What is Chapel?

- A new parallel language being developed by Cray Inc.
- Part of Cray’s entry in DARPA’s HPCS program

**Main Goal:** Improve programmer productivity
- Improve the **programmability** of parallel computers
- Match or beat the **performance** of current programming models
- Provide better **portability** than current programming models
- Improve **robustness** of parallel codes

**Target architectures:**
- multicore desktop machines
- clusters of commodity processors
- Cray architectures
- systems from other vendors

- A work in progress
Chapel’s Setting: HPCS

**HPCS:** High *Productivity* Computing Systems (DARPA *et al.*)

- **Goal:** Raise productivity of high-end computing users by 10×
- **Productivity** = Performance
  + Programmability
  + Portability
  + Robustness

- **Phase II:** Cray, IBM, Sun (July 2003 – June 2006)
  - Evaluated the entire system architecture’s impact on productivity…
    - processors, memory, network, I/O, OS, runtime, compilers, tools, …
  - …and new languages:
    - Cray: Chapel
    - IBM: X10
    - Sun: Fortress

- **Phase III:** Cray, IBM (July 2006 – )
  - Implement the systems and technologies resulting from phase II
  - (Sun also continues work on Fortress, without HPCS funding)
Chapel: Motivating Themes

1) general parallel programming
2) *global-view* abstractions
3) *multiresolution* design
4) control of locality/affinity
5) reduce gap between mainstream & parallel languages
1) General Parallel Programming

- **General software parallelism**
  - *Algorithms*: should be able to express any that come to mind
    - should never hit a limitation requiring the user to return to MPI
  - *Styles*: data-parallel, task-parallel, concurrent algorithms
    - as well as the ability to compose these naturally
  - *Levels*: module-level, function-level, loop-level, statement-level, ...

- **General hardware parallelism**
  - *Types*: multicore desktops, clusters, HPC systems, ...
  - *Levels*: inter-machine, inter-node, inter-core, vectors, multithreading
2) Global-view vs. Fragmented

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

\[
\text{global-view} = \left( \begin{array}{cccc}
\text{\textcolor{blue}{1} & \textcolor{blue}{1} & \textcolor{blue}{1}} \\
\text{\textcolor{blue}{1} & \textcolor{blue}{1} & \textcolor{blue}{1}} \\
\end{array} \right) / 2
\]

\[
\begin{array}{cc}
\text{global-view} & \text{fragmented} \\
\text{\textcolor{yellow}{1} & \textcolor{yellow}{1} & \textcolor{yellow}{1}} & \text{\textcolor{yellow}{1} & \textcolor{yellow}{1} & \textcolor{yellow}{1}} \\
\end{array}
\]
2) Global-view vs. Fragmented

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

\[
\text{global-view} = \frac{1}{2} \left( \begin{array}{cccc}
\text{blue} & \text{blue} & \text{blue} & \text{blue} \\
\text{blue} & \text{blue} & \text{blue} & \text{blue} \\
\text{blue} & \text{blue} & \text{blue} & \text{blue} \\
\text{blue} & \text{blue} & \text{blue} & \text{blue}
\end{array} \right)
\]

\[
\text{fragmented} = \frac{1}{2} \left( \begin{array}{cccc}
\text{yellow} & \text{yellow} & \text{yellow} & \text{yellow} \\
\text{yellow} & \text{yellow} & \text{yellow} & \text{yellow} \\
\text{yellow} & \text{yellow} & \text{yellow} & \text{yellow} \\
\text{yellow} & \text{yellow} & \text{yellow} & \text{yellow}
\end{array} \right)
\]
2) Global-view vs. SPMD Code

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

**Global-view**

```python
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;

    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;
    }
}
```
2) Global-view vs. SPMD Code

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

---

**Global-view**

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

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;
    }
}
```
2) SPMD pseudo-code + MPI

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

```
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;
}
```

SPMD (pseudocode + MPI)

Communication becomes geometrically more complex for higher-dimensional arrays.
2) *rprj3* stencil from NAS MG
2) NAS MG rprj3 stencil in Fortran + MPI

subroutine comm3(u,n1,n2,n3,k)
    use caf_intrinsics
    implicit none
    include 'globals.h'
    integer axis, dir, n1, n2, n3, k, kk, indx
    integer buff_id, indx
    double precision u( n1, n2, n3 )
    do  i1=1,n1
        do  i2=1,n2
            do  i3=2,n3
                buff(i,buff_id) = 0.0D0
            enddo
        enddo
    enddo
    buff_len = 0
    do  i=1,nm2
        buff(i,2) = buff(i,1)
        buff(i,4) = buff(i,3)
    enddo
    do  i3=2,n3
        buff(i,buff_id) = buff(i,buff_id)
    enddo
    buff(buff_len, buff_id ) = u( 2, 1,i3)
    buff_len = buff_len + 1
    do  i1=1,n1
        do  i3=2,n3
            buff(i1,i2,1) = buff(i1,1,i3) + x1(i1+1) + y2)
            buff(i1,i2,i3) = buff(i1,i2,i3) + r(i1, i2+1,i3  ) + r(i1, i2+1,i3  )
            buff(i1,i2,i3) = buff(i1,i2,i3) + x2(i1,i2,i3)
    enddo
    if( axis .eq. 3 )then
        return
    endif
    end
end

end subroutine comm3(u,n1,n2,n3,k)

end subroutine rprj3(r,m1k,m2k,m3k,x1,y1,y2)
2) NAS MG `rprj3` stencil in Chapel

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

Our previous work in ZPL showed that compact, global-view codes like these can result in performance that matches or beats hand-coded Fortran+MPI while also supporting more runtime flexibility.
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;
NAS MG Speedup: ZPL vs. Fortran + MPI

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.
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

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

• the ZPL codes are 5.5–6.5x 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
Global-view models can benefit Productivity

- more programmable, flexible
- able to achieve competitive performance
- more portable; leave low-level details to the compiler
<table>
<thead>
<tr>
<th><strong>2) Classifying HPC Programming Notations</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>communication libraries:</strong></td>
</tr>
<tr>
<td>- MPI, MPI-2</td>
</tr>
<tr>
<td>- SHMEM, ARMCI, GASNet</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>shared memory models:</strong></td>
</tr>
<tr>
<td>- OpenMP, pthreads</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>PGAS languages:</strong></td>
</tr>
<tr>
<td>- Co-Array Fortran</td>
</tr>
<tr>
<td>- UPC</td>
</tr>
<tr>
<td>- Titanium</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>HPCS languages:</strong></td>
</tr>
<tr>
<td>- Chapel</td>
</tr>
<tr>
<td>- X10 (IBM)</td>
</tr>
<tr>
<td>- Fortress (Sun)</td>
</tr>
</tbody>
</table>
3) Multiresolution Languages: Motivation

Two typical camps of parallel language design:
low-level vs. high-level

"Why is everything so tedious?"

"Why don’t I have more control?"
3) Multiresolution Language Design

**Our Approach:** Structure the language in a layered manner, 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
- use appropriate separation of concerns to keep these layers clean

(language concepts)

- Distributions
- Data parallelism
- Task Parallelism
- Locality Control
- Base Language
- Target Machine
4) Ability to Tune for Locality/Affinity

- Large-scale systems tend to store memory with processors
  - a good approach for building scalable parallel systems

- Remote accesses tend to be significantly more expensive than local

- Therefore, placement of data relative to computation matters for scalable performance
  ⇒ programmer should have control over placement of data, tasks

- As multicore chips grow in #cores, locality likely to become more important in desktop parallel programming as well
  - GPUs/accelerators also expose node-level locality concerns
4) A Note on Machine Model

- As with ZPL, the CTA is still present in our design to reason about locality
- That said, it is probably more subconscious for us
- And we vary in some minor ways:
  - no controller node
    - though we do utilize a front-end launcher node in practice
  - nodes can execute multiple tasks/threads
    - through software multiplexing if not hardware
5) Support for Modern Language Concepts

- students graduate with training in Java, Matlab, Perl, C#
- HPC community mired in Fortran, C (maybe C++) and MPI
- we’d like to narrow this gulf
  - leverage advances in modern language design
  - better utilize the skills of the entry-level workforce…
    …while not ostracizing traditional HPC programmers
- examples:
  - build on an imperative, block-structured language design
  - support object-oriented programming, but make its use optional
  - support for static type inference, generic programming to support…
    …exploratory programming as in scripting languages
  - …code reuse