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)

Chapel and Productivity

Chapel’s Productivity Goals:
- vastly improve **programmability** over current languages/models
  - writing parallel codes
  - reading, modifying, porting, tuning, maintaining, evolving them
- support **performance** at least as good as MPI
  - competitive with MPI on generic clusters
  - better than MPI on more capable architectures
- improve **portability** compared to current languages/models
  - as ubiquitous as MPI, but with fewer architectural assumptions
  - more portable than OpenMP, UPC, CAF, …
- improve **code robustness** via improved semantics and concepts
  - eliminate common error cases altogether
  - better abstractions to help avoid other errors
Outline

- Chapel’s Themes, Context, and Goals
- The Parallel Language Landscape
  - Distributed memory programming
  - Shared memory programming
  - PGAS languages
  - HPCS languages
- Programming Model Terminology

Distributed Memory Programming

- **Characteristics:**
  - Execute multiple binaries simultaneously & cooperatively
  - Each binary has its own local namespace
  - Binaries transfer data via communication calls
- **Examples:** MPI, PVM, SHMEM, sockets, …
My Evaluation of MPI

**Strengths**
- a very general parallel programming model
- most scientific HPC results in the past decade achieved using it
- it runs on most parallel platforms
- it is relatively easy to implement (or, that’s the conventional wisdom)
- for many architectures, it can result in near-optimal performance
- it serves as a strong foundation for higher-level technologies

**Weaknesses**
- only supports parallelism at the “cooperating executable” level
  - applications and architectures contain parallelism at many levels
  - doesn’t reflect how one abstractly thinks about parallel algorithms
- encodes too much about “how” data should be transferred rather than simply “what data” (and possibly “when”)
  - can mismatch architectures with different data transfer capabilities
- obfuscates algorithms with many low-level details
  - tedious and error-prone details, arguably best left to the compiler

Shared Memory Programming

- **Characteristics:**
  - execute multiple cooperating threads within one process
  - threads have shared namespace
  - coordinate data accesses via synchronization primitives
- **Examples:** OpenMP, pthreads, Java, …
My Evaluation of OpenMP

**Strengths**
- supports finer-grain parallelism -- e.g., loop-level
- can be mixed with other programming models (parallel & sequential)
  - supports existing languages, code bases
  - supports incremental parallelization
- consortium effort, broad support among vendors

**Weaknesses**
- shared memory bugs can be notoriously difficult to track down
- no precise control over locality and affinity
  - makes it difficult to scale to large-scale systems

(Traditional) PGAS Programming Models

- **Characteristics:**
  - execute an SPMD program (Single Program, Multiple Data)
  - all binaries share a namespace
    - namespace is partitioned, permitting reasoning about locality
    - binaries also have a local, private namespace
  - compiler introduces communication to satisfy remote references
- **Examples:** UPC, Co-Array Fortran (Fortran 2008), Titanium
My Evaluation of Traditional PGAS Languages

**Strengths**
- supports distributed memory architectures
  - particularly ideal for networks with RDMA support
- raises the level of abstraction compared to MPI
- nice support for pointer-based data structures across multiple nodes
- some support for distributed arrays

**Weaknesses**
- SPMD programming/execution model is too restrictive
  - algorithms and architectures support parallelism at many levels
- subject to similar synchronization bugs as shared-memory programs
- distributed arrays more restricted than one would ideally like
  - CAF: bookkeeping challenges when local arrays aren’t uniform
  - UPC: limited to 1D block-cyclic arrays

### PGAS: What’s in a Name?

<table>
<thead>
<tr>
<th>PGAS Languages</th>
<th>memory model</th>
<th>programming model</th>
<th>execution model</th>
<th>data structures</th>
<th>communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI</td>
<td>distributed memory</td>
<td>cooperating seq. processes (often SPMD in practice)</td>
<td>manually created distributed arrays</td>
<td>APIs</td>
<td></td>
</tr>
<tr>
<td>OpenMP</td>
<td>shared memory</td>
<td>global-view parallelism</td>
<td>shared memory multithreaded</td>
<td>shared memory arrays</td>
<td>N/A</td>
</tr>
<tr>
<td>CAF</td>
<td>PGAS</td>
<td>Single Program, Multiple Data (SPMD)</td>
<td>co-arrays</td>
<td>co-array refs</td>
<td></td>
</tr>
<tr>
<td>UPC</td>
<td>PGAS</td>
<td>global-view parallelism</td>
<td>PGAS multithreaded</td>
<td>global-view distributed arrays</td>
<td>implicit</td>
</tr>
<tr>
<td>Titanium</td>
<td>PGAS</td>
<td>global-view parallelism</td>
<td>PGAS multithreaded</td>
<td>global-view distributed arrays</td>
<td>implicit</td>
</tr>
</tbody>
</table>
Asynchronous PGAS (APGAS)

- **Characteristics:**
  - a term coined by IBM’s X10 group (to the best of my knowledge)
  - uses the PGAS memory model
  - programming/execution models are richer than SPMD
    - each node can execute multiple tasks/threads
    - nodes can create work for one another

- **Examples:** X10, Chapel, Fortress (?)

---

X10 in a Nutshell (reflecting my opinions)

- Originally influenced by Java
  - emphasis on type safety, OOP design, small core language
  - also ZPL: support for global-view domains and arrays

- Has since diverged from Java; influenced by Scala, others

- Similar concepts to what you’ll hear about today in Chapel
  - yet a fairly different syntax and design aesthetic

- Main differences from Chapel
  - X10 semantics tend to distinguish between local and remote data
  - X10 is a purer object-oriented language
    - for example, arrays have reference rather than value semantics
      
      ```
      A = B; // alias or copy if A and B are arrays?
      ```

- For more information:
  - [http://x10.codehaus.org/](http://x10.codehaus.org/)
  - [http://sf.net/projects/x10](http://sf.net/projects/x10)
Fortress in a Nutshell (again, my opinions)

- The most blue-sky, clean-slate of the HPCS languages

- **Goal:** define language semantics in libraries, not compiler:
  - data structures and types (including scalars types?)
  - operators, typecasts
  - operator precedence
  - in short, as much as possible to support future changes, languages

- Other themes:
  - implicitly parallel -- most things are parallel by default
  - supports mathematical notation, symbols, operators
  - functional semantics
  - hierarchical representation of target architecture’s structure
  - units of measurement in the type system (meters, seconds, miles, …)

- For more information:
  - [http://projectfortress.sun.com/Projects/Community/](http://projectfortress.sun.com/Projects/Community/)

Outline

- Chapel's Themes, Context, and Goals
- the Parallel Language Landscape
- Programming Model Terminology
  - *global-view* vs. *fragmented* programming models
  - *multiresolution languages*
  - a first taste of Chapel
Global-view vs. Fragmented

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

<table>
<thead>
<tr>
<th>Global-view</th>
<th>Fragmented</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{global-view} = \left( \frac{\text{fragmented} + \text{fragmented}}{2} \right)
\]
Global-view vs. SPMD Code

Problem: “Apply 3-pt stencil to vector”

Global-view

```plaintext
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

```plaintext
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;
    }
}
```

Assumes `numProcs` divides `n`; a more general version would require additional effort.
MPI SPMD pseudo-code

Problem: “Apply 3-pt stencil to vector”

SPMD (pseudocode + MPI)

```
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

---

rprj3 stencil from NAS MG

```
= \begin{align*}
\begin{bmatrix}
W_0 & W_1 & W_2 & W_3
\end{bmatrix}
\end{align*}
```

= \begin{align*}
\begin{bmatrix}
W_0 & W_1 & W_2 & W_3
\end{bmatrix}
\end{align*}

= \begin{align*}
\begin{bmatrix}
W_0 & W_1 & W_2 & W_3
\end{bmatrix}
\end{align*}

Chapel: Background (3)
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.
Summarizing Fragmented/SPMD Models

- **Advantages:**
  - fairly straightforward model of execution
  - relatively easy to implement
  - reasonable performance on commodity architectures
  - portable/ubiquitous
  - lots of important scientific work has been accomplished with them

- **Disadvantages:**
  - blunt means of expressing parallelism: cooperating executables
  - fails to abstract away architecture / implementing mechanisms
  - obfuscates algorithms with many low-level details
    - error-prone
    - brittle code: difficult to read, maintain, modify, *experiment*
    - “MPI: the assembly language of parallel computing”

Current HPC Programming Notations

- **communication libraries:**
  - MPI, MPI-2
  - SHMEM, ARMCI, GASNet

- **shared memory models:**
  - OpenMP, pthreads

- **PGAS languages:**
  - Co-Array Fortran
  - UPC
  - Titanium

- **HPCS languages:**
  - Chapel
  - X10 (IBM)
  - Fortress (Sun)
Parallel Programming Models: Two Camps

**Expose Implementing Mechanisms**
- MPI
- OpenMP
- pthreads

**Target Machine**

Higher-Level Abstractions
- ZPL
- HPF

“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

**language concepts**
- Data Parallelism
- Distributions
- Task Parallelism
- Base Language
- Locality Control
- Target Machine

**task scheduling**
- Stealable Tasks
- Suspendable Tasks
- Run to Completion
- Thread per Task
- Target Machine

**memory management**
- Garbage Collection
- Region-based
- Malloc/Free
- Target Machine
Questions?