Chapel: Background
HPCS: High Productivity Computing Systems

- Overall goal: Raise high-end user productivity by 10x

\[ \text{Productivity} = \text{Performance} + \text{Programmability} + \text{Portability} + \text{Robustness} \]

Phase II: Cray, IBM, Sun (July 2003 – June 2006)

- Goal: Propose new productive system architectures
- Each vendor created a new programming language
  - Cray: Chapel
  - IBM: X10
  - Sun: Fortress

Phase III: Cray, IBM (July 2006 – )

- Goal: Develop the systems proposed in phase II
- Each vendor implemented a compiler for their language
  - Sun also continued their Fortress effort without HPCS funding
Chapel's Productivity Goals

- Vastly improve **programmability** over current languages
  - Writing parallel programs
  - Reading, modifying, porting, tuning, maintaining them

- 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 but more abstract
  - More portable than OpenMP, UPC, and CAF are thought to be

- Improve **robustness** via improved semantics
  - Eliminate common error cases
  - Provide better abstractions to help avoid other errors
Outline

- Chapel’s Context
- Chapel’s Motivating Themes
  1. General parallel programming
  2. *Global-view* abstractions
  3. *Multiresolution* design
  4. Control over locality/affinity
  5. Reduce gap between mainstream & HPC languages
1) General Parallel Programming

With a unified set of concepts...

...express any parallelism desired in a user’s program

- **Styles**: data-parallel, task-parallel, concurrency, nested, ...
- **Levels**: model, function, loop, statement, expression

...target all parallelism available in the hardware

- **Systems**: multicore desktops, clusters, HPC systems, ...
- **Levels**: machines, nodes, cores, instructions
2) Global-View Abstractions

In pictures: “Apply a 3-Point Stencil to a vector”

\[
\text{Global-View} \quad \begin{array}{c}
\text{\( ( \quad \quad \quad )/2 \)} \\
\text{\( + \quad \quad \quad \)} \\
\text{\( = \quad \quad \quad \)}
\end{array}
\]

\[
\text{Local-View} \quad \begin{array}{c}
\text{\( \quad \quad \quad \quad \quad \quad \quad \quad \)} \\
\text{\( \quad \quad \quad \quad \quad \quad \quad \quad \)} \\
\text{\( \quad \quad \quad \quad \quad \quad \quad \quad \)}
\end{array}
\]
2) Global-View Abstractions

In pictures: “Apply a 3-Point Stencil to a vector”

Global-View

\[
\begin{align*}
( & \quad \quad & \quad \quad & \quad \quad )/2 \\
+ & \quad \quad & \quad \quad & \quad \quad )/2 \\
= & \quad \quad & \quad \quad & \quad \quad \\
\end{align*}
\]

Local-View

\[
\begin{align*}
( & \quad \quad & \quad \quad & \quad \quad )/2 \\
+ & \quad \quad & \quad \quad & \quad \quad )/2 \\
= & \quad \quad & \quad \quad & \quad \quad \\
\end{align*}
\]

\[
\begin{align*}
( & \quad \quad & \quad \quad & \quad \quad )/2 \\
+ & \quad \quad & \quad \quad & \quad \quad )/2 \\
= & \quad \quad & \quad \quad & \quad \quad \\
\end{align*}
\]
2) Global-View Abstractions

In code: “Apply a 3-Point Stencil to a vector”

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;
}
```

Local-View (SPMD)

```chapel
def main() {
    var n = 1000;
    var p = numProcs(),
        me = myProc(),
        myN = n/p,
    var A, B: [0..myN+1] real;

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

Bug: Refers to uninitialized values at ends of A
2) Global-View Abstractions

In code: “Apply a 3-Point Stencil to a vector”

```chapel
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;
}
```

Local-View (SPMD)

```chapel
def main() {
    var n = 1000;
    var p = numProcs(),
        me = myProc(),
        myN = n/p,
        iLo = 1,
        iHi = myN;
    var A, B: [0..myN+1] real;

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

Communication becomes geometrically more complex for higher-dimensional arrays

Assumes p divides n
2) rprj3 Stencil from NAS MG

\[ = w_0 \]
\[ = w_1 \]
\[ = w_2 \]
\[ = w_3 \]
double precision u(n1,n2,n3);

include 'mpinpb.h'

integer n1, n2, n3,

double precision u(i1,i2,i3);

include 'globals.h'

integer i1, i2, i3,
2) *rprj3* Stencil from NAS MG in Chapel

```chapel
def rprj3(S: [?SD], R: [?RD]) {
    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 ijk in SD do
        S[ijk] = + reduce [offset in Stencil]
            (W3D[offset] * R[ijk + RD.stride*offset]);
}
```

Our previous work in ZPL demonstrated that such compact codes can result in better performance than Fortran + MPI while also supporting more flexibility at runtime*.

*e.g., the Fortran + MPI *rprj3* code shown previously not only assumes *p* divides *n*, it also assumes that *p* and *n* are specified at compile-time and powers of two.
## 2) Classifying Current Programming Models

<table>
<thead>
<tr>
<th>System</th>
<th>Data Model</th>
<th>Control Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Communication Libraries</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI/MPI-2</td>
<td>Local-View</td>
<td>Local-View</td>
</tr>
<tr>
<td>SHMEM, ARMCI, GASNet</td>
<td>Local-View</td>
<td>SPMD</td>
</tr>
<tr>
<td><strong>Shared Memory</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenMP, Pthreads</td>
<td><strong>Global-View (trivially)</strong></td>
<td><strong>Global-View (trivially)</strong></td>
</tr>
<tr>
<td><strong>PGAS Languages</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Co-Array Fortran</td>
<td>Local-View</td>
<td>SPMD</td>
</tr>
<tr>
<td>UPC</td>
<td>Global-View</td>
<td>SPMD</td>
</tr>
<tr>
<td>Titanium</td>
<td>Local-View</td>
<td>SPMD</td>
</tr>
<tr>
<td><strong>PGAS Libraries</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global Arrays</td>
<td>Global-View</td>
<td>SPMD</td>
</tr>
</tbody>
</table>
### 2) Classifying Current Programming Models

<table>
<thead>
<tr>
<th>System</th>
<th>Data Model</th>
<th>Control Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Communication Libraries</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI/MPI-2</td>
<td>Local-View</td>
<td>Local-View</td>
</tr>
<tr>
<td>SHMEM, ARMCI, GASNet</td>
<td>Local-View</td>
<td>SPMD</td>
</tr>
<tr>
<td><strong>Shared Memory</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenMP, Pthreads</td>
<td>Global-View (trivially)</td>
<td>Global-View (trivially)</td>
</tr>
<tr>
<td><strong>PGAS Languages</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Co-Array Fortran</td>
<td>Local-View</td>
<td>SPMD</td>
</tr>
<tr>
<td>UPC</td>
<td>Global-View</td>
<td>SPMD</td>
</tr>
<tr>
<td>Titanium</td>
<td>Local-View</td>
<td>SPMD</td>
</tr>
<tr>
<td><strong>PGAS Libraries</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global Arrays</td>
<td>Global-View</td>
<td>SPMD</td>
</tr>
<tr>
<td><strong>HPCS Languages</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapel</td>
<td>Global-View</td>
<td>Global-View</td>
</tr>
<tr>
<td>X10 (IBM)</td>
<td>Global-View</td>
<td>Global-View</td>
</tr>
<tr>
<td>Fortress (Sun)</td>
<td>Global-View</td>
<td>Global-View</td>
</tr>
</tbody>
</table>
2) Global-View Programming: A Final Note

- A language may support both global- and local-view programming — in particular, Chapel does

```python
def main() {
    coforall loc in Locales do
        on loc do
            MySPMDProgram(loc.id, Locales.numElements);
}

def MySPMDProgram(me, p) {
    ...
}
```
3) Multiresolution Language Design: Motivation

"Why is everything so difficult?"
"Why don’t I have more control?"
"Why don’t my programs port trivially?"
**Multiresolution Design:** Support multiple tiers of features

- higher levels for programmability, productivity
- lower levels for performance, control
- build the higher-level concepts in terms of the lower-

*Chapel language concepts*

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

- separate concerns appropriately for clean design
Consider:

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

Therefore:

- Placement of data relative to computation affects scalability
- Programmers need control over data and task placement

Note:

- As core counts grow, locality will matter more on desktops
- GPUs and accelerators already expose node-level locality
Consider:
- Students graduate with training in Java, Matlab, Perl, Python
- Yet HPC programming is dominated by Fortran, C/C++, MPI

We’d like to narrow this gulf with Chapel:
- to leverage advances in modern language design
- to better utilize the skills of the entry-level workforce...
- ...while not ostracizing the traditional HPC programmer
  - e.g., support object-oriented programming, but make it optional

Other examples:
- function overloading, name-based argument passing
- scripting-like features: type inference, generic functions
- rich data structures with iterators (e.g., associative arrays)
Questions?

- Chapel’s Context
- Chapel’s Motivating Themes
  1. General parallel programming
  2. *Global-view* abstractions
  3. *Multiresolution* design
  4. Control over locality/affinity
  5. Reduce gap between mainstream & HPC languages