Chapel at the Petascale and on the Desktop
Challenges and Potential

Brad Chamberlain, Cray Inc.

Barcelona Multicore Workshop 2010
October 22, 2010
Five Key Parallel Language Design Decisions
For Multicore, Petascale, and Beyond

Brad Chamberlain, Cray Inc.

Barcelona Multicore Workshop 2010
October 22, 2010
What is Chapel?

- A new parallel language being developed by Cray Inc.
- Part of Cray’s entry in the DARPA 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, developed as open-source (BSD license)
Chapel's Origins

- **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
Outline

• Chapel Background

• Five Parallel Language Design Decisions
  1. Data- vs. Task Parallelism
  2. Global- vs. Local-view Data and Control
  3. High- vs. Low-level Abstractions
  4. Shared- vs. Distributed Memory Model
  5. Locality/Affinity Model

• Next-Generation Nodes: Manycore, GPUs

• Summary

• Possible Bonus: User-defined domain maps
Design Decision 1:
Should a parallel language support data parallelism or task parallelism?
Q1: Data vs. Task Parallelism

**Data Parallel:** driven by collections of data/indices
  - e.g., “for every element in array A do the following…”
  - notable examples: HPF, ZPL, ...

**Task Parallel:** driven by specifying individual tasks
  - e.g., “task 1 should do this while task 2 does that”
  - notable examples: Cilk, pthreads, MPI, ...

**Sub-questions:**
What kinds of data parallel structures should be supported?
Can tasks have dependences between one another or not?
Can the parallel concepts be nested?
Chapel supports a unified set of concepts in order to...

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

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

...target all parallelism available in the hardware

- **Systems:** multicore desktops, clusters, HPC systems, ...
- **Levels:** machines, nodes, cores, instructions

Status quo: most current parallel programming models support only a limited number of styles and system levels, leading to hybrid programming models (e.g., MPI + OpenMP)
Design Decision 2:
Should a parallel language support a global view of data structures and control flow or a local view?
Q2: Global- vs. Local-View Data/Control

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

\[
\text{Global-View} \quad (\begin{array}{cccc}
\text{\color{purple}0} & \text{\color{purple}1} & \text{\color{purple}2} & \text{\color{purple}3} \\
\text{\color{orange}4} & \text{\color{orange}5} & \text{\color{orange}6} & \text{\color{orange}7}
\end{array})/2
\]

\[
\text{Local-View} \quad (\begin{array}{cccc}
\text{\color{purple}0} & \text{\color{purple}1} & \text{\color{purple}2} & \text{\color{purple}3} \\
\text{\color{orange}4} & \text{\color{orange}5} & \text{\color{orange}6} & \text{\color{orange}7}
\end{array})
\]
In pictures: “Apply a 3-Point Stencil to a vector”

Global-View

\[
\frac{(\frac{\text{\#\#\#\#\#}}{\text{\#\#\#\#\#}})}{2} + \frac{\text{\#\#\#\#\#}}{2} = \frac{\text{\#\#\#\#\#}}{2}
\]

Local-View

\[
\frac{(\frac{\text{\#\#\#\#\#}}{\text{\#\#\#\#\#}})}{2} + \frac{\text{\#\#\#\#\#}}{2} = \frac{\text{\#\#\#\#\#}}{2}
\]

\[
\frac{(\frac{\text{\#\#\#\#\#}}{\text{\#\#\#\#\#}})}{2} + \frac{\text{\#\#\#\#\#}}{2} = \frac{\text{\#\#\#\#\#}}{2}
\]

\[
\frac{(\frac{\text{\#\#\#\#\#}}{\text{\#\#\#\#\#}})}{2} + \frac{\text{\#\#\#\#\#}}{2} = \frac{\text{\#\#\#\#\#}}{2}
\]
Q2: Global- vs. Local-View Data/Control

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

**Global-View**

```python
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)**

```python
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
Q2: Global- vs. Local-View Data/Control

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

```
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)

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;
}
```

Assumes p divides n

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

= w_0 + w_1 + w_2 + w_3
Local-view rprj3 Stencil (Fortran + MPI)
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.*

*specifically, the Fortran + MPI rprj3 code shown previously assumes that p and n are both specified at compile-time and powers of two.
This choice is not exclusive: A language can support both global and local views, and we believe it should.

In particular, Chapel does:

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

def MySPMDProgram(me, p) {
    ...
}
```
Design Decision 3:
What level of abstraction should a parallel language support?
Q3: High- vs. Low-level Abstractions

“Why is everything so difficult?”
“Why don’t my programs port trivially?”

“Why don’t I have more control?”

Low-Level Implementation Concepts
- MPI
- OpenMP
- Pthreads

Target Machine

High-Level Abstractions
- HPF
- ZPL

Target Machine
Q3: High- vs. Low-level Abstractions

**Low-level / Control-oriented:** closer to the machine

- *e.g.,* C, MPI, OpenMP, CUDA, ...

+ general; good performance control
+ easier to implement
- tend to require more user effort to program
- more brittle w.r.t. architectural changes
  - *e.g.,* MPI works for clusters, but is inadequate for GPUs

**High-level / Programmability-oriented:** more abstract, hides details

- *e.g.,* ZPL, HPF, NESL, ...

- reverse benefits/liabilities from above
Multiresolution Languages: Layered, multi-tiered design

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

Typically a bigger language, though with good design, not necessarily a kitchen sink
Design Decision 4:
Should a parallel language support a shared-memory or distributed-memory view of data?
Shared Memory

- considered simpler, more like traditional programming
  - “if you want to access something, simply name it”
- no support for expressing locality/affinity; limits scalability
- bugs can be subtle, difficult to track down (race conditions)
- tend to require complex memory consistency models
Distributed Memory

+ a more constrained model; you can only access local data
- communication must be used to get copies of remote data
- only supports coarse-grain task parallelism
- intermixes semantics of data transfer with synchronization
- has frustrating classes of bugs of its own
  - e.g., recvs without matching sends, buffer overflows, etc.
**PGAS**: Partitioned Global Address Space

- supports a shared namespace, like shared-memory
- supports a strong sense of ownership and locality
  - each variable is stored in a particular memory segment
  - tasks can access any visible variable, local or remote
  - local variables are cheaper to access than remote ones
- retains many of the downsides of shared-memory
Design Decision 5:
How should a parallel programming language support the user’s ability to reason about locality/affinity?
Q5: Locality/Affinity Model (w.r.t. Parallelism)

**locality-oblivious**: model has no real notion of locality
  - (see shared-memory bullet from previous question)

**locality-constrained**: locality and parallelism are expressed using the same concept
  - *e.g.*, MPI ranks serve as both the unit of locality and parallelism
  - implications for utilizing multicore processors:
    - programmer has to use a hybrid model
    - or has to ignore locality within a node
    - or work outside of the abstract programming model
A5: Distinct Concepts for Parallelism vs. Locality

**Characteristics:**

- Chapel has distinct concepts for parallelism vs. locality
  - *task*: unit of parallel work that supports concurrent execution
  - *locale*: region of target architecture with processors and memory
- Resulting programming/execution model richer than SPMD
  - each locale can execute multiple tasks
  - tasks can create work for any locale
  - a more appropriate model for multicore
1. Data- vs. Task Parallelism?
   - support both (and composition) for the sake of generality

2. Global- vs. Local-view Data and Control?
   - support both: global- for productivity, local- for control

3. High- vs. Low-level Abstractions?
   - use a multiresolution design to get the best of both worlds

4. Shared- vs. Distributed Memory Model?
   - PGAS supports shared memory advantages with scalability

5. Locality/Affinity Model?
   - use distinct concepts for parallelism vs. locality

Where do your current parallel programming models fall?
Outline

- Chapel Background
- Five Parallel Language Design Decisions
- Next-Generation Nodes: Manycore, GPUs
- Summary
Processor Architecture Trends

Expected Processor Trends:

- multicore -> manycore
- increasing use of accelerators (e.g., GPGPUs)

Impacts on Programming Model:

- growing need to pay attention to locality within a node
  - desktop parallel programming will increasingly resemble cluster
  - HPC parallel programming will only become more complex
- growing need to deal with heterogeneity
  - different processor types/capabilities/limitations
  - different memory types/properties

*We believe that Chapel is well-positioned for these challenges given the choices described earlier*
Next-Generation Nodes and Design Decisions

1. Data- vs. Task Parallelism?
   - task- to launch asynchronous computations
   - data- to leverage SIMD computation units

2. Global- vs. Local-view Data and Control?

3. High- vs. Low-level Abstractions?
   - HW will be complex enough that the value of high-level global-view abstractions will only grow
   - yet desire for lower-level control will always remain

4. Shared- vs. Distributed Memory Model?
   - shared memory doesn’t match hierarchy/heterogeneity
   - yet distributed memory feels like overkill for an accelerator

5. Locality/Affinity Model?
   - will only become more important given trends
Through Chapel’s design choices...

- general forms of composable parallelism
- global- and local-view programming
- multiresolution design
- PGAS memory model
- distinct concepts for locality and parallelism

...we believe it is well-positioned for productive desktop/petascale parallel programming today

...and for the desktop/exascale machines of tomorrow where these decisions become more important
Current/Future Work

• Generalize Locale Concept to Support Hierarchies
  • single level of locality was sufficient for petascale
  • next-generation nodes will require more
• Domain Maps for Next-generation Nodes
  • to support global-view arrays on accelerators, e.g.
• Performance Improvements
  • communication optimizations
  • loop nest idioms
For More Information

- **http://chapel.cray.com**: papers, presentations, language specification, and other general information

- **https://sourceforge.net/projects/chapel**: download Chapel and view/contribute to its development

- **chapel_info@cray.com**: for general questions to the team (SourceForge-based mailing lists also exist)

- Attend our SC10 Tutorial, Monday November 15th
Questions?