The Chapel Language

José Gracia, HLRS

March 2, 2010
Outline

1 Chapel in a Nutshell
   - Language concepts

2 Kernels
   - EuroBen kernels
   - HPCC kernels

3 Conclusions
Outline

1. Chapel in a Nutshell
   - Language concepts

2. Kernels
   - EuroBen kernels
   - HPCC kernels

3. Conclusions
**Portrait**

- *Cascade High Production Language*: Chapel
- developed by University of Washington & Cray within the HPCS project
- ”C-like” (i.e. curly braces & semicolons)
- asynchronous PGAS-like language
- data & task parallelism
- influenced by: ZPL, HPF and Cray MTA extensions to C and Fortran
- resources:
  - www.cray.com
  - chapel.sourceforge.net
  - chapel-users@lists.sourceforge.net
  - chapel/examples/* in distribution

Chapel compiler still considered prototype, i.e. limited use for production environment.
Disclaimer

In general, I have not checked the syntax of the following code snippets ... in particular as it is still floating (eg. distributions).

Please take them with a grain of salt!!
Key concepts

- locale: execution site
- domain: data/array range
- distribution: data distribution / domain decomposition
- forall: parallel loop
- on: affinity
Locale

key concept: *locale*

A hardware execution site with uniform access to memory and other resources. Different locales are homogeneous (see *realm* for inhomogenous execution sites).

In general a *locale* supports concurrent execution threads in order to allow locale-local data & task parallelism.

example:

Each node in a cluster corresponds to a single *locale*.
Domain

key concept: domain

A domain is a range of indices over which an array or collection is defined. All arrays are defined over (anonymous) domains. Domains may have multidimensional integer indices (arithmetic array), indices of arbitrary type (dictionary, associative array), or anonymous indices (opaque domain).

examples:

var ProblemSpace: domain(2) = [1..m, 1..n]; // 2D n x m domain
var Matrix: [ProblemSpace] real; // real-valued matrix
    // declared over ProblemSpace

var People: domain(string); // associative domain
var Age: [People] int; // integer-valued variable
People += "Jose"; // add "Jose" to indices of People
Age("Jose") = beMyGuest(); // array element assignment
Domain, continued

**sparse domain**

Arithmetic *domains* may be dense or sparse. The latter is indicated by the keyword *sparse* and basically allows the index-set to be modified during its lifetime.

**example:**

```chapel
var DenseDomain: domain(2) = [1..m, 1..n]; // dense domain
var SparseDomain: sparse subdomain(DenseDomain); // sparse domain with
// limits (1,1)...

var Matrix: [SparseDomain] real;

SparseDomain += (1,0); // assign to index set
Matrix(1,0) = 1.0; // assign element
writeln(Matrix(1,0), Matrix(3,3)); // unassigned indices are valid
// => 1.0 0.0
```
Distribution

key concept: *distribution*

A *distribution* specifies how a *domain* is distributed across available *locales*. Each *domain* may be associated to a *distribution* explicitly. If unspecified, the default *distribution* applies which keeps all the data on the default *locale*.

Distributions provide iterators over the locale-local indices.

examples

```chapel
var DefaultSpace: domain(2) = [1..m, 1..n]; // with default distribution
var Matrix: [DefaultSpace] = real; // data on first locale

var ProblemDist = new Block(); // block dist from standard lib
var ProblemSpace: domain(2) distributed ProbDist = [1..m, 1..n]; // explicitly distributed domain

var MatrixDist: [ProblemSpace] = real; // data distributed
```
Standard library distributions

As per V1.02 the Chapel standard module library offers the following distributions:

- (default):
- Block: multi-dimensional blockwise
- Cyclic: multi-dimensional cyclic
- BlockCyc: block-cyclic
- CSR: compressed storage row (sparse matrices)
- DistGPU: data stored on GPU
Controlling data distribution

Most distributions accept a "targetLocales" parameter to specify data storage targets. However, interpretation of this parameter is distribution specific.

example:

```chapel
const numLocales = int; // provided by system
const Locales: locale = [1..numLocales]; // provided by system

var Dist1: Block(targetLocales=Locales(2...numLocales-1));
var Dist2: Block(targetLocales=Locales(numLocales..1));
```

For anything beyond that the "experienced user" may implement his own distribution.
Forall

**key concept: forall loop**

Data parallelism is accessible through the *forall* loop which relies on a (distributed) *domain*'s index iterators. On each individual *locale* loops may be traversed simultaneously by multiple threads. I.e two levels of parallelism: node-local and cross-node.

**examples:**

```chapel
forall ij in [1..m, 1..n] do
    writeln("Running on locale", here.id);

var ProblemSpace: domain(2) distributed = [1..m, 1..n];
forall ij in ProblemSpace do
    writeln("Running on locale", here.id);
var Matrix : [ProblemSpace] = real;
Matrix = 1.0 // implicit forall loop
```
On clause

key concept: on clause

Parallel tasks are spawned by `forall`, `begin`, `cobegin`, etc. `Forall` loop iterations are executed on the locale owning the running index.

Almost any statement may be *migrated* to a different locale with the `on` clause.

**Examples**

```chapel
const Locales: locale = [1..numLocales]; // provided by system
on Locales(1) do writeln(here.id); // yields: 1

forall loc in Locales do
    otherLoc = Locales((here.id+1)%numLocales)
    on otherLoc do
        writeln(loc, "->", here.id); // yields: 0 -> 1, etc
```
Outline

1. Chapel in a Nutshell
   - Language concepts

2. Kernels
   - EuroBen kernels
   - HPCC kernels

3. Conclusions
Kernels

- implement the EuroBen kernels in a proper ‘Chapel way’ without falling back to fragmented MPI-style programming
- implies relying on distributions to handle data partitioning and communication
- Chapel’s performance stands or falls with quality of parallel iterators in a given distribution.
  → EuroBen :(  
- show Chapel’s potential through official HPCC kernels
mod2as: declarations

**Approach**

Use Chapel distribution *CSR* (compressed sparse row) and simple forall loops. No explicit communication.

**declaration of distribution, domains and variables**

```chapel
const Distribution = new CSR();
const DenseMatrixDom: domain(2) = [1..nrows, 1..ncols];
const MatrixDom: sparse subdomain(DenseMatrixDom) distributed(Distribution)  
   = sparseIndices(ncols, nrows, nelmts);
   // sparse subdomain; index set initialized to sparseIndices()
const LhsVecDom = [MatrixDom.dim(2)];
const RhsVecDom = [MatrixDom.dim(1)];

var A: [MatrixDom] real;
const b: [LhsVecDom] real;  // const -> replicate to all locales
var c: [RhsVecDom] real = 0.0;
```
mod2as: timed loop

timed loop

startTime = getCurrentTime();
for 1..nrep {
    forall (i, j) in A.domain do
        c(i) += A(i, j) * b(j);
}
execTime = (getCurrentTime() - startTime)/nrep;

Separation of algorithm and implementation

The timed algorithm (ie. the loop) is independent of the implementation (# locales, data distribution method, etc).

In particular forall in A.domain will iterate only over sparse indices if A is sparse.
mod2as: timed loop with reduction

```chapel
timed loop with reduction

startTime = getCurrentTime();
for 1..nrep {
    c = + reduce(dim=2) [(i,j) in MatrixDom] (A(i,j) * b(j));
}
execTime = (getCurrentTime() - startTime)/nrep;
```

Dow!

Unfortunately the partial reduce semantics, i.e. reduce along one dimension only, is not implemented yet. The workaround could be:

```chapel
forall i in MatrixDom.dim(1) do
    c(i) = + reduce [j in MatrixDom.dimIter(2,i)] (A(i,j) * b(j));
```
mod2as: results & issues

Results for 10240 x 10240 @ 7% fill factor on Nehalem cluster
Independently of the number of nodes mod2as yields 25Mflop/s :

Issues

Iterators for the distribution CSR do currently:
- **not** support distributed execution across several locales
- **not** support concurrent multi-threaded execution on a single locale

Comparisons

- Chapel: non-distributed dense matrix (10k x 10k) yields > 400 Mflop/s
- C: serial reference yields > 1200 Mflop/s
- C: MPI reference yields > ?? Mflop/s
mod2am: declarations

Approach
Ideally we would use a block-cyclic distribution; use BlockDist for now

declaration of distribution, domains and variables

const ADom: domain(2) = [1..m, 1..l];
const BDom: domain(2) = [1..l, 1..n];

const Dist = new Block(rank=2 bbox=[ADom.dim(1), BDom.dim(2)],
    tasksPerLocale=tasks);
const CDom: domain(2) distributed(Dist) = [ADom.dim(1), BDom.dim(2)];

const A: [ADom] real;
const B: [BDom] real;
var C: [CDom] real;
mod2am: timed loop

```chapel
mod2am: timed loop

timed loop

startTime = getCurrentTime();
for 1..nrep {
    forall (i, k) in C.domain do
        C(i, k) = + reduce [j in ADom.dim(2)] A(i, j) * B(j, k);
}
execTime = getCurrentTime() - startTime;
```
mod2am: timed loop, blocked

timed loop

config const: blkSize = ....;

startTime = getCurrentTime();
for 1..nrep {
    forall (i, k) in C.domain by (blkSize, blkSize) do
        for j in A.domain.dim(2) by blkSize { // leave this serial!
            dgemm(blkSize, blkSize, blkSize,
                A[row..#blkSize, j..#blkSize],
                B[j..#blkSize, col..#blkSize],
                C[row..#blkSize, col..#blkSize]);
        }
}
execTime = getCurrentTime() - startTime;
mod2am: results & issues

Results for 2048 x 2048 x 2048 on Cray XT5

On a single node **20 Mflop/s**; on \( \geq 2 \) nodes only **10 Mflop/s**.

Issues

- BlockCycDist not yet mature, use BlockDist \( \rightarrow \) possible mismatch between blkSize and data distribution
- As of Chapel V1.02 mod2am **should** scale ... but doesn’t. Why?

Comparisons

- Chapel V1.02: non-distributed multi-threaded version yields \( > 4900 \) Mflop/s
- Chapel V1.02: distributed version on 1 locale \( > 400 \) Mflop/s
- C: serial reference yields \( > 1800 \) Mflop/s
- C: MPI reference yields \( > ?? \) Mflop/s
HPCC: Random Access

Random Access kernel

Update random elements of a distributed vector/table from a random task.

Bild!
Random Access: declarations

Approach

Block-distributed table $T$ of length $m$. Updates stored in second block-distributed vector of length $m^2$.

declarations

```chapel
const TableDist = new Block(rank=1, bbox=[0..m-1]);
const UpdateDist = new Block(rank=1, bbox=[0..(m**2)-1]);
const TableSpace = domain(1) distributed(TableDist) = [0..m-1];
const Updates = domain(1) distributed(TableDist) = [0..(m**2)-1];

var T: [TableSpace] int;
```
Random Access: timed loop

timed loop

startTime = getCurrentTime();

forall (_, r) in (Updates, RAStream()) do
  on T[r] ^= r;

execTime = getCurrentTime() - startTime;
Random Access Performance

Performance of HPCC Random Access (Cray XT4)

Number of Locales

GUP/s

Chapel TPL=1
Chapel TPL=2
Chapel TPL=4
Chapel TPL=8
Random Access Efficiency on 32+ Nodes

Efficiency of HPCC Random Access on 32+ Locales (Cray XT4)

% Efficiency
(of scaled Chapel TPL=4 local GUP/s)

Number of Locales

Chapel TPL=1
Chapel TPL=2
Chapel TPL=4
Chapel TPL=8
MPI PPN=4
MPI No Buckets PPN=4
MPI+OpenMP TPN=4
Outline

1 Chapel in a Nutshell
   - Language concepts

2 Kernels
   - EuroBen kernels
   - HPCC kernels

3 Conclusions
Compiler & runtime

Releases

- V0.9: 2008(?), first public release
- V1.01: late 2009, SC09 version
- V1.02: late 2009, support for realms and DistGPU, technology preview

Pro

- compiler generates faster code for V1.0X
- BlockDist is multi-threaded and multi-node parallel
Compiler & runtime

**Con**

- partial reductions still missing
- still big penalty for distributed execution in general
- no work on CSR distribution, multi-node support for Cyclic and BlockCyclic still lacking

But improvement over time clearly visible!
# HPC Challenge: Chapel Entries (2008-2009)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>2008</th>
<th>2009</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Global STREAM</strong></td>
<td>1.73 TB/s (512 nodes)</td>
<td>10.8 TB/s (2048 nodes)</td>
<td>6.2x</td>
</tr>
<tr>
<td><strong>EP STREAM</strong></td>
<td>1.59 TB/s (256 nodes)</td>
<td>12.2 TB/s (2048 nodes)</td>
<td>7.7x</td>
</tr>
<tr>
<td><strong>Global RA</strong></td>
<td>0.00112 GUPs (64 nodes)</td>
<td>0.122 GUPs (2048 nodes)</td>
<td>109x</td>
</tr>
<tr>
<td><strong>Global FFT</strong></td>
<td>single-threaded single-node</td>
<td>multi-threaded multi-node</td>
<td>multi-node parallel</td>
</tr>
<tr>
<td><strong>Global HPL</strong></td>
<td>single-threaded single-node</td>
<td>multi-threaded single-node</td>
<td>single-node parallel</td>
</tr>
</tbody>
</table>

All timings on ORNL Cray XT4:
- 4 cores/node
- 8 GB/node
- no use of library routines
Language

Pro
- Chapel is beautiful, short readable code
- allows incremental writing of parallel code.
  Ideally, you write serial algorithm code and parallel declarations
- incremental optimization of data locality and affinity

Con
- syntax still not stable, in particular distribution declaration
- poor interoperability with C or other languages !!!
- little tools support, eg. profiling, performance analysis, debugging
HPC Challenge

- **Two classes of competition:**
  - Class 1: “best performance”
  - Class 2: “most productive”
    - **Judged on:** 50% performance 50% elegance
    - **Four recommended benchmarks:** STREAM, RA, FFT, HPL
    - **Use of library routines:** discouraged

- **Why you might care:**
  - provides an alternative to the top-500’s focus on peak performance

- **Historically:** the judges have “split the baby” for class 2
  - 2005: *tie*: Cray (MTA-2) and IBM (UPC)
  - 2006: *overall*: MIT (Cilk); *performance*: IBM (UPC); *elegance*: Mathworks (Matlab);
  - 2007: *research*: IBM (X10)
    - *industry*: Interactive Supercompting (Python/Star-P)
  - 2008: *performance*: IBM (UPC/X10)
    - *productive*: Cray (Chapel), IBM (UPC/X10), Mathworks (Matlab)
  - 2009: *performance*: IBM (UPC+X10)
    - *elegance*: Cray (Chapel)
Future

The Chapel teams aims at winning the 2010 HPC *performance* challenge (in addition to the ellegance).

Personally, I am going to keep an eye on Chapel.
Thank you for your attention

and

be welcome to ask questions!