Chapel: Background
• **HPCS**: High Productivity Computing Systems (DARPA)
  • Goal: Raise HEC user productivity by 10x
    \[ \text{Productivity} = \text{Performance} + \text{Programmability} + \text{Portability} + \text{Robustness} \]
  • Phase II: Cray, IBM, Sun (July 2003 – June 2006)
    • Evaluated entire system architecture
    • Three new languages (Chapel, X10, Fortress)
  • 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
Outline

- Chapel’s Settings and Goals
- Chapel’s Themes
  - Global-view abstractions
  - General parallel programming
  - Multiple levels of design
  - Control of locality
  - Mainstream language features
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

Fragmented

Chapel: Background
Global-View Abstractions

Example: 3-Point Stencil (Computation)

\[
\begin{align*}
\text{Global-View} & \quad \text{Fragmented} \\
( \quad & \quad ( \\
+ & + \\
= & = \\
\frac{1}{2} & \frac{1}{2} \\
\frac{1}{2} & \frac{1}{2} \\
\frac{1}{2} & \frac{1}{2} \\
\frac{1}{2} & \frac{1}{2} \\
\end{align*}
\]
Global-View Abstractions

Example: 3-Point Stencil (Code)

**Global-View**

```chapel
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**

```chapel
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;
}
```

Assumes p divides n
NAS MG Stencil
use caf_intrinsics

implicit none

include 'cafgh.h'
include 'global.h'

integer n1, n2, n3, k,
double precision x(n1,n2,n3)

integer axis
if( .not. dead(kk) ) then
  if( axis .eq. 3 ) then
    call give3( axis, dir, u, n1, n2, n3 )
  else
    call take3( axis, u, n1, n2, n3 )
  endif
endif
!

do  i1=1,n1
  do  i2=2,n2
    do  i3=2,n3
      buff( i, 4 ) = buff( i, 3 )
    enddo
  enddo
enddo
!

do  i1=1,n1
  do  i2=2,n2
    do  i3=2,n3
      indx = i1,i2,i3
      buff_len = 0
      buff_id = 0
    enddo
  enddo
enddo
!

do  i1=2,n1
  do  i2=2,n2
    do  i3=2,n3
      indx = i1,i2,i3
      buff_len = 0
      buff_id = 0
    enddo
  enddo
enddo
!

do  i1=1,n1
  do  i2=2,n2
    do  i3=1,n3
      indx = i1,i2,i3
      buff_len = 0
      buff_id = 0
    enddo
  enddo
enddo
!

if( .not. dead(1) ) then
  call give( axis, u, n1, n2, n3 )
endif
!

do  i1=1,n1
  do  i2=2,n2
    do  i3=2,n3
      indx = i1,i2,i3
      buff_len = 0
      buff_id = 0
    enddo
  enddo
enddo
!

do  i1=2,n1
  do  i2=2,n2
    do  i3=2,n3
      indx = i1,i2,i3
      buff_len = 0
      buff_id = 0
    enddo
  enddo
enddo
!

do  i1=1,n1
  do  i2=2,n2
    do  i3=1,n3
      indx = i1,i2,i3
      buff_len = 0
      buff_id = 0
    enddo
  enddo
enddo
!

if( .not. dead(2) ) then
  call take2( axis, u, n1, n2, n3 )
endif
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

<table>
<thead>
<tr>
<th>Communication Libraries</th>
<th>Data Model</th>
<th>Compute Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI/MPI-2</td>
<td>Fragmented</td>
<td>Fragmented</td>
</tr>
<tr>
<td>SHMEM</td>
<td>Fragmented</td>
<td>Fragmented</td>
</tr>
<tr>
<td>ARMCI</td>
<td>Fragmented</td>
<td>Fragmented</td>
</tr>
<tr>
<td>GASNet</td>
<td>Fragmented</td>
<td>Fragmented</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shared Memory</th>
<th>Communication Libraries</th>
<th>Data Model</th>
<th>Compute Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP, pThreads</td>
<td>MPI/MPI-2</td>
<td>Global-View (trivially)</td>
<td>Global-View (trivially)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PGAS Languages</th>
<th>Communication Libraries</th>
<th>Data Model</th>
<th>Compute Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Co-Array Fortran</td>
<td>MPI/MPI-2</td>
<td>Fragmented</td>
<td>Fragmented</td>
</tr>
<tr>
<td>UPC</td>
<td>MPI/MPI-2</td>
<td>Global-View</td>
<td>Fragmented</td>
</tr>
<tr>
<td>Titanium</td>
<td>MPI/MPI-2</td>
<td>Fragmented</td>
<td>Fragmented</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HPCS Languages</th>
<th>Communication Libraries</th>
<th>Data Model</th>
<th>Compute Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapel</td>
<td>MPI/MPI-2</td>
<td>Global-View</td>
<td>Global-View</td>
</tr>
<tr>
<td>X10 (IBM)</td>
<td>MPI/MPI-2</td>
<td>Global-View</td>
<td>Global-View</td>
</tr>
<tr>
<td>Fortress (Sun)</td>
<td>MPI/MPI-2</td>
<td>Global-View</td>
<td>Global-View</td>
</tr>
</tbody>
</table>
General Parallel Programming

- Express all parallelism in the software
  - Forms: data, task, nested (arbitrary composition thereof)
  - Levels: module, function, loop, statement

- Target all parallelism in the hardware
  - Systems: multicore desktops, clusters, HPC systems
  - Types: multithreading, vector
  - Levels: across cores, across nodes, across systems
Multiple Levels of Design

“Why is everything so difficult?”

“Why can’t I optimize this?”

Low-Level Implementing Mechanisms
- MPI
- OpenMP
- pThreads

Target Machine

High-Level Abstractions
- HPF
- ZPL

Target Machine
Structure the language in layers, permitting it to be used at multiple levels as required/desired

- support high-level features and automation for convenience
- provide the ability to drop down to lower, more manual levels

Language concepts:
- Domain Maps
- Data parallelism
- Task Parallelism
- Locality Control
- Base Language
- Target Machine
Control of Locality

Given
- Scalable systems tend to store memory with processors
- Remote accesses tend to take longer than local accesses

Therefore
- Placement of data relative to computation matters
- Programmers need control over data placement

Note
- As multi-core chips grow, locality matters on desktops
- GPUs/accelerators expose node-level locality
Mainstream Language Features

- Object-oriented programming with value and reference classes
- Generic programming with types and compile-time constants
- Latent typing and a rich set of primitive types
- Modules for libraries and code organization
- Functions with nesting, overloading, and named arguments
- Multi-dimensional and associative arrays with slicing, etc.
- Classes, records, and unions
- Tuples, ranges, and domains
- Standard modules (e.g., Math, Random, Time, BitOps, Norm)
Questions?

- Chapel’s Settings and Goals
- Chapel’s Design
  - Global-view abstractions
  - General parallel programming
  - Multiple levels of design
  - Control of locality
  - Mainstream language features