Chapel: Motivating Themes

Brad Chamberlain
Cray Inc.

CSEP 524
May 20, 2010

What is Chapel?

- A new parallel language being developed by Cray Inc.
- Part of Cray's entry in DARPA's HPCS program
- **Main Goal:** Improve programmer productivity
  - Improve the programmability of parallel computers
  - Match or beat the performance of current programming models
  - Provide better portability than current programming models
  - Improve robustness of parallel codes
- Target architectures:
  - multicore desktop machines
  - clusters of commodity processors
  - Cray architectures
  - systems from other vendors
- A work in progress

Chapel's Setting: HPCS

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

- Goal: Raise productivity of high-end computing users by 10×
- 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 –)
  - Implement the systems and technologies resulting from phase II
  - (Sun also continues work on Fortress, without HPCS funding)

Chapel: Motivating Themes

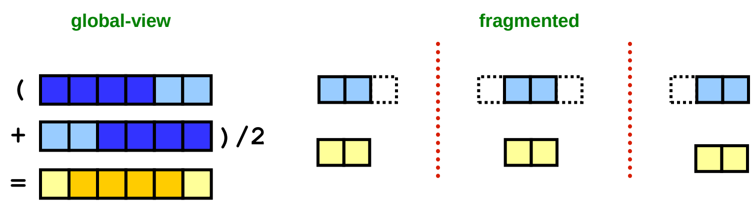
1) **general parallel programming**
2) ***global-view* abstractions**
3) ***multiresolution* design**
4) **control of locality/affinity**
5) **reduce gap between mainstream & parallel languages**

1) General Parallel Programming

- **General software parallelism**
  - *Algorithms:* should be able to express any that come to mind
    - should never hit a limitation requiring the user to return to MPI
  - *Styles:* data-parallel, task-parallel, concurrent algorithms
    - as well as the ability to compose these naturally
  - *Levels:* module-level, function-level, loop-level, statement-level, …
- **General hardware parallelism**
  - *Types:* multicore desktops, clusters, HPC systems, …
  - *Levels:* inter-machine, inter-node, inter-core, vectors, multithreading

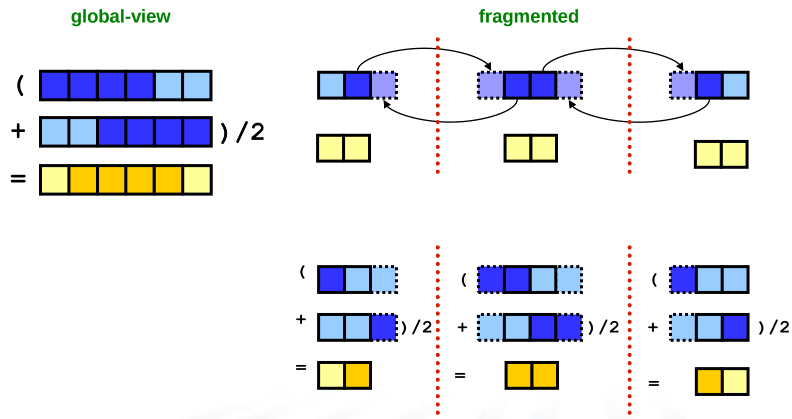
2) Global-view vs. Fragmented

Problem: "Apply 3-pt stencil to vector"



2) Global-view vs. Fragmented

Problem: "Apply 3-pt stencil to vector"



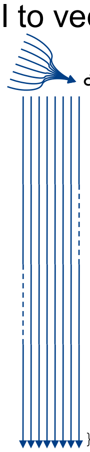
2) Global-view vs. SPMD Code

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



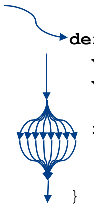
```
def main() {
  var n: int = 1000;
  var locN: int = n/numProcs;
  var a, b: [0..locN+1] real;

  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;
  }
}
```

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;
  }
}
```

2) SPMD pseudo-code + MPI

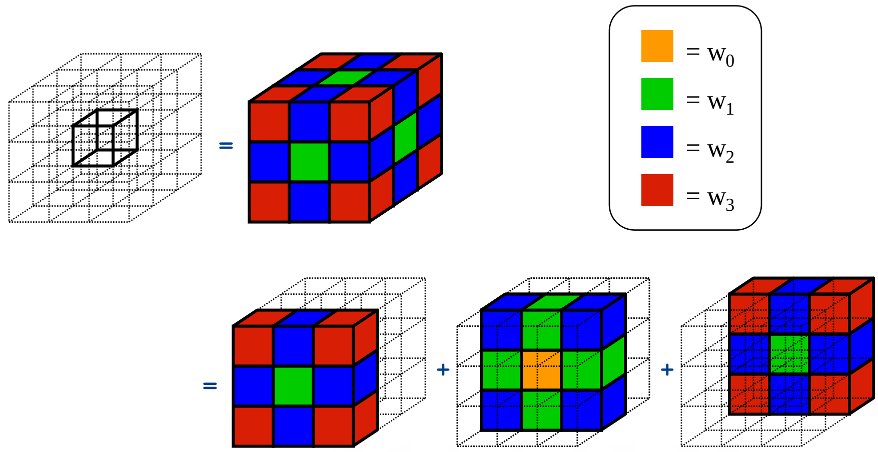
Problem: "Apply 3-pt stencil to vector"

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;
}
```

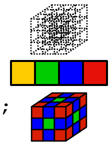
2) *rprj3* stencil from NAS MG



2) 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
```

2) NAS MG *rprj3* stencil in Chapel



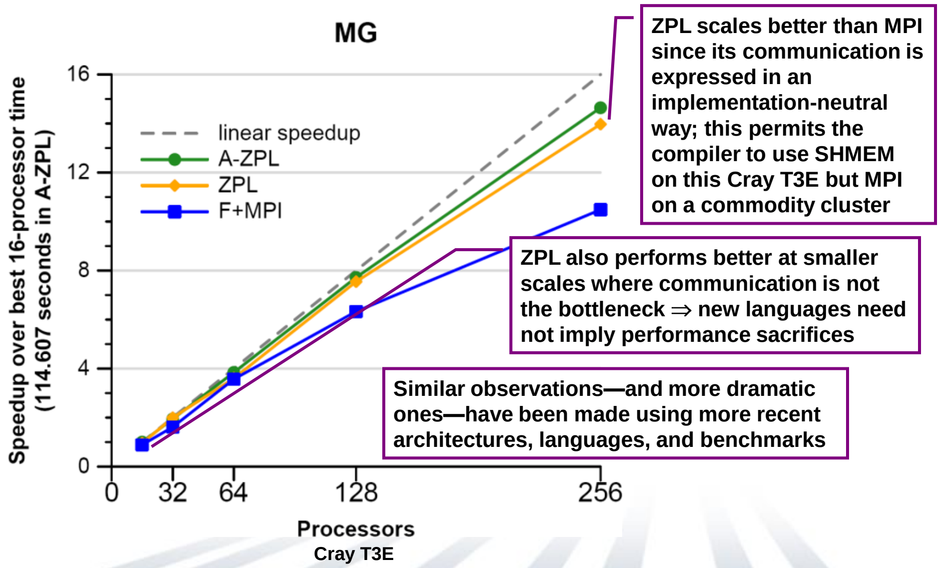
```
def rprj3(S, R) {
  const Stencil = [-1..1, -1..1, -1..1],
        w: [0..3] real = (0.5, 0.25, 0.125, 0.0625),
        w3d = [(i,j,k) in Stencil] w((i!=0) + (j!=0) + (k!=0));

  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 while also supporting more runtime flexibility

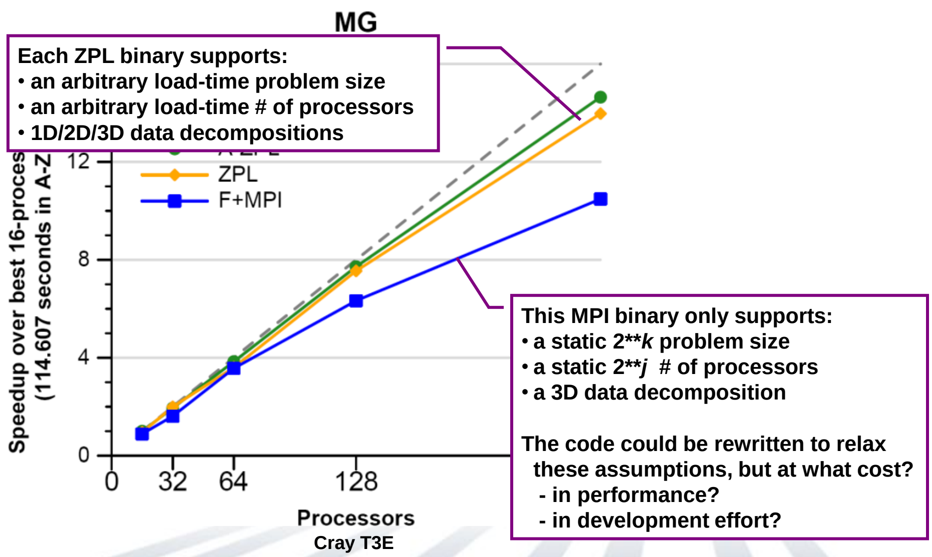
NAS MG *rprj3* stencil in ZPL

```
procedure rprj3(var S,R: [,,] double;
                d: array [] of direction);
begin
  S := 0.5    *  R
     + 0.25   * (R@^d[ 1, 0, 0] + R@^d[ 0, 1, 0] + R@^d[ 0, 0, 1] +
                 R@^d[-1, 0, 0] + R@^d[ 0,-1, 0] + R@^d[ 0, 0,-1])
     + 0.125  * (R@^d[ 1, 1, 0] + R@^d[ 1, 0, 1] + R@^d[ 0, 1, 1] +
                 R@^d[ 1,-1, 0] + R@^d[ 1, 0,-1] + R@^d[ 0, 1,-1] +
                 R@^d[-1, 1, 0] + R@^d[-1, 0, 1] + R@^d[ 0,-1, 1] +
                 R@^d[-1,-1, 0] + R@^d[-1, 0,-1] + R@^d[ 0,-1,-1])
     + 0.0625 * (R@^d[ 1, 1, 1] + R@^d[ 1, 1,-1] +
                 R@^d[ 1,-1, 1] + R@^d[ 1,-1,-1] +
                 R@^d[-1, 1, 1] + R@^d[-1, 1,-1] +
                 R@^d[-1,-1, 1] + R@^d[-1,-1,-1]);
end;
```

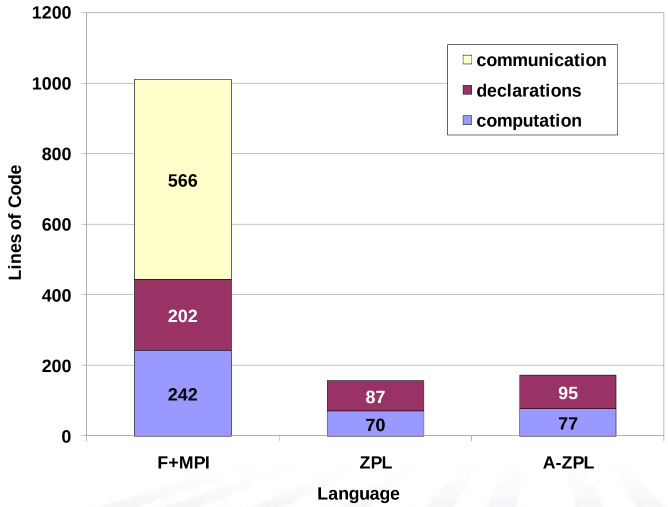
NAS MG Speedup: ZPL vs. Fortran + MPI



Generality Notes



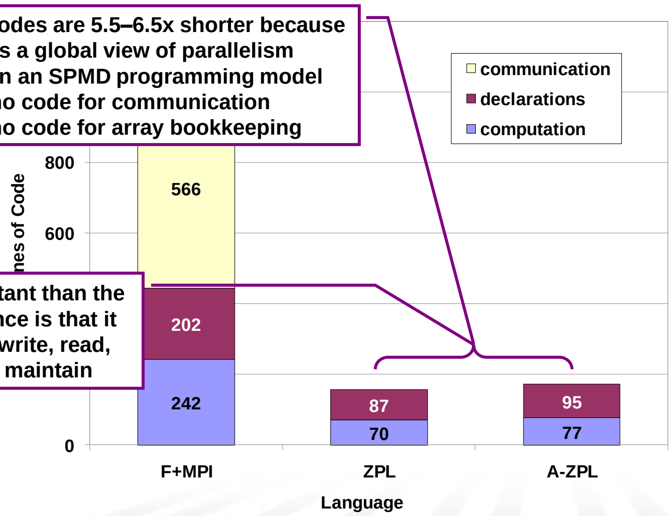
Code Size



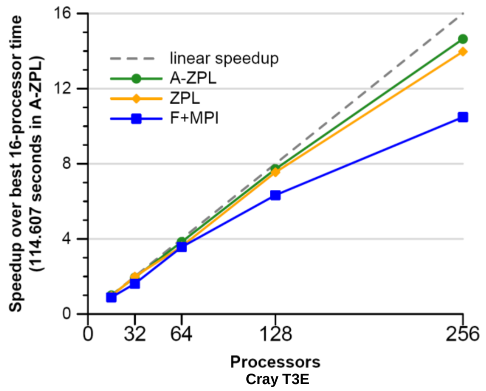
Code Size Notes

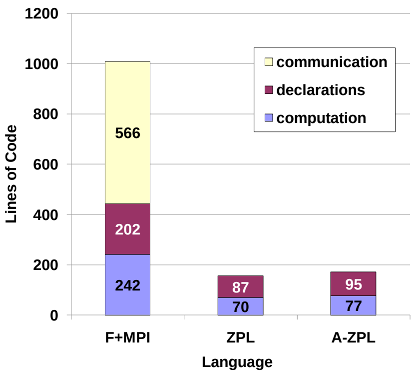
- the ZPL codes are 5.5–6.5x shorter because it supports a global view of parallelism rather than an SPMD programming model
  - ⇒ little/no code for communication
  - ⇒ little/no code for array bookkeeping

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



Global-view models can benefit Productivity





- more programmable, flexible
- able to achieve competitive performance
- more portable; leave low-level details to the compiler

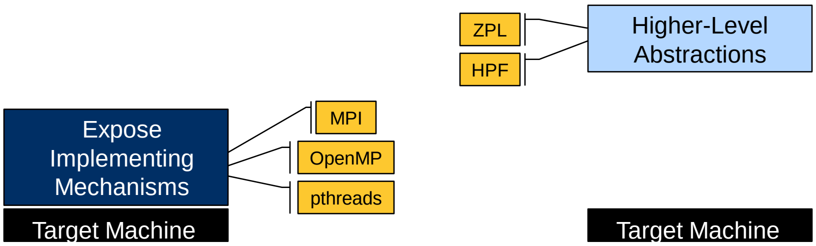
2) Classifying HPC Programming Notations

	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

3) Multiresolution Languages: Motivation

Two typical camps of parallel language design:

low-level vs. high-level



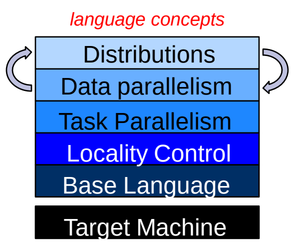
"Why is everything so tedious?"

"Why don't I have more control?"

3) Multiresolution Language Design

Our Approach: Structure the language in a layered manner, permitting it to be used at multiple levels as required/desired

- support 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



4) Ability to Tune for Locality/Affinity

- Large-scale systems tend to store memory w/ processors
  - a good approach for building scalable parallel systems
- Remote accesses tend to be significantly more expensive than local
- Therefore, placement of data relative to computation matters for scalable performance

  ⇒ programmer should have control over placement of data, tasks
- As multicore chips grow in #cores, locality likely to become more important in desktop parallel programming as well
  - GPUs/accelerators also expose node-level locality concerns

4) A Note on Machine Model

- As with ZPL, the CTA is still present in our design to reason about locality
- That said, it is probably more subconscious for us
- And we vary in some minor ways:
  - no controller node
    - though we do utilize a front-end launcher node in practice
  - nodes can execute multiple tasks/threads
    - through software multiplexing if not hardware

5) Support for Modern Language Concepts

- students graduate with training in Java, Matlab, Perl, C#
- HPC community mired in Fortran, C (maybe C++) and MPI
- we'd like to narrow this gulf
  - leverage advances in modern language design
  - better utilize the skills of the entry-level workforce…
    …while not ostracizing traditional HPC programmers
- examples:
  - build on an imperative, block-structured language design
  - support object-oriented programming, but make its use optional
  - support for static type inference, generic programming to support…
    …exploratory programming as in scripting languages
    …code reuse