Chapel: Compiler Challenges

Steve Deitz
Cray Inc.

LCPC
21 October 2005
HPCS: High Productivity Computing Systems

Increase productivity for HEC community by the year 2010 (via HW, architecture, OS, compilers, tools, …)

Productivity = Programmability
+ Performance
+ Portability
+ Robustness

Revolutionary results (not evolutionary)

Marketable to people other than program sponsors

Phase II Competitors: Cray, IBM, Sun
What is Chapel?

- **Chapel**: Cascade High-Productivity Language

- **Goal**
  - Simplify the creation of parallel programs
  - Allow for experimental programming
  - Support evolution from prototype to production
  - Emphasize generality

- **Motivating Technologies**
  - Multithreaded programming
  - Locality-aware programming
  - Object-oriented programming
  - Generic programming
Multithreaded Programming

- Global view of computation, data structures
- Abstractions for data and task parallelism
  - *Data*: domains, arrays, iterators, …
    
    ```
    forall i in X ...
    ```
  - *Task*: cobegins, atomic transactions, syncs, …
    
    ```
    cobegin {
        taskA();
        taskB();
    }
    ```
- Composition of parallelism
Locality-aware Programming

- **Locale**: machine unit of storage and processing
- Programmer specifies number of locales at runtime
  
  ```
prompt> myChapelProg -nl=8
  ```
- Built-in locale array
  ```
  const locales: [1..num_locales()] locale;
  ```
- User-defined locale arrays
  ```
  var CompGrid: [1..GridRows, 1..GridCols] locale = ...;
  ```
- Domains (index sets) distribute across locales
  ```
  var D: domain(2) distributed(Block(2), CompGrid) = ...;
  ```
- Computations on locales
  ```
  cobegin {
    forall i in D on B(i) do
      on ALocs do taskA(); A(i) = B(i);
      on BLocs do taskB();
  }
  ```
Object-Oriented Programming

- Objects help manage program complexity
  - Encapsulate related data and code
  - Facilitate reuse
  - Separate interfaces from implementations

- Chapel supports traditional and value classes
  - Traditional: assign by reference, nominally typed
  - Value: assign by value/name, structurally typed

- OOP is not required (user’s preference)

- Advanced language features expressed using classes
  - User-defined distributions, reductions, …
Generic Programming

- **Type variables and parameters**
  
  ```plaintext
  class Stack {
    type t;
    var buffsize: integer = 128;
    var data: [1..buffsize] t;
    function top(): t { ... };
  }
  ```

- **Type query variables**
  
  ```plaintext
  function copyN(data: [?D] ?t; n: integer): [D] t {
    var newcopy: [D] t;
    forall i in 1..n do
      newcopy(i) = data(i);
    return newcopy;
  }
  ```

- **Elided types**
  
  ```plaintext
  function inc(val): {
    var tmp = val;
    return tmp + 1;
  }
  ```

- **Chapel programs are statically-typed**
Productivity Miscellanea

- General purpose
  - Express arbitrary parallelism
  - Control location of data/computation
  - Provide access to lower levels of implementation
  - Extensible distributions, reductions, scans

- Separation of concerns
  - Number and arrangement of locales
  - Data distribution
  - Numeric types and widths
  - Array implementation, e.g., dense vs. sparse
  - Array rank
  - User should be able to change these without...
    …unnecessarily duplicating code
    …rewriting all references to the data in question
    …changing communication details
Compiler Challenges

- General *global-view language* challenges
  - Definition
  - Impact

- Chapel-specific challenges
  - Object-oriented and generic programming issues
  - User-defined data distributions
  - Performance problems due to programmable focus
  - Commodity implementation issues
  - Language Interoperability
  - Garbage collection
  - Zippered iteration
Global-view: Definition

◆ With a global-view, the programmer writes the program largely independent of the virtual processor layout.

Global-View

```
var n: integer = 1000;
var a, b: [1..n] float;

forall i in 1..n do
    a(i) += b(i);
```

Fragmented

```
var n: integer = 1000;
var ln: integer = n/num_locales();
var a, b: [1..ln] float;

forall i in 1..ln do
    a(i) += b(i);
```
With a global-view, the programmer writes the program largely independent of the virtual processor layout.

**Global-View**  Two-point stencil changes highlighted

\[
\begin{align*}
\text{var } & \text{n: integer } = 1000; \\
\text{var } & \text{a, b: [1..n] float;} \\
\text{forall } i \text{ in } 2..n-1 \text{ do} \\
& a(i) = b(i-1) + b(i+1);
\end{align*}
\]

**Fragmented**

\[
\begin{align*}
\text{var } & \text{n: integer } = 1000; \\
\text{var } & \text{ln: integer } = \text{n/num_locales();} \\
\text{var } & \text{a, b: [1..ln] float;} \\
\text{forall } i \text{ in } 1..\ln \text{ do} \\
& a(i) += b(i);
\end{align*}
\]
With a global-view, the programmer writes the program largely independent of the virtual processor layout.

**Global-View**

```plaintext
var n: integer = 1000;
var a, b: [1..n] float;

forall i in 2..n-1 do
  a(i) = b(i-1) + b(i+1);
```

**Fragmented**

```plaintext
var n: integer = 1000;
var ln: integer = n/num_locales();
var lo: integer = (if left then 0 else 1);
var hi: integer = (if right then ln+1 else ln);
var a, b: [lo..hi] float;

if right { send(right, a(ln)); recv(right, a(ln+1)); }  
if left  { send(left,  a(1));   recv(left,  a(0));  }

forall i in lo+1..hi-1 do
  a(i) = b(i-1) + b(i+1);
```
Challenge: Efficient compilation of the global view

- Fragmented languages obfuscate code
  - User intersperses per-processor management code with program
  - User required to think in SPMD model
- Global-view languages leave detail management to compiler

Plans:

- Leverage work on HPF and ZPL
- Expose locality to user through user-defined distributions

<table>
<thead>
<tr>
<th>fragmented languages</th>
<th>global-view languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI</td>
<td>OpenMP</td>
</tr>
<tr>
<td>SHMEM</td>
<td>HPF</td>
</tr>
<tr>
<td>Co-Array Fortran</td>
<td>ZPL</td>
</tr>
<tr>
<td>UPC</td>
<td>MTA C/Fortran</td>
</tr>
<tr>
<td>Titanium</td>
<td>Matlab</td>
</tr>
<tr>
<td></td>
<td>Chapel</td>
</tr>
</tbody>
</table>
Challenge: Object-oriented and generic programming
- Features require substantial implementation effort
- Not strictly necessary for parallel programming
- Included in order to support large-scale software systems
- Useful for arrays, sequences, distributions, reductions, etc.

Plans:
- Early implementation effort focusing on these features
Challenge: User-defined data distributions

- Chapel intends to support user-defined domain distributions
- Goal is to implement “standard” distributions using the same mechanism (i.e., avoid treating them as special cases)
- This has not been successfully accomplished before

Plans:

- Caltech/JPL actively working in this area, initial whitepapers
- Implementation effort focused on enabling technologies (OOP)
- If this approach fails, fall back on an HPF-/ZPL-like approach
Challenge: Insufficient performance
- Focus on programmability
- May take too much time to optimize, e.g., HPF

Plans:
- Implementing features depth-first
- Perhaps programmability will make this gap worthwhile?
- Can fall back to a stricter semantic model like ZPL
Challenge: Commodity implementation issues
- Chapel designed with idealized architecture in mind
- Commodity architectures, e.g., clusters, are not ideal
- Multi-threaded, one-sided communication layer required

Plans:
- Examine/leverage GASnet and ARMCI
- Similar infrastructure required by all three HPCS languages
Challenge: Language interoperability
   - Substantial engineering required

Plans:
   - Compiling to C
   - Leverage Livermore’s Babel work
Challenge: Garbage collection
  - Parallel garbage collection has traditionally been a challenge

Plans:
  - With Java’s popularity, research in this area has ramped up
  - Leverage academic work in this area
  - Architecture may help
  - Emphasize non-garbage-collected language features
  - Fall back on more help from the user
    - Leverage Titanium’s “regions”
    - Programmer hints to time/place to collect garbage
    - Allow/Force user to manage memory explicitly
Challenge: Zippered iteration

- Difficult to compute multiple iterators at the same time
- Difficult to optimize multiple iterators that are similar

Plans:

- Implement a ‘next’ function from an iterator function
- Implement zippered iteration with multiple threads
- Cloning to optimize similar iterator cases
- Lots of semantic information available for optimization
- Compiler warnings to user
Summary

- Chapel is being designed to...
  - enhance programmer productivity
  - address a wide range of workflows

- Via high-level, extensible abstractions for...
  - multithreaded parallel programming
  - locality-aware programming
  - object-oriented programming
  - generic programming

- Status:
  - Draft language specification
  - Open source implementation proceeding apace
  - Your feedback desired!