CRAY
THE SUPERCOMPUTER COMPANY

An Overview of Chapel

Steve Deitz

Cray Inc.

What is Chapel?

- A new parallel language
  - Under development at Cray Inc.
  - Supported through the DARPA HPCS program
- Status
  - Version 1.1 released April 15, 2010
  - Open source via BSD license

    http://chapel.cray.com/

    http://sourceforge.net/projects/chapel/

The Chapel Team

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

- Former Team Members

  David Callahan, Roxana Diaconescu, Samuel Figueroa, Shannon Hoffswell, Mary Beth Hribar, Mark James, John Plevyak, Wayne Wong, Hans Zima

- Interns

  Mackale Joyner (‘05 – Rice)
  Robert Bocchino (’06 – UIUC)
  James Dinan (’07 – Ohio St.)
  Andy Stone (’08 – Colorado St.)
  Jacob Nelson (‘09 – U. Wash.)
  Albert Sidelnik (‘09 – UIUC)
  Jonathan Turner (’10 – U. Colorado)

Goals For Today

- Introduce you to Chapel with a focus on
  - Data parallelism
  - Task parallelism
  - Execution on distributed-memory systems
- Discuss to-date performance
- Get your feedback on Chapel
- Point you towards resources to use after today

CRAY
THE SUPERCOMPUTER COMPANY

Outline

- Background
- HPCC Benchmarks in Chapel
- Language Basics
- Data Parallelism
- Multi-Locale Execution
- Task Parallelism
- Wrap Up

Chapel Settings

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

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

Chapel Productivity Goals

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

Chapel Themes

- Global-view abstractions
- General parallel programming
  - Express all software parallelism
  - Target all hardware parallelism
- Control of locality
  - Control placement of data and tasks
  - Necessary to target distributed memory with performance
- Mainstream language features
  - Object-oriented programming
  - Generic programming and latent types
  - Modules, nested functions, associative arrays, ...

Global-View Abstractions

Definitions

- **Programming model**

  The mental model of a programmer

- **Fragmented model**

  Programmer takes point-of-view of a single processor/thread

- **SPMD models** (Single Program, Multiple Data)

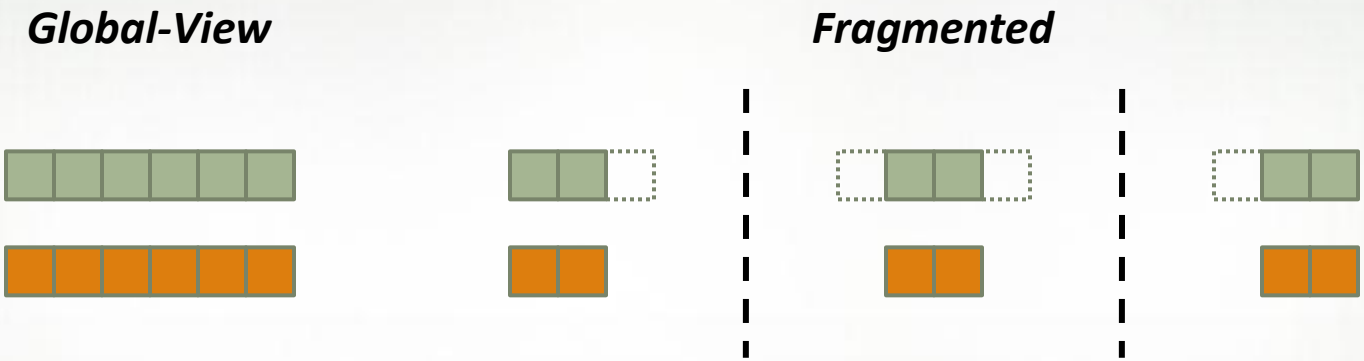
  Fragmented models with multiple copies of one program

- **Global-view model**

  Programmer writes code to describe computation as a whole

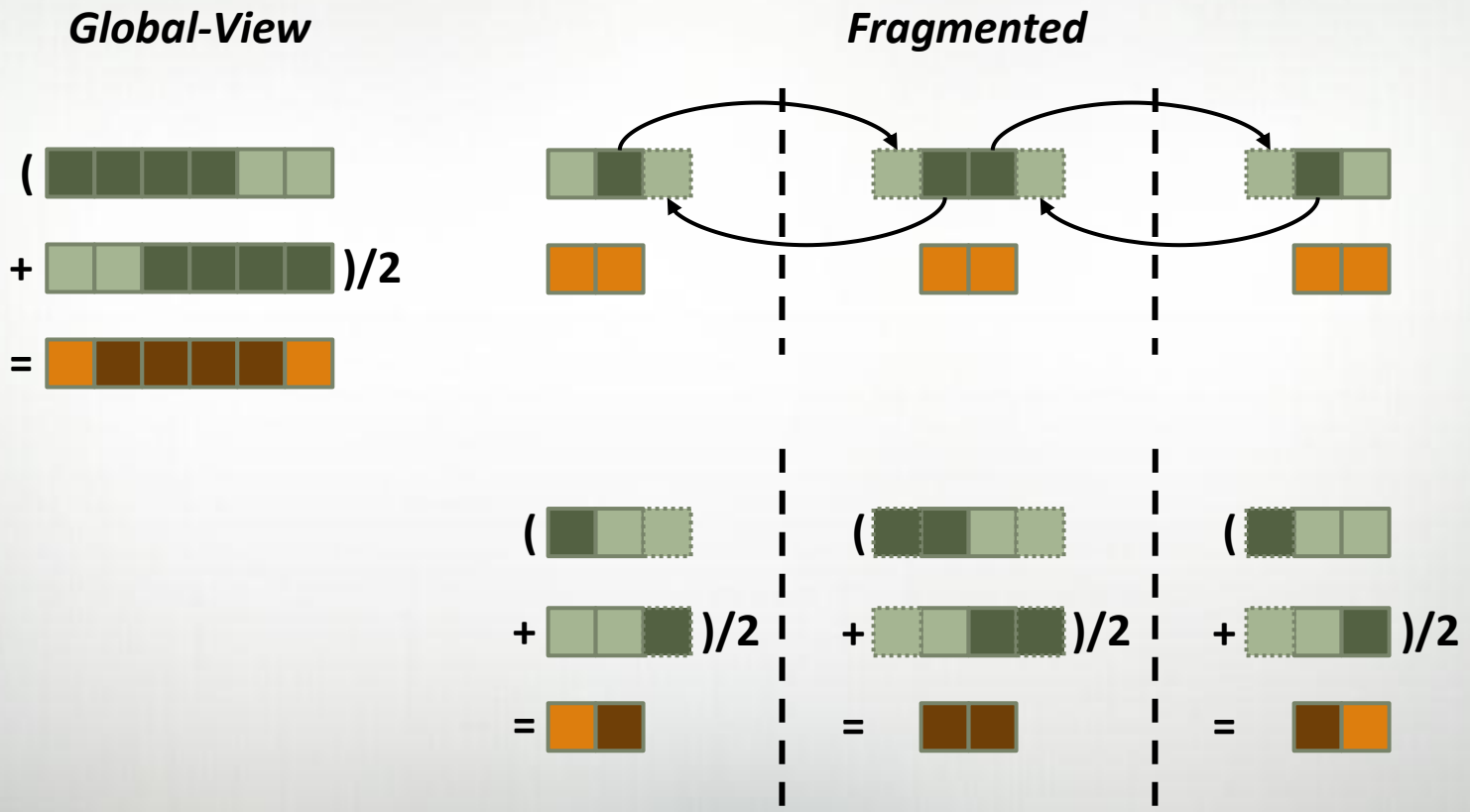
Global-View Abstractions

Example: 3-Point Stencil (Data Declarations)



Global-View Abstractions

Example: 3-Point Stencil (Computation)

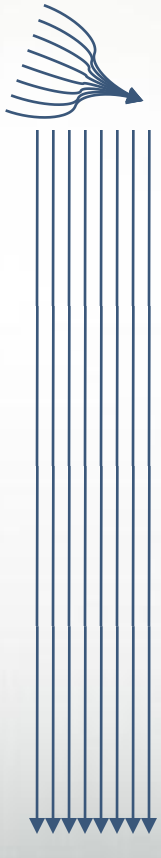


Global-View Abstractions

Example: 3-Point Stencil (Code)

Global-View

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

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

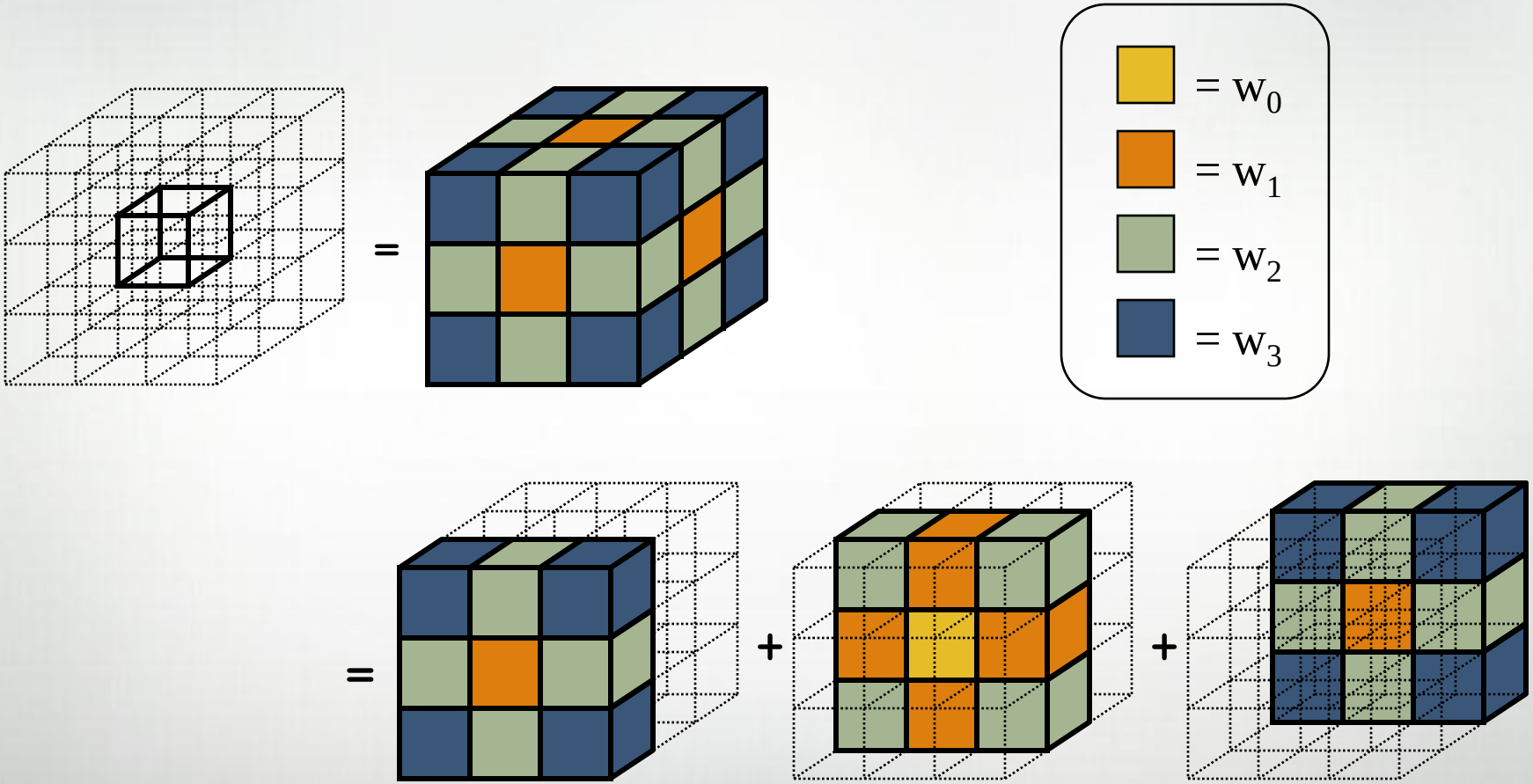


Fragmented

Assumes p divides n

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

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

NAS MG Stencil



NAS MG Stencil in Fortran + MPI

```
use caf_intrinsics

 implicit none

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

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

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

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


 implicit none

 include 'cafnpb.h'
 include 'globals.h'  subroutine
     comm3(u,n1,n2,n3,kk)

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

 integer i3, i2, i1, buff_len,buff_id

 buff_id = 2 + dir
 buff_len = 0

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

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


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

    else if( dir .eq. +1 ) then

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


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

    endif
 endif

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

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


    buff(1:buff_len,buff_id+1)[nbr(axis
     ,dir,k)] =
>      buff(1:buff_len,buff_id)
 else if( dir .eq. +1 ) then

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


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

    endif
 endif

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

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


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

    else if( dir .eq. +1 ) then

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


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

    endif
 endif

 return
 end


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


 implicit none

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

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

 integer buff_id, indx

 integer i3, i2, i1

 buff_id = 3 + dir
 indx = 0

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

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

    else if( dir .eq. +1 ) then

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

    endif
 endif

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

       do  i3=2,n3-1
          do  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

    else if( dir .eq. +1 ) then

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

    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(i1,i2,i3) = buff(indx,
      buff_id )
          enddo
    enddo
 endif

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

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

 return
 end

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

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

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

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

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

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

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

NAS MG Stencil in Chapel

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

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

Our previous work in ZPL has shown that such compact codes can result in better performance than the Fortran + MPI.

Summary of Current Programming Systems

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

CRAY
THE SUPERCOMPUTER COMPANY

Outline

- Background
- HPCC Benchmarks in Chapel
  - The Challenge Competition at SC '09
  - STREAM Triad, RandomAccess, FFT, HPL
- Language Basics
- Data Parallelism
- Multi-Locale Execution
- Task Parallelism
- Wrap Up

Prologue

This is a reprise of a 10-minute presentation I gave at the HPCC Competition at Supercomputing 2009. The finalists were

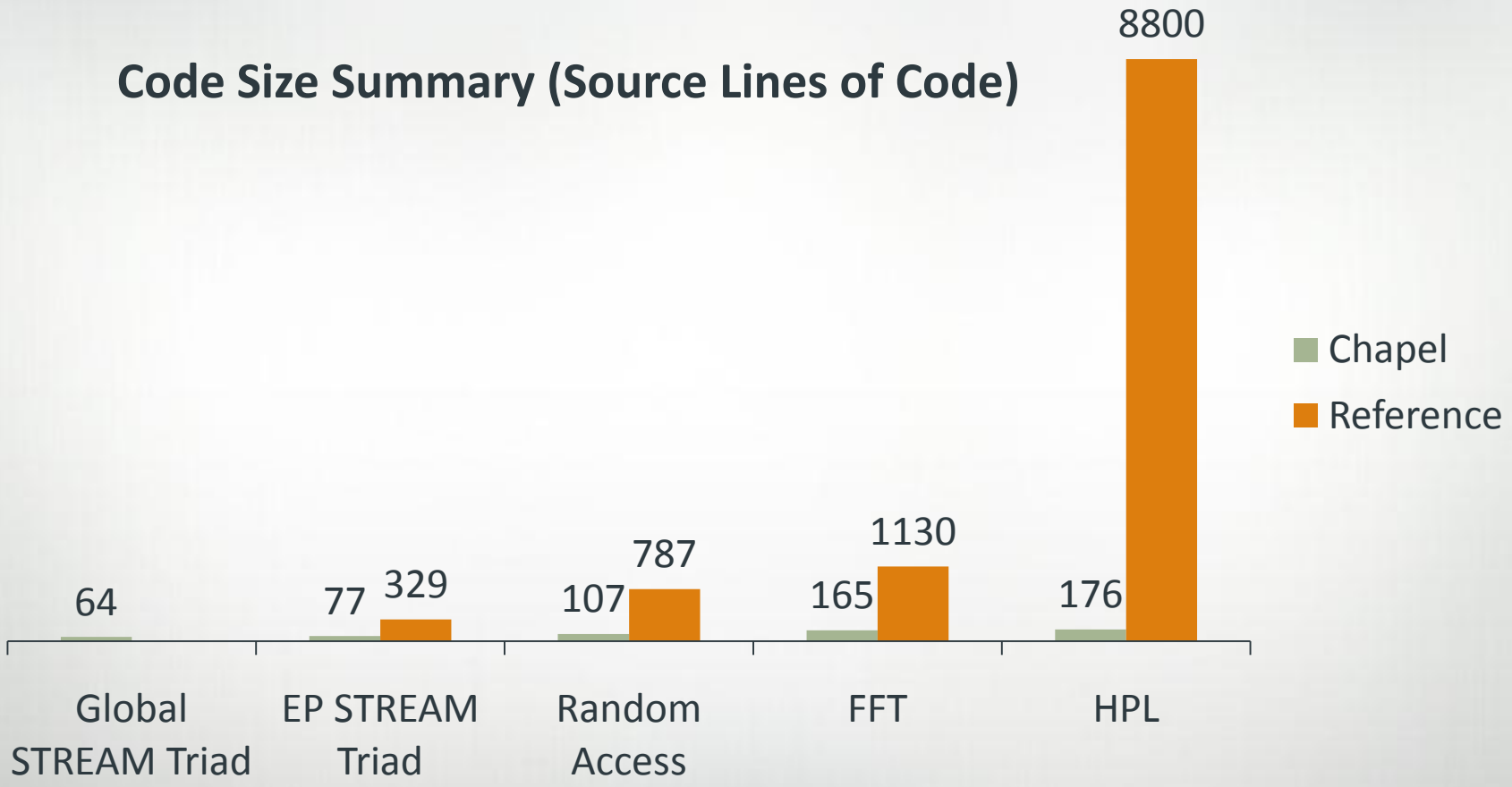
- Cray: Chapel
- IBM: X10
- University of Tsukuba: XcalableMP

The codes have been updated to version 1.1 of Chapel.

Highlights

- Global STREAM Triad 10.8 TB/s (6.4x over 2008)
  - Executed on 2048 nodes (up from 512 nodes in 2008)
  - Better scaling by eliminating extra communication
- EP STREAM Triad 12.2 TB/s
  - More similar to EP STREAM reference version
- Random Access 0.122 GUP/s (111x over 2008)
  - Executed on 2048 nodes (up from 64 nodes in 2008)
  - Optimized remote forks + better scaling as with STREAM
- A distributed-memory implementation of FFT
- A demonstration of portability
  - Cray XT4, Cray CX1, IBM pSeries 575, SGI Altix

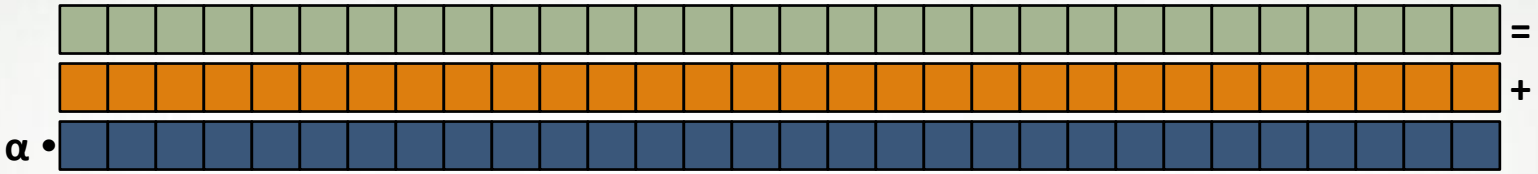
Chapel Implementation Characteristics



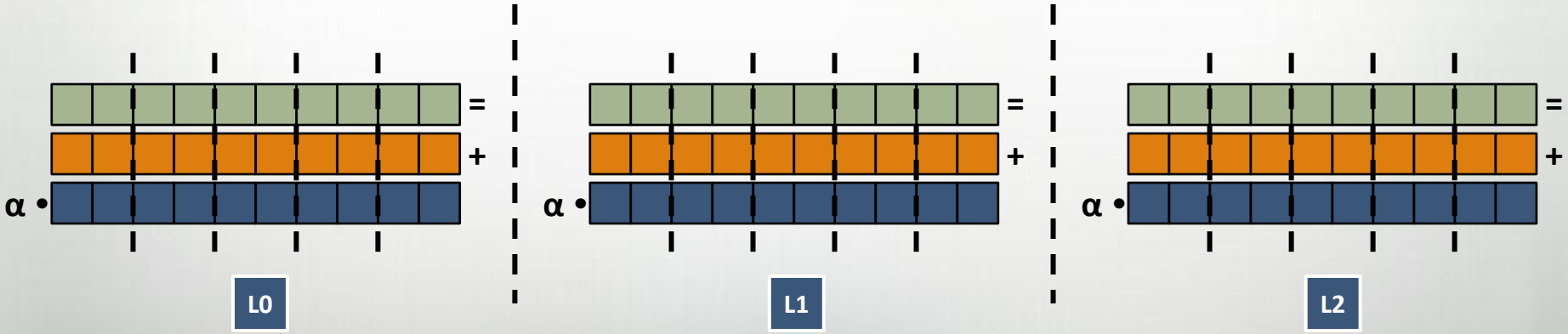
What is a Distribution?

A “recipe” for distributed arrays that...

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



...to a fragmented, per-processor implementation



Chapel Distributions

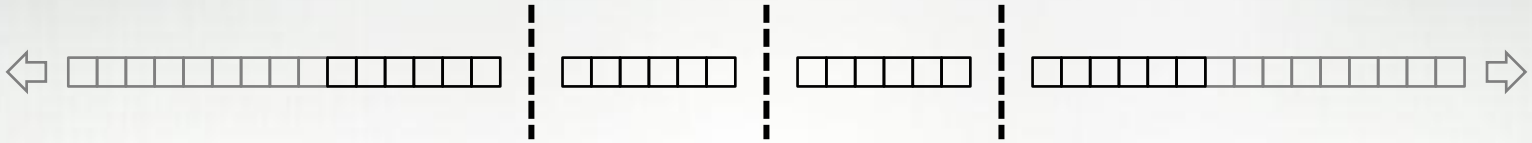
- Distributions are written entirely in Chapel
  - Advanced programmers can write their own
  - Classes define distributions, domains, and arrays
  - Compiler lowers code to a structural interface
  - Task-parallel constructs implement concurrency
- Standard distributions under development
  - Block
  - Cyclic
  - BlockCyclic
  - Associative
  - GPU

FFT and HPL in a Nutshell

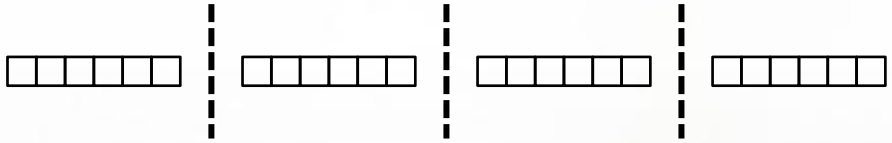
- FFT
  - Uses both Block and Cyclic distributions
  - Butterfly-patterned accesses are completely local
    - Communication with nearby neighbors is local with Block
    - Communication with far off neighbors is local with Cyclic
  - Executes on distributed memory, but is slow
- HPL
  - Implementation is ready for BlockCyclic distribution
  - Executes on single locale only, but is multi-threaded

Global STREAM Triad in Chapel (Excerpts)

```
const BlockDist = new dmap(new Block(…));
```



```
const ProblemSpace:
        domain(1,int(64)) dmapped BlockDist = [1..m];
```

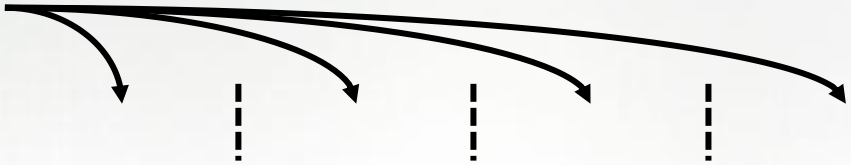


```
var A, B, C: [ProblemSpace] elemType;
```

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

EP STREAM Triad in Chapel (Excerpts)

```
coforall loc in Locales do on loc {
```



```
  local {
    var A, B, C: [1..m] elemType;
```

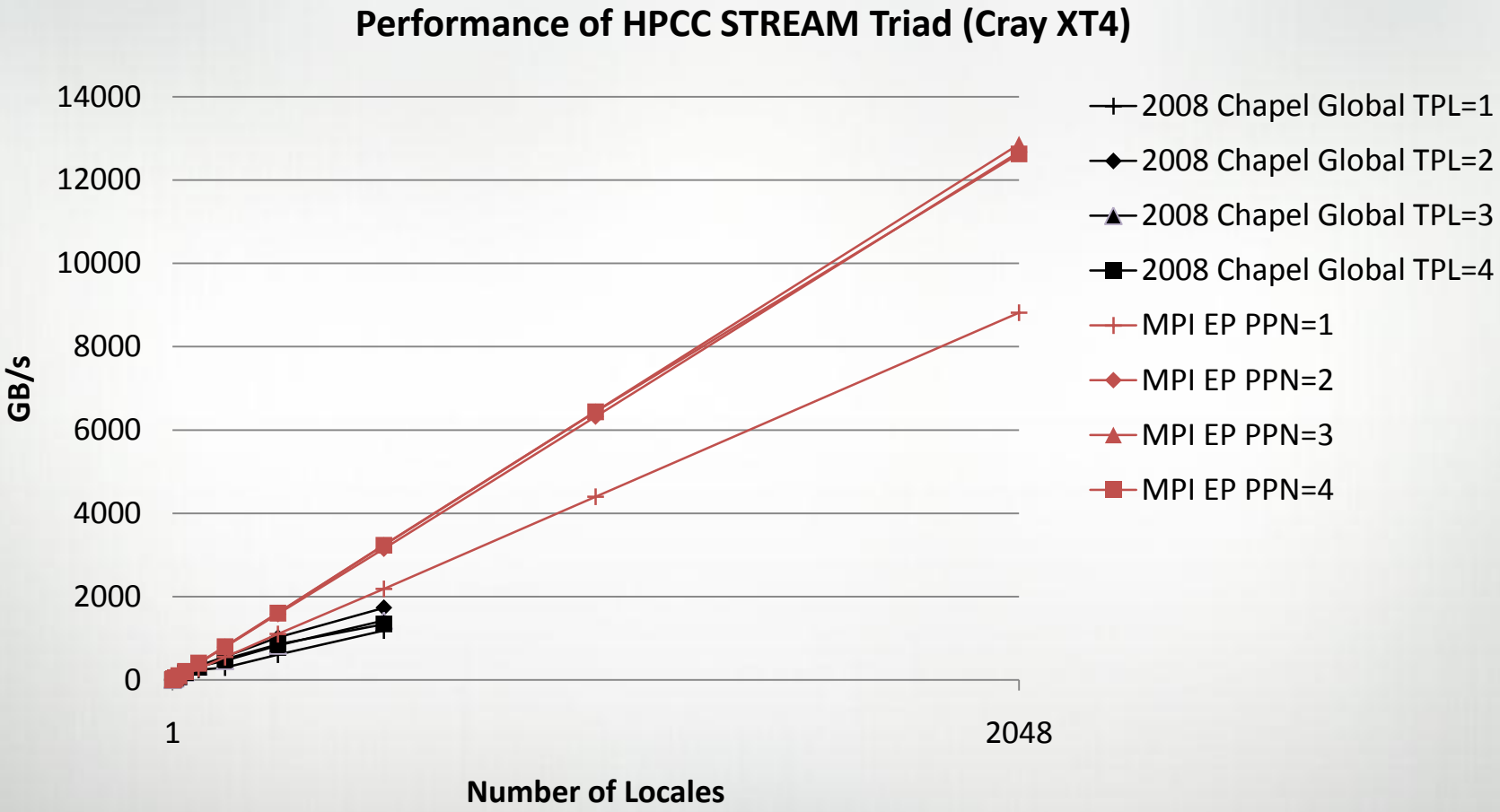
```
    forall (a,b,c) in (A,B,C) do
      a = b + alpha * c;
  }
}
```

Experimental Setup

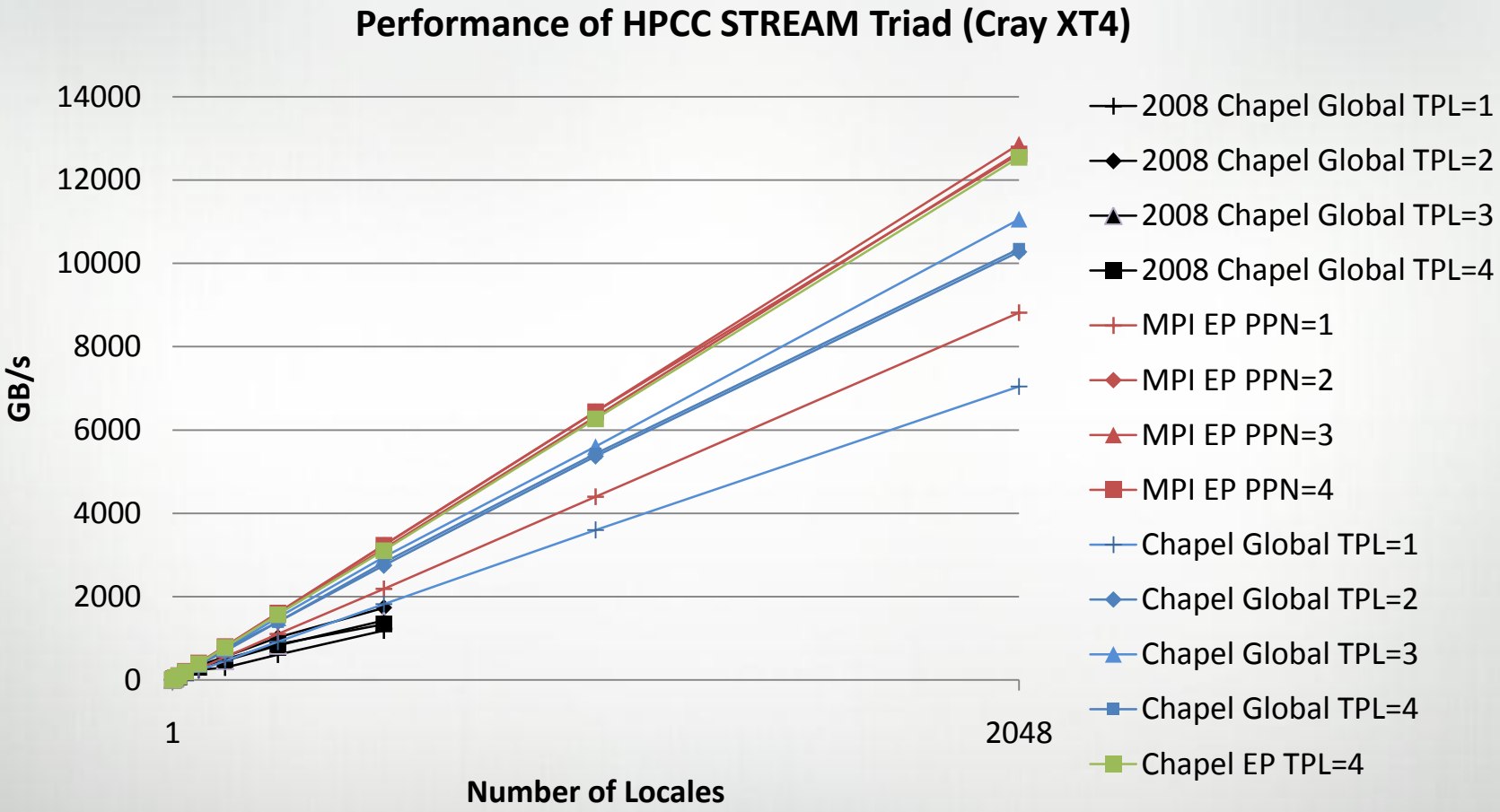
Machine Characteristics	
Model	Cray XT4
Location	ORNL
Nodes	7832
Processor	2.1 GHz Quadcore AMD Opteron
Memory	8 GB per node

Benchmark Parameters	
STREAM Triad Memory	Least value greater than 25% of memory
Random Access Memory	Least power of two greater than 25% of memory
Random Access Updates	2^{n-10} for memory equal to 2^n

STREAM Triad Performance



STREAM Triad Performance



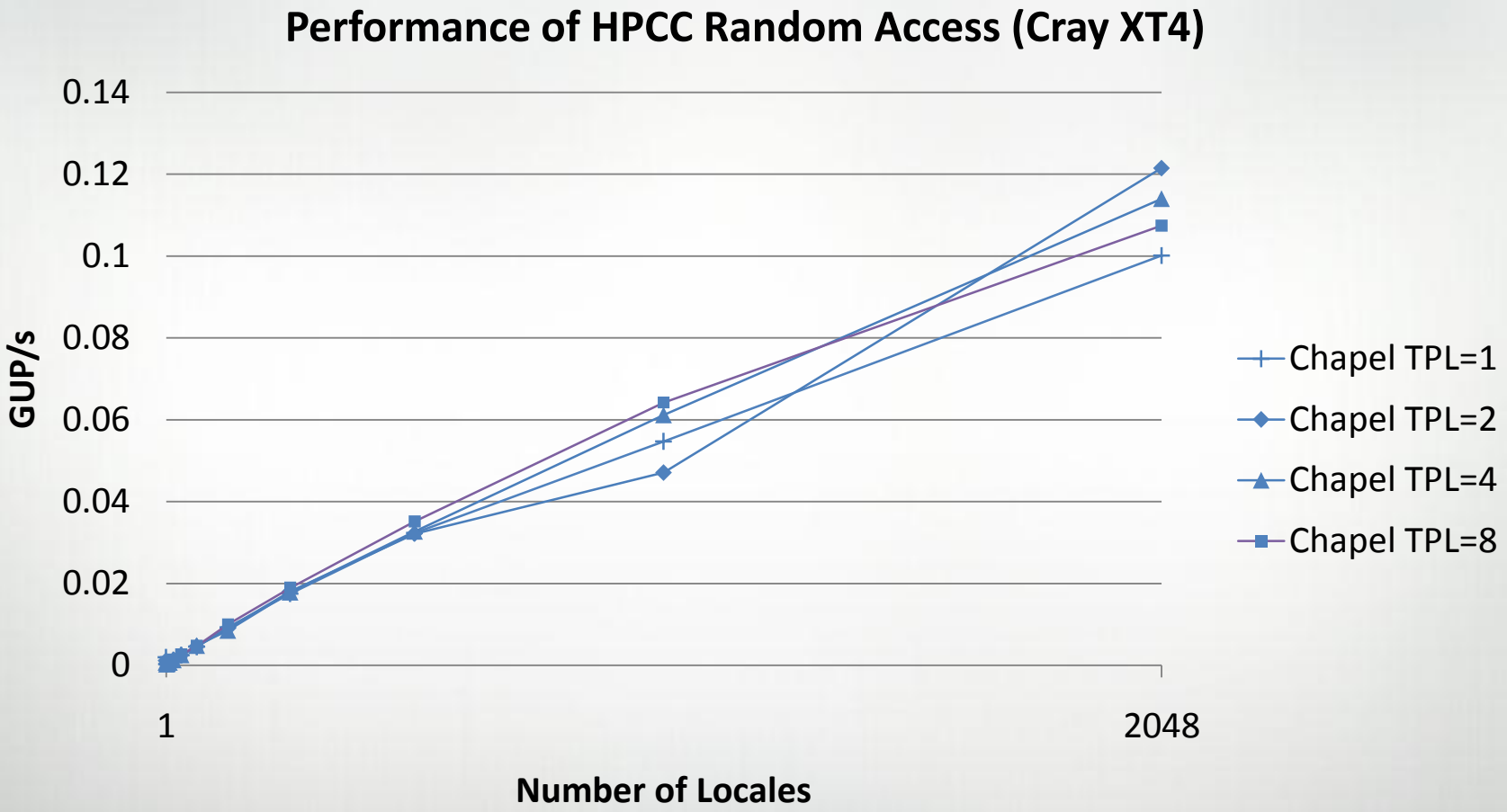
Global Random Access in Chapel (Excerpts)

```
const TableDist = new dmap(new Block(…0..m…)),
      UpdateDist = new dmap(new Block(…0..N_U…));

const TableSpace: domain … dmapped TableDist = …,
      Updates: domain … dmapped UpdateDist = …;

var T: [TableSpace] elemType;

forall ( ,r) in (Updates,RAStream()) do
  on TableDist.idxToLocale(r & indexMask) {
    const myR = r;
    local T(myR & indexMask) ^= myR;
  }
```

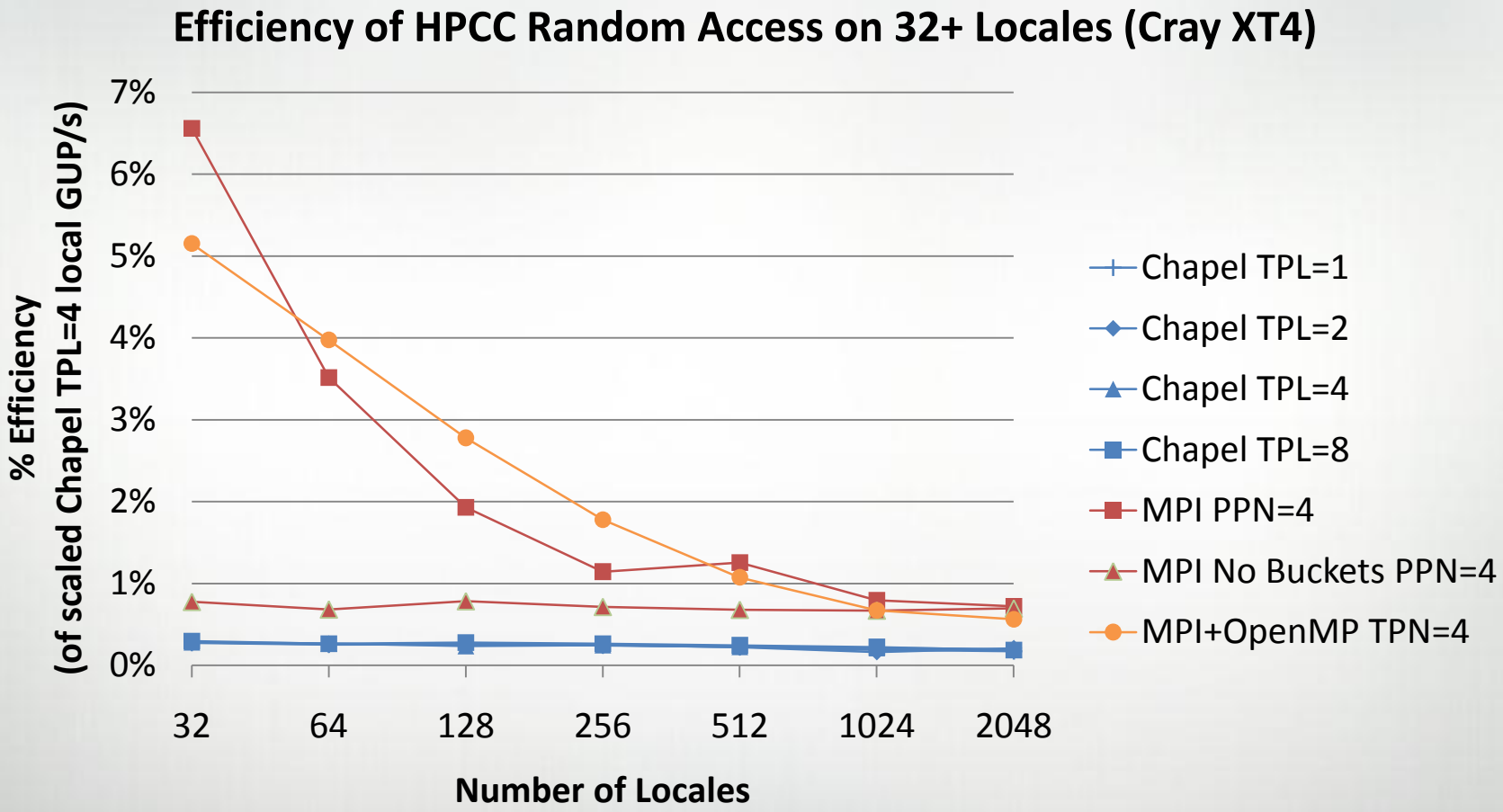
More elegant on-block

```
on T(r&indexMask) do
  T(r&indexMask) ^= r;
```

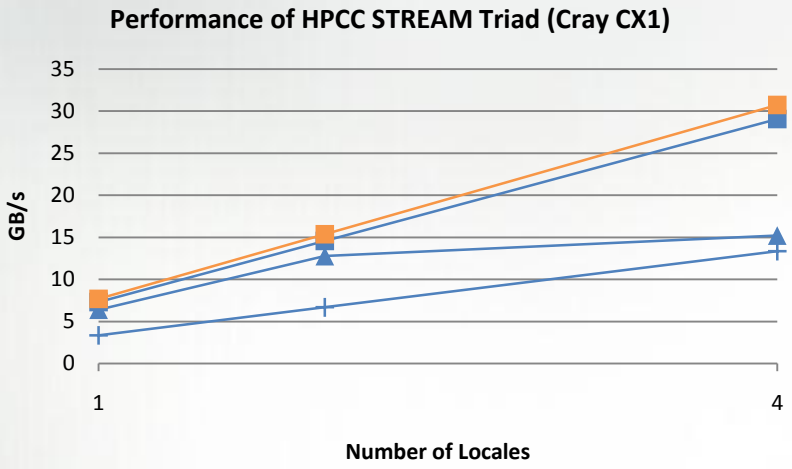
Random Access Performance

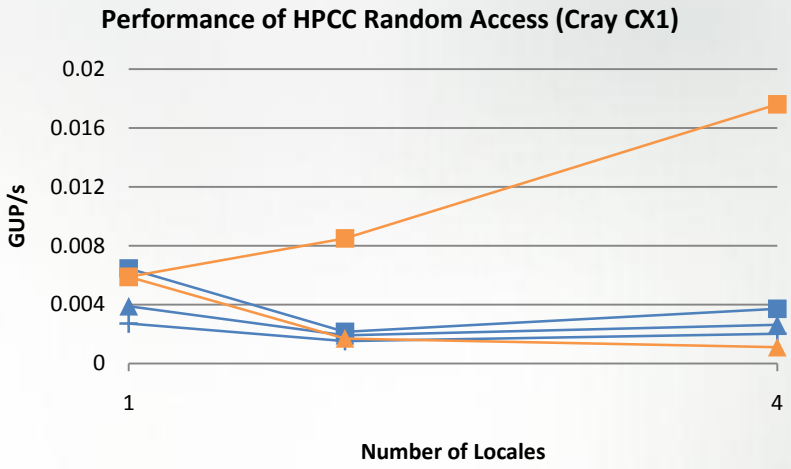


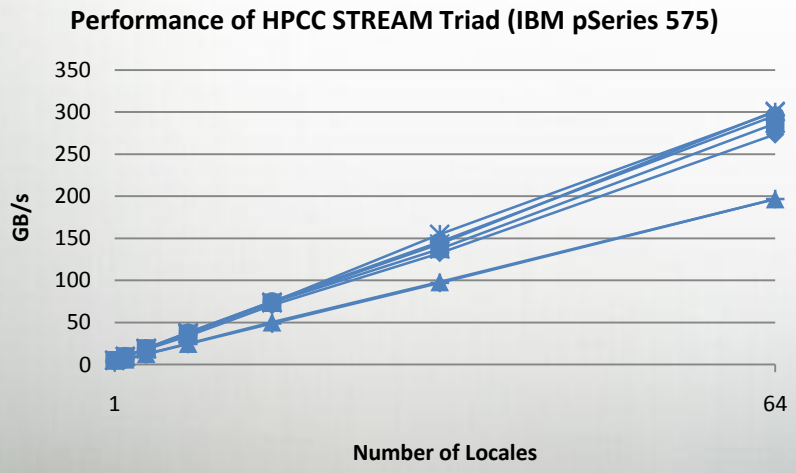
Random Access Efficiency on 32+ Nodes

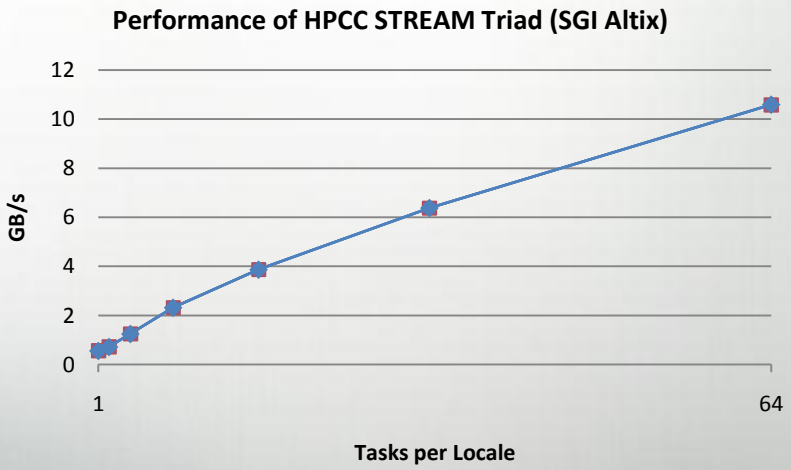


Portability Results









Summary

- Chapel is a work-in-progress
  - Performance is improving
  - Development of distributions is progressing

Score Card	Elegance	Performance
Global STREAM Triad	64 SLOC	10.8 TB/s
EP STREAM Triad	77 SLOC	12.2 TB/s
Random Access	107 SLOC	0.11 GUP/s
FFT	165 SLOC	0.00015 Gflop/s
HPL	176 SLOC	Multi-threaded, single-locale

No library routines were used in this entry.

Epilogue

The prize was split as follows

- $1000 Most Elegant Implementation: Chapel
- $1000 Best Performance: X10
- Honorable Mention: XcalableMP

CRAY
THE SUPERCOMPUTER COMPANY

Outline

- Background
- HPCC Benchmarks in Chapel
- Language Basics
  - Overview and ‘hello, world’
  - Miscellaneous features
- Data Parallelism
- Multi-Locale Execution
- Task Parallelism
- Wrap Up

The Hello World Program

- Fast prototyping

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

- Production-grade

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

Characteristics of Chapel

- Syntax
  - Basics from C and Modula
  - Influences from many other languages
- Semantics
  - Imperative, block-structured
  - Optional object-oriented programming (OOP)
  - Elided types for convenience and generic coding
  - Static typing for performance and safety
- Design points
  - No pointers and few references
  - No compiler-inserted array temporaries

Chapel Influences

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

CRAY MTA C/Fortran: task parallelism, synchronization

CLU, Ruby, Python: iterators

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

Java, C#: OOP, type safety

C++: generic programming/templates

Primitive Types

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

- Syntax

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

- Examples

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

Variables, Constants, and Parameters

- Syntax

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

- Semantics
  - Constness at runtime (**`const`**), at compile-time (**`param`**)
  - Omitted *init-expr*: value is assigned default for type
  - Omitted *type*: type is inferred from *init-expr*
- Examples

```
var count: int;              // initialized to 0
const pi: real = 3.14159;
param debug = true;          // inferred to be bool
```

Config Declarations

- Syntax

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

- Semantics
  - Supports command-line overrides
  - Must be declared at module (file) scope
- Examples

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

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

Input and Output

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

Tuples

- Syntax

```
tuple-expr:
   ( expr, expr-list )

expr-list:
   expr
   expr, expr-list
```

- Semantics
  - Light-weight first-class data structure
- Examples

```
var i3: (int, int, int) = (1, 2, 3);
var i3_2: 3*int = (4, 5, 6);
var triple: (int, string, real) = (7, "eight", 9.0);
```

Ranges

- Syntax

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

- Semantics
  - Regular sequence of integers

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

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

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

Arrays

- Syntax

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

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

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

Much more on arrays in data parallelism part

For Loops

- Syntax

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

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

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

for i in 1..3 do write(A(i));          // DO RE MI
for a in A { a += "LA"; write(a); } // DOLA RELA MILA
```

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

- Syntax

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

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

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

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

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

Function Examples

- Example to compute the area of a circle

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

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

- Example of function arguments

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

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

Iterators

- An abstraction for loop control
  - Yields (generates) values for consumption
  - Otherwise, like a function
- Example

```
def string_chars(s: string) {
  for i in 1..length(s) do
    yield s.substring(i);
}

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

Generic Functions

Generic functions can be defined by explicit type and param arguments:

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

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

```
def goo(x, y) { ...
def sort(A: []) { ...
```

Generic functions are replicated for each unique instantiation:

```
foo(int, x);    // copy of foo() with t==int
foo(string, x); // copy of foo() with t==string
goo(4, 2.2);    // copy of goo() with int and real args
```

Generic Types

Generic types can be defined by explicit type and param fields:

```
class Table { param numFields: int; ...
class Matrix { type eltType; ...
```

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

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

Generic types are replicated for each unique instantiation:

```
// copy of Table with 10 fields
var myT: Table(10);
// copy of Triple with x:int, y:int, z:real
var my3: Triple(int,int,real) = new Triple(1,2,3.0);
```

Other Basic Language Features

- Classes, records, unions, and enumerated types
- Expression forms of conditionals and loops
- Type select statements
- Function instantiation constraints (where clauses)
- Formal argument intents (in, out, inout, const)
- User-defined compiler warnings and errors

CRAY
THE SUPERCOMPUTER COMPANY

Outline

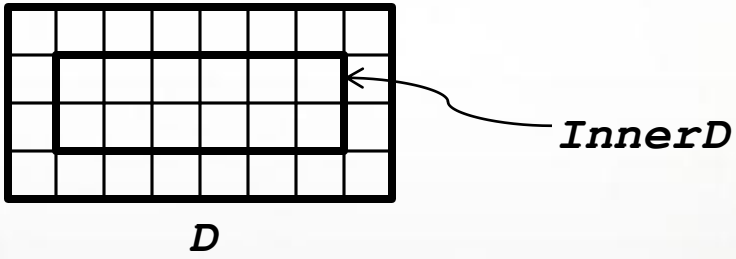
- Background
- HPCC Benchmarks in Chapel
- Language Basics
- Data Parallelism
  - Domains and arrays
  - Forall, reductions, scans, and promotion
- Multi-Locale Execution
- Task Parallelism
- Wrap Up

Domains

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

Sample Arithmetic Domains

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

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

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



Domains Define Arrays

- Syntax

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

- Semantics
  - Stores element for each index in *domain-expr*
- Example

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

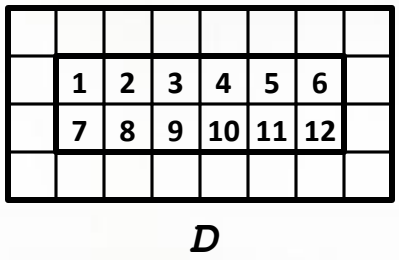
- Revisited example

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

Domain Iteration

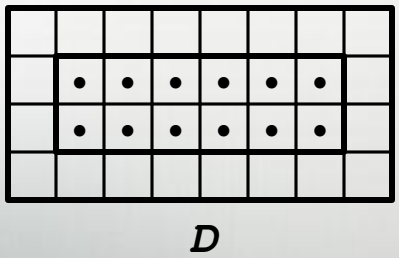
- For loops (discussed already)
  - Executes loop body once per loop iteration
  - Order is serial

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



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

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



Other Forall Loops

Forall loops also support...

- A shorthand:

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

- An expression-based form:

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

- A shorthand expression-based form:

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

A

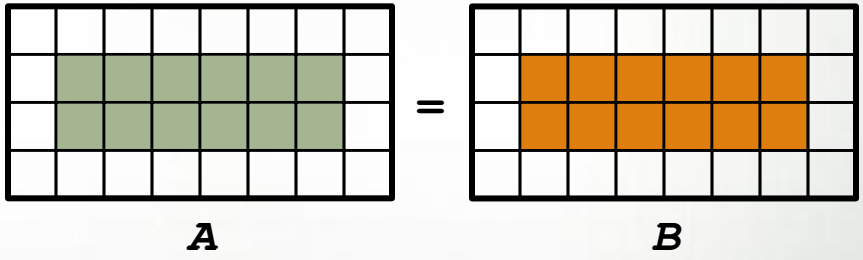
Data Parallelism Configuration Constants

- **--dataParTasksPerLocale=#**
  - Specify # of tasks to execute forall loops
  - Default: number of cores *(in current implementation)*
- **--dataParIgnoreRunningTasks=[true|false]**
  - If false, reduce # of forall tasks by # of running tasks
  - Default: true *(in current implementation)*
- **--dataParMinGranularity=#**
  - If > 0, reduce # of forall tasks if any task has fewer iterations
  - Default: 1 *(in current implementation)*

Other Domain Functionality

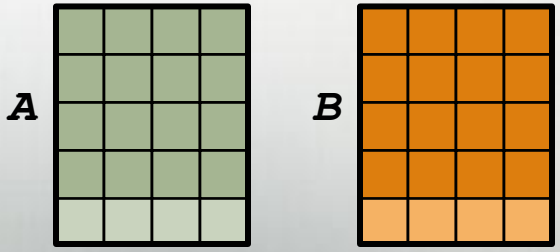
- Domain methods (exterior, interior, translate, ...)
- Domain slicing (intersection)
- Array slicing (sub-array references)

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



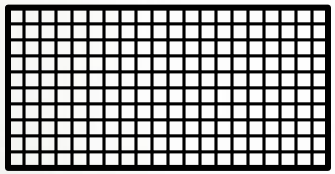
- Array reallocation
  - Reassign domain → change array
  - Values are preserved (new elements initialized)

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

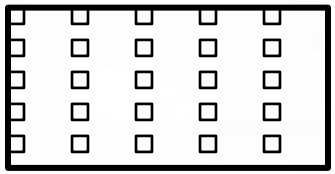


The Varied Kinds of Domains

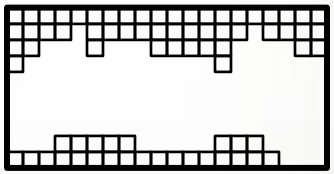
```
var Dense: domain(2) = [1..10, 1..20],
    Strided: domain(2) = Dense by (2, 4),
    Sparse: sparse subdomain(Dense) = genIndices(),
    Associative: domain(string) = readNames(),
    Opaque: domain(opaque);
```



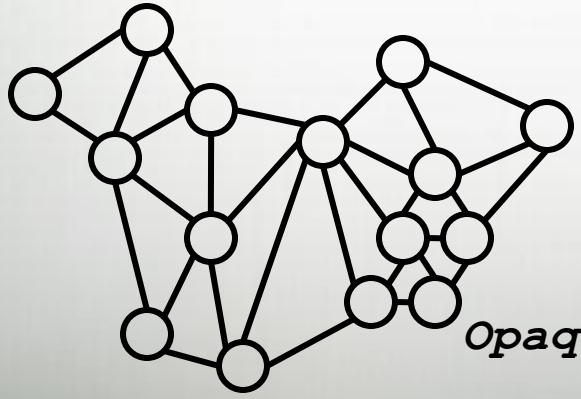
Dense



Strided



Sparse

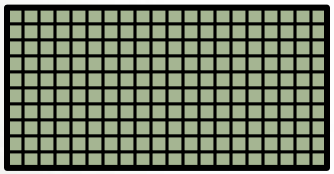


Opaque

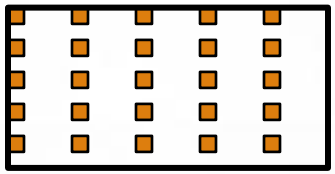
Associative

The Varied Kinds of Arrays

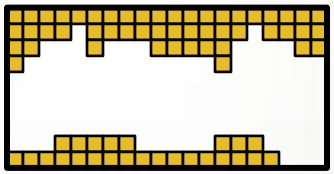
```
var DenseArr: [Dense] real,
    StridedArr: [Strided] real,
    SparseArr: [Sparse] real,
    AssociativeArr: [Associative] real,
    OpaqueArr: [Opaque] real;
```



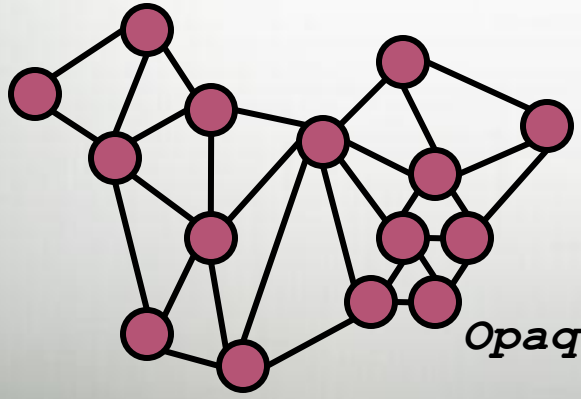
DenseArr



StridedArr



SparseArr

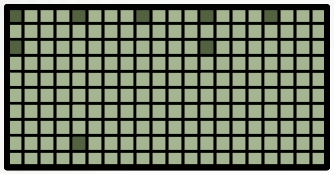


OpaqueArr

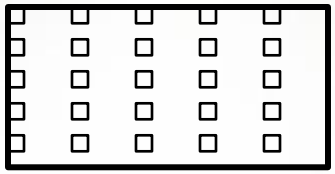
AssociativeArr

All Domains Support Iteration

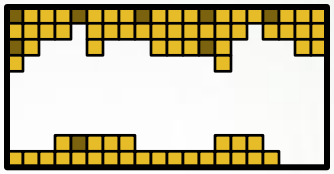
```
forall (i,j) in Strided {
  DenseArr(i,j) += SparseArr(i,j);
}
```



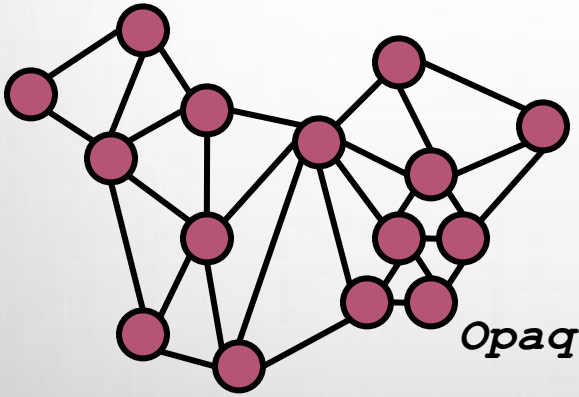
DenseArr



Strided



SparseArr



OpaqueArr

AssociativeArr

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

Data Parallel Promotion

Functions/operators expecting scalars can also take...

- Arrays, causing each element to be passed

```
sin(A)
2*A
```
≈
```
forall a in A do sin(a)
forall a in A do 2*a
```

- Domains, causing each index to be passed

```
foo(Sparse)
```
≈
```
forall i in Sparse do foo(i)
```

Multiple arguments can promote using either...

- Zipper promotion

```
pow(A, B)
```
≈
```
forall (a,b) in (A,B) do pow(a,b)
```

- Tensor product promotion

```
pow[A, B]
```
≈
```
forall (a,b) in [A,B] do pow(a,b)
```

Reductions

- Syntax

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

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

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

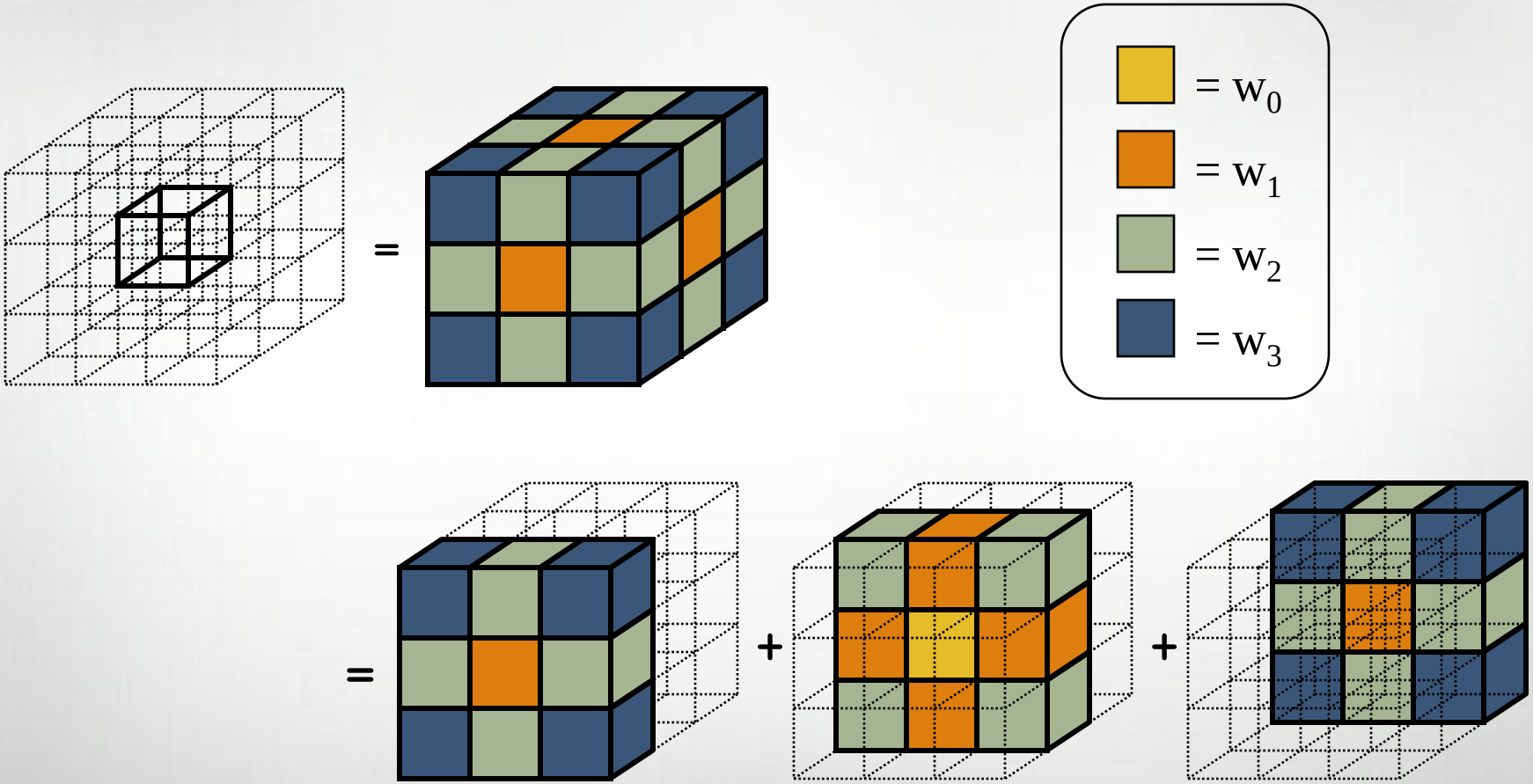
Reduction Operators

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

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

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

NAS MG Stencil Revisited



NAS MG Stencil in Chapel Revisited

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

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

CRAY
THE SUPERCOMPUTER COMPANY

Outline

- Background
- HPCC Benchmarks in Chapel
- Language Basics
- Data Parallelism
- Multi-Locale Execution
  - Locales and ‘on’
  - Distributions and layouts
- Task Parallelism
- Wrap Up

The Locale Type

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

Program Startup

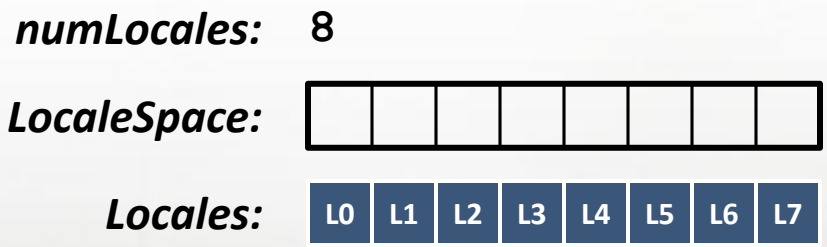
- Execution Context

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

- Specify # of locales when running executable

```
% a.out --numLocales=8
```

```
% a.out -nl 8
```



- Execution begins as a single task on a locale 0

Locale Methods

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

  Returns index in LocaleSpace

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

  Returns name of locale (like uname -a)

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

  Returns number of cores available to locale

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

  Returns physical memory available to user programs on locale

  Example

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

The On Statement

- Syntax

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

- Semantics
  - Executes *stmt* on the locale that stores *expr*
  - Does not introduce concurrency
- Example

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

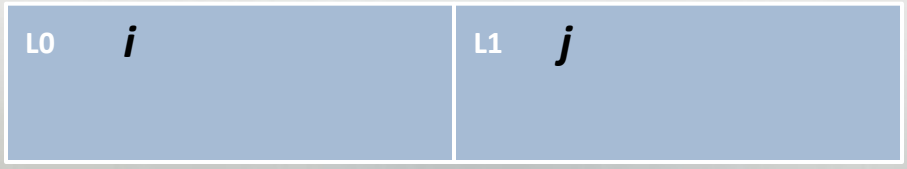
Querying a Variable's Locale

- Syntax

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

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

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



Here

- Built-in locale

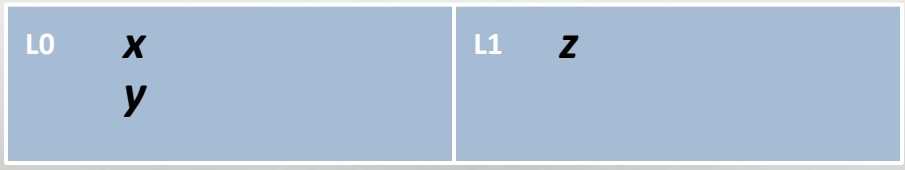
```
const here: locale;
```

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

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

Serial Example with Implicit Communication

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

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

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

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



Asynchronous Remote Tasks

```
on loc do begin f();
```

- Combining concurrency with locality
  - On-statement evaluates locale of *expr*

    Then executes *stmt* on that locale
  - Begin-statement creates a new task to execute *stmt*

    Original task continues with the next statement

Locales vs. Tasks

- Locales
  - Abstraction of memory and processing capability
  - Architecture-dependent definition optimizes local accesses
- Tasks
  - Abstraction of computation or thread
  - Execution is *on* a locale
- Comparisons with other programming models

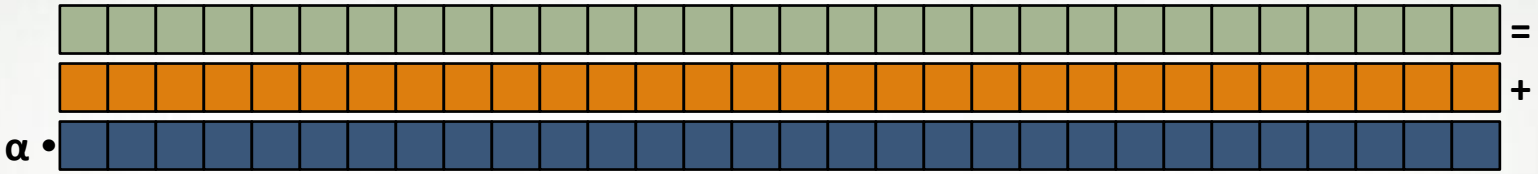
Chapel	OpenMP	MPI	UPC	CAF	Titanium
Locales		Processes	Threads	Images	Demesnes
Tasks	Threads				

Conflated concepts

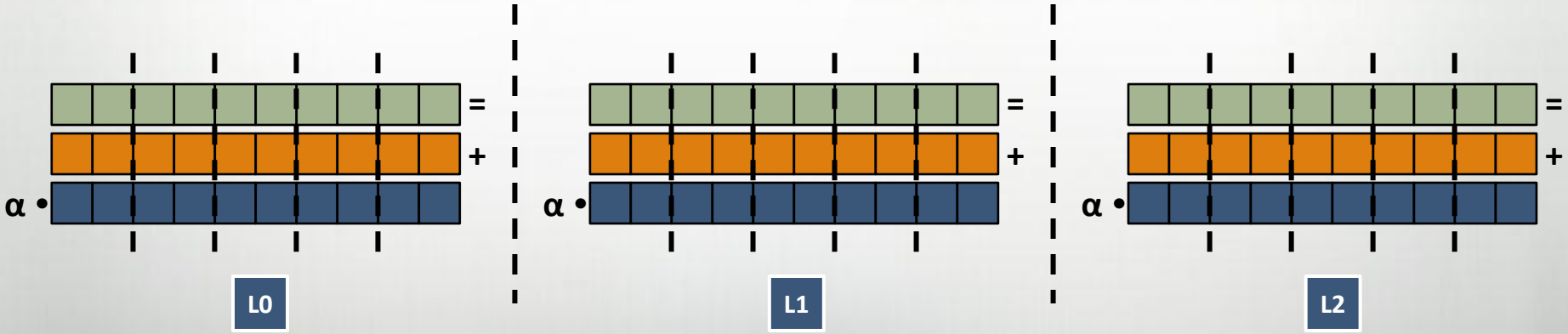
Domain Maps

A “recipe” for arrays that...

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



...to memory and/or to locales



More on Domain Maps

A domain map defines:

- Ownership of domain indices and array elements
- Underlying representation
- Standard set of operations on domains and arrays
  - E.g, slicing, reindexing, rank change
- How to farm out work
  - E.g., forall loops over distributed domains/arrays

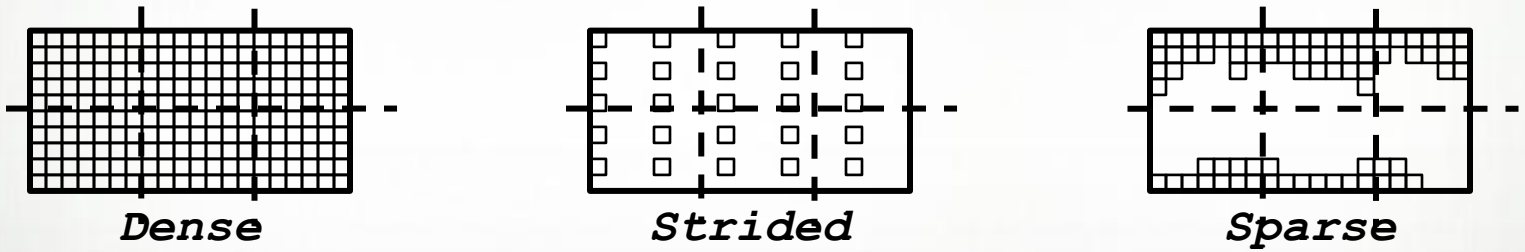
Domain maps are built using language-level constructs

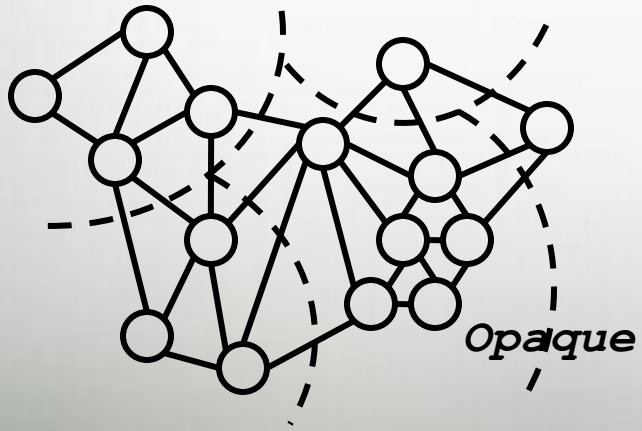
Domain Map Types

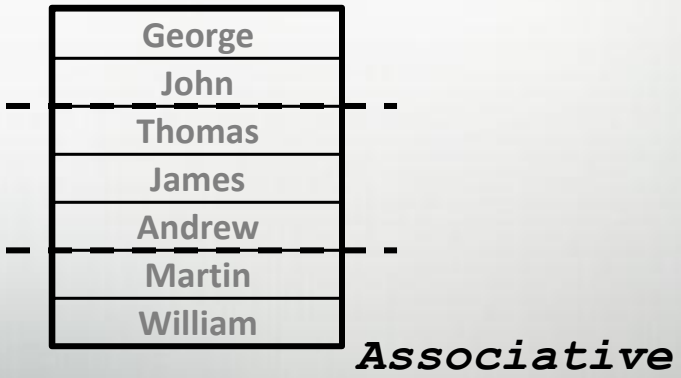
All domain types can be dmapped.

Semantics are independent of domain map.

(Though performance and parallelism will vary...)







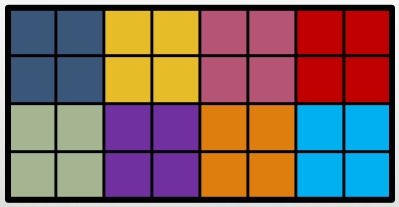
Layouts and Distributions

- Layouts are single-locale domain maps
  - Uses begin, cobegin, coforall to implement data parallelism
  - May take advantage of locale resources, *e.g.*, multiple cores
  - Examples: Sparse CSR, GPU, ...
- Distributions are multi-locale domain maps
  - Uses begin, cobegin, coforall to implement data parallelism
  - Handles both inter- and intra-locale parallelism
  - Uses on to control data and task locality
  - May use layouts for per-locale implementation
  - Examples: block, cyclic, block CSR, recursive bisection, ...
- Standard distributions include both block and cyclic variants

The Block Distribution

The Block Distribution maps the indices of a domain in a dense fashion across the target Locales according to the `boundingBox` argument

```
const Dist = new dmap(new Block(boundingBox=[1..4, 1..8]));

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

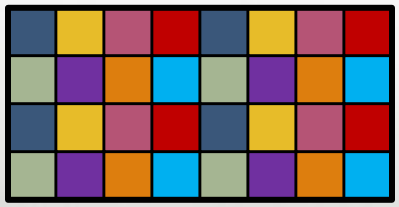


distributed over

The Cyclic Distribution

The Cyclic Distribution maps the indices of a domain in a round-robin fashion across the target Locales according to the `startIdx` argument

```
const Dist = new dmap(new Cyclic(startIdx=(1,1)));

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



distributed over

User-defined Domain Maps

(Advanced) programmers can write domain maps

- The compiler uses a structural interface to build domain maps:
  - Create domains and arrays
  - Map indices to locales
  - Access array elements
  - Iterate over indices/elements sequentially, in parallel, zippered
  - …

Standard Domain Maps ***are*** user-defined domain maps

Design goal: User-defined domain maps should perform as well as the Chapel Standard Domain Maps

CRAY
THE SUPERCOMPUTER COMPANY

Outline

- Background
- HPCC Benchmarks in Chapel
- Language Basics
- Data Parallelism
- Multi-Locale Execution
- Task Parallelism
  - Primitive task-parallel constructs
  - Structured task-parallel constructs
- Wrap Up

Unstructured Task Creation: Begin

- Syntax

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

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

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

- Possible output

```
hello world
good bye
```

```
good bye
hello world
```

Synchronization via Sync-Types

- Syntax

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

- Semantics
  - Default read blocks until written (until “full”)
  - Default write blocks until read (until “empty”)
- Examples: futures and critical sections

```
var future$: sync real;

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

```
var lock$: sync bool;

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

Sync-Type Methods

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

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

Block-Structured Task Invocation: Cobegin

- Syntax

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

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

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

Cobegin is Unnecessary

Any cobegin statement

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

can be rewritten in terms of begin statements

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

but the compiler may miss out on optimizations.

Loop-Structured Task Invocation: Coforall

- Syntax

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

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

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

Usage of Begin, Cobegin, and Coforall

- Use begin when
  - Creating tasks with unbounded lifetimes
  - Load balancing requires dynamic task creation
  - Cobegin and coforall are insufficient for task structuring
- Use cobegin when
  - Invoking a fixed # of tasks (potentially heterogeneous)
  - The tasks have bounded lifetimes
- Use coforall when
  - Invoking a fixed or dynamic # of homogeneous task
  - The tasks have bounded lifetimes

Usage of For, Forall, and Coforall

- Use for when
  - A loop must be executed serially
  - One task is sufficient for performance
- Use forall when
  - The loop can be executed in parallel
  - The loop can be executed serially
  - Degree of concurrency << # of iterations
- Use coforall when
  - The loop must be executed in parallel
    (And not just for performance reasons!)
  - Each iteration has substantial work

Structuring Sub-Tasks: Sync-Statements

- Syntax

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

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

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

Program Termination and Sync-Statements

Where the cobegin statement is static,

```
cobegin {
  functionWithBegin();
  functionWithoutBegin();
}
```

the sync statement is dynamic.

```
sync {
  begin functionWithBegin();
  begin functionWithoutBegin();
}
```

Program termination is defined by an implicit sync.

```
sync main();
```

Atomic Transactions (Unimplemented)

- Syntax

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

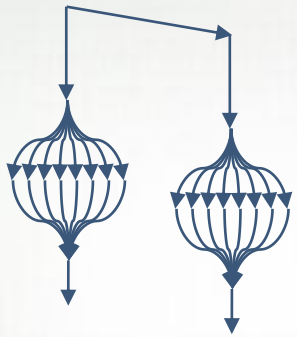
- Semantics
  - Executes stmt so it appears as a single operation
  - No other task sees a partial result
- Example

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

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

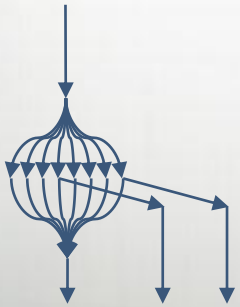
Nested Parallelism

- Task parallelism of data parallelism



```
cobegin {
  forall (a, b, c) in (A, B, C) do
    a = b + alpha * c;
  forall (d, e, f) in (D, E, F) do
    d = e + beta * f;
}
```

- Data parallelism of task parallelism



```
forall i in D do
  if i >= 0 then
    A(i) = f(i);
  else
    on A(i) do begin A(i) = g(i);
```

CRAY
THE SUPERCOMPUTER COMPANY

Outline

- Background
- HPCC Benchmarks in Chapel
- Language Basics
- Data Parallelism
- Multi-Locale Execution
- Task Parallelism
- Wrap Up

Chapel Version 1.1

- Features
  - Open source at http://sourceforge.net/projects/chapel/
  - Distributed under the BSD Open Source license
  - Ported to Linux/Unix, Mac, Cygwin
- Contents
  - Compiler, runtime, standard modules, third-party libraries
  - Language spec, quick reference, numerous examples
- Highlights
  - Most data-parallel operations execute in parallel
  - Improved control of data parallelism
  - Completed Block and Cyclic distributions

Implementation Status

- Language Basics
  - No support for inheritance from multiple or generic classes
  - Incomplete support for sparse arrays and domains
  - No support for skyline arrays
  - Several internal memory leaks
- Task Parallelism
  - No support for atomic statements
  - Memory consistency model is not guaranteed
- Data Parallelism
  - Promoted functions/operators do not preserve shape
  - No partial scans or reductions

Collaborations I

- **Notre Dame/ORNL** (Peter Kogge, Srinivas Sridharan, Jeff Vetter)
  Asynchronous software transactional memory over distributed memory
- **UIUC** (David Padua, Albert Sidelnik, Maria Garzaran)
  Chapel for hybrid CPU-GPU computing
- **BSC/UPC** (Alex Duran)
  Chapel over Nanos++ user-level tasking
- **U. Malaga** (Rafa Asenio, Maria Gonzales, Rafael Larossa)
  Parallel file I/O for arrays
- **OSU** (Gagan Agrawal, Bin Ren)
  User-defined reductions over FREERIDE for data intensive computing

Collaborations II

- **U. Colorado** (Jeremy Siek, Jonathan Turner)
  Interfaces and modular generics for Chapel
- **PNNL/CASS-MT** (John Feo, Daniel Chavarria)
  Hybrid computing in Chapel; Cray XMT performance tuning; ARMCI port
- **ORNL** (David Bernholdt *et al.*, Steve Poole *et al.*)
  Code studies – Fock matrices, MADNESS, Sweep3D, coupled models, ...
- **Berkeley** (Dan Bonachea, Paul Hargrove *et al.*)
  Efficient GASNet support for Chapel; collective communication
- **U. Oregon/Paratools Inc.** (Sameer Shende)
  Performance analysis with Tau

http://chapel.cray.com/ ◆ http://sourceforge.net/projects/chapel ◆ chapel_info@cray.com