Scalability at Cray

Cray’s business: Allowing customers to achieve sustained high performance at large machine scales

My job: Enabling users to program our machines more productively through innovative language design

Productivity = Performance
+ Programmability
+ Portability
+ Robustness
Scalability Limiters in HPC Programming Models

- Restricted programming and execution models (e.g., SPMD)
- Exposure of low-level implementation mechanisms
- Lack of programmability: ability to _________________ code
  ...write...
  ...read...
  ...modify...
  ...tune...
  ...maintain...
  ...experiment with...
Chapel

*Chapel*: a new parallel language being developed by Cray Inc.

**Themes:**

- **general parallel programming**
  - data-, task-, and nested parallelism
  - express general levels of software parallelism
  - target general levels of hardware parallelism
- **reduce gap between mainstream & parallel languages**
- **global-view abstractions**
- **control of locality**
- **multiresolution design**
Outline

- Motivation for Chapel
  - Global-view Programming Models and Scalability
    - Language Overview
    - Wrap-up
Global-view vs. Fragmented

Problem: “Apply 3-pt stencil to vector”

\[
\begin{align*}
\text{global-view} & : ( \begin{array}{cccc}
\text{red} & \text{red} & \text{red} & \text{red} \\
\text{blue} & \text{blue} & \text{blue} & \text{blue} \\
\end{array} ) / 2 \\
+ & : ( \begin{array}{cccc}
\text{red} & \text{red} & \text{red} & \text{red} \\
\text{blue} & \text{blue} & \text{blue} & \text{blue} \\
\end{array} ) \\
= & : ( \begin{array}{cccc}
\text{red} & \text{red} & \text{red} & \text{red} \\
\text{blue} & \text{blue} & \text{blue} & \text{blue} \\
\end{array} )
\end{align*}
\]
Global-view vs. Fragmented

**Problem:** “Apply 3-pt stencil to vector”

For the global-view:

\[
\left( \begin{array}{cccc}
1 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 \\
\end{array} \right) / 2
\]

= \left( \begin{array}{cccc}
1 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 \\
\end{array} \right)

For the fragmented:

\[
\left( \begin{array}{cccc}
1 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 \\
\end{array} \right) / 2
\]

= \left( \begin{array}{cccc}
1 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 \\
\end{array} \right)

\[
\left( \begin{array}{cccc}
1 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 \\
\end{array} \right) / 2
\]

= \left( \begin{array}{cccc}
1 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 \\
\end{array} \right)

\[
\left( \begin{array}{cccc}
1 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 \\
\end{array} \right) / 2
\]

= \left( \begin{array}{cccc}
1 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 \\
\end{array} \right)
Global-view vs. SPMD Code

**Problem:** “Apply 3-pt stencil to vector”

**global-view**

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

    forall i in 2..n-1 {
        b(i) = (a(i-1) + a(i+1))/2;
    }
}
```

**SPMD**

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

    if (iHaveRightNeighbor) {
        send(right, a(locN));
        recv(right, a(locN+1));
    }
    if (iHaveLeftNeighbor) {
        send(left, a(1));
        recv(left, a(0));
    }

    forall i in 1..locN {
        b(i) = (a(i-1) + a(i+1))/2;
    }
}
```
Global-view vs. SPMD Code

Problem: “Apply 3-pt stencil to vector”

Assumes `numProcs` divides `n`; a more general version would require additional effort.

**global-view**

```chapel
def main() {
    var n: int = 1000;
    var a, b: [1..n] real;
    forall i in 2..n-1 {
        b(i) = (a(i-1) + a(i+1))/2;
    }
}
```

**SPMD**

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

    if (iHaveRightNeighbor) {
        send(right, a(locN));
        recv(right, a(locN+1));
    } else {
        innerHi = locN-1;
    }

    if (iHaveLeftNeighbor) {
        send(left, a(1));
        recv(left, a(0));
    } else {
        innerLo = 2;
    }

    forall i in innerLo..innerHi {
        b(i) = (a(i-1) + a(i+1))/2;
    }
}
```
**MPI SPMD pseudo-code**

**Problem:** “Apply 3-pt stencil to vector”

```chapel
var n: int = 1000, locN: int = n/numProcs;
var a, b: [0..locN+1] real;
var innerLo: int = 1, innerHi: int = locN;
var numProcs, myPE: int;
var retval: int;
var status: MPI_Status;

MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
MPI_Comm_rank(MPI_COMM_WORLD, &myPE);
if (myPE < numProcs-1) {
    retval = MPI_Send(&(a(locN)), 1, MPI_FLOAT, myPE+1, 0, MPI_COMM_WORLD);
    if (retval != MPI_SUCCESS) { handleError(retval); }
    retval = MPI_Recv(&(a(locN+1)), 1, MPI_FLOAT, myPE+1, 1, MPI_COMM_WORLD, &status);
    if (retval != MPI_SUCCESS) { handleErrorWithStatus(retval, status); }
} else
    innerHi = locN-1;
if (myPE > 0) {
    retval = MPI_Send(&(a(1)), 1, MPI_FLOAT, myPE-1, 1, MPI_COMM_WORLD);
    if (retval != MPI_SUCCESS) { handleError(retval); }
    retval = MPI_Recv(&(a(0)), 1, MPI_FLOAT, myPE-1, 0, MPI_COMM_WORLD, &status);
    if (retval != MPI_SUCCESS) { handleErrorWithStatus(retval, status); }
} else
    innerLo = 2;
forall i in (innerLo..innerHi) {
    b(i) = (a(i-1) + a(i+1))/2;
}
```

Communication becomes geometrically more complex for higher-dimensional arrays.
rprj3 stencil from NAS MG

\[ w_0 = w_1 = w_2 = w_3 \]
NAS MG rprj3 stencil in Fortran + MPI

```fortran
! use caf_intrinsics
! implicit none
!
include 'globals.h'
!
double precision u(n1,n2,n3)
integer n1, n2, n3, kk
include 'cafnpb.h'
use caf_intrinsics
!
subroutine rprj3(axis, dir, u, n1, n2, n3, k)
end
!
! do 1, n3-1
! ! dir = +1
! ! buff_len = 0
! ! buff_id = 1
! ! buff_len = buff_len + 1
! ! buff(buff_len, buff_id) = u(1, i2, i3)
! ! u(1,i2,i3) = buff(indx, buff_id)
! ! indx = indx + 1
! ! enddo
!
do 1, n3-1
! ! dir = -1
! ! buff_len = 0
! ! buff_id = 1
! ! buff_len = buff_len + 1
! ! buff(buff_len, buff_id) = u(1, i2, i3)
! ! u(1,i2,i3) = buff(indx, buff_id)
! ! indx = indx + 1
! ! enddo
!
if( axis .eq. 1 )then
  if( dir .eq. +1 )then
    do 1 i2=1,n2
      do 1 i1=1,n1
        buff(i,2) = buff(i,1)
        buff(i,4) = buff(i,3)
      enddo
    enddo
  !endif
  endif
endif
!
! do 1, n3-1
! ! dir = +1
! ! buff_len = 0
! ! buff_id = 3 + dir
! ! buff_len = buff_len + 1
! ! buff(buff_len, buff_id) = u(1, i2, i3)
! ! u(1,i2,i3) = buff(indx, buff_id)
! ! indx = indx + 1
! ! enddo
!
do 1, n3-1
! ! dir = -1
! ! buff_len = 0
! ! buff_id = 3 + dir
! ! buff_len = buff_len + 1
! ! buff(buff_len, buff_id) = u(1, i2, i3)
! ! u(1,i2,i3) = buff(indx, buff_id)
! ! indx = indx + 1
! ! enddo
!
if( axis .eq. 2 )then
  if( dir .eq. +1 )then
    do 1 i3=2,n3
      do 1 i2=1,n2
        buff(i,2) = buff(i,1)
        buff(i,4) = buff(i,3)
      enddo
    enddo
  !endif
  endif
endif
!
! do 1, n3-1
! ! dir = +1
! ! buff_len = 0
! ! buff_id = 3 + dir
! ! buff_len = buff_len + 1
! ! buff(buff_len, buff_id) = u(1, i2, i3)
! ! u(1,i2,i3) = buff(indx, buff_id)
! ! indx = indx + 1
! ! enddo
!
do 1, n3-1
! ! dir = -1
! ! buff_len = 0
! ! buff_id = 3 + dir
! ! buff_len = buff_len + 1
! ! buff(buff_len, buff_id) = u(1, i2, i3)
! ! u(1,i2,i3) = buff(indx, buff_id)
! ! indx = indx + 1
! ! enddo
!
if( axis .eq. 3 )then
  if( dir .eq. +1 )then
    do 1 i1=1,n1
      do 1 i3=2,n3
        buff(i,2) = buff(i,1)
        buff(i,4) = buff(i,3)
      enddo
    enddo
  !endif
  endif
endif
!
! end
```

Chapel (16)
NAS MG *rprj3* stencil in Chapel

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

    forall ijk in S.domain do
        S(ijk) = + reduce [offset in Stencil]
            (w3d(offset) * R(ijk + R.stride*offset));
}
```

- Unfortunately, Chapel is a work in progress, so I do not have scalability results to show today
- However, these Chapel features are based on our previous work in ZPL, for which I do have scalability results
NAS MG *rprj3* stencil in ZPL

```chapel
procedure rprj3(var S,R: [,,] double;
    d: array [] of direction);
begin
    S := 0.5 * R
        + 0.25 * (R@d[ 1, 0, 0] + R@d[ 0, 1, 0] + R@d[ 0, 0, 1] +
                   R@d[-1, 0, 0] + R@d[ 0,-1, 0] + R@d[ 0, 0,-1])
        + 0.125 * (R@d[ 1, 1, 0] + R@d[ 1, 0, 1] + R@d[ 0, 1, 1] +
                      R@d[ 1,-1, 0] + R@d[ 1, 0,-1] + R@d[ 0, 1,-1] +
                      R@d[-1, 1, 0] + R@d[-1, 0, 1] + R@d[ 0,-1, 1] +
                      R@d[-1,-1, 0] + R@d[-1, 0,-1] + R@d[ 0,-1,-1])
        + 0.0625 * (R@d[ 1, 1, 1] + R@d[ 1, 1,-1] +
                      R@d[ 1,-1, 1] + R@d[ 1,-1,-1] +
                      R@d[-1, 1, 1] + R@d[-1, 1,-1] +
                      R@d[-1,-1, 1] + R@d[-1,-1,-1]);
end;
```
ZPL scales better than MPI since its communication is expressed in an implementation-neutral way; this permits the compiler to use SHMEM on this Cray T3E but MPI on a commodity cluster.

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

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

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

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

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

- **F+MPI**
  - Communication: 566
  - Declarations: 202
  - Computation: 242

- **ZPL**
  - Communication: 87
  - Declarations: 70
Code Size Notes

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

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

F+MPI: 242
- communication: 202
- declarations: 242
- computation: 566

ZPL: 70
- communication: 87
- declarations: 70
- computation: 0
Summarizing Fragmented/SPMD Models

- **Advantages:**
  - fairly straightforward model of execution
  - relatively easy to implement
  - reasonable performance on commodity architectures
  - portable/ubiquitous
  - lots of important scientific work has been accomplished with them

- **Disadvantages:**
  - blunt means of expressing parallelism: cooperating executables
  - fails to abstract away architecture / implementing mechanisms
  - obfuscates algorithms with many low-level details
    - error-prone
    - brittle code: difficult to read, maintain, modify, *experiment*
    - “MPI: the assembly language of parallel computing”
Current HPC Programming Notations

- **communication libraries:**
  - MPI, MPI-2
  - SHMEM, ARMCI, GASNet

- **shared memory models:**
  - OpenMP, pthreads

- **PGAS languages:**
  - Co-Array Fortran
  - UPC
  - Titanium

Chapel (24)
Scalability Limiters in HPC Programming Models

- **Restricted programming and execution models**
  - limits applicability to multiple levels of parallelism in HW and SW

- **Exposure of low-level implementation mechanisms**
  - exposes too much about implementation semantics
    - *e.g.*, saying “how” data should be communicated rather than simply “what” and possibly “when”
  - binds language too tightly to a particular implementation in hardware
    - *e.g.*, MPI for inter-node parallelism + OpenMP/pthreads for inter-core parallelism + directives for intra-core parallelism

- **Lack of programmability**
  - to a large degree, resulting from the previous two bullets
  - limits ability to modify code – a necessity to tune for these scales
Outline

✓ Motivation for Chapel
✓ Global-view Programming Models and Scalability

➤ Language Overview
  ➤ Base Language
  ➤ Parallel Features
    ➤ Task Parallel
    ➤ Data Parallel
  ➤ Locality Features

☐ Wrap-up
Chapel Design

- Block-structured, imperative programming
- Intentionally not an extension to an existing language
- Instead, select attractive features from others:
  - **ZPL, HPF**: data parallelism, index sets, distributed arrays
    (see also APL, NESL, Fortran90)
  - **Cray MTA C/Fortran**: task parallelism, lightweight synchronization
  - **CLU**: iterators (see also Ruby, Python, C#)
  - **ML**: latent types (see also Scala, Matlab, Perl, Python, C#)
  - **Java, C#**: OOP, type safety
  - **C++**: generic programming/templates (without adopting its syntax)
  - **C, Modula, Ada**: syntax
Base Language: Overview

- **Syntax**
  - adopt C family of syntax whenever possible/useful
  - main departures: declarations/casts, generics, for loops

- **Language Elements**
  - standard scalar types, expressions, statements
  - value- and reference-based OOP (optional)
  - no pointers, restricted opportunities for aliasing
  - argument intents similar to Fortran/Ada

- **My favorite base language features**
  - iterators (in the CLU/Ruby sense, not C++/Java)
  - tuples
  - latent types / simple static type inference
  - rich compile-time language
  - configuration variables
Task Parallelism: Task Creation

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

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

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

```chapel
sync {
    begin recursiveTreeSearch(root);
}
```
Task Parallelism: Task Coordination

**sync variables:** store full/empty state along with value

```chapel
var result$: sync real; // result is initially empty
sync {
    begin ... = result$; // block until full, leave empty
    begin result$ = ...; // block until empty, leave full
}
result$.readFF(); // read when full, leave full;
// other variations also supported
```

**single-assignment variables:** writable once only

```chapel
var result$: single real = begin f(); // result initially empty
...
// do some other things
total += result$; // block until f() has completed
```

**atomic sections:** support transactions against memory

```chapel
atomic {
    newnode.next = insertpt;
    newnode.prev = insertpt.prev;
    insertpt.prev.next = newnode;
    insertpt.prev = newnode;
}
```
Task Parallelism: Structured Task Creation

**cobegin**: creates a task per component statement:

```chapel
cobegin:
creates a task per component statement:

  computePivot(lo, hi, data);
  cobegin {
    Quicksort(lo, pivot, data);
    Quicksort(pivot, hi, data);
  } // implicit join here

  cobegin {
    computeTaskA(...);
    computeTaskB(...);
    computeTaskC(...);
  } // implicit join
```

**coforall**: creates a task per loop iteration

```chapel
coforall e in Edges {
  exploreEdge(e);
} // implicit join here
```
Data Parallelism: Domains

**domains:** first-class index sets, whose indices can be...

...integer tuples...

...anonymous...  ...or arbitrary values.

```
DnsDom  (10,24)
StrDom  (10,24)
SpsDom  (10,24)
GrphDom
NameDom
```

Chapel (36)
Data Parallelism: Domain Declarations

\textbf{var} DnsDom: \texttt{domain}(2) = [1..10, 0..24],
StrDom: \texttt{subdomain}(\texttt{DnsDom}) = \texttt{DnsDom \ by \ (2,4)},
SpsDom: \texttt{subdomain}(\texttt{DnsDom}) = \texttt{genIndices}();

\textbf{var} GrphDom: \texttt{domain}(\texttt{opaque}),
NameDom: \texttt{domain}(\texttt{string}) = \texttt{readNames}();
Data Parallelism: Domains and Arrays

Domains are used to declare arrays...

```chapel
var DnsArr: [DnsDom] complex,
SpsArr: [SpsDom] real;
...```

“steve”
“mary”
“wayne”
“david”
“john”
“pete”
“peg”
Data Parallelism: Domain Iteration

…to iterate over index spaces…

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

“steve”  “mary”  “wayne”  “david”  “john”  “pete”  “peg”
Data Parallelism: Array Slicing

...to slice arrays...

\[ \text{DnsArr}[\text{StrDom}] += \text{SpsArr}[\text{StrDom}]; \]
Data Parallelism: Array Reallocation

...and to reallocate arrays

$$\text{StrDom} = \text{DnsDom} \text{ by } (2,2);$$
$$\text{SpsDom } += \text{ genEquator}();$$
Locality: Locales

- **locale**: architectural unit of locality
  - has capacity for storage and processing
  - threads within a locale have ~uniform access to local memory
  - memory within other locales is accessible, but at a price
  - *e.g.*, a multicore processor or SMP node
Locality: Locales

- user specifies # locales on executable command-line
  
prompt> myChapelProg –nl=8

- Chapel programs have built-in locale variables:

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

- Programmers can create their own locale views:

  ```
  var CompGrid = Locales.reshape([1..GridRows, 1..GridCols]);
  ```

  ```
  var TaskALocs = Locales[..numTaskALocs];
  var TaskBLocs = Locales[numTaskALocs+1..];
  ```
Locality: Task Placement

on clauses: indicate where tasks should execute

Either in a data-driven manner...

\[
\text{computePivot(lo, hi, data); cobegin}
\]
\[
\begin{array}{c}
\text{on A(lo) do Quicksort(lo, pivot, data);}
\text{on A(pivot) do Quicksort(pivot, hi, data);}
\end{array}
\]

…or by naming locales explicitly

\[
\text{cobegin}
\]
\[
\begin{array}{c}
\text{on Locales(0) do producer();}
\text{on Locales(1) do consumer();}
\end{array}
\]

\[
\text{cobegin}
\]
\[
\begin{array}{c}
\text{on TaskALocs do computeTaskA(…);}
\text{on TaskBLocs do computeTaskB(…);}
\text{on Locales(0) do computeTaskC(…);}
\end{array}
\]
Locality: Domain Distribution

Domains may be distributed across locales

```
var D: domain(2) distributed Block on CompGrid = ...;
```
Locality: Domain Distributions

Distributions specify…

…a mapping of indices to locales

…per-locale storage for domain indices and array elements

…the implementation of parallel operations on domains/arrays
Locality: Domain Distributions

Distributions specify...
...a mapping of indices to locales
...per-locale storage for domain indices and array elements
...the implementation of parallel operations on domains/arrays
Locality: Distributions Overview

**Distributions**: “recipes for distributed arrays”

- Intuitively, distributions support the lowering…
  ...from: the user’s global view of a distributed array
  ...to: the fragmented implementation on a distributed memory machine

- Users can implement custom distributions:
  - written using task parallel features, on clauses, domains/arrays
  - must implement standard interface:
    - **allocation/reallocation** of domain indices and array elements
    - **mapping functions** (e.g., index-to-locale, index-to-value)
    - **iterators**: parallel/serial × global/local
    - optionally, communication idioms

- Chapel provides a standard library of distributions…
  …written using the same mechanism as user-defined distributions
  …tuned for different platforms to maximize performance
Multiresolution Language Design

Conventional Wisdom: By providing higher-level concepts in a language, programmers’ hands are tied, preventing them from manually optimizing for performance.

My Belief: With appropriate design, this need not be the case.
- provide high-level features and automation for convenience
  ▪ knowledge of such features can aid in compiler optimization
- provide capabilities to drop down to lower, more manual levels
- use appropriate separation of concerns to keep this clean
  ▪ support the 90/10 rule

<table>
<thead>
<tr>
<th>Distributions</th>
<th>Domains / Arrays</th>
<th>Task Parallelism</th>
<th>Base Language</th>
<th>Locality Control</th>
<th>Target Machine</th>
</tr>
</thead>
</table>
Outline

✓ Motivation for Chapel
✓ Global-view Programming Models and Scalability
✓ Language Overview

☐ Wrap-up
Chapel Work

- Chapel Team’s Focus:
  - specify Chapel syntax and semantics
  - implement prototype compiler for Chapel
  - support users of preliminary releases
  - code studies of benchmarks, applications, and libraries in Chapel
  - community outreach to inform and learn from users/researchers
  - refine language based on all these activities
Prototype Implementation

- **Approach:**
  - source-to-source compiler for portability (Chapel-to-C)
  - link against runtime libraries to hide machine details
    - threading layer currently implemented using pthreads
    - communication currently implemented using Berkeley’s GASNet

- **Status:**
  - **base language:** solid, usable (a few gaps remain)
  - **task parallel:** multiple threads, multiple locales
  - **data parallel:** single-threaded, single-locale
  - **performance:** has received little effort (but much planning)

- **Current Focus:**
  - multi-threaded implementation of data parallel features
  - distributed domains and arrays
  - performance optimizations

- **Early releases to ~40 users at ~20 sites** (academic, gov’t, industry)
# HPC vs. Datacenter concerns

<table>
<thead>
<tr>
<th>Concern</th>
<th>HPC</th>
<th>Datacenter*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalability</td>
<td>Crucial</td>
<td>Crucial</td>
</tr>
<tr>
<td>Locality/affinity</td>
<td>Crucial for performance</td>
<td>Crucial for performance</td>
</tr>
<tr>
<td>Portability to future technologies</td>
<td>Important</td>
<td>Important</td>
</tr>
<tr>
<td>Power/cooling</td>
<td>Increasingly an issue</td>
<td>Increasingly an issue</td>
</tr>
<tr>
<td>Flavor of Parallelism</td>
<td>Lots of data dependence, communication</td>
<td>Pleasingly parallel + reductions</td>
</tr>
<tr>
<td>Reliability/robustness</td>
<td>Checkpoint/restart</td>
<td>Redundancy + dynamic monitoring</td>
</tr>
<tr>
<td>Data types</td>
<td>Floating point (historically)</td>
<td>Strings, integers</td>
</tr>
<tr>
<td>Data structures</td>
<td>Multidimensional arrays / unstructured graphs</td>
<td>???</td>
</tr>
<tr>
<td>Memory Use</td>
<td>Lots (but generally in-core)</td>
<td>Lots (&amp; generally out-of-core)</td>
</tr>
</tbody>
</table>

* Based on my rather limited understanding…

Chapel (53)
Chapel for Datacenter Computations?

Some appropriate features:
- large, distributed data structures ("arrays")
- application of scalar functions to arrays
- reductions: standard and user-defined
- ability to reason about locality, machine resources
- abstraction away from implementing mechanisms
- designed for generality

Yet also some areas requiring innovation/research:
- language-level support for redundancy/reliability?
- extend domains and distributions to out-of-core computations?
- interpreted Chapel for interactive data exploration?
- your ideas here…

A potentially interesting collaboration?
(We’re open to others as well…)
Summary

*Programming languages can help with scalability, given appropriate design and abstractions*

- Abstractions must map well to hardware capabilities
  - capable of resulting in good performance
  - avoid encoding more about hardware than necessary
- Should support ability to drop to lower levels when required
- Should support ability to control and reason about locality
- Must be as general-purpose as target community requires
Chapel Team

Steve Deitz, Brad Chamberlain
David Iten, Samuel Figueroa, Mary Beth Hribar
Questions?

bradc@cray.com
chapel_info@cray.com

http://chapel.cs.washington.edu