Adaptive Mesh Refinement in Chapel: An Acid Test for High Productivity Programming

Jonathan Claridge
Cray Inc. Tech Forum
December 2, 2010
Overview

Goal

- Initially modest: Isolate “motifs” of adaptive mesh refinement, suitable for benchmarking
Overview

Goal
• Initially modest: Isolate “motifs” of adaptive mesh refinement, suitable for benchmarking

Results
• Fully-functional, dimension-independent AMR framework in under 4 months, with no prior Chapel experience
Overview

Goal

- Initially modest: Isolate “motifs” of adaptive mesh refinement, suitable for benchmarking

Results

- Fully-functional, **dimension-independent** AMR framework in under 4 months, with no prior Chapel experience
- Code is drastically shorter than existing libraries:

<table>
<thead>
<tr>
<th>Language</th>
<th>Parallelism</th>
<th>SLOC$^1$</th>
<th>Tokens</th>
<th>Relative size (tokens)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapel (any D)</td>
<td>Shared mem.</td>
<td>1988</td>
<td>13783</td>
<td>1</td>
</tr>
<tr>
<td>Fortran (2D+3D)$^2$</td>
<td>Serial</td>
<td>16562</td>
<td>151992</td>
<td>11.03</td>
</tr>
<tr>
<td></td>
<td>2D</td>
<td>8297</td>
<td>71639</td>
<td>5.20</td>
</tr>
<tr>
<td></td>
<td>3D</td>
<td>8265</td>
<td>80353</td>
<td>5.83</td>
</tr>
<tr>
<td>C/C++ (any D)$^3$</td>
<td>Dist. mem.</td>
<td>40200</td>
<td>261427</td>
<td>20.22</td>
</tr>
</tbody>
</table>

$^1$ source ines of code, $^2$ AMRClaw, $^3$ Chombo BoxTools+AMRTools
Adaptive Mesh Refinement (AMR)

Provide enhanced resolution in regions where features are poorly resolved
Adaptive Mesh Refinement (AMR)

Provide enhanced resolution in regions where features are poorly resolved
Adaptive Mesh Refinement (AMR)

Provide enhanced resolution in regions where features are poorly resolved
AMR Terminology

grid
AMR Terminology

(refinement) level
AMR Terminology

- coarse level
- fine level
AMR Terminology

coarse level

fine level
AMR Terminology

AMR hierarchy

top level
config param dimension = 2;

const dimensions = 1..dimension;
Spatial dimension

May specify value with a command-line flag

```plaintext
config param dimension = 2;
const dimensions = 1..dimension;
```
Spatial dimension

May specify value with a command-line flag

Value known at compile-time

```c
config param dimension = 2;
const dimensions = 1..dimension;
```
Spatial dimension

May specify value with a command-line flag

Value known at compile-time

Default value; type is elided

```c
config param dimension = 2;

const dimensions = 1..dimension;
```
Spatial dimension

May specify value with a command-line flag

`config param dimension = 2;`

Value known at compile-time

`const dimensions = 1..dimension;`

Default value; type is elided

Not modified after assignment; use `var` otherwise
Spatial dimension

May specify value with a command-line flag

Value known at compile-time

Default value; type is elided

```c
config param dimension = 2;

const dimensions = 1..dimension;
```

Not modified after assignment; use `var` otherwise

**Range:** Arithmetic sequence of integers

- Supports iteration:
  ```c
  for d in dimensions do ... 
  ```
Grids

Basic geometry described by tuples

```c
const x_low, x_high: dimension*real;
```
Grids

Basic geometry described by tuples

```cpp
const x_low, x_high: dimension*real;
const i_low, i_high: dimension*int;
```
Grids

Basic geometry described by tuples

```c
const x_low, x_high: dimension*real;
const i_low, i_high: dimension*int;
const n_cells: dimension*int;
```

```plaintext
n_cells(1) = 4
n_cells(2) = 5
```
Grids

Basic geometry described by tuples

\[
\begin{align*}
\text{const } \& x_{\text{low}}, \ x_{\text{high}}: \text{dimension}^*\text{real}; \\
\text{const } \& i_{\text{low}}, \ i_{\text{high}}: \text{dimension}^*\text{int}; \\
\text{const } \& n_{\text{cells}}: \text{dimension}^*\text{int}; \\
\text{const } \& n_{\text{ghost}_{\text{cells}}}: \text{dimension}^*\text{int}; \\
\end{align*}
\]

\[
\begin{align*}
n_{\text{ghost}_{\text{cells}}}(1) &= 1 \\
n_{\text{ghost}_{\text{cells}}}(2) &= 2
\end{align*}
\]
Grids: Indexing

Conventional approach – number the cells sequentially

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
Grids: Indexing

Conventional approach – number the cells sequentially

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>2</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>1</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

```
const cells = [1..4, 1..3];
```
Grids: Indexing

Conventional approach – number the cells sequentially

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

const cells = [1..4, 1..3];

**Arithmetic domain**: Multidimensional index space

- Supports storage:
  ```
  var my_array: [cells] real;
  ```
- Supports (parallel) iteration:
  ```
  for(all) cell in cells do ...
  ```
- The reason Chapel is so useful for AMR
Grids: Indexing

Conventional approach – number the cells sequentially

```
const cells = [1..4, 1..3];
```

Problem with conventional indexing:

- How are interfaces and vertices indexed?
Grids: Indexing

Conventional approach – number the cells sequentially

```
const cells = [1..4, 1..3];
```

Problem with conventional indexing:
• How are interfaces and vertices indexed?

Many objects will have the same indices
Grids: Indexing

Modified approach – denser index space
Grids: Indexing

Modified approach – denser index space

```
const cells = [1..7 by 2, 1..5 by 2];
```
Grids: Indexing

Modified approach – denser index space

```
const cells = [1..7 by 2, 1..5 by 2];
const x_interfaces = [0..8 by 2, 1..5 by 2];
```
Grids: Indexing

Modified approach – denser index space

```plaintext
const cells = [1..7 by 2, 1..5 by 2];
const x_interfaces = [0..8 by 2, 1..5 by 2];
const y_interfaces = [1..7 by 2, 0..6 by 2];
```
Grids: Indexing

Modified approach – denser index space

const cells = [1..7 by 2, 1..5 by 2];
const x_interfaces = [0..8 by 2, 1..5 by 2];
const y_interfaces = [1..7 by 2, 0..6 by 2];
const vertices = [0..8 by 2, 0..6 by 2];
Grids: Indexing

Modified approach – denser index space

\[
\begin{array}{cccc}
6 & & & \\
5 & & & \\
4 & & & \\
\end{array}
\]

**Strided domains**

- Array and iteration syntax are **unchanged**
- Chapel helps us describe the physical problem **much more effectively**

```plaintext
const cells = [1..7 by 2, 1..5 by 2];
const x_interfaces = [0..8 by 2, 1..5 by 2];
const y_interfaces = [1..7 by 2, 0..6 by 2];
const vertices = [0..8 by 2, 0..6 by 2];
```
Levels

Essentially a union of grids
Levels

Essentially a union of grids

\texttt{var grids: domain(Grid);}
Levels

Essentially a union of grids

```plaintext
var grids: domain(Grid);
```

Associative domain

- A list of indices of *any* type
- Array and iteration syntax remains *unchanged*
Levels: Overlaps

Each grid has a layer of **ghost cells** to facilitate data transfer.
Levels: Overlaps

Each grid has a layer of **ghost cells** to facilitate data transfer

```cpp
const extended_cells = cells.expand(n_ghost_cells);
```
Levels: Overlaps

Each grid has a layer of **ghost cells** to facilitate data transfer

```cpp
const extended_cells = cells.expand(n_ghost_cells);
```

Calculating the **overlaps**
Levels: Overlaps

Each grid has a layer of **ghost cells** to facilitate data transfer

```javascript
const extended_cells =
cells.expand(n_ghost_cells);
```

Calculating the **overlaps**

```javascript
var neighbors: domain(Grid);
var overlaps: domain(dimension,stridable=true);
for sibling in parent_level.grids {
  var overlap = extended_cells(sibling.cells);
  if overlap.numIndices > 0 && sibling != this {
    neighbors.add(sibling);
    overlaps(sibling) = overlap;
  }
}
```

Declare associative domain to store neighbors; initializes to empty.
Levels: Overlaps

Each grid has a layer of **ghost cells** to facilitate data transfer

```cpp
const extended_cells = cells.expand(n_ghost_cells);
```

Calculating the **overlaps**

```cpp
var neighbors: domain(Grid);
var overlaps: [neighbors] domain(dimension, stridable=true);

for sibling in parent_level.grids {
    var overlap = extended_cells(sibling.cells);
    if overlap.numIndices > 0 && sibling != this {
        neighbors.add(sibling);
        overlaps(sibling) = overlap;
    }
}
```

An array of domains; stores one domain for each neighbor.
New space allocated as `neighbors` grows.
Levels: Overlaps

Each grid has a layer of **ghost cells** to facilitate data transfer

```
const extended_cells = cells.expand(n_ghost_cells);
```

Calculating the **overlaps**

```
var neighbors: domain(Grid);
var overlaps: [neighbors] domain(dimension,stridable=true);

for sibling in parent_level.grids {
    var overlap = extended_cells(sibling.cells);
    if overlap.numIndices > 0 && sibling != this {
        neighbors.add(sibling);
        overlaps(sibling) = overlap;
    }
}
```

Loop over all grids on the same level, checking for neighbors.
Levels: Overlaps

Each grid has a layer of **ghost cells** to facilitate data transfer

```javascript
const extended_cells = cells.expand(n_ghost_cells);
```

Calculating the **overlaps**

```javascript
var neighbors: domain(Grid);
var overlaps: [neighbors] domain(dimension,stridable=true);

for sibling in parent_level.grids {
    var overlap = extended_cells(sibling.cells);
    if overlap.numIndices > 0 && sibling != this {
        neighbors.add(sibling);
        overlaps(sibling) = overlap;
    }
}
```

**Computes intersection of the domains**

`extended_cells and sibling.cells`. 
Levels: Overlaps

Each grid has a layer of **ghost cells** to facilitate data transfer

```cpp
const extended_cells = cells.expand(n_ghost_cells);
```

Calculating the **overlaps**

```cpp
var neighbors: domain(Grid);
var overlaps: [neighbors] domain(dimension, stridable=true);

for sibling in parent_level.grids {
  var overlap = extended_cells(sibling.cells);

  if overlap.numIndices > 0 && sibling != this {
    neighbors.add(sibling);
    overlaps(sibling) = overlap;
  }
}
```

If overlap is nonempty, and sibling is distinct from this grid, then update stored data.
Levels: Interpolation

Remaining ghost cells will receive data interpolated from the coarser level.
Levels: Interpolation

Remaining ghost cells will receive data interpolated from the coarser level

- Region is naturally described by a union of domains
Levels: Interpolation

Remaining ghost cells will receive data interpolated from the coarser level

- Region is naturally described by a union of domains
- Region is naturally calculated by subtraction of domains
Levels: Interpolation

Remaining ghost cells will receive data interpolated from the coarser level

- Region is naturally described by a union of domains
- Region is naturally calculated by subtraction of domains

New object: **MultiDomain**

- Stores a collection of domains
- Supports subtraction of domains with a set-minus interpretation
Levels: Interpolation

Remaining ghost cells will receive data interpolated from the coarser level

• Region is naturally described by a union of domains
• Region is naturally calculated by subtraction of domains

Computing the interpolation region
Levels: Interpolation

Remaining ghost cells will receive data interpolated from the coarser level

- Region is naturally described by a union of domains
- Region is naturally calculated by subtraction of domains

Computing the interpolation region

```java
var interp_region =
    new MultiDomain(dimension, stridable=true);
```

Declare a new MultiDomain; initializes to empty.

```java
for neighbor in neighbors do
    interp_region.subtract( overlaps(neighbor) );
```
Levels: Interpolation

Remaining ghost cells will receive data interpolated from the coarser level

• Region is naturally described by a union of domains
• Region is naturally calculated by subtraction of domains

Computing the interpolation region

```javascript
var interp_region =
    new MultiDomain(dimension,stridable=true);
interp_region.add( extended_cells );
interp_region.subtract( cells );
```

for neighbor in neighbors do
    interp_region.subtract( overlaps(neighbor) );

Fill with only the ghost cells.
Levels: Interpolation

Remaining ghost cells will receive data interpolated from the coarser level

• Region is naturally described by a union of domains
• Region is naturally calculated by subtraction of domains

Computing the interpolation region

```javascript
var interp_region =
    new MultiDomain(dimension, stridable=true);
interp_region.add( extended_cells );
interp_region.subtract( cells );
```

```javascript
for neighbor in neighbors do
    interp_region.subtract( overlaps(neighbor) );
```

Remove any regions of overlap with a neighboring grid.
Levels: Interpolation

Remaining ghost cells will receive data interpolated from the coarser level

- Region is naturally described by a union of domains

- This code is dimension-independent

- Underlying code for MultiDomains is relatively simple because domains do most of the hard work

```javascript
var interp_region =
    new MultiDomain(dimension,stridable=true);
interp_region.add( extended_cells );
interp_region.subtract( cells );

for neighbor in neighbors do
    interp_region.subtract( overlaps(neighbor) );
```
Regridding (very briefly)

- Primarily the work of a **partitioning algorithm**: Given a set of flags, find rectangles that cover them (Berger & Rigoutsos, 1991)
Regridding (very briefly)

- Primarily the work of a **partitioning algorithm**: Given a set of flags, find rectangles that cover them (Berger & Rigoutsos, 1991)

- More rectangles and unions of rectangles – naturally described by domains and MultiDomains
Regridding (very briefly)

- Primarily the work of a **partitioning algorithm**: Given a set of flags, find rectangles that cover them (Berger & Rigoutsos, 1991)

- More rectangles and unions of rectangles – naturally described by domains and MultiDomains

- Dimension-independent code is ≈200 lines
Parallelism

Shared-memory parallelism

- Genuinely trivial:

  ```plaintext
  forall cell in grid.cells do ...
  ```
Parallelism

Shared-memory parallelism

- Genuinely trivial:
  
  ```
  forall cell in grid.cells do ...
  ```

Distributed-memory parallelism

- Not implemented yet, but not difficult once grids have been mapped to processors
  
  ```
  forall grid in level.grids { 
    forall cell in grid.cells do ...
  }
  ```
Performance

- Chapel performance in general is an open issue
Performance

- Chapel performance in general is an open issue
- Will at least test:
  - What fraction of time is spent number crunching?
  - How good is scaling on a multicore machine?
Summary

• The challenges of AMR are mostly gymnastics with rectangles. Chapel makes this immensely easier.
Summary

• The challenges of AMR are mostly gymnastics with rectangles. Chapel makes this immensely easier.

• Dimension-independence and parallelism are so simple that you can almost forget about them.
Summary

- The challenges of AMR are mostly gymnastics with rectangles. Chapel makes this immensely easier.

- Dimension-independence and parallelismism are so simple that you can almost forget about them.

- Massive reductions in code size:

<table>
<thead>
<tr>
<th>Language</th>
<th>Parallelism</th>
<th>SLOC$^1$</th>
<th>Tokens</th>
<th>Relative size (tokens)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapel (any D)</td>
<td>Shared mem.</td>
<td>1988</td>
<td>13783</td>
<td>1</td>
</tr>
<tr>
<td>Fortran (2D+3D)$^2$</td>
<td>Serial</td>
<td>16562</td>
<td>151992</td>
<td>11.03</td>
</tr>
<tr>
<td></td>
<td>2D</td>
<td>8297</td>
<td>71639</td>
<td>5.20</td>
</tr>
<tr>
<td></td>
<td>3D</td>
<td>8265</td>
<td>80353</td>
<td>5.83</td>
</tr>
<tr>
<td>C/C++ (any D)$^3$</td>
<td>Dist. mem.</td>
<td>40200</td>
<td>261427</td>
<td>20.22</td>
</tr>
</tbody>
</table>

$^1$ source ines of code, $^2$ AMRClaw, $^3$ Chombo BoxTools+AMRTools