Chapel: an HPC language in a mainstream multicore world

Brad Chamberlain

UW CSE/MSR Summer Research Institute
August 6, 2008
Chapel

*Chapel*: a new parallel language being developed by Cray Inc.

**Themes:**

- general parallel programming
  - data-, task-, and nested parallelism
  - express general levels of software parallelism
  - target general levels of hardware parallelism
- *global-view* abstractions
- *multiresolution* design
- control of locality
- reduce gap between mainstream & parallel languages
Chapel’s Setting: HPCS

**HPCS:** High *Productivity* Computing Systems (DARPA *et al.*)
- **Goal:** Raise HEC user productivity by 10× for the year 2010
- **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 – 2010)
  - Implement the systems and technologies resulting from phase II
  - (Sun also continues work on Fortress, without HPCS funding)
HPC vs. Mainstream Multicore

- Differences:
  - machine scale
  - performance requirements (?)
  - memory requirements (?)
  - robustness requirements (?)
  - workloads
  - programming community sizes and expertise areas

- Some interesting HPC(S) trends:
  - growing desire for software productivity, programmability
  - desire to better support nontraditional users
    - students just out of school with no C/Fortran/vi/emacs experience
    - scientists without strong parallel CS background
  - desire to leverage multicore technologies in larger systems
    - ideally without requiring hybrid programming models
  - and others that match opinions expressed in this meeting…

Chapel (4)
Outline

✓ Chapel Context

➢ Defining my terms

☐ Some Chapel features

☐ Wrap-up

(with an emphasis on themes and topics from this meeting)
Global-view vs. Fragmented

Problem: “Apply 3-pt stencil to vector”

\[
\begin{align*}
\text{global-view} & : \left( \begin{array}{c}
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\end{array} \right) / 2 \\
= & \begin{array}{c}
\text{yellow} \\
\text{yellow} \\
\text{yellow} \\
\text{yellow} \\
\text{yellow} \\
\text{yellow} \\
\text{yellow} \\
\text{yellow} \\
\text{yellow} \\
\text{yellow} \\
\text{yellow} \\
\text{yellow} \\
\text{yellow} \\
\text{yellow} \\
\text{yellow} \\
\text{yellow} \\
\text{yellow} \\
\text{yellow} \\
\text{yellow} \\
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{fragmented} & : \\
& \begin{array}{c}
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\text{blue} \\
\end{array}
\end{align*}
\]
Problem: “Apply 3-pt stencil to vector”

**Global-view vs. Fragmented**

\[
\text{global-view} = \left( \begin{array}{c}
\text{Blue} \\
\text{Blue} \\
\text{Blue} \\
\text{Blue} \\
\text{Blue} \\
\text{Blue} \\
\end{array} \right) / 2
\]

\[
\text{fragmented} = \left( \begin{array}{c}
\text{Blue} \\
\text{Blue} \\
\text{Blue} \\
\text{Blue} \\
\text{Blue} \\
\text{Blue} \\
\end{array} \right) / 2
\]

\[
\text{Global-view vs. Fragmented}
\]
**Problem:** “Apply 3-pt stencil to vector”

```chapel
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;
    }
}
```

```chapel
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;
    }
}
```
Global-view vs. SPMD Code

Problem: “Apply 3-pt stencil to vector”

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

SPMD

```python
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;
    }
}
```

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;
    }
}
```
MPI SPMD pseudo-code

Problem: “Apply 3-pt stencil to vector”

**SPMD (pseudocode + MPI)**

```plaintext
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;
}
```

Communication becomes geometrically more complex for higher-dimensional arrays
NAS MG $rprj3$ stencil

\[
\begin{align*}
\text{\[image\]} & = \text{\[image\]} + \text{\[image\]} + \text{\[image\]} \\
\text{\[image\]} & = \text{\[image\]} \\
\text{\[image\]} & = \text{\[image\]}
\end{align*}
\]

- \(\text{\[image\]} = w_0\)
- \(\text{\[image\]} = w_1\)
- \(\text{\[image\]} = w_2\)
- \(\text{\[image\]} = w_3\)
subroutine comm3(s,i1,i2,i3,k)  
use caf_intrinsics  
implicit none  
include 'globals.h'  
include 'cafnpb.h'  
implicit none  
use caf_intrinsics  

subroutine take3( axis, dir, u, n1, n2, n3 )
  integer axis, dir, n1, n2, n3
  double precision u(n1,n2,n3)
  include 'cafnpb.h'
  implicit none
  use caf_intrinsics
  do  i1=1,n1
    do  i2=1,n2
      do  i3=2,n3
        buff(buff_len,buff_id ) = u( i1, i2,2)
        buff_len = buff_len + 1
      enddo
    enddo
  enddo
  endif
  if( axis .eq.  1 )then
    if( dir .eq. +1 ) then
      buff(i,2) = buff(i,1)
      buff(i,4) = buff(i,3)
      buff(i,3) = 0.0D0
      do  i=1,nm2
        buff(i,2) = buff(i,1)
        buff(i,4) = buff(i,3)
        buff(i,3) = 0.0D0
      enddo
    endif
    if( dir .eq. -1 ) then
      buff(i,4) = buff(i,3)
      buff(i,3) = 0.0D0
      if( i .eq. 1)then
        buff(i,4) = buff(i,3)
        buff(i,3) = 0.0D0
      endif
      endif
  endif
  if( axis .eq.  2 )then
    if( dir .eq. +1 ) then
      buff(i,1) = buff(i,2)
      buff(i,3) = buff(i,4)
      buff(i,4) = 0.0D0
      do  i=1,nm2
        buff(i,1) = buff(i,2)
        buff(i,3) = buff(i,4)
        buff(i,4) = 0.0D0
      enddo
    endif
    if( dir .eq. -1 ) then
      buff(i,3) = buff(i,4)
      buff(i,4) = 0.0D0
      if( i .eq. 1)then
        buff(i,3) = buff(i,4)
        buff(i,4) = 0.0D0
      endif
      endif
  endif
  if( axis .eq.  3 )then
    if( dir .eq. +1 ) then
      buff(i,2) = buff(i,4)
      buff(i,4) = 0.0D0
      do  i=1,nm2
        buff(i,2) = buff(i,4)
        buff(i,4) = 0.0D0
      enddo
    endif
    if( dir .eq. -1 ) then
      buff(i,4) = 0.0D0
      if( i .eq. 1)then
        buff(i,4) = 0.0D0
      endif
      endif
  endif
endif
enddo

enddo

endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endif
endid
def rprj3(S, R) {
    param 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));
}

NAS MG rprj3 stencil in Chapel
NAS MG \textit{rprj3} stencil in ZPL

\textbf{procedure} \textit{rprj3}(\texttt{var S,R: [,] double;
\hspace{1em}d: array [] of direction});

\textbf{begin}
\begin{align*}
S := & \hspace{1em} 0.5 \hspace{1em} * \hspace{1em} R \\
& + 0.25 \hspace{1em} * \hspace{1em} (R \hat{\otimes}^d[1, 0, 0] + R \hat{\otimes}^d[0, 1, 0] + R \hat{\otimes}^d[0, 0, 1] +
R \hat{\otimes}^d[-1, 0, 0] + R \hat{\otimes}^d[0,-1, 0] + R \hat{\otimes}^d[0, 0, -1]) \\
& + 0.125 \hspace{1em} * \hspace{1em} (R \hat{\otimes}^d[1, 1, 0] + R \hat{\otimes}^d[1, 0, 1] + R \hat{\otimes}^d[0, 1, 1] +
R \hat{\otimes}^d[1,-1, 0] + R \hat{\otimes}^d[1, 0,-1] + R \hat{\otimes}^d[0, 1,-1] +
R \hat{\otimes}^d[-1, 1, 0] + R \hat{\otimes}^d[-1, 0, 1] + R \hat{\otimes}^d[0,-1, 1] +
R \hat{\otimes}^d[-1,-1, 0] + R \hat{\otimes}^d[-1, 0,-1] + R \hat{\otimes}^d[0,-1,-1]) \\
& + 0.0625 \hspace{1em} * \hspace{1em} (R \hat{\otimes}^d[1, 1, 1] + R \hat{\otimes}^d[1, 1,-1] +
R \hat{\otimes}^d[1,-1, 1] + R \hat{\otimes}^d[1,-1,-1] +
R \hat{\otimes}^d[-1, 1, 1] + R \hat{\otimes}^d[-1, 1,-1] +
R \hat{\otimes}^d[-1,-1, 1] + R \hat{\otimes}^d[-1,-1,-1])
\end{align*}
\textbf{end};
NAS MG: Fortran + MPI vs. ZPL

The graph on the left shows the distribution of lines of code across different languages: F+MPI, ZPL, and A-ZPL. The breakdown is as follows:

- **F+MPI**:
  - Communication: 566 lines
  - Declarations: 202 lines
  - Computation: 242 lines

- **ZPL**:
  - Communication: 87 lines
  - Declarations: 70 lines
  - Computation: 95 lines

- **A-ZPL**:
  - Communication: 95 lines
  - Declarations: 77 lines
  - Computation: 95 lines

The graph on the right illustrates the speedup over 16-processor time for A-ZPL and F+MPI compared to ZPL. The speedup is measured in units of 114,607 seconds in A-ZPL.

- **Linear speedup**
- **A-ZPL**
- **ZPL**
- **F+MPI**

The graph shows the performance improvement as the number of processors increases, with A-ZPL and F+MPI consistently outperforming ZPL.
Parallel Programming Models: Two Camps

“Why is everything so painful?”

“Why do my hands feel tied?”
Multiresolution Language Design

Our Approach: Permit the language to be utilized at multiple levels, as required by the problem/programmer

- provide 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

<table>
<thead>
<tr>
<th>Language Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Parallelism</td>
</tr>
<tr>
<td>Distributions</td>
</tr>
<tr>
<td>Task Parallelism</td>
</tr>
<tr>
<td>Base Language</td>
</tr>
<tr>
<td>Locality Control</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task Scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stealable Tasks</td>
</tr>
<tr>
<td>Suspendable Tasks</td>
</tr>
<tr>
<td>Run to Completion</td>
</tr>
<tr>
<td>Thread per Task</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Garbage Collection</td>
</tr>
<tr>
<td>Region-based</td>
</tr>
<tr>
<td>Malloc/Free</td>
</tr>
</tbody>
</table>

Chapel (17)
Outline

- Chapel Context
- Defining my terms
- Some Chapel features
- Wrap-up
Domains

*domain*: a first-class index set

```chapel
var m = 4, n = 8;

var D: domain(2) = [1..m, 1..n];
```
Domains

*domain*: a first-class index set

```chapel
var m = 4, n = 8;
var D: domain(2) = [1..m, 1..n];
var Inner: subdomain(D) = [2..m-1, 2..n-1];
```

![Diagram of domain and subdomain]
Domains: Some Uses

- **Declaring arrays:**
  \[
  \text{var } A, B : [D] \text{ real;} \]

- **Iteration (sequential or parallel):**
  \[
  \text{for } \ ij \ \text{in} \ \text{Inner} \ \{ ... \} \]
  \[
  \text{or: for all } \ ij \ \text{in} \ \text{Inner} \ \{ ... \} \]
  \[
  \text{or: } ... \]

- **Array Slicing:**
  \[
  A[\text{Inner}] = B[\text{Inner}]; \]

- **Array reallocation:**
  \[
  D = [1..2*m, 1..2*n]; \]
Domains: Different Flavors

- Dense
- Strided
- Sparse

Graphs

Distributions
Task Parallelism
Base Language
Locality Control
Target Machine

Data Parallelism
Locale: Locales

**locale:** architectural unit of locality
- has capacity for processing and storage
- threads within a locale have ~uniform access to local memory
- memory within other locales is accessible, but at a price
- *e.g.*, a multicore processor or SMP node could be a locale
Locality: Locales

- user specifies # locales on executable command-line
  
  prompt> myChapelProg -nl=8

- Chapel programs have built-in domain/array of locales:

  config const numLocales: int;
  const LocaleSpace = [0..numLocales-1], Locales: [LocaleSpace] locale;

- Programmers can create their own locale views:

  const CompGrid = Locales.reshape([1..GridRows, 1..GridCols]);
Distributions: Overview

Domains may be distributed across locales

\[ \text{var } D: \text{domain}(2) \text{ distributed Block on CompGrid} = \ldots; \]

- A distribution implies…
  - …ownership of the domain’s indices (and its arrays’ elements)
  - …the default work ownership for operations on the domain/arrays

- Advanced users can write their own distributions
  (and standard distributions are written using the identical mechanism)

- A distribution must implement…
  - …the mapping from indices to locales
  - …the per-locale representation of domain indices and array elements
  - …the compiler’s target interface for lowering global-view operations

Chapel (25)
Other Chapel Features

- **unstructured task parallelism**
  - begins
  - sync statements
- **structured task parallelism**
  - cobegins
  - coforall
- **synchronization**
  - sync/single variables
  - atomic blocks

- value- and reference-based OOP (optional)
- generic programming features
- latent types / shallow static type inference
- rich compile-time language
- iterators (generators)
- tuples

- subdomains and index types
- zippered and tensor iteration
- scalar function promotion
- reductions and scans

- fine-grain control over task and data placement via on clauses
Outline

✓ Chapel Context
✓ Defining my terms
✓ Some Chapel features
➢ Wrap-up
Chapel and Mainstream Multicore

- While Chapel doesn’t specifically target mainstream multicore, it could be applicable
  - removes much tedium and nitpicky details from data parallelism
  - raises level of discourse for task parallelism beyond threads
  - though not a dialect of a mainstream language, not far afield either
    - programmers today seem more multilingual than in the past
    - interoperability more crucial than extending a language

- Chapel’s locales and distributions are likely overkill for today’s multicore processors
  - but what about for future generations of multicore?

- Chapel team does most of our development and testing on mainstream multicore machines
  - Linux, Mac, Windows, …
  - AMD, Intel, …
Takeaways

- Key themes for mainstream multicore languages
  - generality, to the extent possible
  - global-view abstractions (for Joe & Joan)
  - multiresolution design (for Steve & Stephanie)

- Other Lessons from HPC
  - abstract implementing mechanisms further from user’s view than we traditionally have
  - SPMD: an execution case to optimize for, not the only tool in the box
Future Directions (with an eye on multicore)

- expand locale concept
  - hierarchical locales
    - to expose hierarchy within a node
  - heterogeneous locale types
    - to describe coarse-grain HW heterogeneity
  - dynamically varying numbers of locales

- equivalent of “distribution” concept for task parallelism
  - (Edward Lee’s directors?)

- and of course, many others…
Collaborations

UIUC (Vikram Adve and Rob Bocchino): Software Transactional Memory (STM) over distributed memory (PPoPP `08)

ORNL (David Bernholdt et al.): Chapel code studies – Fock matrix computations, MADNESS, Sweep3D, … (HIPS `08)

PNNL (Jarek Nieplocha et al.): ARMCI port of comm. layer

EPCC (Michele Weiland, Thom Haddow): performance study of single-locale task parallelism

CMU (Franz Franchetti): Chapel as portable parallel back-end language for SPIRAL

(Your name here?)
Chapel Contributors

- **Current Team**
  - Brad Chamberlain
  - Steve Deitz
  - Samuel Figueroa
  - David Iten
  - Andy Stone (not shown)

- **Alumni**
  - Robert Bocchino
  - David Callahan
  - James Dinan
  - Roxana Diaconescu
  - Shannon Hoffswell
  - Mary Beth Hribar
  - Mark James
  - Mackale Joyner
  - John Plevyak
  - Wayne Wong
  - Hans Zima
Questions?

bradc@cray.com
chapel_info@cray.com

http://chapel.cs.washington.edu