Chapel: Data Parallelism

Brad Chamberlain

Outline

- Domains and Arrays
  - overview
  - arithmetic domains
  - domain roles
- Other Domain Types
- Data Parallel Operations
- Example Computations
Domains

- **domain**: a first-class index set
  - specifies size and shape of arrays
  - supports iteration, array operations
  - potentially distributed across machine

- Three main classes:
  - **arithmetic**: indices are Cartesian tuples
    - rectilinear, multidimensional
    - optionally strided and/or sparse
  - **associative**: indices serve as hash keys
    - supports hash tables, dictionaries
  - **opaque**: indices are anonymous
    - supports sets, graph-based computations

- Fundamental Chapel concept for data parallelism
- A generalization of ZPL’s *region* concept

Sample Arithmetic Domains

```chapel
var m = 4, n = 8;
var D: domain(2) = [1..m, 1..n];
```
Sample Arithmetic Domains

\[
\begin{align*}
\text{var } m &= 4, \ n = 8; \\
\text{var } D: \text{domain}(2) &= [1..m, 1..n]; \\
\text{var } \text{InnerD}: \text{subdomain}(D) &= [2..m-1, 2..n-1];
\end{align*}
\]

Domain Roles: Declaring Arrays

- **Syntax**

  \[
  \text{array-type:} \\
  \quad [\text{domain-expr}] \text{ type}
  \]

- **Semantics**
  * for each index in domain-expr, stores an element of type

- **Example**

  \[
  \begin{align*}
  \text{var } A, B: [D] \text{ real;}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Revisiting our previous array declarations:}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{var } A: [1..3] \text{ int; // creates an anonymous domain [1..3]}
  \end{align*}
  \]
Domain Roles: Supporting Iteration

- **Syntax**
  
  \[
  \text{for identifier in domain-expr loop-body} \\
  \text{forall identifier in domain-expr loop-body}
  \]

- **Semantics**
  
  - *for* – same as previous for loops we’ve seen; indices are const
  - *forall* – asserts the loop iterations can/should be executed in parallel  
    - also that they are serializable (can be run by a single task)

- **Example**

  ```chapel
  forall (i,j) in InnerD do
    A(i,j) = i + j/10.0;

  for ind in InnerD { write(A(ind), " "); }
  ```

- **Output**

  
  2.2 2.3 2.4 2.5 2.6 2.7 3.2 3.3 3.4 3.5 3.6 3.7

Other forall loop forms

- Forall loops also support...

  ...an expression-based form:

  ```chapel
  var A: [D] real = forall (i,j) in D do i + j/10.0;
  ```

  ...a symbolic shorthand:

  ```chapel
  [(i,j) in D] A(i,j) = i + j/10.0;
  ```

  ...and a sugar that combines it with the array type declaration syntax:

  ```chapel
  var A: [(i,j) in D] real = i + j/10.0;
  ```
Loops and Parallelism

- **for loops**: one task executes all iterations
- **forall loops**: some number of tasks executes the iterations
  - as determined by the iterator expression controlling the loop
    - for domains/arrays, specified as part of its distribution
    - for other objects/iterators, author specifies using task parallelism
- **coforall loops**: one task per iteration

Domain Roles: Domain Slicing

- **Syntax**
  
  \[
  \text{domain-slice:} \\
  \text{domain-expr}[\text{domain-expr}]
  \]

- **Semantics**
  - evaluates to the intersection of the two domains
- **Example**
  
  ```chapel
cost SubD: subdomain(D) = D[2.., ..7];
  ```
Domain Roles: Array Slicing

- Syntax
  
  ```plaintext
  array-slice:
  array-expr[domain-expr]
  ```

- Semantics
  - evaluates to the sub-array referenced by `domain-expr`
  - `domain-expr`'s indices must be in-bounds

- Example
  ```plaintext
  A[InnerD] = B[InnerD];
  ```

---

Domain Roles: Array Reallocation

- Semantics
  - re-assigning a domain's index set causes its arrays to be reallocated
  - array values are preserved for indices that remain in the index set
  - elements for new indices are initialized to the type’s default value

- Example
  ```plaintext
  D = [1..2*m, 1..2*n];
  ```
Outline

- Domains
- Other Domain Types
  - strided
  - sparse
  - associative
  - opaque
- Data Parallel Operations
- Example Computations

Other Domain Types

Domain indices can be...

- integer tuples...
- dense
- strided
- sparse
- graphs
- associative
- ...anonymous...
- ...or arbitrary values.

"George"
"John"
"Thomas"
"James"
"Andrew"
"Martin"
"William"
Domain Declarations

```chapel
var DnsDom: domain(2) = [1..10, 0..24],
StrDom: subdomain(DnsDom) = DnsDom by (2,4),
SpsDom: subdomain(DnsDom) = genIndices();
```

Array Declarations

```chapel
var DnsArr: [DnsDom] complex,
StrArr: [StrDom] real(32),
SpsArr: [SpsDom] real;
```

Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.
Data Parallelism: Domain Iteration

All domain types support iteration...

```chapel
forall ij in StrDom {
    DnsArr(ij) += SpsArr(ij);
}
```

Data Parallelism: Array Slicing

...array slicing...

```chapel
DnsArr[StrDom] += SpsArr[StrDom];
```
Data Parallelism: Array Reallocation

...array reallocation (as well as other domain/array operations)
StrDom = DnsDom by (2,2);
SpsDom += genEquator();
### Associative Domains and Arrays

```chapel
var People: domain(string);
var Age: [People] int,
    Birthdate: [People] string;

People += "john";
Age("john") = 60;
Birthdate("john") = "12/11/1943";
...
forall person in People {
    if (Birthdate(person) == today) {
        Age(person) += 1;
    }
}
```

### Opaque Domains and Arrays

```chapel
var Vertices: domain(opaque);
for i in (1..5) {
    Vertices.create();
}
var AV, BV: [Vertices] real;
```
Opaque Domains and Arrays

```chapel
var Vertices: domain(opaque);
var left, right: [Vertices] index(Vertices);
var root: index(Vertices);
root = Vertices.create();
left(root) = Vertices.create();
right(root) = Vertices.create();
left(right(root)) = Vertices.create();
```

conceptually:

more precisely:

Outline

- Domains
- Other Domain Types
- Data Parallel Operations
  - promotion
  - reductions and scans
- Example Computations
Promotion

- Functions/operators expecting scalar values can also take...
  
  ...arrays, causing each element to be passed in

  \[
  A = \sin(B); \\
  B = 2 * A;
  \]

  ...domains, causing each index to be passed in

  ```
  def foo(x: (int, int)) { ... } \\
  foo(SpsDom);  // calls foo once per index in SpsDom
  ```

- When multiple arguments are promoted, calls may use...
  
  ...zippered promotion:

  \[
  X = \text{pow}(A, B);  // X is 2D; X(i,j) = \text{pow}(A(i,j), B(i,j))
  \]

  ...tensor promotion:

  \[
  Y = \text{pow}[A, B];  // Y is 2x2D;  \\
  // Y(i,j)(k,l) = \text{pow}(A(i,j), B(k,l))
  \]

Promotion and Parallelism

- Promoted functions/operators are executed in parallel
  
  - as if a zipzer/tensor forall loop was implementing the promotion

  ```
  A = \sin(B);  \\
  B = 2 * A;  \\
  foo(SpsDom);  \\
  X = \text{pow}(A, B); \\
  X = \text{pow}[A, B];
  ```
Reductions

- Syntax

  \[
  \text{reduce-expr:} \quad \text{reduce-type reduce iterable-expr}
  \]

- Semantics
  - combines elements generated by \textit{iteratable-expr} using \textit{reduce-type}
  - \textit{reduce-type} may be one of several built-in operators, or user-defined

- Examples

  \[
  \text{tot} = \text{reduce } A; \quad // \text{tot is the sum of all elements in } A
  \text{big} = \text{max reduce } [i \text{ in InnerD} ] \text{abs}(A(i) + B(i));
  \]

- Future work:
  - support for partial reductions to reduce only a subset of an array's dimensions

Scans

- Syntax

  \[
  \text{scan-expr:} \quad \text{scan-type scan iterable-expr}
  \]

- Semantics
  - combines elements generated by \textit{iteratable-expr} using \textit{scan-type},
    generating partial results along the way
  - \textit{scan-type} may be one of several built-in operators, or user-defined

- Examples

  \[
  \text{var A, B, C: [1..5] real;}
  A = 1.1; \quad // A is: 1.1 1.1 1.1 1.1 1.1
  B = + \text{scan } A; \quad // B is: 1.1 2.2 3.3 4.4 5.5
  B(3) = -B(3); \quad // B is: 1.1 2.2 -3.3 4.4 5.5
  C = \text{min scan } B; \quad // C is: 1.1 1.1 -3.3 -3.3 -3.3
  \]
Reduction/Scan operators

- Built-in:
  - $+, -, *, /, \&, ^, \&\&, ||, \min, \max$: do the obvious things
  - $\minloc, \maxloc$: generate a tuple of the min/max and its index

- User-defined:
  - user must define a class that supports a number of methods to:
    - generate a new identity state value
    - combine the state element with a new element
    - combine two state elements
    - generate an output result
    - ...
  - the compiler generates a code template to compute the operation in parallel, utilizing the user’s class methods
  - for more information, see:

Outline

- Domains
- Other Domain Types
- Data Parallel Operations
- Example Computations
  - Jacobi iteration
  - Multigrid
Jacobi Iteration in Pictures

\[ \sum \left( \begin{array}{c}
1.0 \\
\vdots \\
1.0 
\end{array} \right) \div 4 \]

repeat until max change < \varepsilon

Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;

do {
    [(i,j) in D] Temp(i,j) = (A(i-1,j) + A(i+1,j)
        + A(i,j-1) + A(i,j+1)) / 4;

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Brad Chamberlain, Steve Deitz,
Samuel Figueroa, David Iten; Cray Inc.
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;
do {
    var [i,j] in D
    Temp(i,j) = (A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1)) / 4;
    var delta = max reduce abs(A(D) - Temp(D));
    A(D) = Temp(D);
}
while (delta > epsilon);
writeln(A);
```

Declare program arguments

- `const` \(\Rightarrow\) can't change values after initialization
- `config` \(\Rightarrow\) can be set on executable command-line

```
prompt> jacobi --n=10000 --epsilon=0.0001
```

note that no types are given; inferred from initializer
- `n` \(\Rightarrow\) integer (current default, 32 bits)
- `epsilon` \(\Rightarrow\) floating-point (current default, 64 bits)

Declare domains (first class index sets)

- `domain(2)` \(\Rightarrow\) 2D arithmetic domain, indices are integer 2-tuples
- `subdomain(P)` \(\Rightarrow\) a domain of the same type as P whose indices are guaranteed to be a subset of P's

`exterior` \(\Rightarrow\) one of several built-in domain generators
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
    [ (i,j) in D ] Temp(i,j) = (A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1)) / 4;
    var delta = max reduce abs(A(D) - Temp(D));
    A(D) = Temp(D);
}
while (delta > epsilon);

writeln(A);
```

Declare arrays

- `var` can be modified throughout its lifetime
- `T` declares variable to be of type `T`
- `[D] T` array of size `D` with elements of type `T`
- *(no initializer)* values initialized to default value (0.0 for reals)

Set Explicit Boundary Condition

- indexing by domain ⇒ slicing mechanism
- array expressions ⇒ parallel evaluation
Jacobi Iteration in Chapel

**Compute 5-point stencil**

\[
((i, j) \in D) \Rightarrow \text{parallel forall expression over } D\text{'s indices, binding them to new variables } i \text{ and } j
\]

**Note:** since \((i, j) \in D \text{ and } D \subseteq \text{BigD} \text{ and Temp: [BigD]}\)
\(\Rightarrow \text{no bounds check required for Temp}(i,j)
\text{with compiler analysis, same can be proven for A's accesses}

\[
\sum \left( \begin{array}{c}
A(i-1,j) + A(i+1,j) \\
A(i, j-1) + A(i, j+1)
\end{array} \right) / 4
\]

\[
\text{const delta = max reduce abs(A[D] - Temp[D]);}
\]

\[
A[D] = \text{Temp}[D];
\]

\[
\text{while (delta > epsilon);}\]

\[
\text{writeln(A);}
\]

---

Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],

var A, Temp : [BigD] real;
A[LastRow] = 1.0;
do {
    [(i, j) in D] Temp(i, j) = (A(i-1,j) + A(i+1,j) \\
                      + A(i,j-1) + A(i,j+1)) / 4;
    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

---

**Compute maximum change**

**op reduce** \(\Rightarrow\) collapse aggregate expression to scalar using **op**

**Promotion:** abs() and – are scalar operators, automatically promoted to work with array operands

```chapel
do {
    [(i, j) in D] Temp(i, j) = (A(i-1,j) + A(i+1,j) \\
                      + A(i,j-1) + A(i,j+1)) / 4;
    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1],
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do 
    
    [[(i,j) in D] Temp(i,j) = (A(i-1,j) + A(i+1,j)
        + A(i,j-1) + A(i,j+1)) / 4;

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Write array to console

If written to a file, parallel I/O would be used

Copy data back & Repeat until done

uses slicing and whole array assignment
standard do...while loop construct
Jacobi Iteration in Chapel

```chapel
cfg const n = 6,
    epsilon = 1.0e-5;

const BigD: domain(2) = [0..n+1, 0..n+1] dist Block,
    D: subdomain(BigD) = [1..n, 1..n],
    LastRow: subdomain(BigD) = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;

do {
    [(i,j) in D] Temp(i,j) = (A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1)) / 4;
    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

dln(A);
```
Multigrid Example

V: input array

U: hierarchical work arrays

R: hierarchical work arrays

Hierarchical Arrays

conceptually:

level 0

level 1

level 2

level 3

dense indexing:

(1:8,1:8)

(1:4,1:4)

(1:2,1:2)

(1:1,1:1)

strided indexing:

(1:8:1,1:8:1)

(1:8:2,1:8:2)

(1:8:4,1:8:4)

(1:8:8,1:8:8)
Hierarchical Arrays

conceptually:

dense indexing:

(1:8,1:8)

(1:4,1:4)

(1:2,1:2)

(1:1,1:1)

strided indexing:

(1:8:1,1:8:1)

(2:8:2,2:8:2)

(4:8:4,4:8:4)

(8:8:8,8:8:8)

Hierarchical Array Declarations in Chapel

```chapel
config const n = 1024,
    numLevels = lg2(n);

const Levels = [0..#numLevels];
const ProblemSpace = [1..n, 1..n, 1..n];

var V: [ProblemSpace] real;

const HierSpace: [lvl in Levels] subdomain(ProblemSpace) = ProblemSpace by -2**lvl;

var U, R: [lvl in Levels] [HierSpace(lvl)] real;
```
Overview of NAS MG

MG’s Timed Portion

Configurations

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Size</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>32³</td>
<td>4 iterations</td>
</tr>
<tr>
<td>W</td>
<td>64³</td>
<td>40 iterations</td>
</tr>
<tr>
<td>A</td>
<td>256³</td>
<td>4 iterations</td>
</tr>
<tr>
<td>B</td>
<td>256³</td>
<td>20 iterations</td>
</tr>
<tr>
<td>C</td>
<td>512³</td>
<td>20 iterations</td>
</tr>
<tr>
<td>D</td>
<td>1024³</td>
<td>50 iterations</td>
</tr>
</tbody>
</table>
MG’s projection/interpolation cycle

NAS MG: \( rprj3 \) stencil
Multigrid: Stencils in Chapel

- Can write them out explicitly, as in Jacobi...

```chapel
def rprj3(S, R) {
    const w: [0..3] real = (0.5, 0.25, 0.125, 0.0625);
    const Rstr = R.stride;

    forall ijk in S.domain do
        S(ijk) = w(0) * R(ijk)
            + w(1) * (R(ijk+Rstr*(1,0,0)) + R(ijk+Rstr*(-1,0,0))
                        + R(ijk+Rstr*(0,1,0)) + R(ijk+Rstr*(0,-1,0))
                        + R(ijk+Rstr*(0,0,1)) + R(ijk+Rstr*(0,0,-1)))
            + w(2) * (R(ijk+Rstr*(1,1,0)) + R(ijk+Rstr*(1,-1,0))
                        + R(ijk+Rstr*(1,0,1)) + R(ijk+Rstr*(1,0,-1))
                        + R(ijk+Rstr*(-1,0,1)) + R(ijk+Rstr*(-1,0,-1))
                        + R(ijk+Rstr*(0,1,1)) + R(ijk+Rstr*(0,1,-1))
                        + R(ijk+Rstr*(0,-1,1)) + R(ijk+Rstr*(0,-1,-1)))
            + w(3) * (R(ijk+Rstr*(1,1,1)) + R(ijk+Rstr*(1,1,-1))
                        + R(ijk+Rstr*(1,-1,1)) + R(ijk+Rstr*(1,-1,-1))
                        + R(ijk+Rstr*(-1,1,1)) + R(ijk+Rstr*(-1,1,-1))
                        + R(ijk+Rstr*(-1,-1,1)) + R(ijk+Rstr*(-1,-1,-1)));
}
```

Multigrid: Stencils in Chapel

- ...or, note that a stencil is simply a reduction over a small subarray expression
- Thus, stencils can be written in a “syntactically scalable” way using reductions:

```chapel
def rprj3(S, R) {
    const Stencil: domain(3) = [-1..1, -1..1, -1..1], // 27-points
                        w: [0..3] real = (0.5, 0.25, 0.125, 0.0625), // 4 weights
                        w3d = [(i,j,k) in Stencil] w((i!=0) + (j!=0) + (k!=0));

    forall ijk in S.domain do
        S(ijk) = + reduce [off in Stencil]
                        (w3d(off) * R(ijk + R.stride*off));
}
```
NAS MG `rprj3` stencil in Fortran+MPI

Questions?