HPC Programming Models: Current Practice, Emerging Promise

Brad Chamberlain
Chapel Team, Cray Inc.

SIAM Conference on
Parallel Processing for Scientific Computing (PP10)
February 25, 2010

CRAY
THE SUPERCOMPUTER COMPANY

Why I'm Glad You're Here

- It's 8am in the morning == way too early for a technical talk
- I seem to have submitted the most boring abstract ever:

  Abstract: In this talk, I will give an overview of parallel programming models for high performance computing (HPC). I will begin by providing an overview of today's dominant notations: MPI and OpenMP. I will then introduce the notion of Partitioned Global Address Space (PGAS) languages which strive to simplify programming while supporting scalability on large-scale machines. I will describe traditional PGAS languages as well as those that are emerging as a result of the DARPA High Produductivity Computing Systems program (HPCS) including Cray's new language, Chapel. As I describe each model, I will also evaluate it, pointing out what I view as its strengths and weaknesses.

Abstract (revised): I will rant about the ongoing lack of truly productive HPC programming models while trying to provide rationale for some of the themes we are pursuing in Chapel.

Disclaimers

In the interest of being an engaging 8am speaker, I have tried not to shy away from potentially controversial statements.

As a result, this talk's contents should be considered my personal beliefs (or at least one facet of them) and not necessarily those of Cray Inc. or my funding sources.

Terminology

Programming Models:

1. abstract models that permit users to reason about how their programs will execute with respect to parallelism, memory, communication, performance, etc.
   e.g., "what can I/should I be thinking about when writing my programs?"

2. concrete notations used to write programs: languages, libraries, pragmas/annotations, …
   i.e., the union of programming languages, libraries, annotations, …

HPC Programming Model Taxonomy (2010)

(or: my original boring mental talk outline)

- **Communication Libraries**
  - **MPI**, PVM, SHMEM, ARMCI, GASNet, …
- **Shared Memory Programming Models**
  - **OpenMP**, pthreads, …
- **GPU Programming Models**
  - **CUDA**, OpenCL, PGI annotations, CAPS, …
- **Hybrid Models**
  - **MPI+OpenMP**, MPI+CUDA, MPI+OpenCL, …
- **Traditional PGAS Languages**
  - **UPC**, **Co-Array Fortran (CAF)**, Titanium
- **HPCS Languages**
  - Chapel, X10, Fortress
- **Others** (for which I don't have a neat unifying category)
  - Charm++, ParalleX, Cilk, TBB, PPL, parallel Matlabs, Star-P, PLINQ, Map-Reduce, DPJ, Yada, …

Shameless Plug

- Many of the programming models that I'll be describing or mentioning will be covered today in a 3-part minisymposium:

Emerging Programming Paradigms for Large-Scale Scientific Computing

chairs: Leonid Oliker, Rupak Biswas, Rajesh Nishtala

(MS24, MS31, MS39)

- The carrot:
  - you'll get more technical detail than I'll be able to give here
  - from the proponents of the various programming models

- Even More Shameless Plug: the Chapel talk is at 2:00ish

Outline

- ✓ Preliminaries
- ➢ well, let's start with MPI and see where that takes us…

- ❑ oh, and we'll want to touch on the PGAS and HPCS languages and exascale computing before we're done…

Panel Question: What problems are poorly served by MPI?

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

"well-served": MPI is a natural/productive way of expressing them

- **embarrassingly parallel:** arguably
- **data parallel:** not particularly, due to cooperating executable model
  - bookkeeping details related to manual data decomposition
  - local vs. global indexing issues
  - data replication, communication, synchronization
- **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

- Its base languages have issues as well
  - **Fortran:** age leads to baggage + failure to track modern concepts
  - **C/C++:** impoverished support for arrays, pointer aliasing issues

Is MPI the best we can do?

- Today? Perhaps yes…
- But is it what you want to be using in 5, 10, 20, 40 years?

If your answer is…

…Yes!: This might be a good time to get some coffee.

…No: Then you should find a way to be part of the solution

- evaluate emerging or academic languages
- provide constructive criticism, not just skepticism
- look for ways to collaborate
  - languages are fertile soil: libraries, tools, visualizations, I/O, resiliency, algorithms, applications, …

Brad, why do you hate MPI so much?

- Honestly, I don't
  - I believe it to be one of the unparalleled successes in HPC
  - And I think it will play a crucial role for some time to come
- Good software is about appropriate layers of abstraction
  - MPI wonderfully abstracts away the complexities of distinct network architectures
  - Yet we're arguably overdue to add some standardized higher-level abstractions above message passing
- So, please don't interpret my goal as "let's bury MPI", but rather to encourage the pursuit of higher-level alternatives
  - ideally by building on top of MPI or using it as a compiler target
  - ideally ones that can interoperate with MPI to preserve legacy code

Exciting Directions in MPI 3.0

- The MPI 3.0 committee is hard at work on a number of promising features…
  - Improved resilience
  - Better support for hybrid computing (e.g., MPI + …)
  - Purer one-sided communication
  - Active messages
  - Asynchronous collective communications
  - Improved scalability
  - …and much more

- Particularly important for…

  …exascale computing

  …serving as a richer foundation for higher-level languages

MPI (Message Passing Interface)

MPI strengths

+ users can get real work done 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 can serve as a strong foundation for higher-level technologies

MPI weaknesses

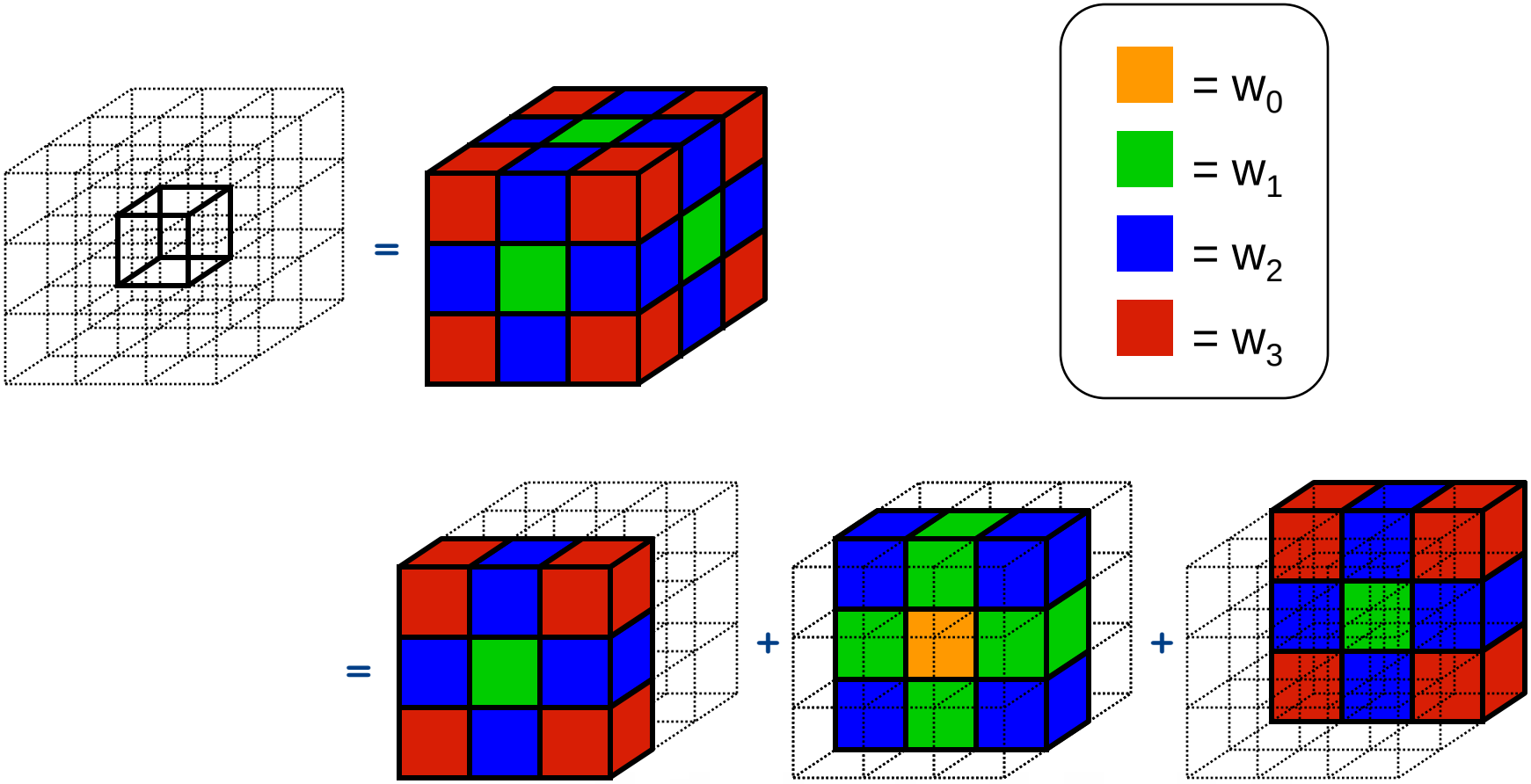
– encodes too much about "how" data should be transferred rather than simply "what data" (and possibly "when")
  - can mismatch architectures with different data transfer capabilities
– only supports parallelism at the "cooperating executable" level
  - applications and architectures contain parallelism at many levels
  - doesn't reflect how one abstractly thinks about parallel algorithm
– no abstractions for distributed data structures
  - places a significant bookkeeping burden on the programmer

A F'r'Instance: how we could do better

- Consider three (fictitious) architectures
  - **A:** prefers non-blocking receives, blocking sends, long messages
  - **B:** does fine with shorter, more synchronous messages
  - **C:** prefers one-sided communications

- An MPI enthusiast might argue "yes, our interface supports calls for all three of these cases" (and many, many more!)
  - but at what level of programmer effort?
  - isn't this selection something we'd really like a compiler, runtime, or library to handle for us rather than embedding it into our sources?

NAS MG *rprj3* stencil



NAS MG *rprj3* stencil in ZPL

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

NAS MG *rprj3* stencil in Fortran+MPI

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

implicit none

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

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

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


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


implicit none

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

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

integer i3, i2, i1, buff_len,buff_id

buff_id = 2 + dir
buff_len = 0

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

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

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

   else if( dir .eq. +1 ) then

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

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

   endif
endif

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

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

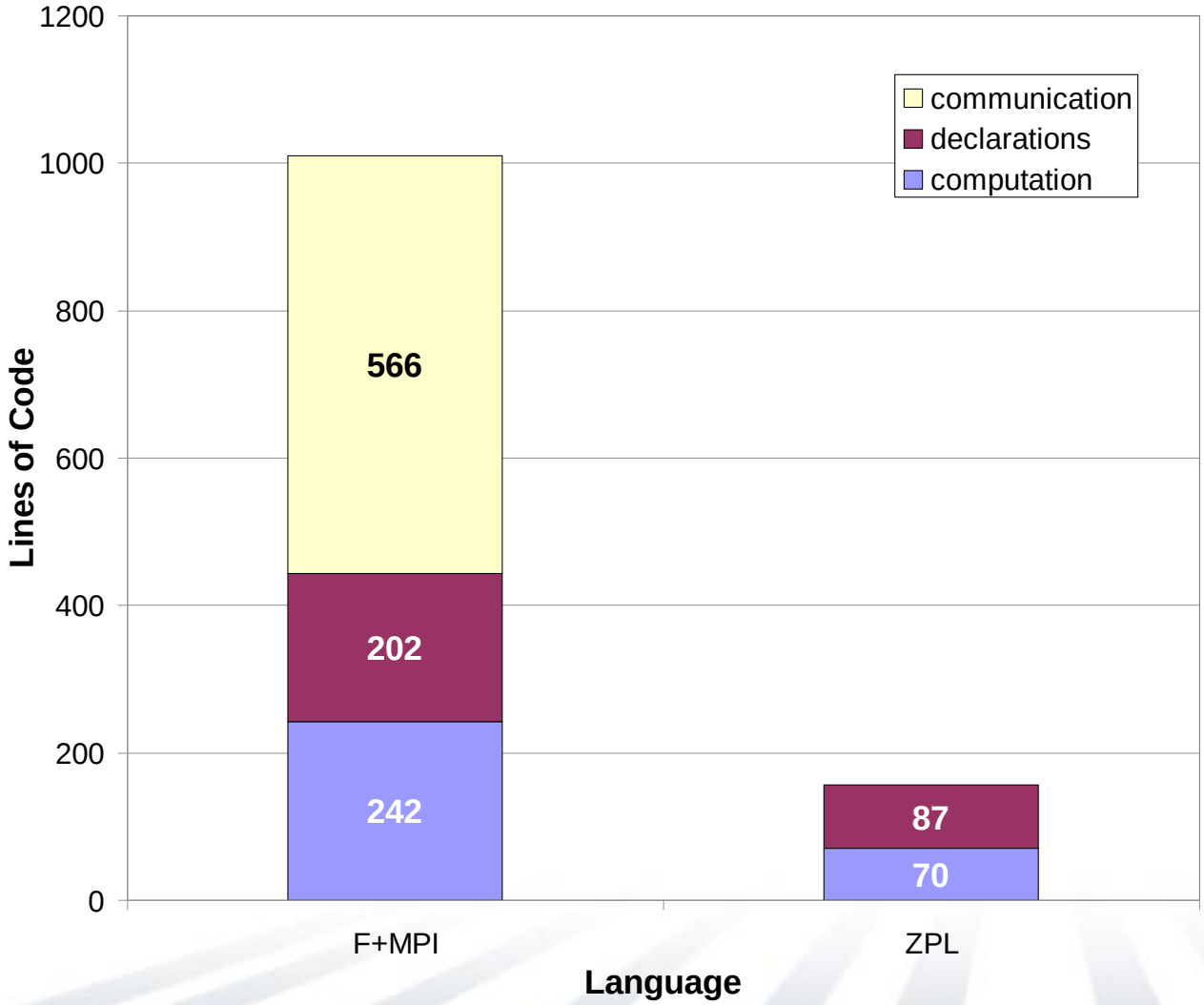
```
else if( dir .eq. +1 ) then

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

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

   endif
endif

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

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

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

   else if( dir .eq. +1 ) then

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

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

   endif
endif

return
end


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


implicit none

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

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

integer buff_id, indx

integer i3, i2, i1

buff_id = 3 + dir
indx = 0

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

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

   else if( dir .eq. +1 ) then

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

   endif
endif

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

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

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

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

   endif
endif

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

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

   else if( dir .eq. +1 ) then

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

   endif
endif

return
end


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


implicit none

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

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

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

dir = -1

buff_id = 3 + dir
buff_len = nm2

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


dir = +1

buff_id = 3 + dir
buff_len = nm2

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

dir = +1

buff_id = 2 + dir
buff_len = 0

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

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

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

dir = -1

buff_id = 2 + dir
buff_len = 0

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

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

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

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

dir = -1

buff_id = 3 + dir
indx = 0

if( axis .eq.  1 )then
   do  i3=2,n3-1
      do  i2=2,n2-1
         indx = indx + 1
         u(nl,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
```

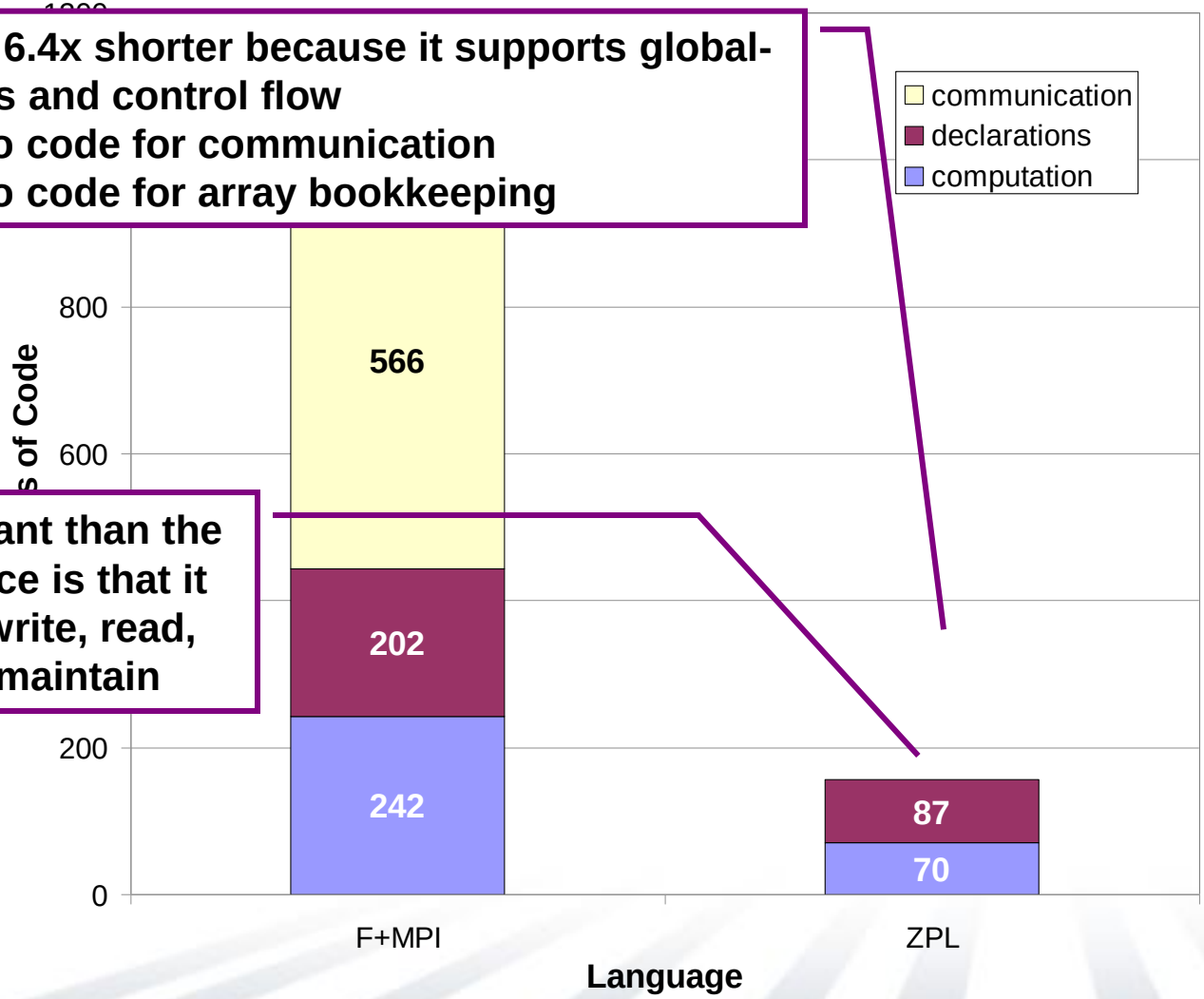
```
if( axis .eq.  2 )then
   do  i3=2,n3-1
      do  i1=1,n1
         indx = indx + 1
         u(i1,1,i3) = buff(indx, buff_id )
      enddo
   enddo
endif

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
```

Fortran+MPI vs. ZPL: Code Size



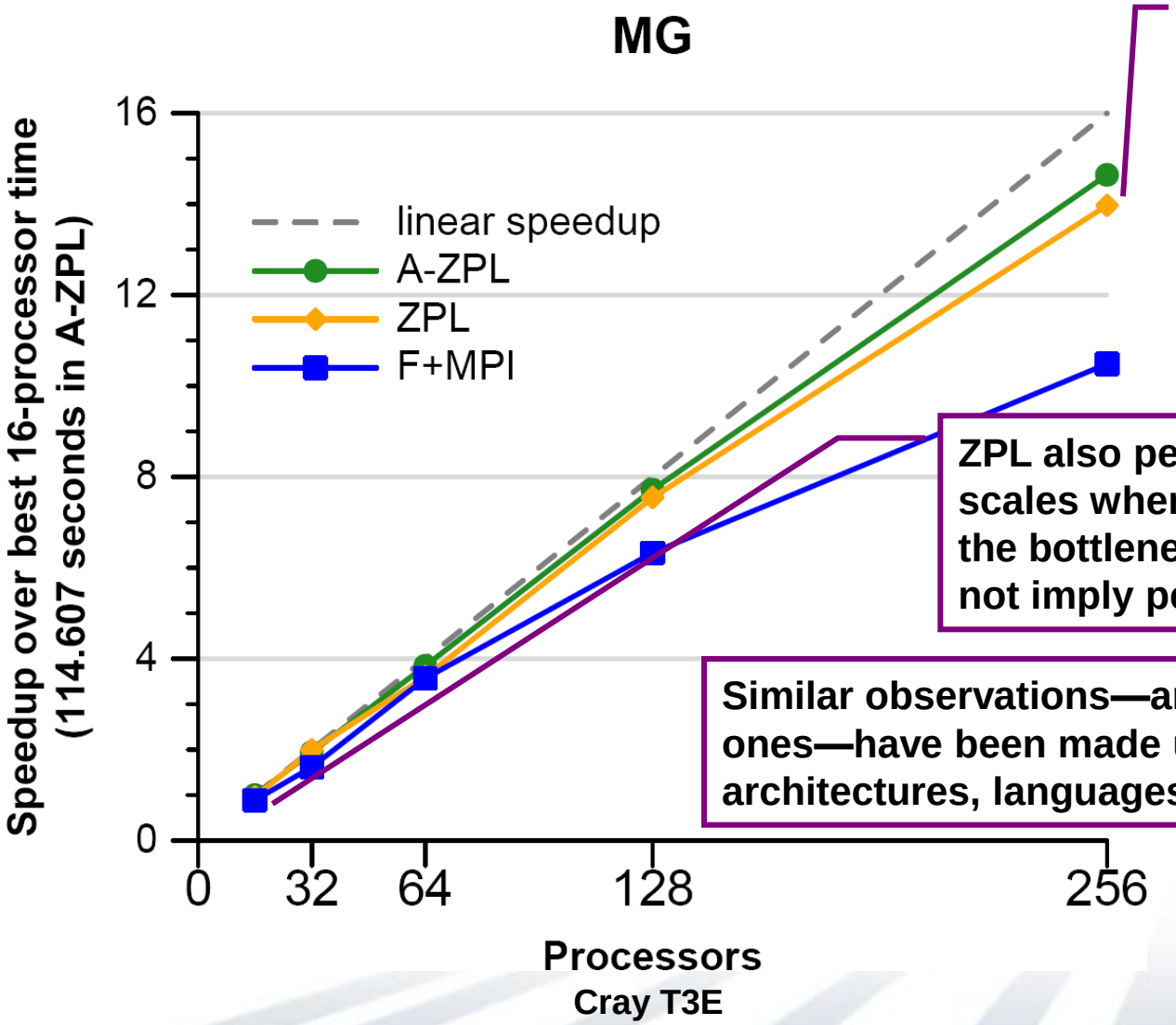
Fortran+MPI vs. ZPL: Code Size

- the ZPL is 6.4x shorter because it supports global-view arrays and control flow
  - ⇒ 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

Fortran+MPI vs. ZPL: Performance

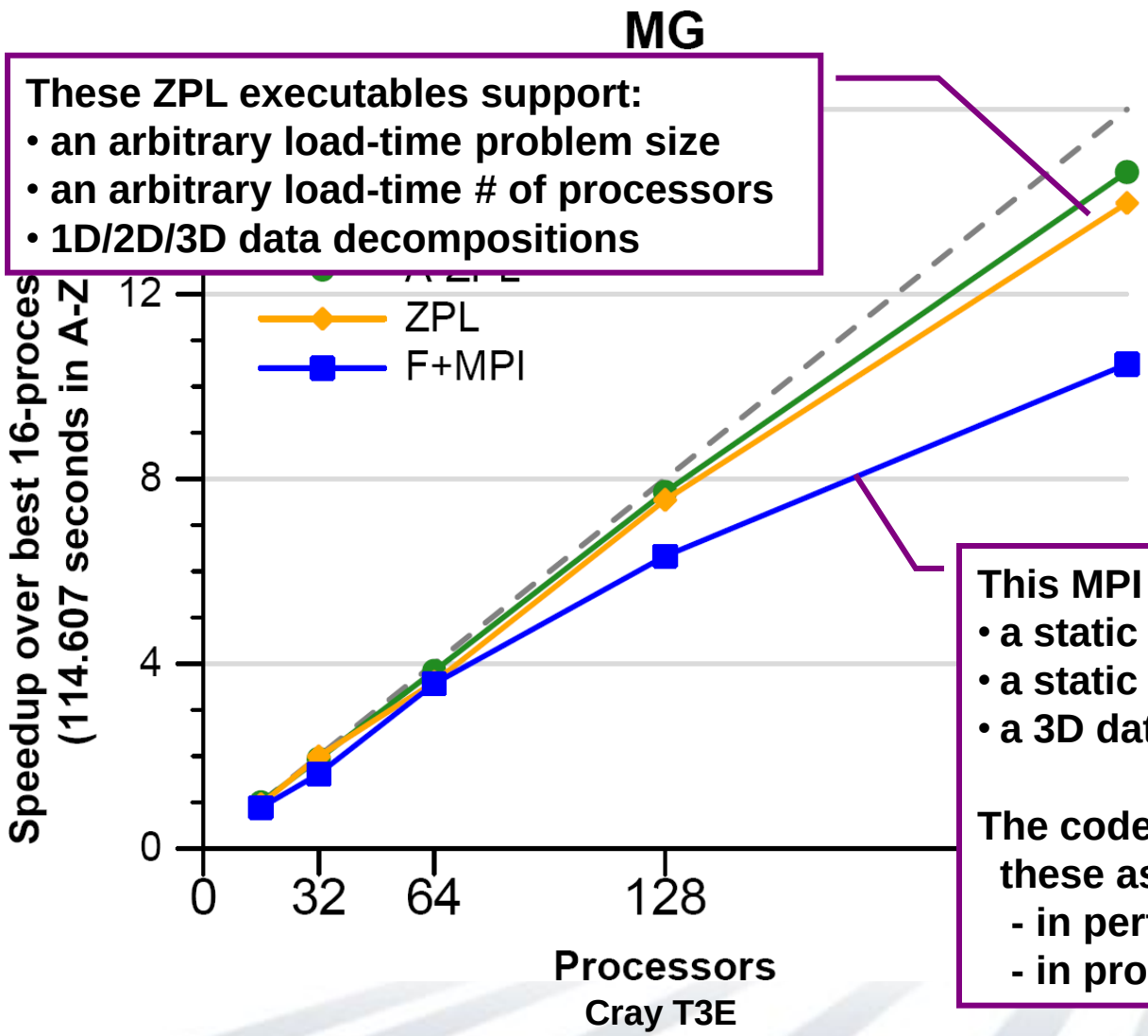


ZPL scales better than MPI since its communication is expressed in an implementation-neutral way; this permits the compiler to use SHMEM on this Cray T3E but MPI on a commodity cluster

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

Fortran+MPI vs. ZPL: Performance

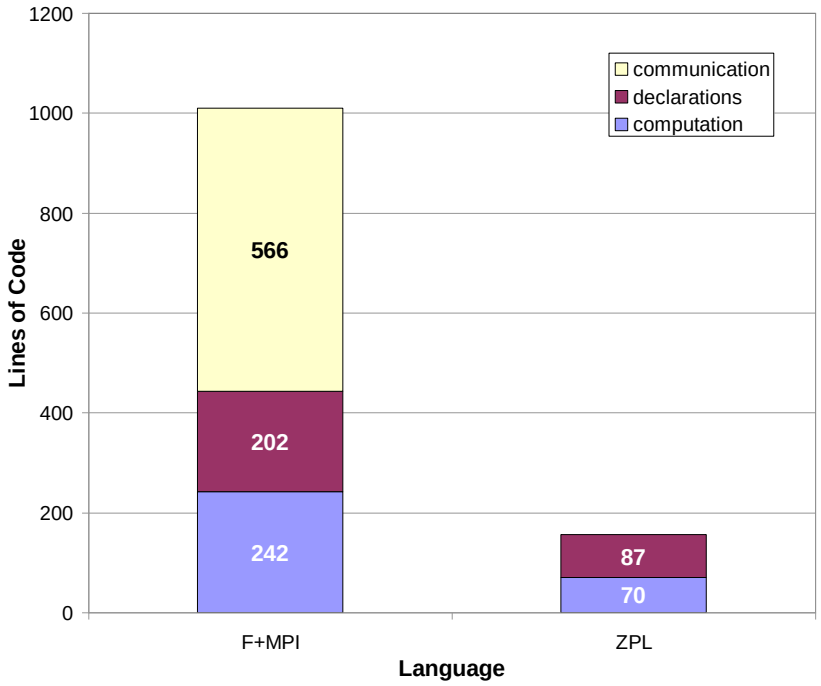


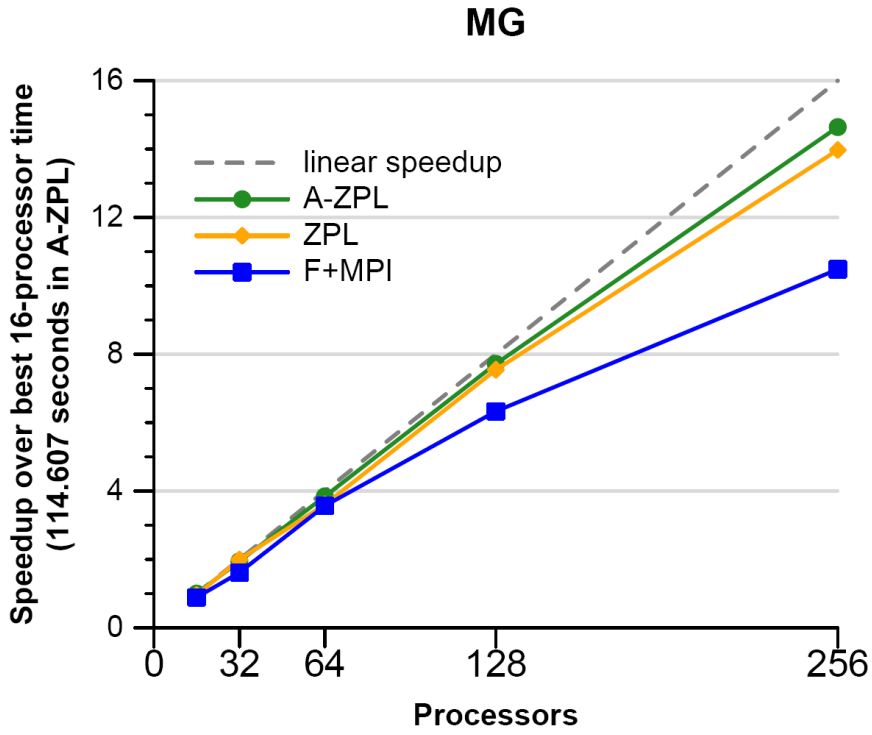
These ZPL executables support:
- **an arbitrary load-time problem size**
- **an arbitrary load-time # of processors**
- **1D/2D/3D data decompositions**

This MPI executable 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 programmer effort?**

So, are we done?





Q: Concise, fast, flexible code – what more could you want?

A: Increased generality

ZPL

ZPL strengths

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

ZPL weaknesses

– only supports one level of data parallelism; no true task parallelism
– distinct concepts for parallel (distributed) vs. serial (non-) arrays
– only supports a small number of built-in distributions

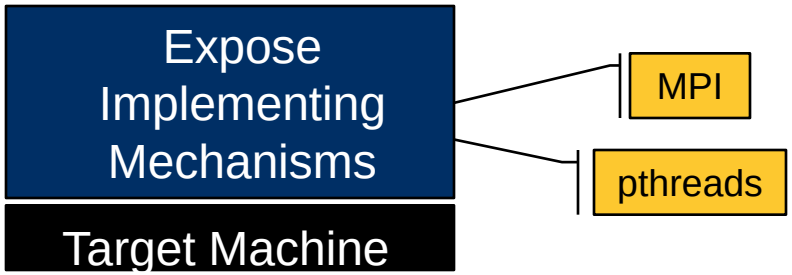
But rather than giving up, let's take the lessons from ZPL that we can and keep pushing forward…

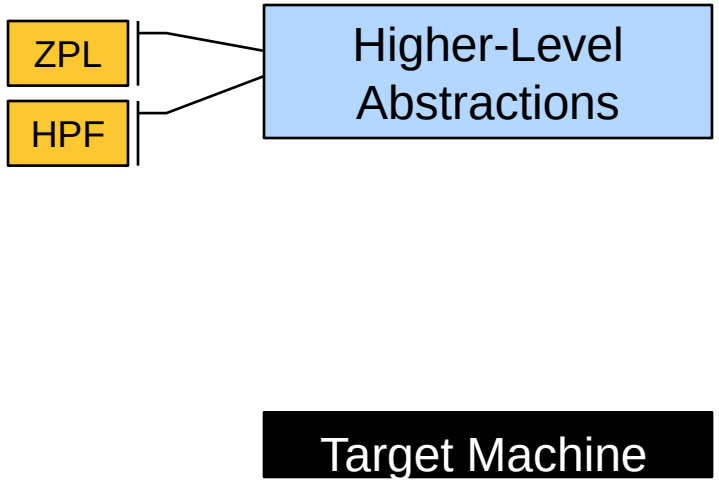
(and ditto for other "failed" 1990's parallel languages as well)

Multiresolution Languages: Motivation

Two typical camps of parallel language design:

low-level vs. high-level





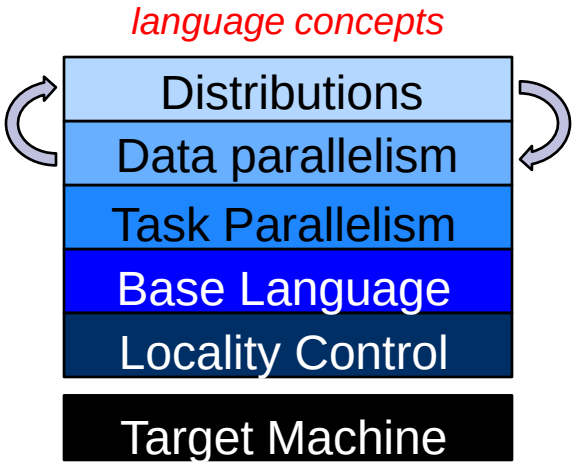
"Why is everything so tedious?"

"Why don't I have more control?"

Multiresolution Language Design

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

- 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



Partitioned Global Address Space Languages

(Or perhaps: partitioned global namespace languages)

- abstract concept:
  - support a shared namespace
    - permit any parallel task to access any lexically visible variable
  - establish a strong sense of ownership
    - local variables are cheaper to access than remote ones
- founding fathers: UPC, Co-Array Fortran, Titanium
  - extensions to C, Fortran, and Java, respectively
  - details vary, but potential for:
    - arrays that are decomposed across nodes
    - pointers that refer to remote objects
  - note that earlier languages could also be considered PGAS, but that the term didn't exist yet

Traditional PGAS Languages: in a Nutshell

- **Co-Array Fortran:** extend Fortran by adding…
  - …a new array dimension to refer to processor space
  - …collectives and synchronization routines
- **UPC:** extend C by adding support for…
  - …block-cyclic distributed arrays
  - …pointers to variables on remote nodes
  - …a memory consistency model
- **Titanium:** extend Java by adding support for…
  - …multidimensional arrays
  - …pointers to variables on remote nodes
  - …synchronization safety via the type system
  - …region-based memory management
  - …features to help with halo communications and other array idioms

PGAS: What's in a Name?

		memory model	*programming model*	*execution model*	*data structures*	*communication*
	MPI	distributed memory	cooperating executables (often SPMD in practice)		manually fragmented	APIs
	OpenMP	shared memory	global-view parallelism	shared memory multithreaded	shared memory arrays	N/A
PGAS Languages	CAF	PGAS	Single Program, Multiple Data (SPMD)		co-arrays	co-array refs
PGAS Languages	UPC	PGAS	Single Program, Multiple Data (SPMD)		1D block-cyc arrays/ distributed pointers	implicit
PGAS Languages	Titanium	PGAS	Single Program, Multiple Data (SPMD)		class-based arrays/ distributed pointers	method-based
	Chapel	PGAS	global-view parallelism	distributed memory multithreaded	global-view distributed arrays	implicit

PGAS Evaluation

PGAS strengths

+ Implicit expression of communication through variable names
+ Ability to reason about locality/affinity supports scalable performance

Traditional PGAS language strengths

+ Elegant, reasonably minimalist extensions to established languages

Traditional PGAS language weaknesses

– **CAF:** Problems that don't divide evenly impose bookkeeping details
– **UPC:** Like C, 1D arrays seem impoverished for many HPC codes
– **Titanium:** Perhaps too pure an OO language for HPC
  – e.g., arrays should have value rather than reference semantics

– **all:** Imposes an SPMD programming + execution model on the user

Hybrid Programming Models

- abstract concept:
  - use multiple models for the concerns they handle best
  - support a natural division of labor
- for example, MPI+OpenMP
  - MPI for the inter-node concerns
  - OpenMP for the intra-node
- see also:
  - MPI+pthreads
  - MPI+CUDA
  - MPI+OpenCL
  - …

MPI+OpenMP (or other hybrid models)

strengths:

+ Supports a division of labor: let each technology do what it does best

weaknesses:

– Requires two distinct notations to express a single logical parallel computation

Why must we use multiple completely distinct notations to express the same key concerns—parallelism and locality—for different architectural levels or types?

Case Study: STREAM (current practice)

CUDA

```
#define N       2000000

int main() {
  float *d_a, *d_b, *d_c;
  float scalar;

  cudaMalloc((void**)&d_a, sizeof(float)*N);
  cudaMalloc((void**)&d_b, sizeof(float)*N);
  cudaMalloc((void**)&d_c, sizeof(float)*N);

  dim3 dimBlock(128);
  dim3 dimGrid(N/dimBlock.x );
  if( N % dimBlock.x != 0 ) dimGrid.x+=1;

  set_array<<<dimGrid,dimBlock>>>(d_b, .5f, N);
  set_array<<<dimGrid,dimBlock>>>(d_c, .5f, N);

  scalar=3.0f;
  STREAM_Triad<<<dimGrid,dimBlock>>>(d_b, d_c, d_a, scalar,  N);
  cudaThreadSynchronize();

  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);
}

__global__ void set_array(float *a,  float value, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) a[idx] = value;
}

__global__ void STREAM_Triad( float *a, float *b, float *c,
                              float scalar, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) c[idx] = a[idx]+scalar*b[idx];
}
```

MPI + OpenMP

```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Comm_size( comm, &commSize );
  MPI_Comm_rank( comm, &myRank );

  rv = HPCC_Stream( params, 0 == myRank);
  MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm );

  return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
  register int j;
  double  scalar;

  VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(double), 0 );

  a = HPCC_XMALLOC( double, VectorSize );
  b = HPCC_XMALLOC( double, VectorSize );
  c = HPCC_XMALLOC( double, VectorSize );

  if (!a || !b || !c) {
    if (c) HPCC_free(c);
    if (b) HPCC_free(b);
    if (a) HPCC_free(a);
    if (doIO) {
      fprintf( outFile, "Failed to allocate memory (%d).\n", VectorSize );
      fclose( outFile );
    }
    return 1;
  }

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 0.0;
  }

  scalar = 3.0;

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];

  HPCC_free(c);
  HPCC_free(b);
  HPCC_free(a);

  return 0;
}
```

Case Study: STREAM (current practice)

CUDA

```
#define N       2000000

int main() {
  float *d_a, *d_b, *d_c;
  float scalar;

  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) c[idx] = a[idx]+scalar*b[idx];
}
```

MPI + OpenMP

```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
  int myRank, commSize;
  int rv, errCount;
  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Comm_size( comm, &commSize );
  MPI_Comm_rank( comm, &myRank );

#ifdef _OPENMP
#pragma omp parallel for
#endif
  for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];

  HPCC_free(c);
  HPCC_free(b);
  HPCC_free(a);

  return 0;
}
```

Chapel (today)

```
config const m = 1000,
             tpb = 256;
const alpha = 3.0;

const gpuDist = new GPUDist(rank=1, tpb);

const ProbSpace: domain(1) = [1..m];
const GPUProbSpace: domain(1) distributed gpuDist
                  = ProbSpace;

var hostA, hostB, hostC: [ProbSpace] real;
var gpuA, gpuB, gpuC: [GPUProbSpace] real;

hostB = …;
hostC = …;

gpuB = hostB;
gpuC = hostC;

forall (a, b, c) in (gpuA, gpuB, gpuC) do
  a = b + alpha * c;

hostA = gpuA;
```

Chapel (ultimate goal)

```
config const m = 1000,
             tpl = here.numCores,
             tpb = 256;

const alpha = 3.0;

const ProbDist = new BlockCPUGPU(rank=1, tpl, tpb);

const ProbSpace: domain(1) distributed ProbDist
                = [1..m];

var A, B, C: [ProbSpace] real;

B = …;
C = …;

forall (a,b,c) in (A,B,C) do
  a = b + alpha * c;
```

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: Characterization via Motivators

- We've encountered several motivators throughout this talk:
  - general parallelism: data, task, concurrency, nested
    - finer-grain, more dynamic parallelism
  - rich array support (e.g., multidimensional)
    - global-view
    - unified types for local and distributed arrays
  - user-defined distributions
  - global namespace / PGAS memory model
    - more abstract specification of communication
  - modern language concepts
    - OOP available, but not required
  - interoperability with legacy models
  - multiresolution design
  - unified concepts across architectural types and levels
  - productivity

(Come to the mini-symposium for more details)

X10 and Fortress: Similarities to Chapel

- PGAS memory model
  - plus, language concepts for referring to realms of locality
- more dynamic (“post-SPMD”) execution model
  - one logical task executes main()
  - any task can create additional tasks--local or remote
- global-view data structures
  - ability to declare and access distributed arrays holistically rather than piecemeal

X10 and Fortress: Distinguishing Themes

- X10:
  - takes a purer object-oriented approach
    - originally based on Java, more recently on Scala
  - a bit more minimalist and purer
    - e.g., less likely to add abstractions to the language if expressible using objects
  - stronger story for exceptions
  - semantics distinguish between local and remote more strongly
- Fortress:
  - one view: how can we write code as mathematically as possible?
  - a more accurate view: how can we define a language that defines as little about the language semantics as possible?
    - including data types, operator precedence, …
  - Follows more of a functional language design
  - I believe recent work has focused less on large-scale machines
  - Many other intriguing features: OO design, dimensional units, …

Exascale

- Exascale is coming and will bring many new challenges
  - increased hierarchy and heterogeneity in the node architecture
    => our abstract machine model will need to change
  - increased machine size and degree of parallelism
    => computations will need to be more dynamic, resilient

- We can view this as a scary time, or one of great opportunity

- Programming model recommendation from Dec09 Workshop on Architectures & Technology: invest in two paths…
  1) evolutionary, hybrid approach (e.g., MPI 3.0 + OpenMP 4.0?)
  2) unified, holistic approach (e.g., Chapel, X10, ParalleX, …)

Some Other Notable Programming Models

- **Distributed objects/remote method invocation:**
  - Charm++, …
- **Massive multithreading:**
  - ParalleX, Cilk, Cray MTA/XMT C…
- **Interactive HPC, linear algebra:**
  - parallel Matlab, Star-P, MatlabMPI, …
- **Data-intensive computing:**
  - Map-Reduce, PLINQ, DryadLINQ, …
- **Increased determinism and safety:**
  - DPJ, Yada, …

Note that, as alluded to in Burton's talk, many of these are motivated by or coming from (or have been purchased by) mainstream rather than HPC computing

Emerging Programming Paradigms Mini-Symposium

- **Part I (9:50am – 11:50am, Eliza Anderson):**
  - autotuning
  - multicore/accelerator programming
  - MPI+OpenMP
  - CPU+Cell Hybrid Computing
- **Part II (1:20pm – 3:20pm, Leonesa II):**
  - UPC
  - ~~X10~~
  - Chapel
  - CAF
- **Part III (4:30pm – 6:30pm, Eliza Anderson):**
  - Yada
  - DryadLINQ
  - CUDA
  - Hadoop

Summary

- This is an exciting time for parallel programming models
  - HPC community seems open to new models for first time since HPF
    - in part thanks to DARPA HPCS
    - in part due to threat/opportunity of exascale computing
  - the mainstream is wrestling with parallel programmability as well
    - due to multicore and GPUs
    - a good opportunity to learn from one another
    - an opportunity for HPC to leverage the broader community

- HPC needs to continue to push itself to invest time and resources into new programming models
  - to deal with limitations in our current approaches
  - to prepare for the anticipated changes as we move to exascale

Questions?