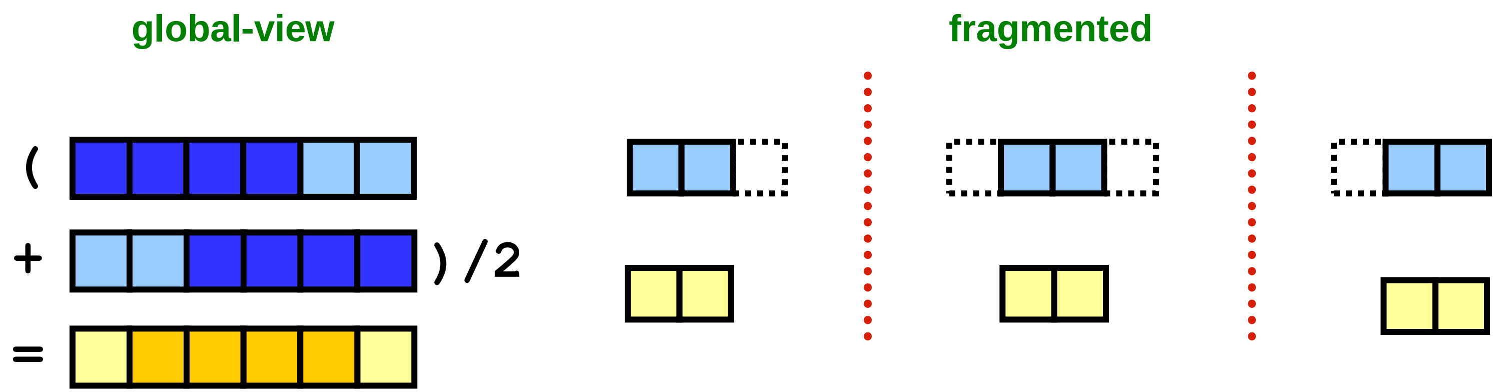
**Our Approach:** Permit the language to be utilized at multiple levels, as required by the problem/programmer

- 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



# Global-view vs. Fragmented

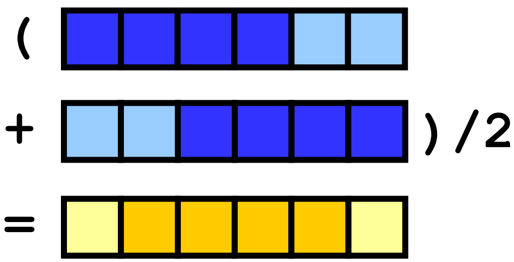
**Problem:** "Apply 3-pt stencil to vector"



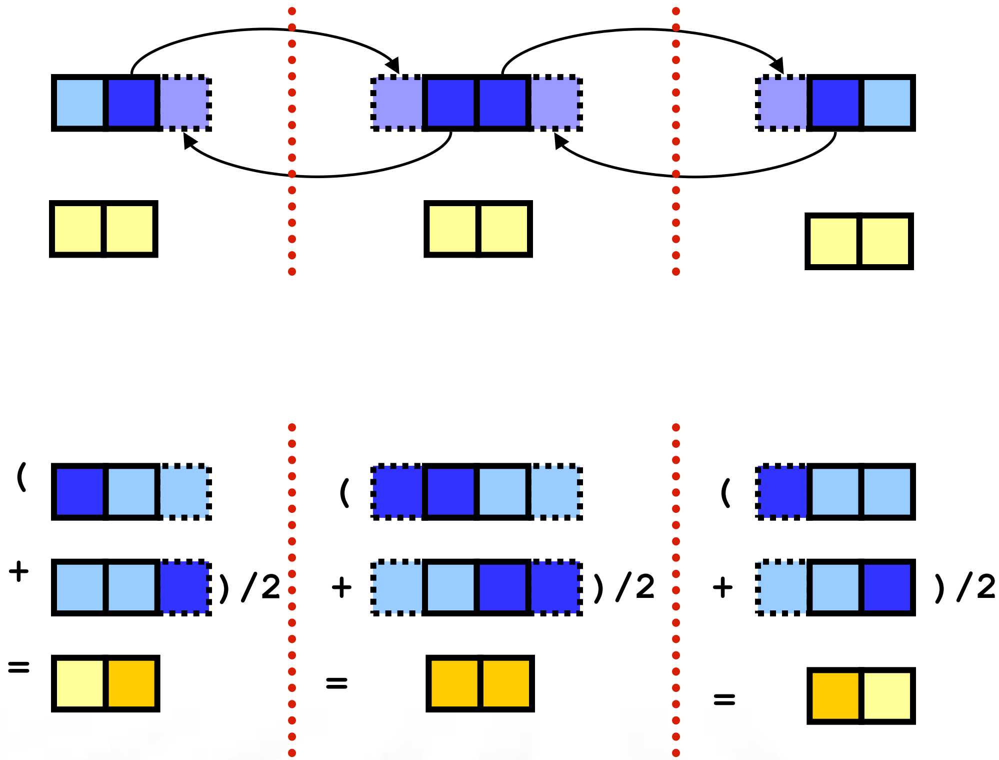
# Global-view vs. Fragmented

**Problem:** "Apply 3-pt stencil to vector"

global-view



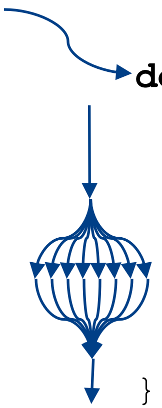

fragmented



# Global-view vs. SPMD Code

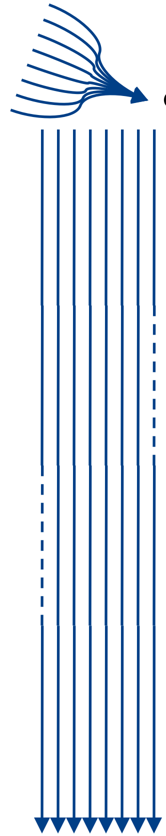
**Problem:** "Apply 3-pt stencil to vector"

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



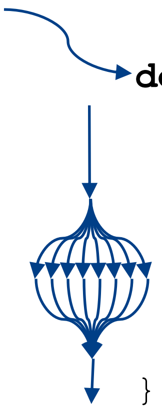
```
def main() {
  var n: int = 1000;
  var locN: int = n/numProcs;
  var a, b: [0..locN+1] real;

  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



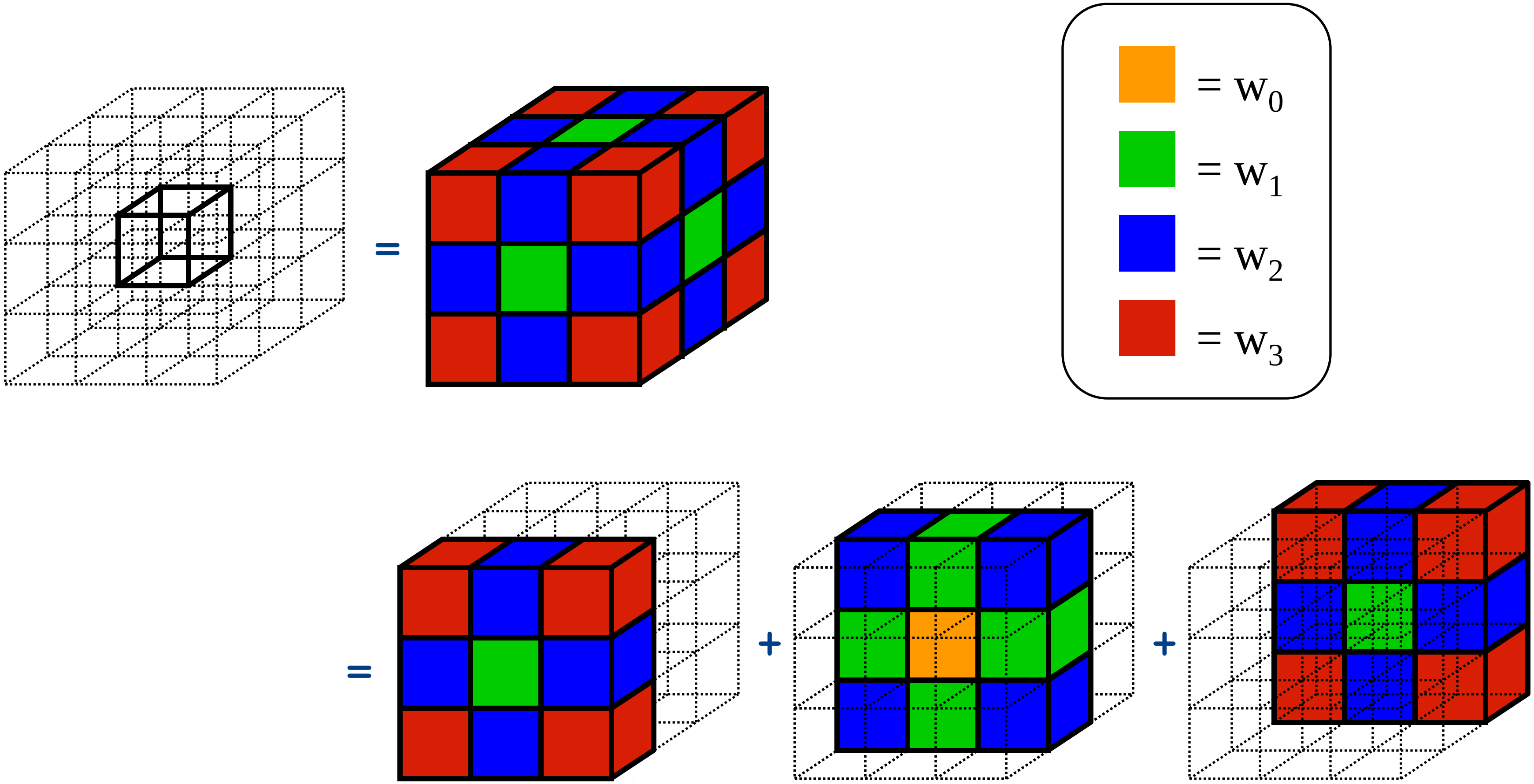
```
def main() {
  var n: int = 1000;
  var a, b: [1..n] real;

  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;
  }
}
```

**Communication becomes geometrically more complex for higher-dimensional arrays**

# *rprj3* stencil from NAS MG

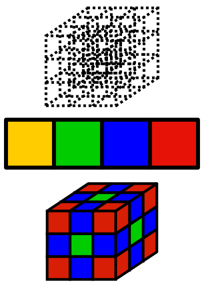


# NAS MG *rprj3* stencil in Fortran + MPI

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

implicit none

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

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

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


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


implicit none

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

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

integer i3, i2, i1, buff_len,buff_id

buff_id = 2 + dir
buff_len = 0

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

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

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

   else if( dir .eq. +1 ) then

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

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

   endif
endif

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

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

   else if( dir .eq. +1 ) then

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

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

   endif
endif

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

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

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

   else if( dir .eq. +1 ) then

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

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

   endif
endif

return
end


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


implicit none

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

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

integer buff_id, indx

integer i3, i2, i1

buff_id = 3 + dir
indx = 0

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

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

   else if( dir .eq. +1 ) then

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

   endif
endif

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

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

   else if( dir .eq. +1 ) then

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

   endif
endif

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

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

   else if( dir .eq. +1 ) then

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

   endif
endif

return
end


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


implicit none

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

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

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

dir = -1

buff_id = 3 + dir
buff_len = nm2

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


dir = +1

buff_id = 3 + dir
buff_len = nm2

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

dir = +1

buff_id = 2 + dir
buff_len = 0

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

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

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

dir = -1

buff_id = 2 + dir
buff_len = 0

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

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

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

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

dir = -1

buff_id = 3 + dir
indx = 0

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

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

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


dir = +1

buff_id = 3 + dir
indx = 0

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

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

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

return
end

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

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

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

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

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

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

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

# 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 this can result in performance that matches or beats hand-coded Fortran+MPI while also supporting more runtime flexibility*

*(see backup slides for more details)*

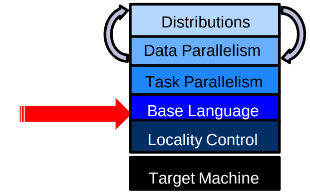
# Current HPC Programming Notations

| | 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 |

# Outline

- ✓ Chapel Context
- ✓ Terminology: Global-view & Multiresolution Prog. Models
- ➢ Language Overview
  - Base Language
  - Parallel Features
    - task parallel
    - data parallel
  - Locality Features
- ❑ Chapel and Mainstream Multicore
- ❑ 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#)

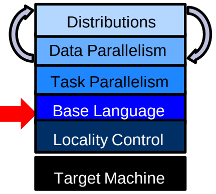
  **ML:** latent types (see also Scala, Matlab, Perl, Python, C#)

  **Java, C#:** OOP, type safety

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

  **C, Modula, Ada:** syntax

# Base Language: Standard Stuff

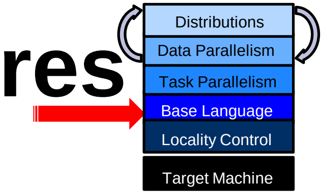



- Lexical structure and syntax based largely on C family
  - main departures: variable/function declarations and for loops

```
{ 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 (think: C++-style and Java-style)
  - yet, no strong requirement to use OOP
- Modules for namespace management
- Generic functions and classes

# Base Language: My Favorite Departures

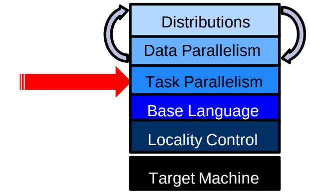



- **Rich compile-time language**
  - parameter values (compile-time constants)
  - folded conditionals, unrolled for loops, expanded tuples
  - type and parameter functions – evaluated at compile-time
- **Latent types:**
  - ability to omit type specifications for convenience or reuse
  - type specifications 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 ./a.out --n=1000000
```

- **Tuples**
- **Iterators (in the CLU, Ruby sense)**

# Task Parallelism: Task Creation



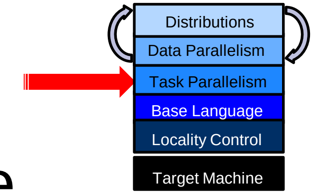

## *begin:* creates a task for future evaluation

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

## *sync:* waits on all begins created within a dynamic scope

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

def treeSearch(node) {
  if node == nil then return;
  begin treeSearch(node.right);
  begin treeSearch(node.left);
}
```

# 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$.readXX();         // read value, leave state unchanged;
                          // other variations also supported
```

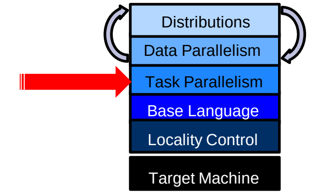
*single-assignment variables:* writable once only

```
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



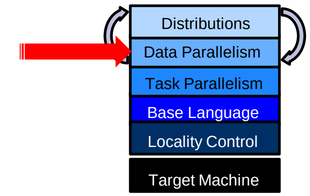

*cobegin:* creates a task per component statement:

```
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

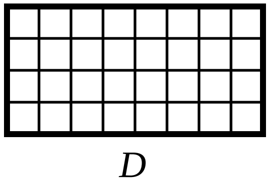
```
coforall e in Edges {
  exploreEdge(e);
} // implicit join here
```



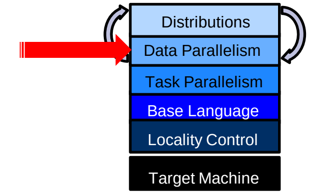

# Domains

*domain:* a first-class index set

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

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

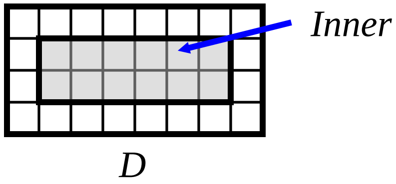


# Domains

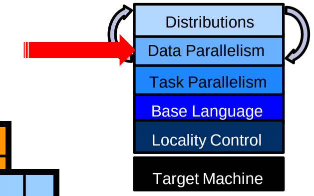



*domain:* a first-class index set

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

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

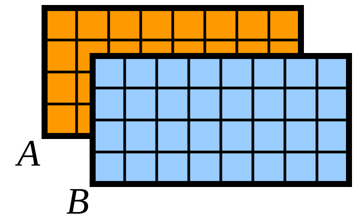


# Domains: Some Uses



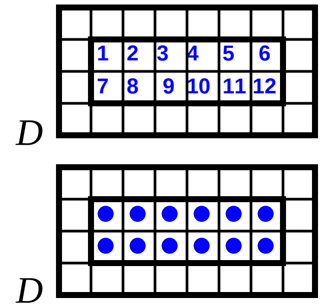

- Declaring arrays:

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



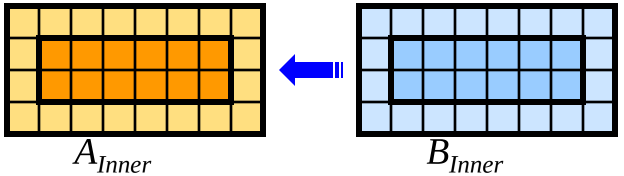

- Iteration (sequential or parallel):

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



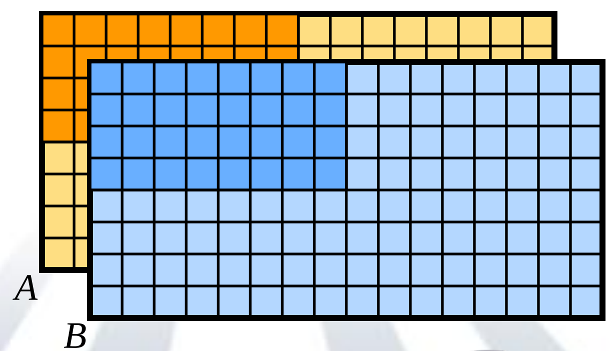

- Array Slicing:

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

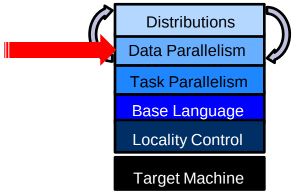


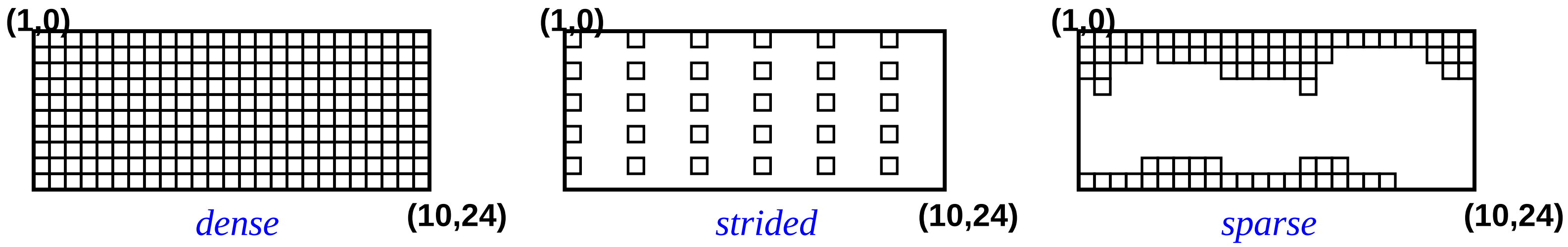

- Array reallocation:

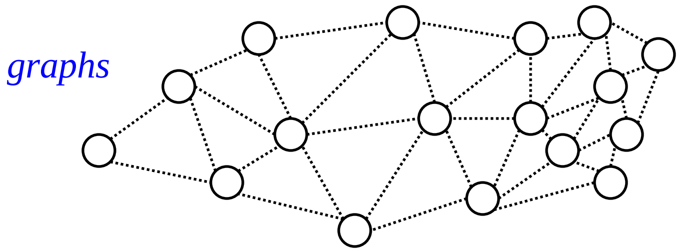
```
D = [1..2*m, 1..2*n];
```

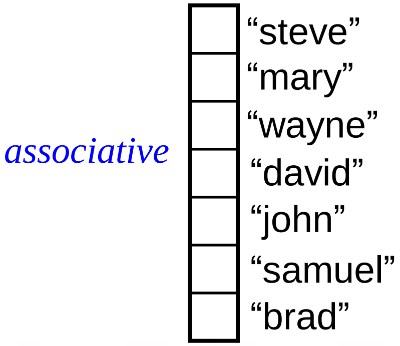


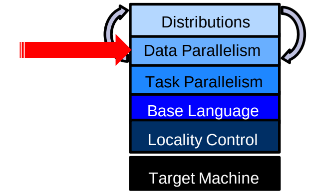
# Data Parallelism: Other Domains



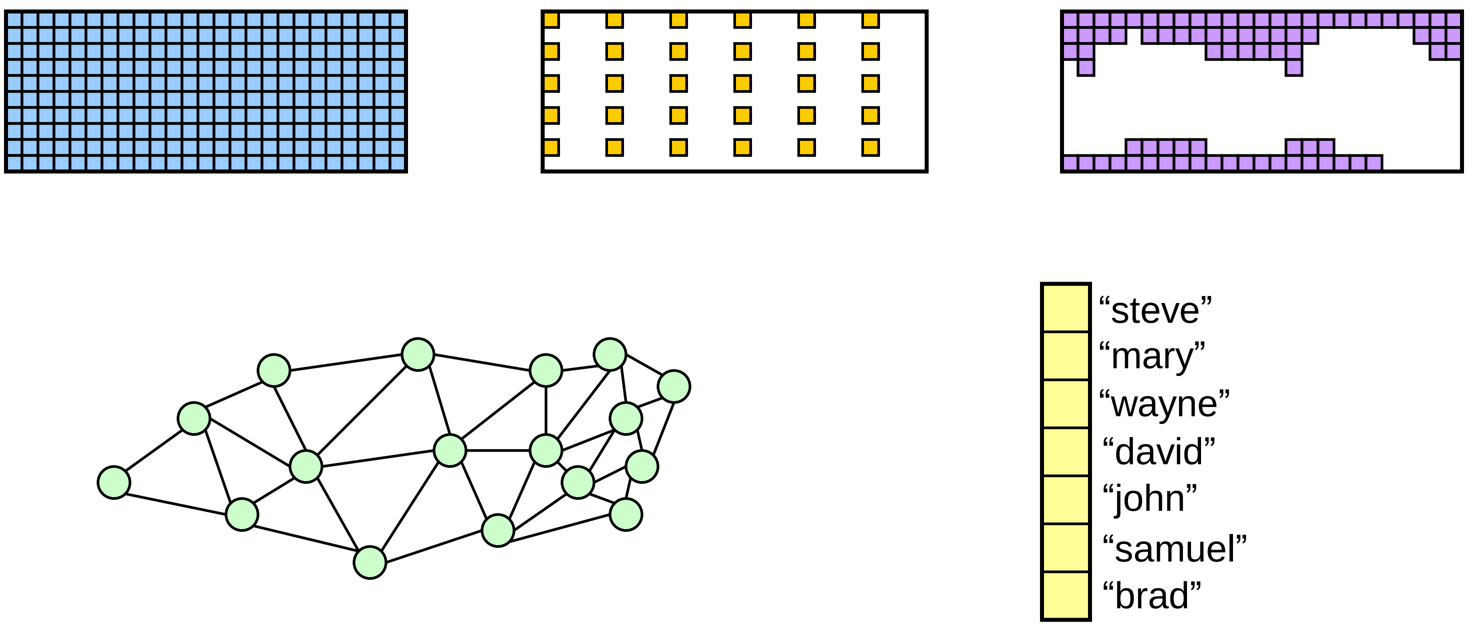






# Data Parallelism: Domain Uses

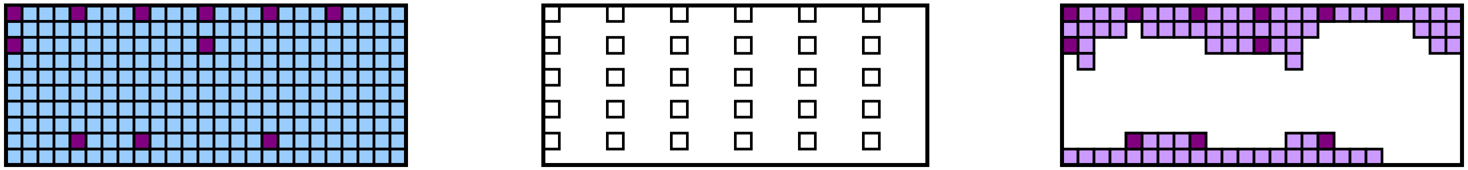
Domains are used to declare arrays…

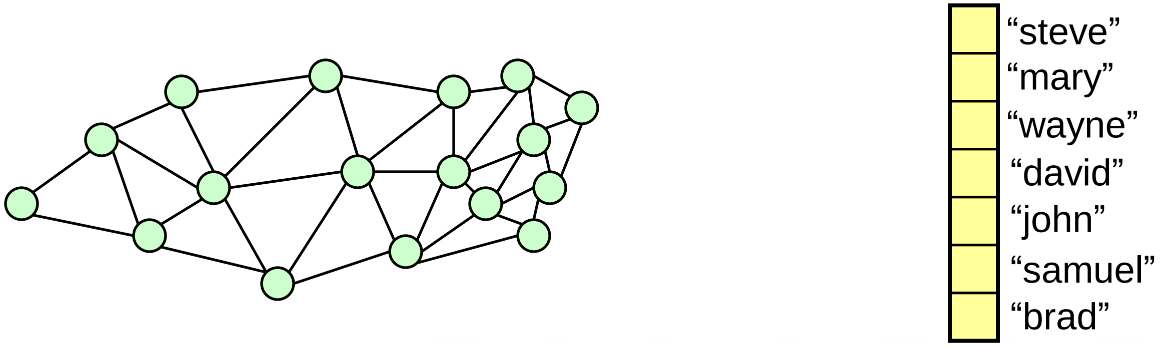


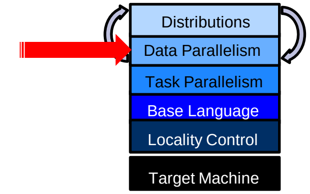
# Data Parallelism: Domain Uses

…to iterate over index sets…

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



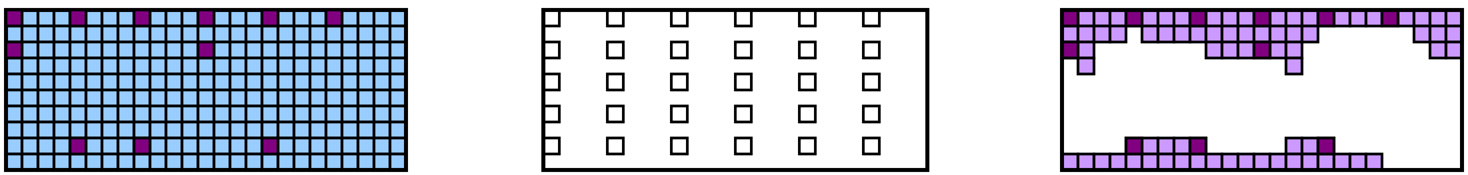


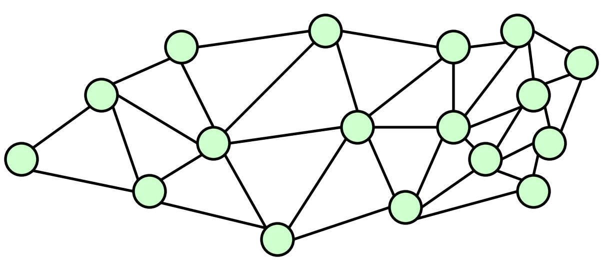


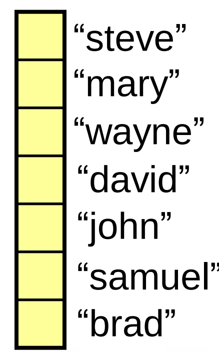

# Data Parallelism: Domain Uses

…to slice arrays…

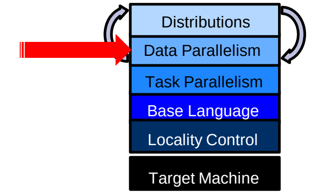
```
DnsArr[StrDom] += SpsArr[StrDom];
```





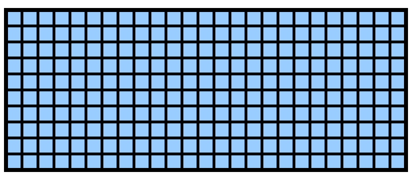


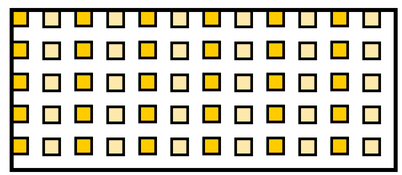
# Data Parallelism: Domain Uses

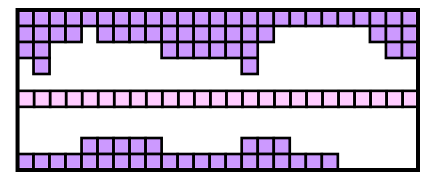


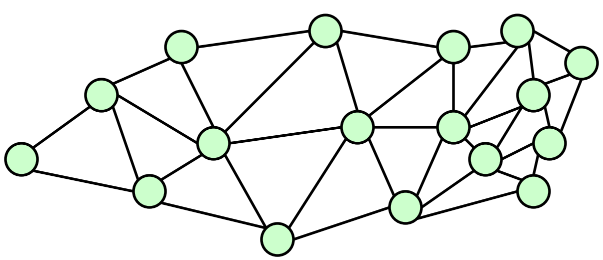

…and to reallocate arrays

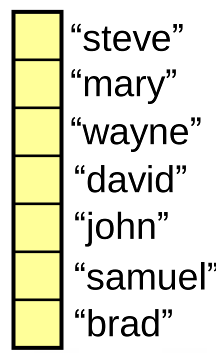
```
StrDom = DnsDom by (2,2);
SpsDom += genEquator();
```

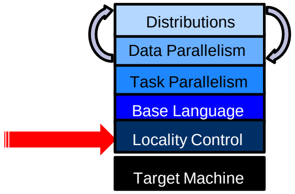








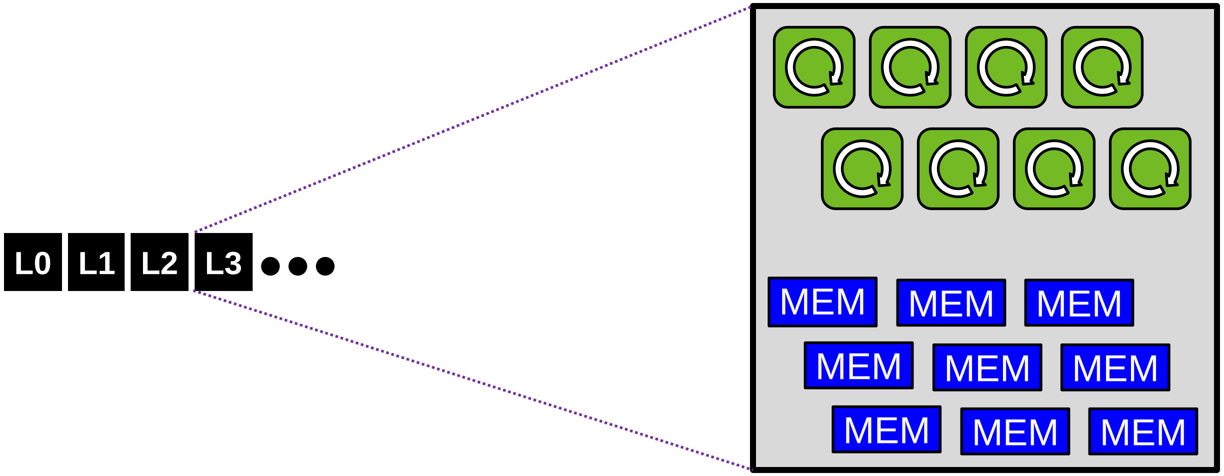




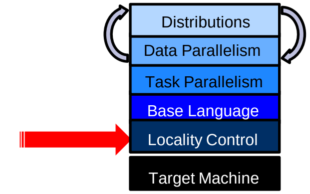

# Locality: Locales

*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

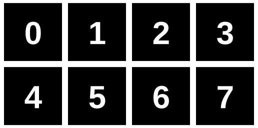
```
prompt> myChapelProg -nl=8
```

- Chapel programs have built-in locale variables:

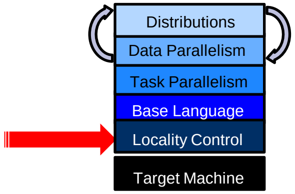
```
config const numLocales: int;
const LocaleSpace = [0..numLocales-1],
      Locales: [LocaleSpace] locale;
```


- 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

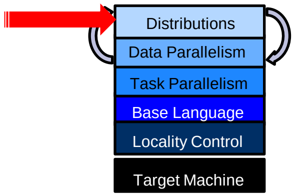
Either in a data-driven manner…

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

…or by naming locales explicitly

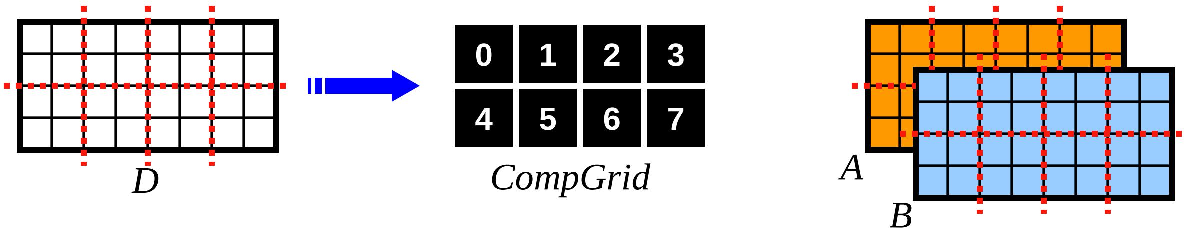
```
cobegin {
  on TaskALocs do computeTaskA(…);
  on TaskBLocs do computeTaskB(…);
  on Locales(0) do computeTaskC(…);
}
```

# Locality: Domain Distribution




Domains may be distributed across locales

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




A distribution implies…

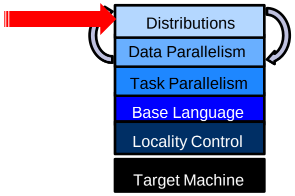
…ownership of the domain’s indices (and its arrays’ elements)

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

Chapel provides…

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

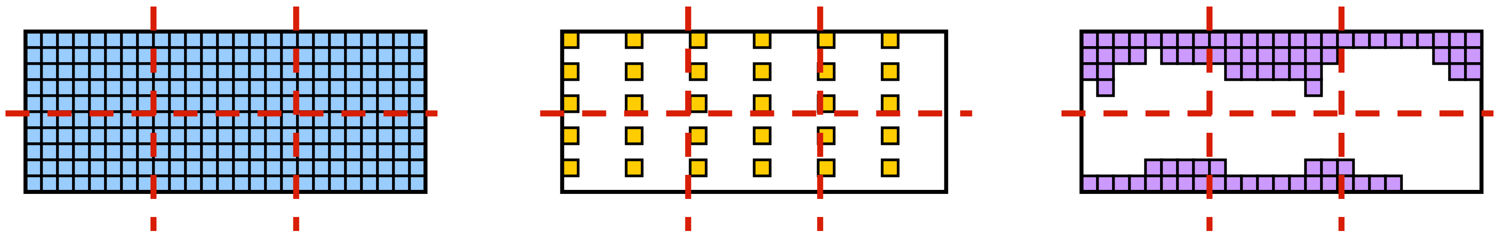
…the means for advanced users to author their own distributions

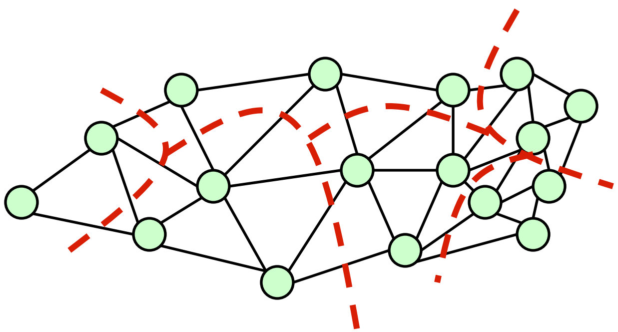



# Locality: Domain Distributions

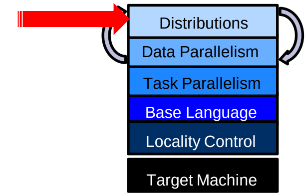
A distribution must implement…

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





# Locality: Domain Distributions

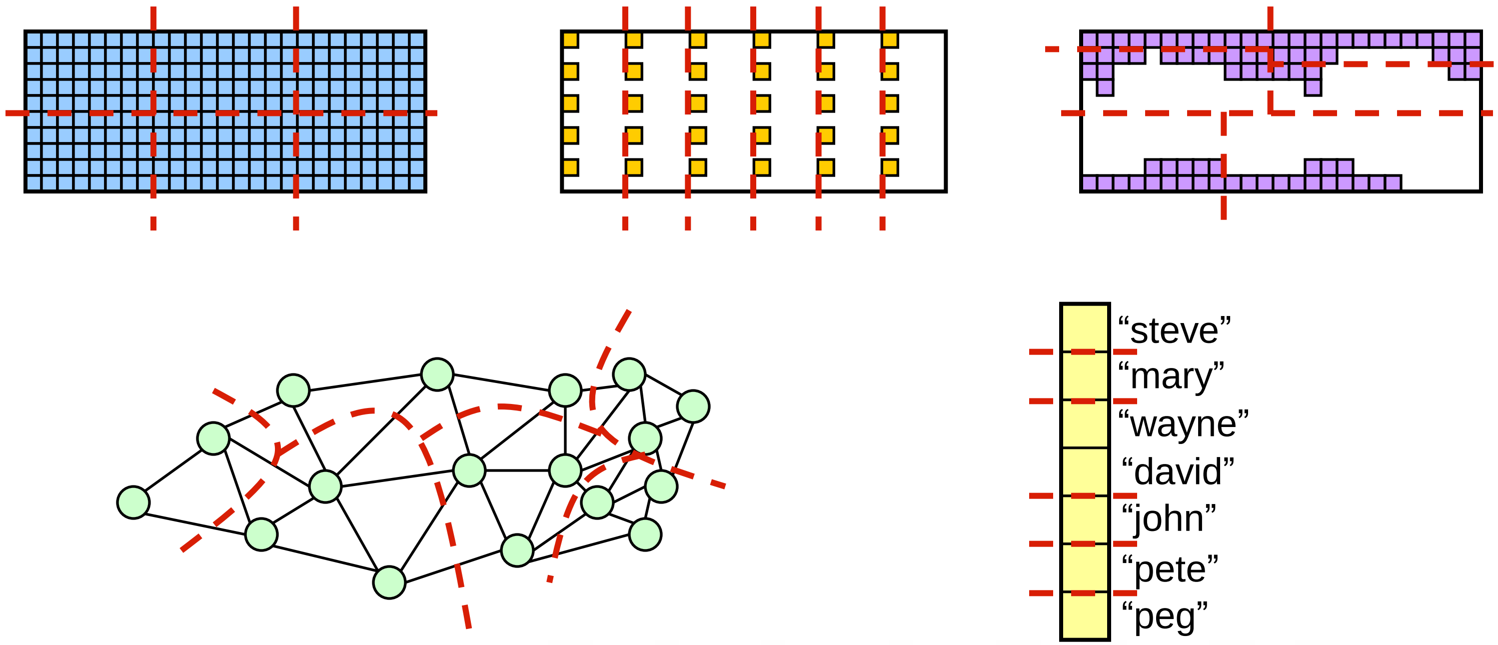



A distribution must implement…

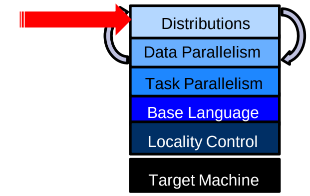
…the mapping from indices to locales

…the per-locale representation of domain indices and array elements

…the compiler's target interface for lowering global-view operations



# Locality: Distributions Overview




***Distributions:*** "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

# Outline

- ✓ Chapel Context
- ✓ Global-view Programming Models
- ✓ Language Overview
- ➢ Chapel and Mainstream Multicore
- ❑ Status, Future Work, Collaborations

# HPC vs. Mainstream Multicore

- The mainstream has a multicore gap too, it's just different
  - *i.e.*, programmers that are not experienced in parallel programming
- Differences between HPC and mainstream:
  - machine scales
  - performance/memory requirements (?)
  - robustness requirements (?)
  - workloads
  - programming community sizes and expertise areas
- Some interesting HPC(S) trends:
  - growing desire for software productivity, programmability
  - desire to better support non-expert users
    - students just out of school with no C/Fortran experience
    - scientists without strong parallel CS background
  - desire to leverage multicore technologies in larger systems
    - ideally without requiring hybrid programming models

# Chapel and Mainstream Multicore

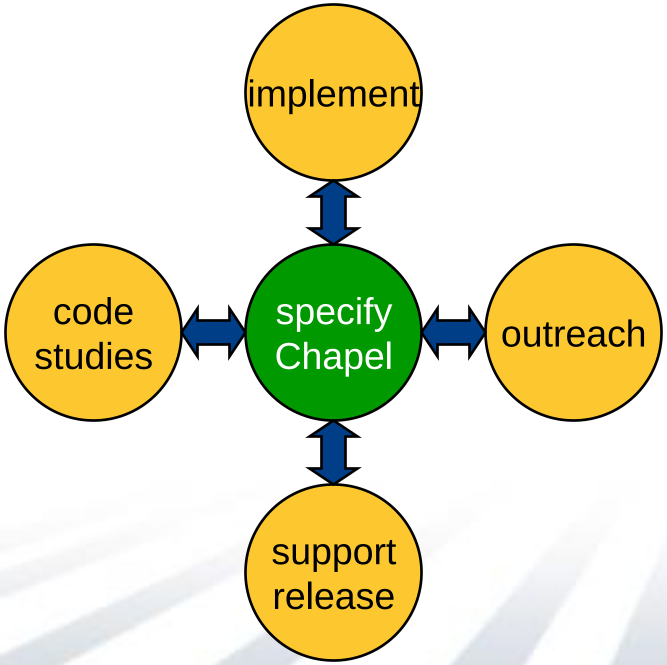
- While Chapel doesn't specifically target mainstream multicore programmers, it could be applicable
  - supports data parallelism at a high-level with clean concepts
  - raises level of discourse for task parallelism above threads
  - though not a dialect of a mainstream language, not far afield either
    - programmers today seem more multilingual than in the past
- Chapel's locales and distributions are likely overkill for today's multicore processors
  - yet, what about for future generations of multicore?
- Chapel team does most of our development and testing on mainstream multicore machines
  - Linux, Mac, Windows, … AMD, Intel, …
- Plus, some enthusiastic responses from open source users

# Outline

- ✓ Chapel Context
- ✓ Global-view Programming Models
- ✓ Language Overview
- ✓ Chapel and Mainstream Multicore
- ❑ Status, Future Work, Collaborations

# Chapel Work

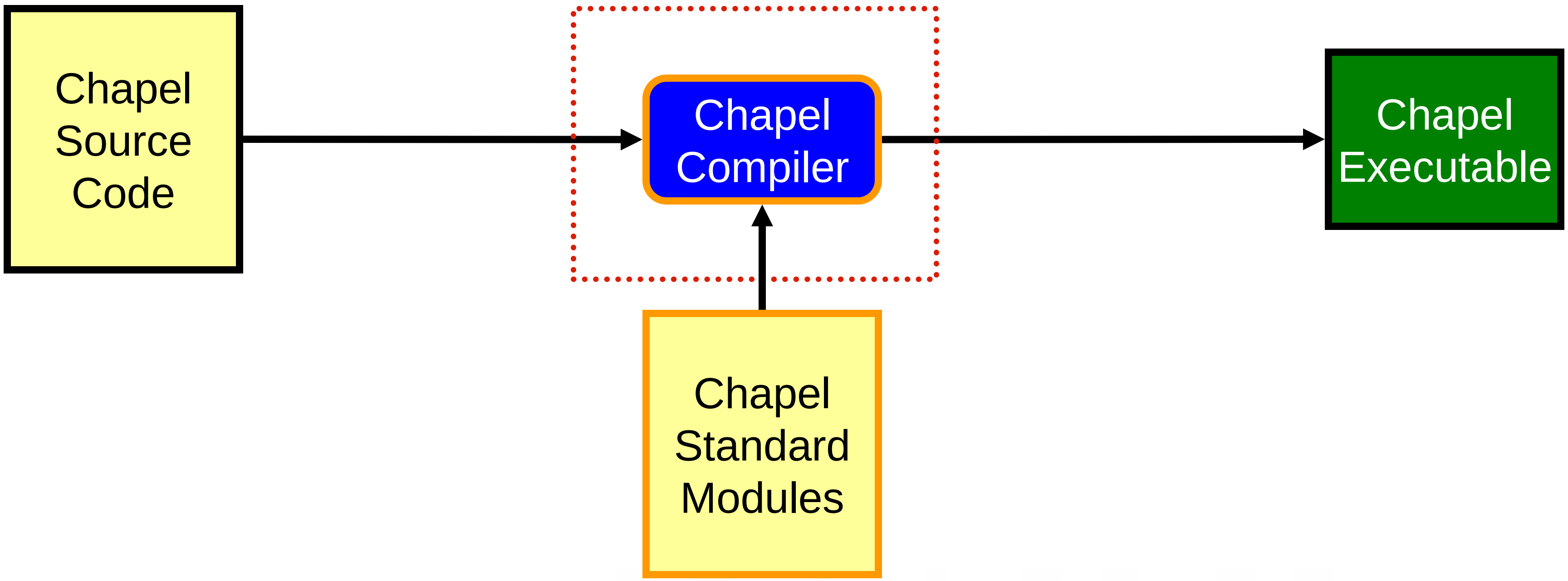
- Chapel Team's Focus:
  - specify Chapel syntax and semantics
  - implement open-source prototype compiler for Chapel
  - perform code studies of benchmarks, apps, and libraries in Chapel
  - do community outreach to inform and learn from users/researchers
  - support users of code releases
  - refine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



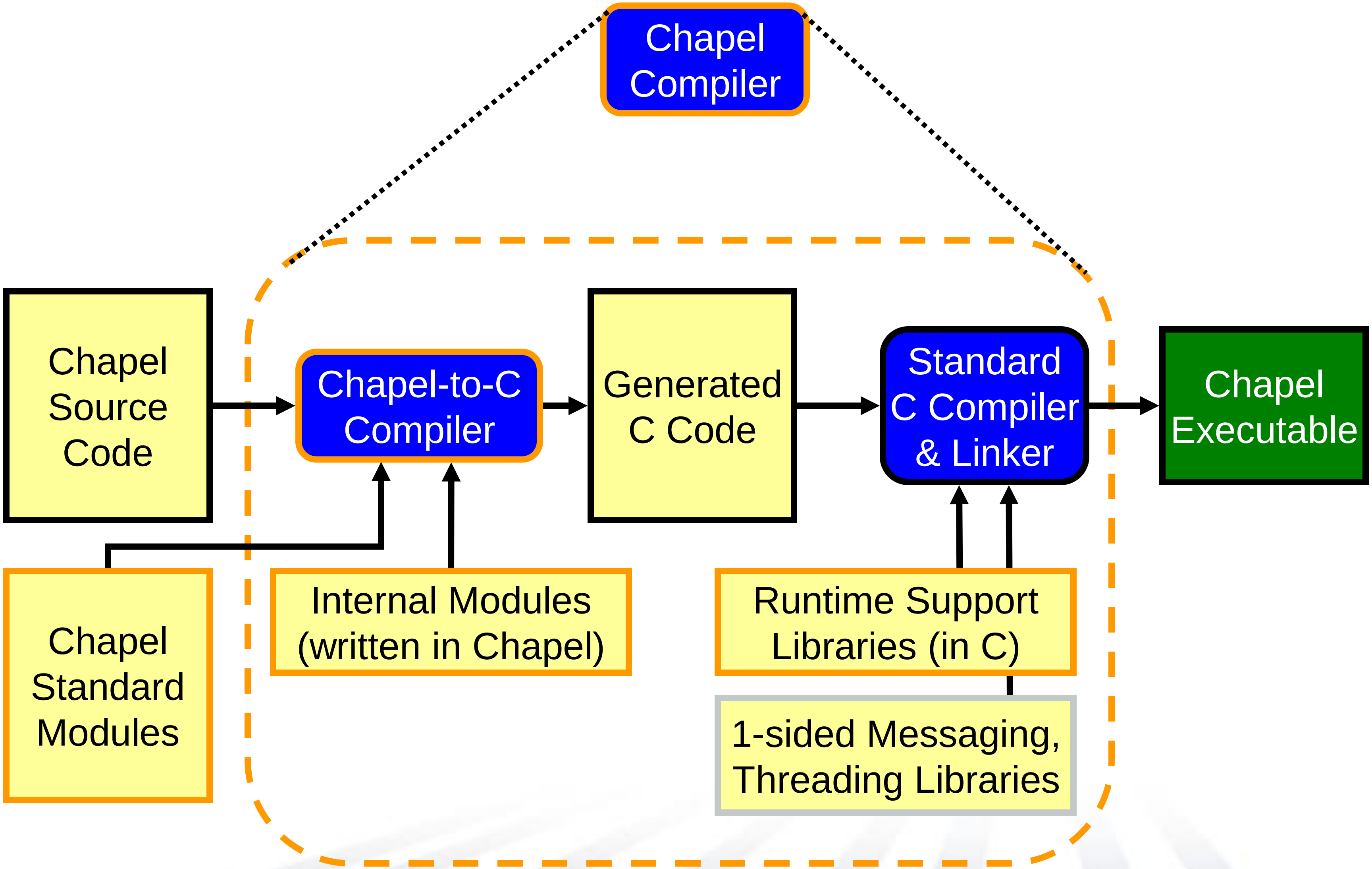
# Language/Compiler Development Strategy

- start by incubating Chapel within Cray under HPCS
- past few years: released to small sets of “friendly” users
  - ~90 users at ~30 sites (government, academia, industry)
- this past weekend: first public release!
- longer-term: turn over to community when it’s ready to stand on its own

# Compiling Chapel



# Chapel Compiler Architecture



# Chapel and Research

- Chapel contains a number of research challenges
- 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 the Parallel Community

- Our philosophy:
  - Help parallel users 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)

**UND/ORNL (Peter Kogge, Srinivas Sridharan, Jeff Vetter):** Asynchronous STM over distributed memory

**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…)

# 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

# Chapel at SC08

- ✓ **Just prior:** First public release of Chapel made available
- ✓ **Sunday:** Chapel tutorial with hands-on session
- ➢ **Monday:** joint PGAS tutorial with UPC, X10 (w/ hands-on)
- ➢ **Monday:** *"Chapel: an HPC language in a multicore world"*
  - at "*Bridging Multicore's Programmability Gap*" workshop
- **Tuesday:** HPC Challenge BOF @ 12:15
  - Chapel's entry was selected as a finalist for "most productive" class
- **Tuesday:** "MADNESS in Chapel" @ 5:15 poster session
  - Ongoing Chapel application study by ORNL and Ohio State
- **Thursday:** PGAS BOF @ 12:15
- **In print:** Chapel interview in HPCwire
- **Throughout:** available for technical discussions; poster
  - inquire at the Cray or PGAS booths to set up a meeting
  - Chapel poster at the PGAS booth

# Release Overview

- Our release is a snapshot of a work in progress
- missing features:
  - data parallelism is a single-threaded, local implementation by default
  - we got our first user-defined distribution running two months ago
  - atomic sections are an active area of research
- not suitable for performance studies
  - performance was a key factor in Chapel's design
  - yet our implementation effort to date has focused almost exclusively on correctness
- license: BSD

# For More Information

chapel_info@cray.com

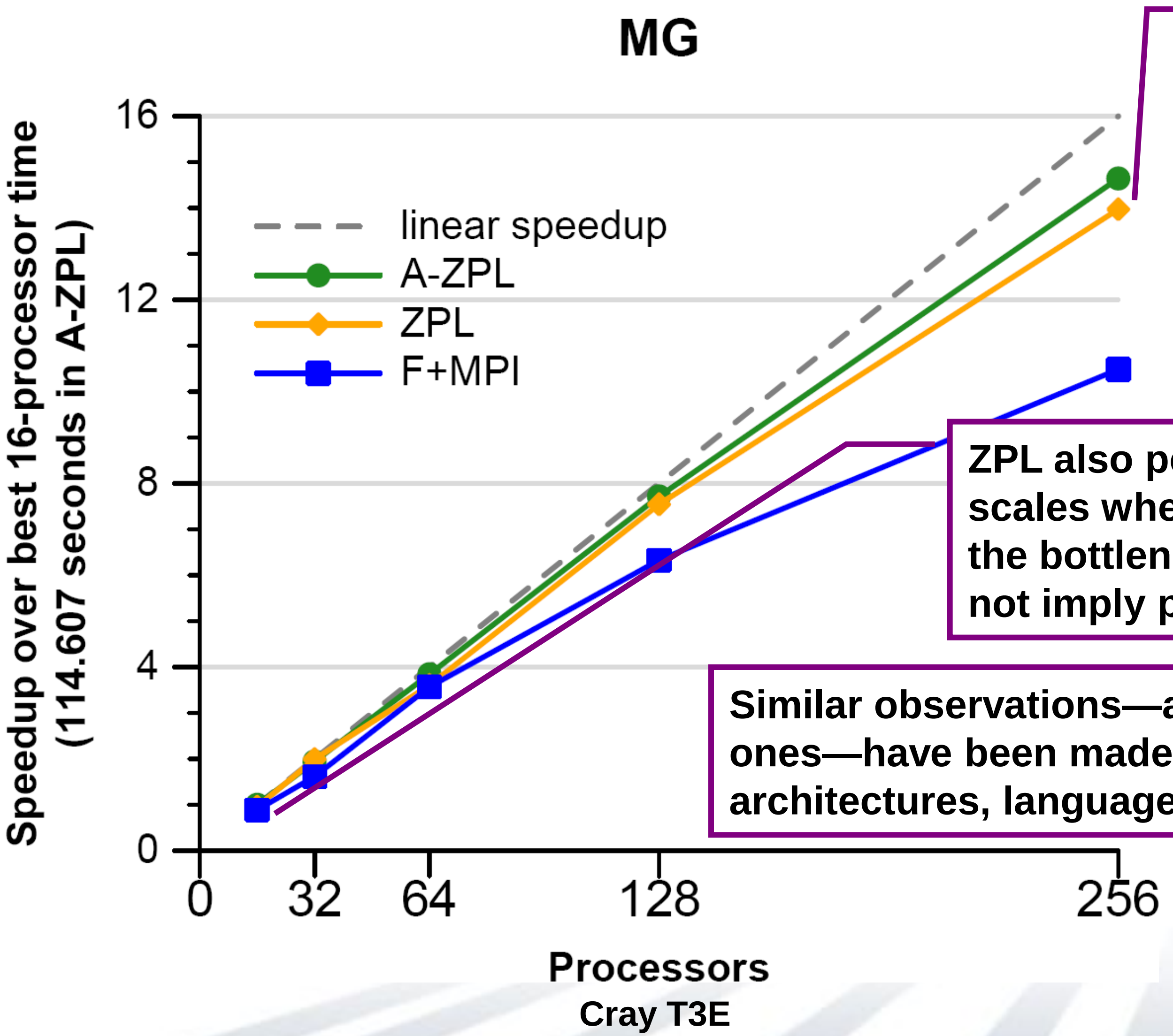
http://chapel.cs.washington.edu

SC08 tutorials

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

# Questions?

# NAS MG Speedup: ZPL vs. Fortran + MPI

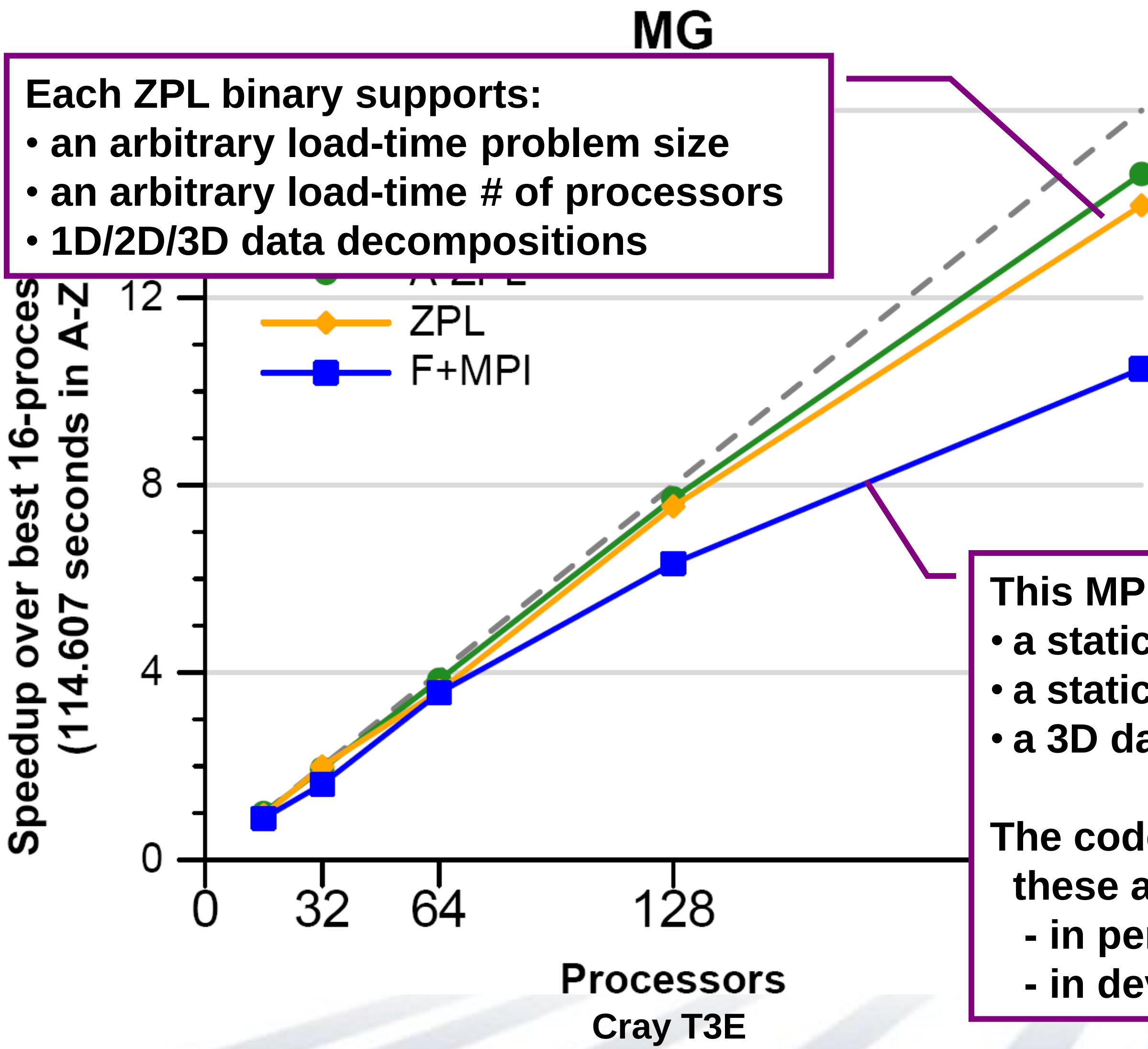



**ZPL scales better than MPI since its communication is expressed in an implementation-neutral way; this permits the compiler to use SHMEM on this Cray T3E but MPI on a commodity cluster**

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

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

# Generality Notes

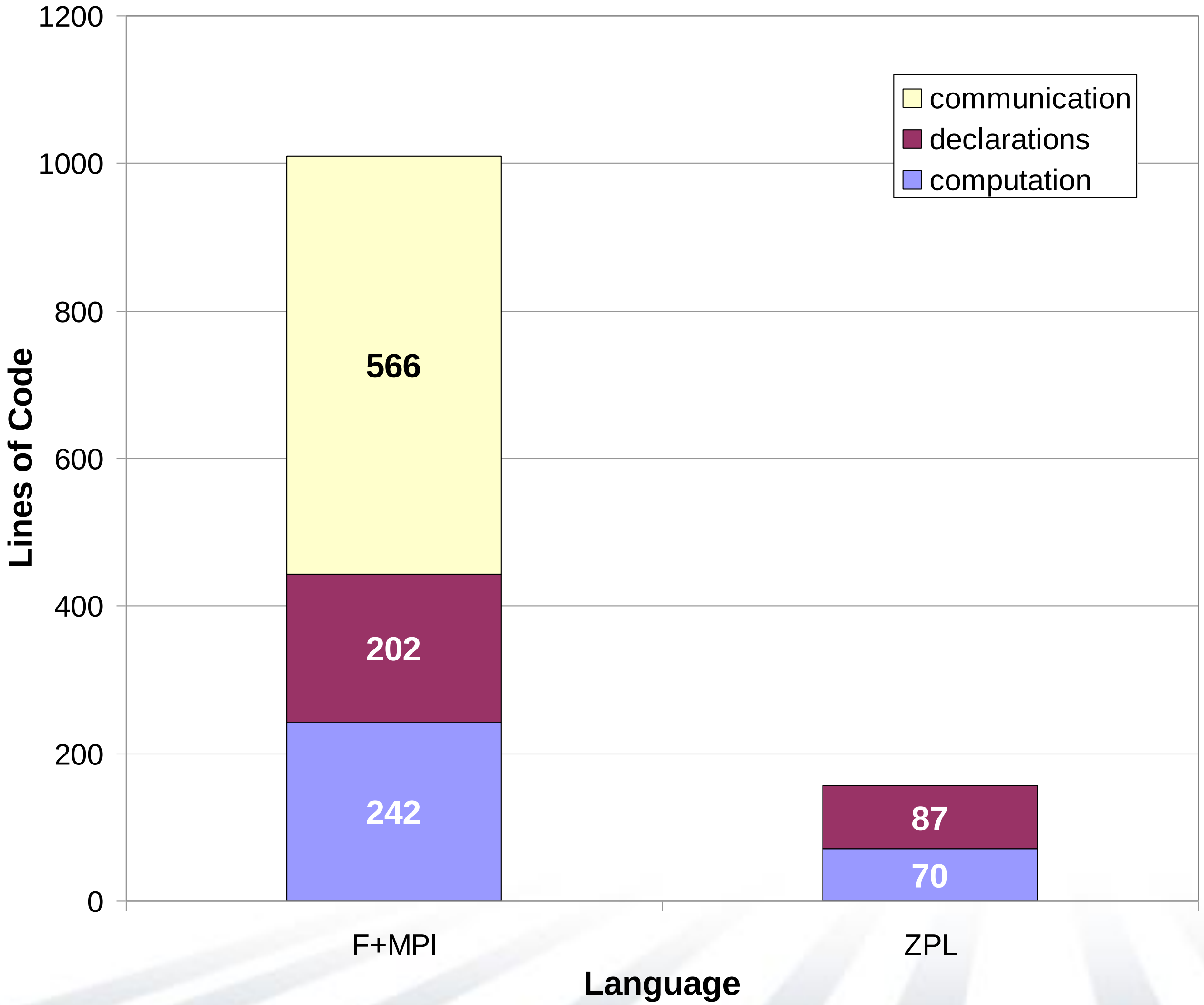



**Each ZPL binary supports:**
- **an arbitrary load-time problem size**
- **an arbitrary load-time # of processors**
- **1D/2D/3D data decompositions**

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

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

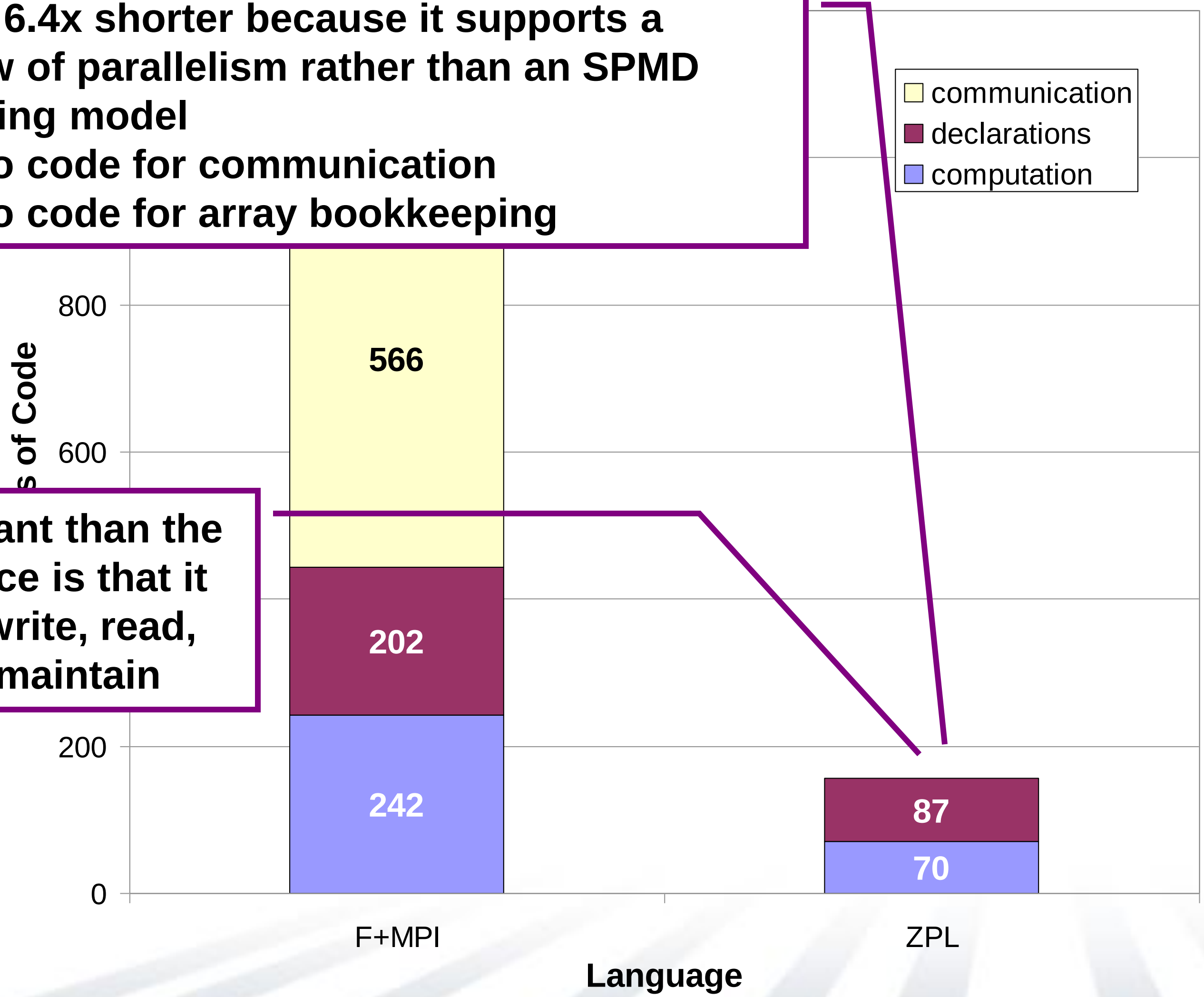
# Code Size



1200
1000
800
600
400
200
0
Lines of Code
communication
declarations
computation
566
202
242
87
70
F+MPI
ZPL
Language

# Code Size Notes

- the ZPL is 6.4x shorter because it supports a global view of parallelism rather than an SPMD programming model
  - ⇒ little/no code for communication
  - ⇒ little/no code for array bookkeeping




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

# NAS MG: Fortran + MPI vs. ZPL