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 10x
- 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”

- **global-view**
  
  \[
  \frac{\begin{array}{c}
    \text{\textcolor{blue}{\texttt{\textbullet\textbullet\textbullet\textbullet}}}
  \end{array} + \begin{array}{c}
    \text{\textcolor{blue}{\texttt{\textbullet\textbullet\textbullet\textbullet}}}
  \end{array}}{2} = \begin{array}{c}
    \text{\textcolor{yellow}{\texttt{\textbullet\textbullet\textbullet\textbullet}}}
  \end{array}
  \]

- **fragmented**

  \[
  \begin{array}{c}
    \text{\textcolor{blue}{\texttt{\textbullet\textbullet\textbullet\textbullet}}}
  \end{array}, \begin{array}{c}
    \text{\textcolor{blue}{\texttt{\textbullet\textbullet\textbullet\textbullet}}}
  \end{array}, \begin{array}{c}
    \text{\textcolor{yellow}{\texttt{\textbullet\textbullet\textbullet\textbullet}}}
  \end{array}
  \]
2) Global-view vs. Fragmented

Problem: “Apply 3-pt stencil to vector”

\[
\begin{align*}
\text{global-view} & : \quad ( + )/2 \\
\text{fragmented} & : \quad ( + )/2 + ( + )/2 + ( + )/2
\end{align*}
\]

2) Global-view vs. SPMD Code

Problem: “Apply 3-pt stencil to vector”

```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;
    }
}
```

```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”

```
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;
    }
}
```

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

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, 0, 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.
2) rprj3 stencil from NAS MG

![Diagram of rprj3 stencil]

2) NAS MG rprj3 stencil in Fortran + MPI

[Fortran code and MPI implementation details]

- Orange = $W_0$
- Green = $W_1$
- Blue = $W_2$
- Red = $W_3$
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*S.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.*

NAS MG *rprj3* stencil in ZPL

```zpl
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[ 0,-1, 0] + R@d[ 0, 0,-1] + R@d[-1, 0, 0] + R@d[-1, 0, 0] + R@d[-1, 0, 0] + R@d[-1, 0, 0]) + 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] + R@d[-1,-1,-1]);
end;
```
Cray 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.

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?
Cray

Code Size

- Communication
- Declarations
- Computation

Lines of Code

F+MPI  ZPL  A-ZPL

Language

242  70  77
202  87  95
566  0  200
400  600  800
1000  1200

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

2) Classifying HPC Programming Notations

- **communication libraries:**
  - MPI, MPI-2
  - SHMEM, ARMCI, GASNet
- **data / control**
  - fragmented / fragmented/SPMD
  - fragmented / SPMD

- **shared memory models:**
  - OpenMP, pthreads
- **global-view / global-view** (trivially)

- **PGAS languages:**
  - Co-Array Fortran
  - UPC
  - Titanium
- **fragmented / SPMD**
  - global-view / SPMD
  - fragmented / SPMD

- **HPCS languages:**
  - Chapel
  - X10 (IBM)
  - Fortress (Sun)
- **global-view / global-view**
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
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
  \[\Rightarrow\] 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