Lessons Learned in Array Programming: from ZPL to Chapel

Brad Chamberlain
Chapel Team, Cray Inc.
ARRAY 2016, June 14, 2016
Safe Harbor Statement

This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.
My Background

Education:

- Earned Ph.D. from University of Washington CSE in 2001
- Remain associated with UW CSE as an Affiliate Professor

Industrial Experience:

- Currently a Principal Engineer at Cray Inc.
- Also spent an educational year at a start-up between UW and Cray
My R&D Interests

Designing and Implementing Parallel Languages
- targeting HPC (High-Performance Computing) system scales
- making use of array abstractions
  (as well as other abstractions)

Parallel Array Languages I’ve Worked on:

ZPL: at UW CSE

Chapel: at Cray
Chapel’s Founders / Relationship to ZPL

Hans Zima
Vienna Fortran, HPF
Fortran-oriented
Performance
Minimal features

David Callahan
MTA C, Fortran
Multithreading
Generality
User-optimizable

Brad Chamberlain
ZPL
Array Programming
Syntactic performance model
SPMD execution

John Plevyak
D, Python, OCaml
Type inference
Generics
OOP
Today’s Talk

**Theme:**
- Comparison of array features in ZPL vs. Chapel
  - and by extension, differences between academic vs. practical languages
    - **ZPL:** very pure, focused language (“array-based data parallelism”)
    - **Chapel:** very general, intended for adoption (“any parallelism, including arrays”)

**Outline:**
- **Background**
- **ZPL**
  - Arrays in ZPL
  - ZPL’s greatest hits
  - ZPL’s drawbacks
- **Chapel arrays** (learning from ZPL’s mistakes)
- **Q&A / Discussion**
**Local-view vs. Global-view**

**Local-view:**
- code describes per-task behavior
- programmer manages coordination details

**Global-view:**
- code describes algorithm as a whole
- lower layers manage coordination (compiler, runtime, libraries)
Parallel Programming Stereotypes

local-view

MPI

non-transparent parallel / distributed implementation

transparent parallel / distributed implementation

HPF

ZPL

global-view

non-transparent parallel / distributed implementation
ZPL Factsheet

**ZPL:** an array-based data-parallel language

**Developed at:** University of Washington

**Timeframe:** 1991 – 2003 (can still download today)

**Target systems:** 1990’s-era HPC systems
- clusters of commodity processors or SMPs
- custom parallel architectures:
  - Cray T3E, KSR, SGI Origin, IBM SP2, Sun Enterprise, ...

**Related languages:** HPF

**Main concepts:**
- abstract machine model: *the CTA*
- data parallelism via *regions* and arrays
- *WYSIWYG performance model*
Regions: index sets that...

...can be named

\[
\text{region } R = [1..n, 1..n];
\]

\[
\text{BigR} = [0..n+1, 0..n+1];
\]

...are used to declare parallel arrays

\[
\text{var } A, B, C: [\text{BigR}] \text{ integer;}
\]

...specify indices for a statement’s array references

\[
[R] A := B + C;
\]
Regions Eliminate Redundancy

C:
for (i=1; i<=n; i++) {
    for (j=1; j<=n; j++) {
        A[i][j] = B[i][j] + C[i][j];
    }
}

F90:  \( A(1:n,1:n) = B(1:n,1:n) + C(1:n,1:n) \)

ZPL:  [1..n,1..n] A := B + C;
or:   [R] A := B + C;
Array Operators: modify the current region’s index set for an array reference

E.g., @-operator translates indices by a direction:

direction east = [0, 1];
west = [0,-1];

[R] A := B@east + C@west;
Regions Emphasize Differences

\[
\begin{array}{c}
\text{C: } \\
\text{\begin{verbatim}
for (i=1; i<=n; i++) {
    for (j=1; j<=n; j++) {
        A[i][j] = B[i][j+1] + C[i][j-1];
    }
}
\end{verbatim}}
\end{array}
\]

\[
\begin{array}{c}
\text{F90: } \\
\text{A(1:n,1:n) = B(1:n,2:n+1) + C(1:n,0:n-1)}
\end{array}
\]

\[
\begin{array}{c}
\text{ZPL: } \\
\text{[1..n,1..n] A := B@[0,1] + C@[0,-1];}
\end{array}
\]

or:
\[
\begin{array}{c}
\text{[R] A := B@east + C@west;}
\end{array}
\]
More on Regions / Parallel Arrays

• Region scopes can nest:

```plaintext
[R] begin
  A := Index1 + Index2 / 10.0;
  B := A@east;
  [i, 1..n] B := 0.0;
  foo(B);
end;
```

• Constrained, by design:
  – Parallel arrays do not support “normal” indexing
  – Regions do not support direct iteration

```plaintext
for (i,j) in R do A[i,j] = B[i,j+1];
```
Parallel Interpretation of Regions

- Users specify virtual processor grid at runtime
  
  ```c
  > ./a.out -p4  \(\text{uses a 2x2 grid } \begin{array}{cc}
  & \\
  \end{array}\) \\
  > ./a.out -p4 -r1 (uses a 1x4 grid \begin{array}{cccc}
  & & & \\
  \end{array}\)
  ```

- Region indices are distributed to this grid
  - defines data distribution for arrays
  - defines work distribution for computations
Region Distribution

1) Regions are block-distributed:

2) Interacting regions are distributed identically.

[R] A := B + C;
Distribution rules imply a performance model:

- Traditional operators are perfectly parallel
  \[
  [R] \quad A := B + C;
  \]

- Array operators indicate likely communication
  • of a particular style
  \[
  [R] \quad A := B@east + C@west;
  \]
\textbf{@-induced Communication}

@ is used to refer to neighboring elements.

- data must be transferred to neighboring processors

\[ [R] \ A := A@east; \]

\begin{itemize}
\item \textit{global view}
\item \textit{local view}
\end{itemize}
Flood Communication

Floods replicate data across array dimensions
– data must be broadcast over sub-dimensions of the processor grid

\[ [R] \ A := \text{>>}[i,] \ A; \]

**global view**

**local view**
Remap Communication

Remap arbitrarily re-arranges data
– indexes into an array with other arrays
– tends to require all-to-all communication

\[ R \ A := A[#(B,C)] ; \]
## Array Operator Summary

<table>
<thead>
<tr>
<th>operator</th>
<th>effect</th>
<th>sample</th>
<th>communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>translate indices</td>
<td>A@east</td>
<td>point-to-point</td>
</tr>
<tr>
<td>flood</td>
<td>replicate array elements</td>
<td>$\gg [i,] A$</td>
<td>sub-grid broadcast</td>
</tr>
<tr>
<td>reduction</td>
<td>collapse array elements</td>
<td>$\ll &lt;&lt; [R] A$</td>
<td>sub-grid reduction</td>
</tr>
<tr>
<td>scan</td>
<td>parallel scan</td>
<td>$+</td>
<td></td>
</tr>
<tr>
<td>remap</td>
<td>arbitrary gather/scatter</td>
<td>A#[X, Y]</td>
<td>all-to-all</td>
</tr>
</tbody>
</table>

This is what we refer to as ZPL’s *WYSIWYG performance model*: Communication implications are visible in a program’s syntax.
ZPL’s Greatest Hits: Hierarchical Arrays in NAS MG
Mathematically: use a 3D multigrid method to find an approximate solution to a discrete Poisson problem ($\nabla^2 u = v$)
MG’s arrays

V:
input array

U:
hierarchical work arrays

R:
Hierarchical Arrays

conceptually:

dense indexing:

(1:8,1:8) (1:4,1:4) (1:2,1:2) (1:1,1:1)

strided indexing:

(1:8:1,1:8:1) (1:8:2,1:8:2) (1:8:4,1:8:4) (1:8:8,1:8:8)
Hierarchical Arrays

conceptually:

<table>
<thead>
<tr>
<th>level 0</th>
<th>level 1</th>
<th>level 2</th>
<th>level 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="level 0" /></td>
<td><img src="image2" alt="level 1" /></td>
<td><img src="image3" alt="level 2" /></td>
<td><img src="image4" alt="level 3" /></td>
</tr>
</tbody>
</table>

dense indexing:

<table>
<thead>
<tr>
<th>level 0</th>
<th>level 1</th>
<th>level 2</th>
<th>level 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="level 0" /></td>
<td><img src="image2" alt="level 1" /></td>
<td><img src="image3" alt="level 2" /></td>
<td><img src="image4" alt="level 3" /></td>
</tr>
</tbody>
</table>

strided indexing:

<table>
<thead>
<tr>
<th>level 0</th>
<th>level 1</th>
<th>level 2</th>
<th>level 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="level 0" /></td>
<td><img src="image2" alt="level 1" /></td>
<td><img src="image3" alt="level 2" /></td>
<td><img src="image4" alt="level 3" /></td>
</tr>
</tbody>
</table>
Distributed Hierarchical Arrays

conceptually:

- dense indexing:  
  - level 0: (1:8, 1:8)  
  - level 1: (1:4, 1:4)  
  - level 2: (1:2, 1:2)  
  - level 3: (1:1, 1:1)

- strided indexing:  
  - level 0: (1:8:1, 1:8:1)  
  - level 1: (2:8:2, 2:8:2)  
  - level 2: (4:8:4, 4:8:4)  
  - level 3: (8:8:8, 8:8:8)
MG’s Timed Portion

repeat \( nit \) times

\[
\text{resid} \rightarrow \text{resid} \rightarrow \text{norm2u3}
\]

\[
\text{mg3p} \quad \boxed{\text{rmr2}} \quad \boxed{\text{rmrnu}}
\]
MG’s Guts ($mg3P$)
27-point stencils

\[ \begin{align*}
\text{27-point stencils} & = w_1 + w_2 + w_3 + w_4 \\
\text{27-point stencils} & = w_1 + w_2 + w_3 + w_4 \\
\text{27-point stencils} & = w_1 + w_2 + w_3 + w_4 \\
\text{27-point stencils} & = w_1 + w_2 + w_3 + w_4
\end{align*} \]
\[ \text{rprj3}(S, R) \]

**conceptually:**

\[
S = \text{convolve}( \quad , R )
\]

**strided indexing:**

\[
S = \sum R
\]
Proposed at other levels:

caseually:

\[ S \] = \sum

constred indexing:

\[ S \] = \sum
Per-processor Data Allocation

In addition to its local block of values...

...each processor allocates ghost cells to cache neighboring values.
Stencil Communication

Prior to computing a stencil, communication is typically required to refresh the ghost cells.

Notes:
- Lots of optimization opportunities (e.g., comm/comp overlap, bulk transfers)
- Have to eventually start skipping processors at coarser levels
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
    enddo
  enddo
enddo

if(m1j.eq.3)then
  d1 = 2
else
  d1 = 1
endif

if(m2j.eq.3)then
  d2 = 2
else
  d2 = 1
endif

if(m3j.eq.3)then
  d3 = 2
else
  d3 = 1
endif

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
Local-view \textit{rprj3} in Fortran + MPI

\begin{verbatim}
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
    x2 = r(i1,  i2-1,i3 ) + r(i1,  i2+1,i3 ) + r(i1,  i2,  i3-1) + r(i1,  i2,  i3+1)
    y2 = r(i1,  i2-1,i3-1) + r(i1,  i2-1,i3+1) + r(i1+1,i2,i3 ) + r(i1+1,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
  call comm3(s,m1j,m2j,m3j,j)
enddo
j = k-1
return
end
\end{verbatim}
comm3: Communication in MPI
procedure rprj3(var S, R: [,,] double;
    d: array [] of direction);
begin
    S := 0.5000 * R +
        0.2500 * (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] +
                   0.1250 * (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;
**rprj3 kernel in Fortran, naively**

```fortran
     do  j3=2,m3j-1
        i3 = 2*j3-d3
     do  j2=2,m2j-1
        i2 = 2*j2-d2
     do  j1=2,m1j-1
        i1 = 2*j1-d1
        s(j1,j2,j3) =
           >       0.5D0 * r(i1,i2,i3)
           >       + 0.25D0 * (r(i1-1,i2,i3) + r(i1+1,i2,i3) + r(i1, i2-1,i3 ) + r(i1, i2+1,i3 ) + r(i1, i2,  i3-1) + r(i1, i2,  i3+1))
           >       + 0.125D0 * (r(i1-1,i2-1,i3 ) + r(i1-1,i2+1,i3 ) + r(i1-1,i2,  i3-1) + r(i1-1,i2  ,i3+1) + r(i1+1,i2-1,i3-1) + r(i1+1,i2-1,i3+1) + r(i1+1,i2  ,i3-1) + r(i1+1,i2  ,i3+1))
           >       + 0.0625D0 * (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) + 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
     enddo
enddo
```
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 )
        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)
        x2 = r(i1, i2-1,i3 ) + r(i1, i2+1,i3 )
        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)
        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
enddo
enddo

Stencil Optimization (2D)

- Adjacent stencils use common subexpressions:

- Observation: Cache partial sums for reuse...

- Benefits are greater for 3D stencils...
MG Stencil Optimization
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
    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
enddo
procedure rprj3(var S, R: [,,] double;
    d: array [] of direction);
begin
    S := 0.5000 * R +
        0.2500 * (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] +
        0.1250 * (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]);
As important as the size difference is that it is easier to write, read, modify, and maintain.
How did ZPL outperform MPI?

• MPI embeds specific communication idioms
  – non-blocking sends/receives
    • carries buffering & synchronization assumptions

• ZPL only expresses computation’s intent
  – permits implementation to map to puts/gets
    • RDMA is the optimal choice on this Cray T3E

• Of course, MPI could use RDMA as well…
  ...but only via modifications to the code...
  ...and what would happen on non-RDMA systems?

• Meanwhile, ZPL can target MPI on clusters…
NAS MG Speedup: Myrinet cluster

MG Class B -- Linux cluster (myrinet)

- Linear speedup
- F77+MPI
- ZPL
- HPF

Speedup over best 1-processor time (308.51 seconds in ZPL)

Processors
ZPL was also fastest at small-scale
The Fortran+MPI version...

...only supports $2^k$ problem sizes

...only supports running on $2^p$ compute nodes
  • moreover, both values must be specified at compile-time

...only supports a single 3D data decomposition

...could be rewritten to avoid these assumptions...
  ...but at what cost in terms of level of effort? code clarity?

In contrast, the ZPL version...

...supports arbitrary problem sizes and node counts
  • and permits them to be specified at run-time

...supports decomposing in 1, 2, or all 3 dimensions
NAS MG in ZPL: Summary

A resounding success for array programming!
- global-view of computation
- ability to reason about communication needs
- clear, concise, compact code
- optimized stencil computations
- system-specific communication idioms
- optimized communication (not described here)
- flexible parameterization (problem size, grid, …)
NAS MG in ZPL: Summary

But also a significant downside:

– expression of stencil required $O(points)$ code
  • OK for 27-point stencils
  • less so for, say, 216-point stencils in FMM
proc rprj3(S: ??SD?, R) {
    const Stencil = {-1..1, -1..1, -1..1},
    w = (0.5, 0.25, 0.125, 0.0625),
    w3d: [Stencil] real
        = [(i,j,k) in Stencil]
            w((i!=0) + (j!=0) + (k!=0));
    forall ijk in SD do
        S[ijk] = + reduce (for offset in Stencil do
            (w3d[offset] * R[ijk + offset*R.stride]));
}
ZPL’s Greatest Hits: Sparse Arrays in NAS CG
Sparse Arrays

• Good sparse arrays are a must for scientific computation
  – should be built-in
  – should be semantically similar to dense arrays

• NAS CG in a nutshell:
  – lots of sparse matrix-vector multiplications in a loop
Compressed Sparse Row (CSR) Format

A:

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 3 & 0 & 4 & 0 & 9 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 5 & 0 & 2 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 7 & 0 & 0 \\
\end{array}
\]

= dense data vector

+ structure information

A: 3 4 9 5 2 1 7

r: 

col: 2 4 6 3 5 2 4
Fortran Mat-Vect Multiplication

Dense Matrix-Vector Multiplication:

```fortran
integer  n
real*8   A(n,n)
real*8   t, V(n), S(n)

do  i = 1,n
   t = 0.d0
   do  j = 1,n
      t = t + A(i,j)*V(j)
   enddo
   S(i) = t
enddo
```

Sparse Matrix-Vector Multiplication:

```fortran
integer  n, nnz
real*8   A(nnz)
real*8   t, V(n), S(n)
integer  r(n+1), col(nnz)

do  i = 1,n
   t = 0.d0
   do  j = r(i),r(i+1)-1
      t = t + A(j)*V(col(j))
   enddo
   S(i) = t
enddo
```
ZPL Mat-Vect Multiplication

declarations:

region \( R = [1..n,1..n] \);
   Row = [*,1..n];
   Col = [1..n,n];

var \( A:[R] \) double;
   V:[Row] double;
   S:[Col] double;
ZPL Mat-Vect Multiplication

computation:

[Col] \( S := +<<[R] \ (A*V) ; \)

[Row] \( V := S#[n, Index1] ; \)
ZPL Mat-Vect Multiplication

computation (in parallel):

\[ \text{[Col]} \quad S := +<<[R] \quad (A*V) \; ; \]

\[ \text{[Row]} \quad V := S[n,\text{Index1}] ; \]
ZPL Mat-Vect Multiplication

Dense Matrix-Vector Multiplication:

region R = [1..n, 1..n];
    Row = [*, 1..n];
    Col = [1..n, n];

var A:[R] double;
    V:[Row] double;
    S:[Col] double;

[Col] S := +<<[R] (A*V);
**Sparse Regions and Arrays**

region Rd = [1..n,1..n];

region Rs = Rd where <pattern>;

var Ad, Bd: [Rd] double;

As, Bs: [Rs] double;
Sparse Assignments

- **Sparse assignment**
  \[
  [Rs] \text{As} := Bs;
  \]

- **Sparse assignment with dense array**
  \[
  [Rs] \text{As} := Bd;
  \]
  \[
  [Rs] \text{Ad} := Bs;
  \]

- **Dense read of sparse array**
  \[
  [Rd] \text{Ad} := Bs;
  \]

- **Dense assignment of sparse array** – illegal
  \[
  [Rd] \text{As} := Bd;
  \]
ZPL Mat-Vect Multiplication

**Dense Matrix-Vector Multiplication:**

```plaintext
region R = [1..n, 1..n];
Row = [*, 1..n];
Col = [1..n, n];

var A:[R] double;
V:[Row] double;
S:[Col] double;

[Col] S := +<<[R] (A*V);
```

**Sparse Matrix-Vector Multiplication:**

```plaintext
region R = [1..n, 1..n];
Row = [*, 1..n];
Col = [1..n, n];
Rs = R where ...

var A:[Rs] double;
V:[Row] double;
S:[Col] double;

[Col] S := +<<[Rs] (A*V);
```
Sparse Matrix-Vector Multiplication:

[Col] \( S := +<<[Rs] \ (A*V); \)

[Row] \( V := S#[n,Index1]; \)
**Sparse Arrays in ZPL**

sparse array...

\[
\begin{array}{cccccc}
\text{A:} & 0 & 0 & 0 & 0 & 0 \\
& 0 & 3 & 0 & 4 & 0 \\
& 0 & 0 & 0 & 0 & 0 \\
& 0 & 0 & 5 & 0 & 2 \\
& 0 & 1 & 0 & 0 & 0 \\
& 0 & 0 & 0 & 7 & 0 \\
0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

= 1D dense array + sparse region

\[
\begin{array}{cccccc}
\text{A:} & 0 & 3 & 4 & 9 & 5 & 2 & 1 & 7 \\
\end{array}
\]

**Diagram:**

- **general iteration**
- **“random” access**
- **Rs**
- **logical indices**
- **physical index**
General Sparse Region Format

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 3 & 0 & 4 & 0 & 9 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 5 & 0 & 2 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 7 & 0 & 0 \\
\end{array}
\]
General Sparse Region Format

4
(4, 3)
General Sparse Region Format

\[
\begin{array}{c}
\text{(2, min)} \\
\text{(4, 3)} \\
\text{(max, 2)}
\end{array}
\]
General Sparse Region Format

- \((2, \text{min})\)
- \((4, 3)\)
- \((\text{max}, 2)\)
- \((\text{min}, \text{min})\)
Sparse Format: Array of Records

- ID
- next in row
- col index
- prev in row
- next in col
- row index
- prev in col

Directory

corners
Sparse Format: Record of Arrays

- ID
- next in row
- col index
- prev in row
- next in col
- row index
- prev in col

Directory

Corners
Sparse Format: Reordered

ID
next in row
col index
prev in row
next in col
row index
prev in col

directory

... corners...
Sparse Format: Optimized

<table>
<thead>
<tr>
<th>ID</th>
<th>next in row</th>
<th>col index</th>
<th>prev in row</th>
<th>next in col</th>
<th>row index</th>
<th>prev in col</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

-6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6

directory

corners
Sparse Format: True Requirements

<table>
<thead>
<tr>
<th>ID</th>
<th>next in row</th>
<th>col index</th>
<th>prev in row</th>
<th>next in col</th>
<th>row index</th>
<th>prev in col</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

-6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6

directory

corners
Sparse Format: Fully Optimized

- ID
- next in row
- col index
- prev in row
- next in col
- row index
- prev in col

- directory

- corners
CG Memory Requirements

CG Class C -- memory usage

F77+MPI

A-ZPL

gigabytes of memory

0.31

0.31

0.32

0.31

0.31

CSR

Rs

A

P

W
CG Code Size

CG Line Counts

Productive Lines of Code

- Communication: 280
- Declarations: 146
- Computation: 198

F77+MPI

- Communication: 38
- Declarations: 37
- Computation: 0

A-ZPL

- Communication: 0
- Declarations: 0
- Computation: 0
though ZPL’s sparsity pattern performed well at small node counts and scaled well, it couldn’t beat hand-coded CSR at scale...
Sparse Arrays in MG

- MG’s input array, $V$, has 20 non-zeroes regardless of problem size
- Computes resid against $V$ twice per iteration
- Wasted time and space
MG Memory Usage

MG Class C -- memory usage

dense F77+MPI, ZPL

A-ZPL

RS
V
R
U

gigabytes of memory

1.00
1.14
1.14
1.14
1.14
NAS MG Speedup: Cray T3E

![Graph showing speedup over best 16-processor time for different algorithms (A-ZPL, ZPL, F+MPI) across varying numbers of processors (0 to 256). Linear speedup is also shown for comparison.](image-url)
Another victory for arrays!

- Sparse computation expressed like dense
- Compact, clean, comprehensible code again
  (not seen in this presentation)
- Code expresses algorithm, not implementation
  – compiler optimizes implementation based on need
ZPL Summary
ZPL’s Successes

- **First-class concept for representing index sets**
  - makes clouds of scalars in array declarations and loops concrete
  - supports global-view of data and control; improved productivity
  - useful abstraction for user and compiler


- **Semantics constraining alignment of interacting arrays**
  - communication requirements visible to user and compiler in syntax


- **Implementation-neutral expression of communication**
  - supports implementation on each architecture using best paradigm

ZPL’s Drawbacks

1) ZPL only supports one level of data parallelism
   - No task parallelism / concurrent programming
   - No nested parallelism (data or otherwise)
   - ZPL’s parallelism can never be very dynamic / unpredictable
     - single-threaded SPMD only

**Takeaway:**

*Users want more general forms of parallelism (as do modern architectures and algorithms)*

**Challenge:**

*What would it take to meaningfully support general parallel programming with ZPL-like regions and arrays?*
ZPL’s Drawbacks

2) WYSIWYG depends on having two distinct array types:
   i. parallel arrays, which you’ve been hearing about
   ii. traditional arrays which support indexing, yet are serial / local only

```cpp
var ThreeParArrs: array [1..3] of [R] double;
var ParArrofTriples: [R] array [1..3] of double;

```
ZPL’s Drawbacks

2) WYSIWYG depends on having two distinct array types:
   i. parallel arrays, which you’ve been hearing about
   ii. traditional arrays which support indexing, yet are serial / local only

```
var ThreeParArrs: array [1..3] of [R] double;
var ParArrofTriples: [R] array [1..3] of double;
```

● this breaks code reuse and results in unfortunate code clones
  ● “I need to rewrite this operation twice, once for each array type”
    ● (or $2^k$ times for $k$ array arguments?)

● there’s a real tension here: WYSIWYG would be lost if...
  ...parallel arrays supported indexing
  ...serial arrays supported array operators
2) WYSIWYG depends on having two distinct array types:
   i. parallel arrays, which you’ve been hearing about
   ii. traditional arrays which support indexing, yet are serial / local only

```haskell
var ThreeParArrs: array [1..3] of [R] double;
var ParArrofTriples: [R] array [1..3] of double;
```

**Takeaway:**

*Having distinct types for parallel vs. serial arrays is regrettable*

**Challenge:**

*How can ZPL’s region/array benefits best be carried forward?*
ZPL’s Drawbacks

3) ZPL only supports Block-distributed arrays
   ● real algorithms may want other distributions: block-cyclic, MRB, …
   ● distribution baked into the compiler, not extensible
   ● memory layouts are also hard-coded
     ● RMO for dense arrays
     ● special ZPL format for sparse arrays

Takeaway:

While block is an important common case, it’s not a panacea. Inability to get CSR sparse arrays hurt CG performance.

Challenge:

How to support more general distributions and layouts without giving up performance?
ZPL’s Drawbacks

4) Dated in several respects

- syntactically follows Modula rather than C (say)
- no support for OOP
- ...

- a good example of academic vs. practical language design

  - for academic work, such things may not matter (they didn’t for ZPL)
  - for practical adoption, they can be show-stoppers (they were for ZPL)

Takeaway:

*Users value attractive, modern, feature-rich language design.*

Challenge:

*How to keep the kitchen sink from dragging down a sleek design.*
Main ZPL Sacrifice in moving to Chapel

Q: Any guesses?

A: Chapel has no WYSIWYG model
   ● Chapel has one type of array for simplicity, code re-use
     ● parallelism is a property of operators / loops, not arrays
     ● distributed arrays support indexing
   ● Interacting arrays need not be distributed identically

```plaintext
var ThreeParArrs: [1..3] [R] real;
var ParArrofTriples: [R] [1..3] real;

// ambiguous:
[1..3][i, 1..n] ThreeParArrs = ParArrofTriples;
// unless we use a tag to distinguish:
[i in 1..3] [jk in {i, 1..n}]
    ThreeParArrs[i][jk] = ParArrOfTriples[jk][i];
// which is essentially an index…
```
Yet, Chapel users can still reason about communication
- semantic model is explicit about where data is placed / tasks execute
- execution-time queries support reasoning about locality
  - here – where is this task running?
  - `x.locale` – where is x stored?
- new `chplvis` tool supports visualization of communication
  - developed by Phil Nelson, WWU
ZPL vs. Chapel: Other Differences

**ZPL**
- data parallel only
- built-in data distributions
- designed in an evolutionary manner “what can we do well today, tomorrow?”
- academic focus: pick one problem to address (array-based data parallelism)
- single-threaded, distributed-memory execution model
- create something that matches today’s architectures

**Chapel**
- task-, nested parallelism
- user-defined data distributions
- designed in a blue-sky manner: “what would an ideal parallel language support?”
- market focus: what does a general, broad-market language require?
- multithreaded, shared address space execution model
- create something productive; unproductive architectures may suffer
Chapel
What is Chapel?

**Chapel**: An emerging parallel programming language

- portable
- open-source
- a collaborative effort
- a work-in-progress

**Goals:**

- Support general parallel programming
  - “any parallel algorithm on any parallel hardware”
- Make parallel programming far more productive
- Designed for practical adoption, not publications
The Chapel Team at Cray (May 2016)

14 full-time employees + 2 summer interns
(one of each started after photo taken)
Chapel Community R&D Efforts

(AND SEVERAL OTHERS, SOME OF WHOM YOU WILL HEAR FROM TODAY…)

http://chapel.cray.com/collaborations.html
Chapel’s Multiresolution Design: Motivation

“Why is everything so tedious/difficult?”
“Why don’t my programs trivially port to new systems?”

“Why don’t I have more control?”
**Chapel’s Multiresolution Design**

**Multiresolution Design:** Support multiple tiers of features
- higher levels for programmability, productivity
- lower levels for greater degrees of control

**Chapel language concepts**

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target Machine

- build the higher-level concepts in terms of the lower
- permit the user to intermix layers arbitrarily
Higher-Level Features

Chapel language concepts

Domain Maps
Data Parallelism
Task Parallelism
Base Language
Locality Control
Target Machine

Higher-level Chapel
Data Parallelism, by example

dataParallel.chpl

```chpl
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D do
    A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl -o dataParallel
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```
Data Parallelism, by example

config const n = 1000;
var D = {1..n, 1..n};
var A: [D] real;
forall (i, j) in D do
    A[i, j] = i + (j - 0.5)/n;
writeln(A);

prompt> chpl dataParallel.chpl -o dataParallel
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
Data Parallelism, by example

```chpl
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i, j) in D do
    A[i, j] = i + (j - 0.5)/n;
writeln(A);
```

Prompt:
```
chpl dataParallel.chpl -o dataParallel
```
```
./dataParallel --n=5
```
```
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```
Data Parallelism, by example

```chpl
config const n = 1000;
var D = {1..n, 1..n};
var A: [D] real;
forall (i, j) in D do
   A[i, j] = i + (j - 0.5)/n;
writeln(A);
```

Prompt>
chpl dataParallel.chpl -o dataParallel
Prompt>
./dataParallel --n=5

1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
Distributed Data Parallelism, by example

```
use CyclicDist;
config const n = 1000;
var D = {1..n, 1..n}
    dmapped Cyclic(startIdx = (1,1));
var A: [D] real;
forall (i,j) in D do
    A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

prompt> chpl dataParallel.chpl -o dataParallel
prompt> ./dataParallel --n=5 --numLocales=4

```
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```
Distributed Data Parallelism, by example

```
use CyclicDist;
config const n = 1000;
var D = {1..n, 1..n}
    dmapped Cyclic(startIdx = (1,1));
var A: [D] real;
forall (i,j) in D do
    A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl -o dataParallel
prompt> ./dataParallel --n=5 --numLocales=4
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```
Chapel Data Parallel Operations

- **Data Parallel Iteration**
  
  ```chapel
  forall (i,j) in D do
  A[i,j] = i + j/10.0;
  ```

- **Array Slicing; Domain Algebra**
  
  ```chapel
  A[InnerD] = B[InnerD+(0,1)];
  ```

- **Promotion of Scalar Functions and Operators**
  
  ```chapel
  A = exp(B, C);
  A = foo("hi", B, C);
  A = B + alpha * C;
  ```

- **And many others: reductions, scans, reallocation, reshaping, remapping, set operations, aliasing, ...**
Chapel has Many Types of Domains/Arrays

- dense
- strided
- sparse
- associative
- unstructured
Domain Maps

Domain maps are “recipes” that instruct the compiler how to map the global view of a computation...

\[ A = B + \alpha \cdot C; \]

...to the target locales’ memory and processors:
Chapel’s Domain Map Philosophy

1. Chapel provides a library of standard domain maps
   - to support common array implementations effortlessly

2. Expert users can write their own domain maps in Chapel
   - to cope with any shortcomings in our standard library

1. Chapel’s standard domain maps are written using the same end-user framework
   - to avoid a performance cliff between “built-in” and user-defined cases
Impacts of Chapel’s Domain Map Philosophy

● Chapel supports both global- and local-view programming
  ● domain maps form a gateway between the two worldviews

● Our approach put us at a performance disadvantage
  ● all Chapel arrays are written as Chapel code
    ● compiler has no built-in knowledge of arrays, as in ZPL, Fortran, C
    ● a distinct performance handicap compared to those languages
  
  ● this has slowed our start, but we don’t think it’s inherently problematic
    ● that said, we have not achieved parity with ZPL yet at scale
      ● (though admittedly, our focus has been different)
    ● nor have we re-implemented many key ZPL optimizations in Chapel (yet)
      ● e.g., communication optimizations, stencil optimization that helped NAS MG
Lower-Level Features

Chapel language concepts

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control

Lower-level Chapel

Target Machine
Base Language Features, by example

```plaintext
iter fib(n) {
    var current = 0,
    next = 1;

    for i in 1..n {
        yield current;
        current += next;
        current <=> next;
    }
}
```

```plaintext
for (i, f) in zip(0..#n, fib(n)) do
    writeln("fib ", i, " is ", f);
```

fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
...
Base Language Features, by example

CLU-style iterators

```plaintext
iter fib(n) {
    var current = 0,
        next = 1;

    for i in 1..n {
        yield current;
        current += next;
        current <=< next;
    }
}
```

```plaintext
for (i,f) in zip(0..#n, fib(n)) do
    writeln("fib ", i, " is ", f);
```

```
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
...
```
Base Language Features, by example

```
iter fib(n) {
  var current = 0,
    next = 1;

  for i in 1..n {
    yield current;
    current += next;
    current <=> next;
  }
}
```

```
for (i,f) in zip(0..#n, fib(n)) do
  writeln("fib ", i, " is ", f);
```

```
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
...```
Base Language Features, by example

```javascript
iter fib(n) {
    var current = 0,
        next = 1;

    for i in 1..n {
        yield current;
        current += next;
        current <=> next;
    }
}
```

```javascript
for (i, f) in zip(0..#n, fib(n)) do
    writeln("fib #", i, " is ", f);
```

- `fib #0 is 0`
- `fib #1 is 1`
- `fib #2 is 1`
- `fib #3 is 2`
- `fib #4 is 3`
- `fib #5 is 5`
- `fib #6 is 8`
- ...
Base Language Features, by example

```javascript
iter fib(n) {
  var current = 0,
      next = 1;

  for i in 1..n {
    yield current;
    current += next;
    current <=> next;
  }
}
```

```javascript
for (i, f) in zip(0..#n, fib(n)) do
  writeln("fib ", i, " is ", f);
```

fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
...

Copyright 2016 Cray Inc.
Base Language Features, by example

Static Type Inference for:
- arguments
- return types
- variables

iter fib(n) {
  var current = 0,
    next = 1;
    
  for i in 1..n {
    yield current;
    current += next;
    current //= next;
  }
}

for (i, f) in zip(0..#n, fib(n)) do
  writeln("fib ", i, " is ", f);
  
  fib #0 is 0
  fib #1 is 1
  fib #2 is 1
  fib #3 is 2
  fib #4 is 3
  fib #5 is 5
  fib #6 is 8
...
Base Language Features, by example

iter fib(n) {
    var current = 0,
        next = 1;

    for i in 1..n {
        yield current;
        current += next;
        current <=> next;
    }
}
Task Parallelism, Locality Control, by example

taskParallel.chpl

coforall loc in Locales do
  on loc {
    const numTasks = here.maxTaskPar;
    coforall tid in 1..numTasks do
      writef("Hello from task %n of %n "+
             "running on %s\n",
             tid, numTasks, here.name);
  }

prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
High-Level Task Parallelism

taskParallel.chpl

```chpl
coforall loc in Locales do
  on loc {
    const numTasks = here.maxTaskPar;
    coforall tid in 1..numTasks do
      writef("Hello from task %n of %n "+
        "running on %s\n",
        tid, numTasks, here.name);
  }
```

prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
Abstraction of System Resources

```
taskParallel.chpl

coforall loc in Locales do
  on loc {
    const numTasks = here.maxTaskPar;
    coforall tid in 1..numTasks do
      printf("Hello from task %n of %n "+
            "running on %s\n", tid, numTasks, here.name);
  }
```

```
prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```
Task Parallelism, Locality Control, by example

```chpl
coforall loc in Locales do
  on loc {
    const numTasks = here.maxTaskPar;
    coforall tid in 1..numTasks do
      printf("Hello from task %d of %d running on %s\n",
             tid, numTasks, here.name);
  }
```

```
prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```
Task Parallelism, Locality Control, by example

Abstraction of System Resources

taskParallel.chpl

coforall loc in Locales do
  on loc {
    const numTasks = here.maxTaskPar;
    coforall tid in 1..numTasks do
      printf("Hello from task \%n of \%n " +
             "running on \%s\n",
             tid, numTasks, here.name);
  }

prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
Task Parallelism, Locality Control, by example

High-Level Task Parallelism

taskParallel.chpl

```chpl
coforall loc in Locales do
  on loc {
    const numTasks = here.maxTaskPar;
    coforall tid in 1..numTasks do
      printf("Hello from task %n of %n "+
             "running on %s\n", tid, numTasks, here.name);
  }
```

prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
Task Parallelism, Locality Control, by example

```chpl
coforall loc in Locales do
  on loc {
    const numTasks = here.maxTaskPar;
    coforall tid in 1..numTasks do
      printf("Hello from task %n of %n " +
        "running on %s\n",
        tid, numTasks, here.name);
  }
```

prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032

Not seen here:
Data-centric task coordination via atomic and full/empty vars
Task Parallelism, Locality Control, by example

```
taskParallel.chpl

coforall loc in Locales do
  on loc {
    const numTasks = here.maxTaskPar;
    coforall tid in 1..numTasks do
      printf("Hello from task %d of %d running on %s\n", tid, numTasks, here.name);
  }

prompt> chpl taskParallel.chpl -o taskParallel
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```
Parallelism and Locality: Orthogonal in Chapel

- This is a **parallel**, but local program:
  ```chapel
coforall i in 1..msgs do
  writeln("Hello from task ", i);
```

- This is a **distributed**, but serial program:
  ```chapel
writeln("Hello from locale 0!");
on Locales[1] do writeln("Hello from locale 1!");
on Locales[2] do writeln("Hello from locale 2!");
```

- This is a **distributed parallel** program:
  ```chapel
coforall i in 1..msgs do
  on Locales[i%numLocales] do
  writeln("Hello from task ", i,
  " running on locale ", here.id);
```
ZPL and Chapel Resources
ZPL Resources

● Project Page:
http://research.cs.washington.edu/zpl/home/index.html

● Recommended Reading:

Chapel Websites

Project page: http://chapel.cray.com
- overview, papers, presentations, language spec, ...

GitHub: https://github.com/chapel-lang
- download Chapel; browse source repository; contribute code

Facebook: https://www.facebook.com/ChapelLanguage

Twitter: https://twitter.com/ChapelLanguage
Suggested Reading

Chapel chapter from *Programming Models for Parallel Computing*

- a detailed overview of Chapel’s history, motivating themes, features
- chapter is now also available [online](http://chapel.cray.com/papers.html)

Other Chapel papers/publications available at [http://chapel.cray.com/papers.html](http://chapel.cray.com/papers.html)
Chapel Blog Articles

- a short-and-sweet introduction to Chapel

**Chapel Springs into a Summer of Code**, Cray Blog, April 2016.
- a run-down of some current events

**Six Ways to Say “Hello” in Chapel** (parts 1, 2, 3), Cray Blog, Sep-Oct 2015.
- a series of articles illustrating the basics of parallelism and locality in Chapel

**Why Chapel?** (parts 1, 2, 3), Cray Blog, Jun-Oct 2014.
- a series of articles answering common questions about why we are pursuing Chapel in spite of the inherent challenges

- a series of technical opinion pieces designed to argue against standard reasons given for not developing high-level parallel languages
Chapel Mailing Lists

**low-traffic (read-only):**
- chapel-announce@lists.sourceforge.net: announcements about Chapel

**community lists:**
- chapel-users@lists.sourceforge.net: user-oriented discussion list
- chapel-developers@lists.sourceforge.net: developer discussions
- chapel-education@lists.sourceforge.net: educator discussions
- chapel-bugs@lists.sourceforge.net: public bug forum

*(subscribe at SourceForge: [http://sourceforge.net/p/chapel/mailman/]*)

**To contact the Cray team:**
- chapel_info@cray.com: contact the team at Cray
- chapel_bugs@cray.com: for reporting non-public bugs
Acknowledgements

Thanks to all the past members of the ZPL team for their contributions to this talk’s content, particularly my advisor Larry Snyder.

Thanks also to all the past and present members of the Chapel team for their contributions, particularly Burton Smith and David Callahan for opening the gate and leading the charge.
Questions?
Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, and URIKA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.