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

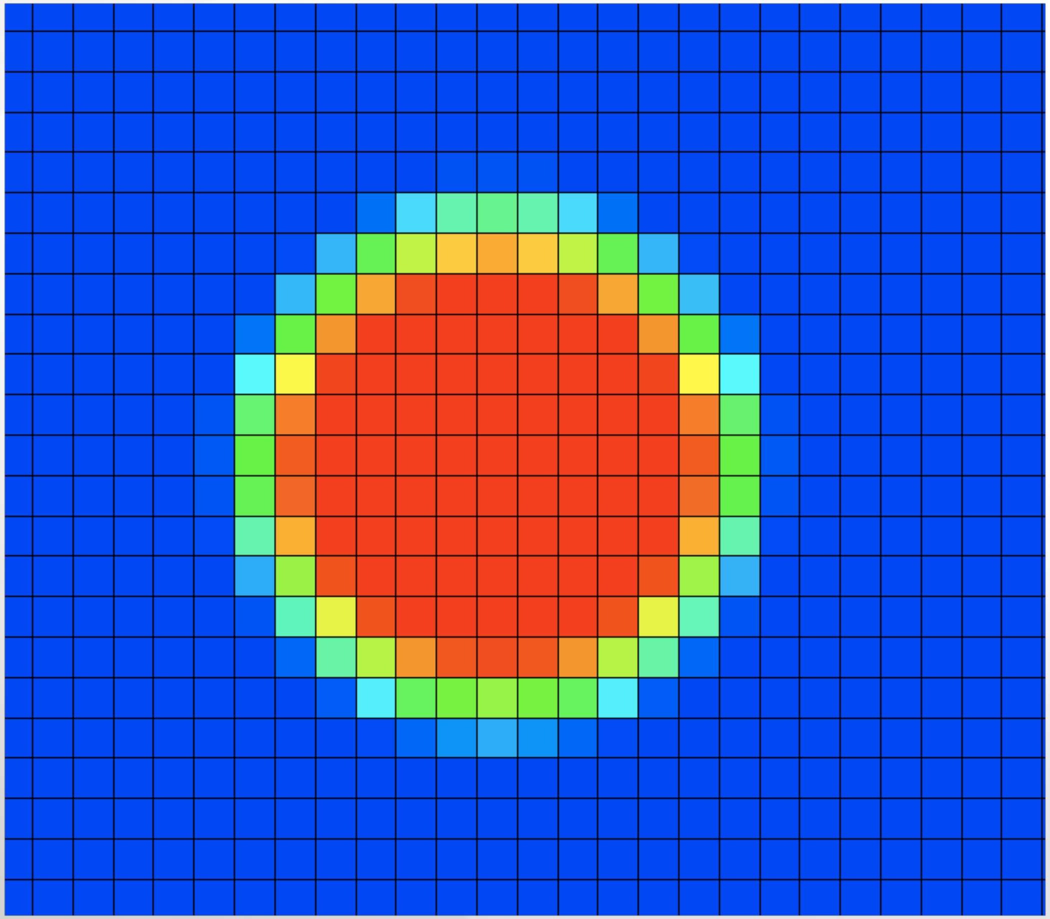
- Fully-functional, **dimension-independent** AMR framework in under 4 months, with no prior Chapel experience
- Code is drastically shorter than existing libraries:

Language	Parallelism	SLOC[1]	Tokens	Relative size (tokens)
Chapel (any D)	Shared mem.	1988	13783	1
Fortran (2D+3D)[2]	Serial	16562	151992	11.03
2D		8297	71639	5.20
3D		8265	80353	5.83
C/C++ (any D)[3]	Dist. mem.	40200	261427	20.22

[1] source ines of code, [2] AMRClaw, [3] Chombo BoxTools+AMRTools

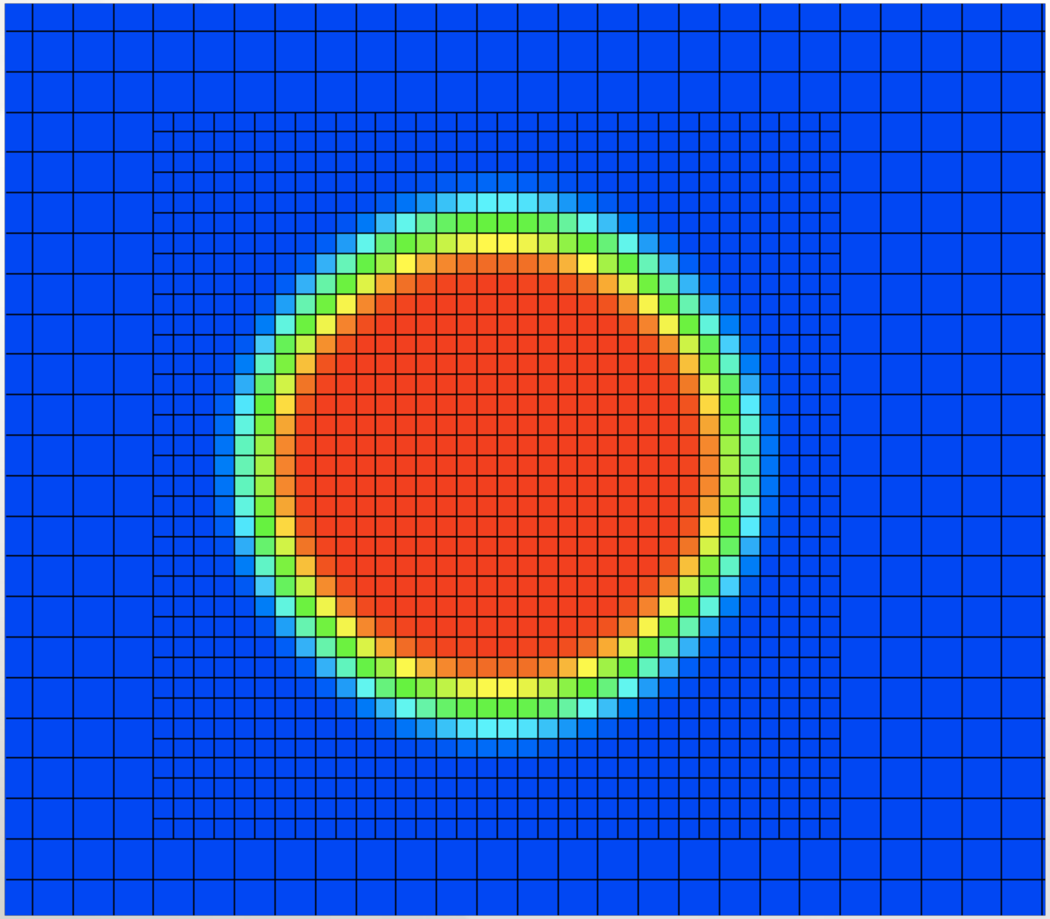
Adaptive Mesh Refinement (AMR)

Provide enhanced resolution in regions where features are poorly resolved



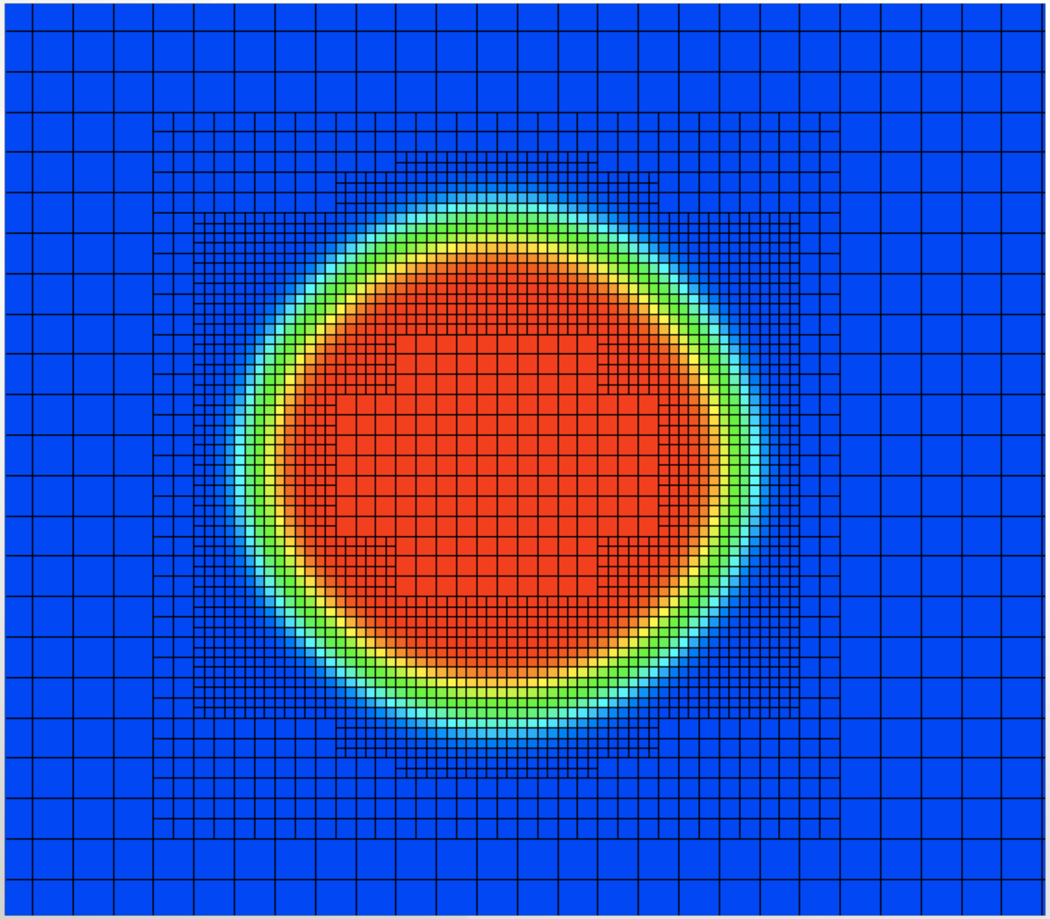
Adaptive Mesh Refinement (AMR)

Provide enhanced resolution in regions where features are poorly resolved

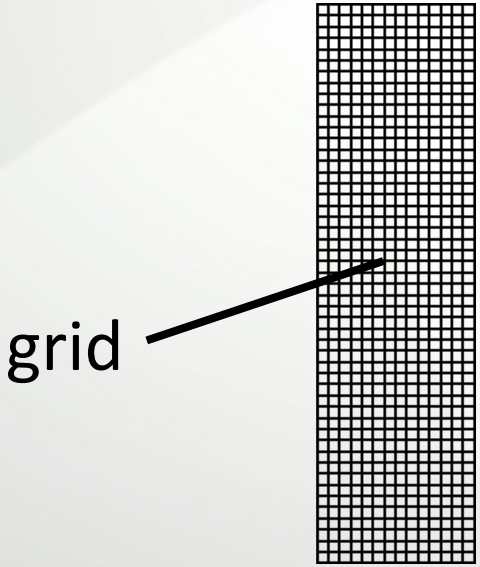


Adaptive Mesh Refinement (AMR)

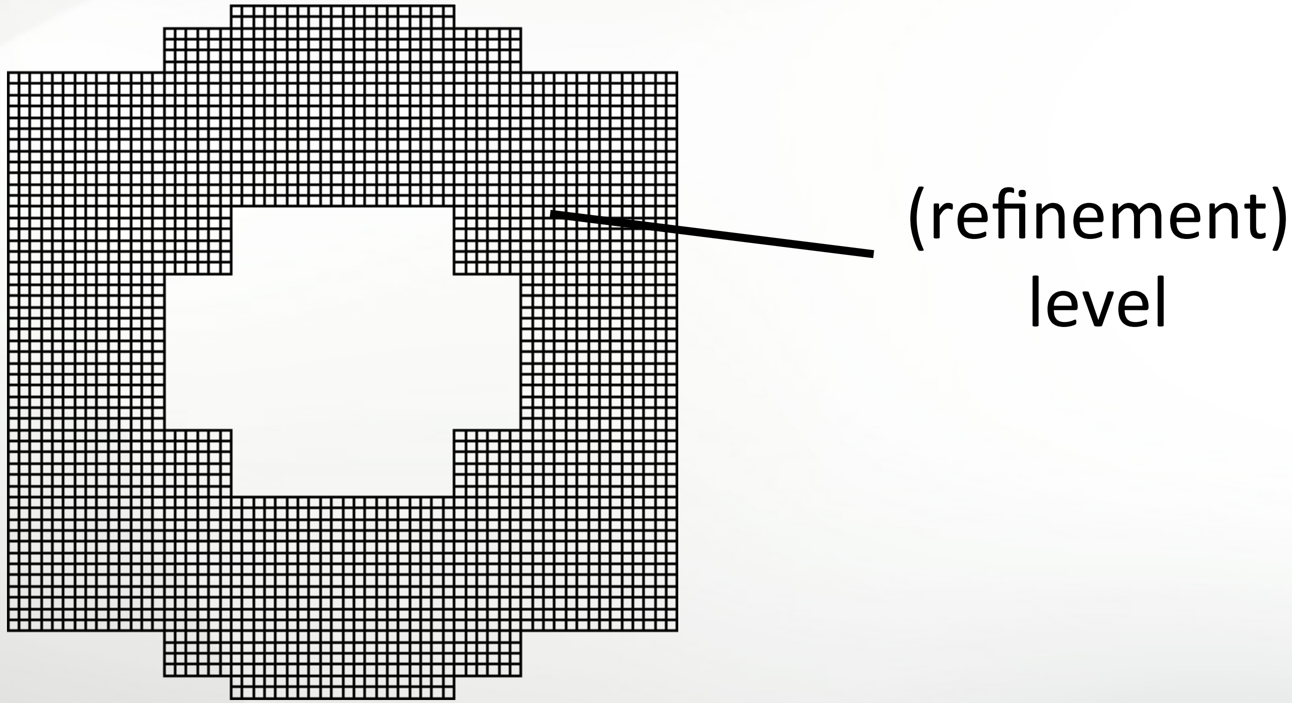
Provide enhanced resolution in regions where features are poorly resolved



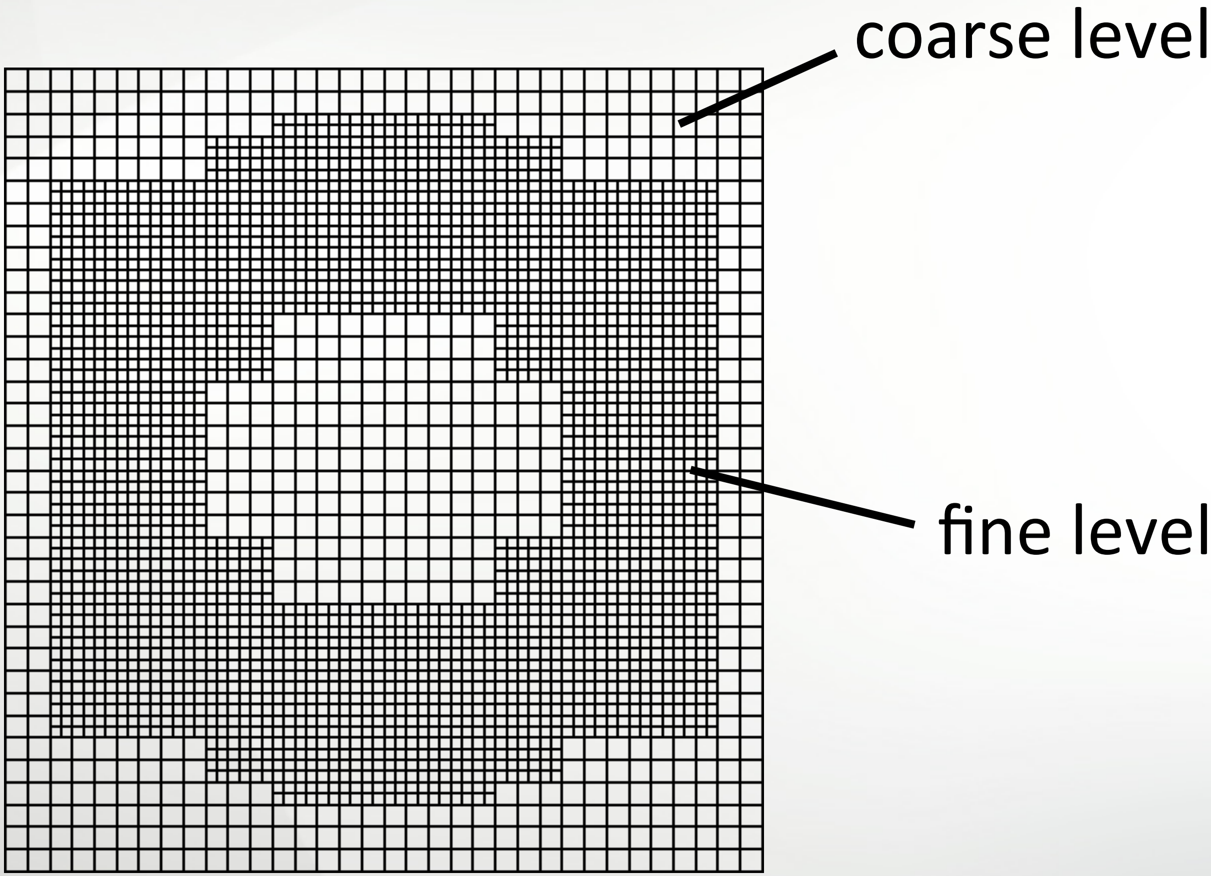
AMR Terminology



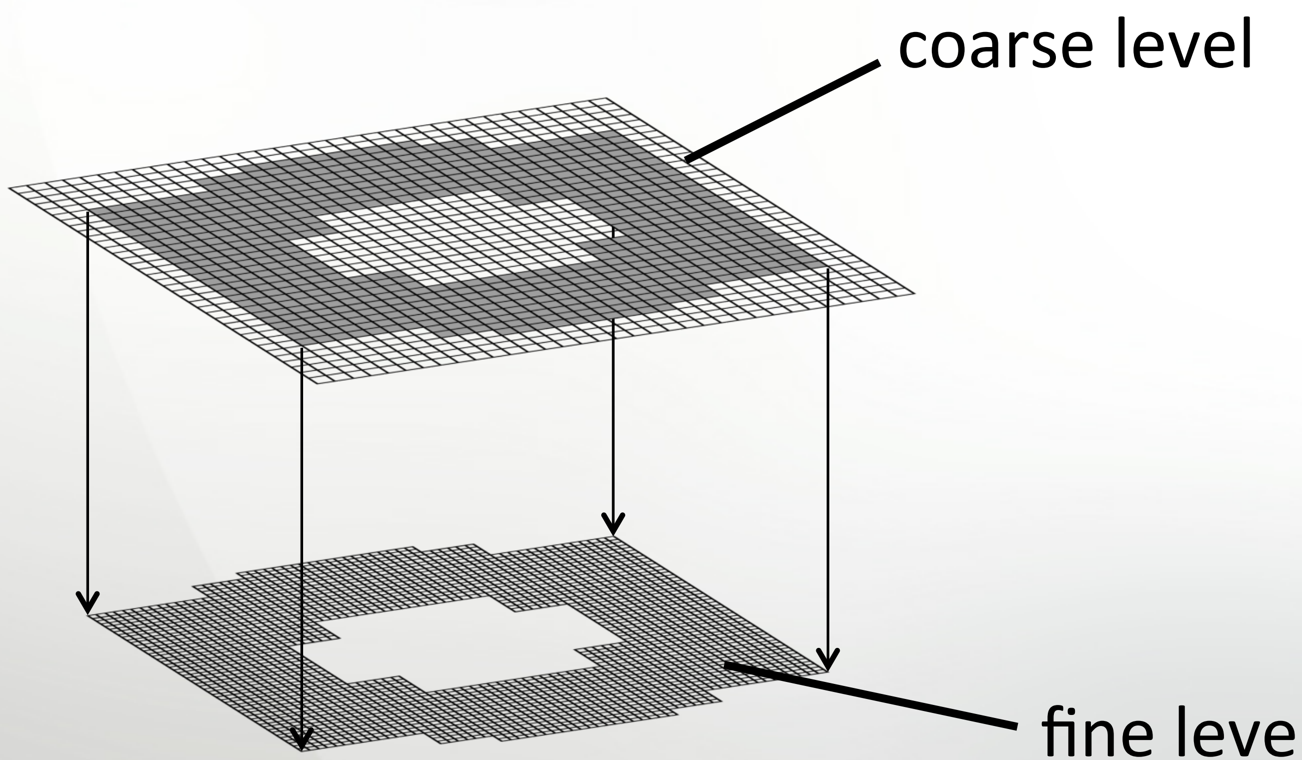
AMR Terminology



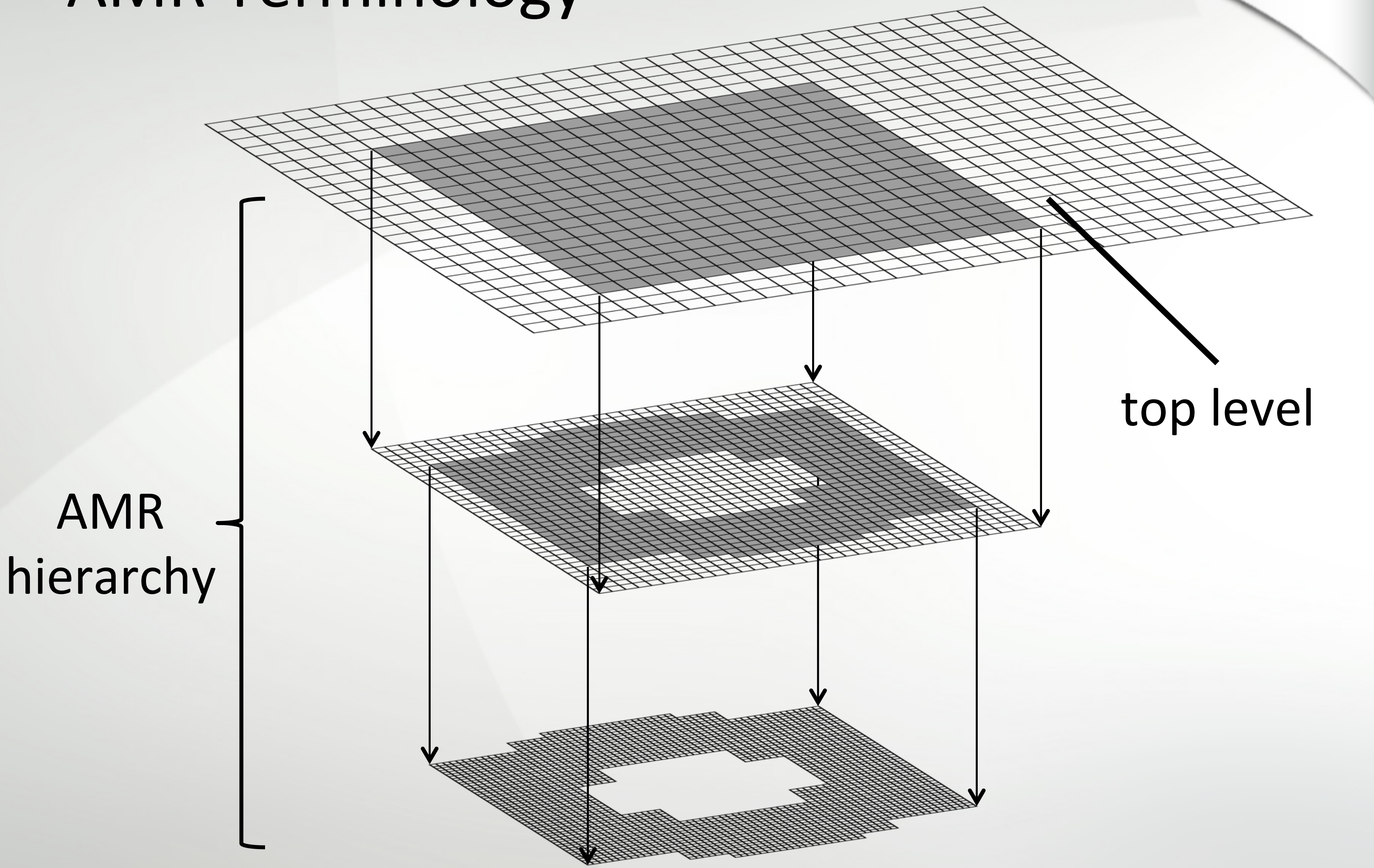
AMR Terminology



AMR Terminology



AMR Terminology



Spatial dimension

```
config param dimension = 2;

const dimensions = 1..dimension;
```

Spatial dimension

May specify value with a command-line flag

```
config param dimension = 2;

const dimensions = 1..dimension;
```

Spatial dimension

May specify value with a command-line flag

Value known at compile-time

```
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

```
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

```
config param dimension = 2;

const dimensions = 1..dimension;
```

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

```
config param dimension = 2;

const dimensions = 1..dimension;
```

Not modified after assignment; use **`var`** otherwise

Range: Arithmetic sequence of integers

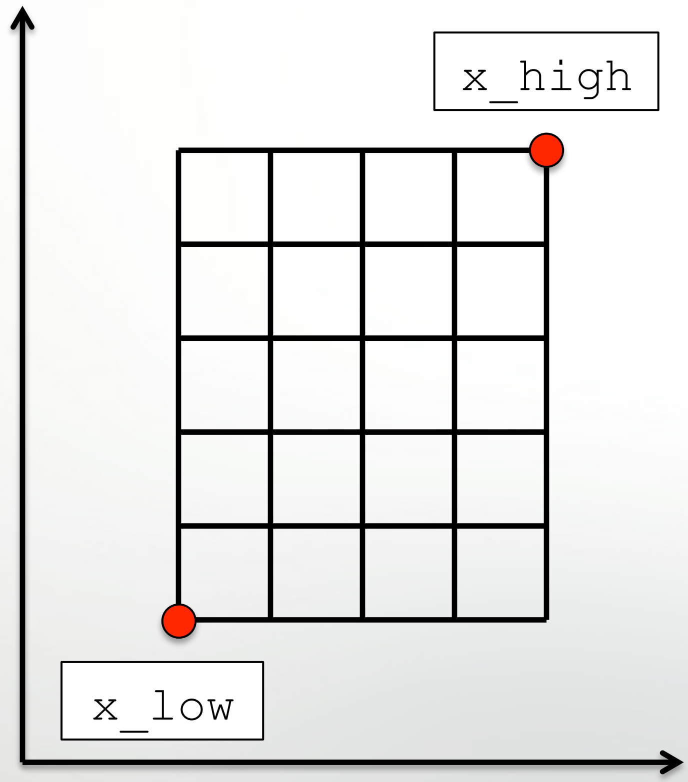
- Supports iteration:

```
for d in dimensions do ...
```

Grids

Basic geometry described by tuples

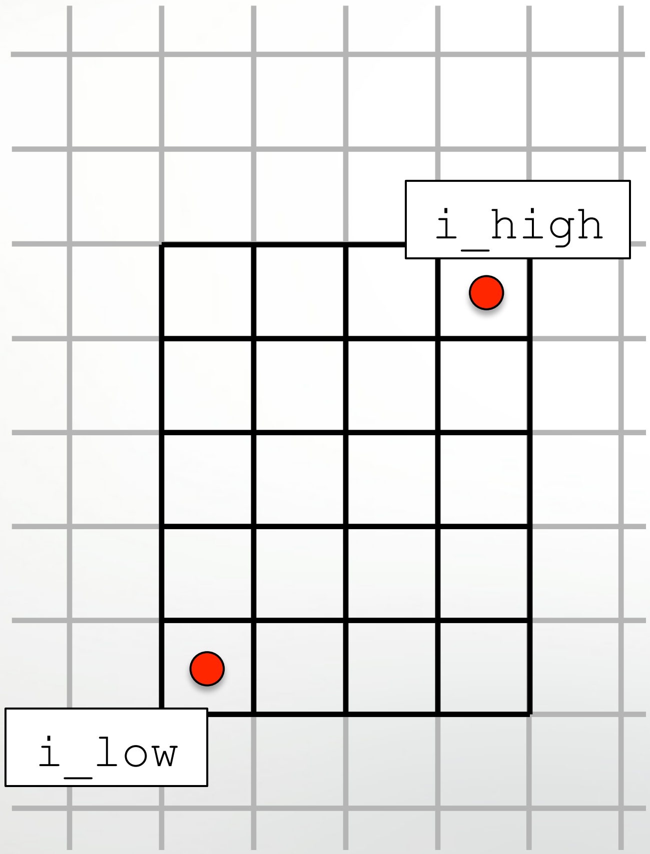
```
const x_low, x_high: dimension*real;
```



Grids

Basic geometry described by tuples

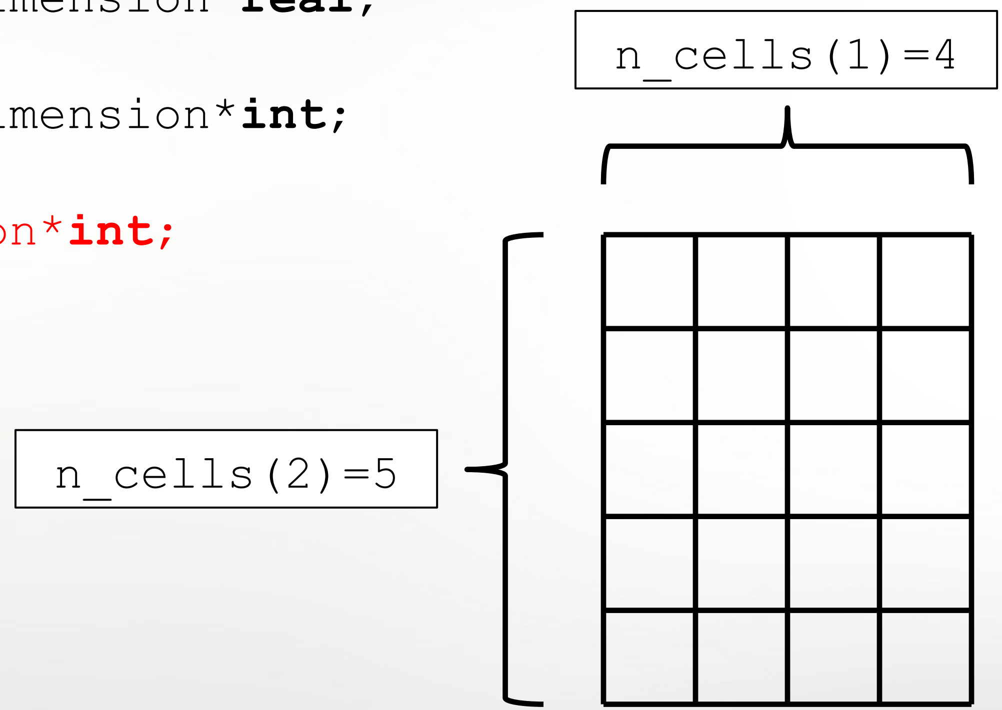
```
const x_low, x_high: dimension*real;

const i_low, i_high: dimension*int;
```



Grids

Basic geometry described by tuples

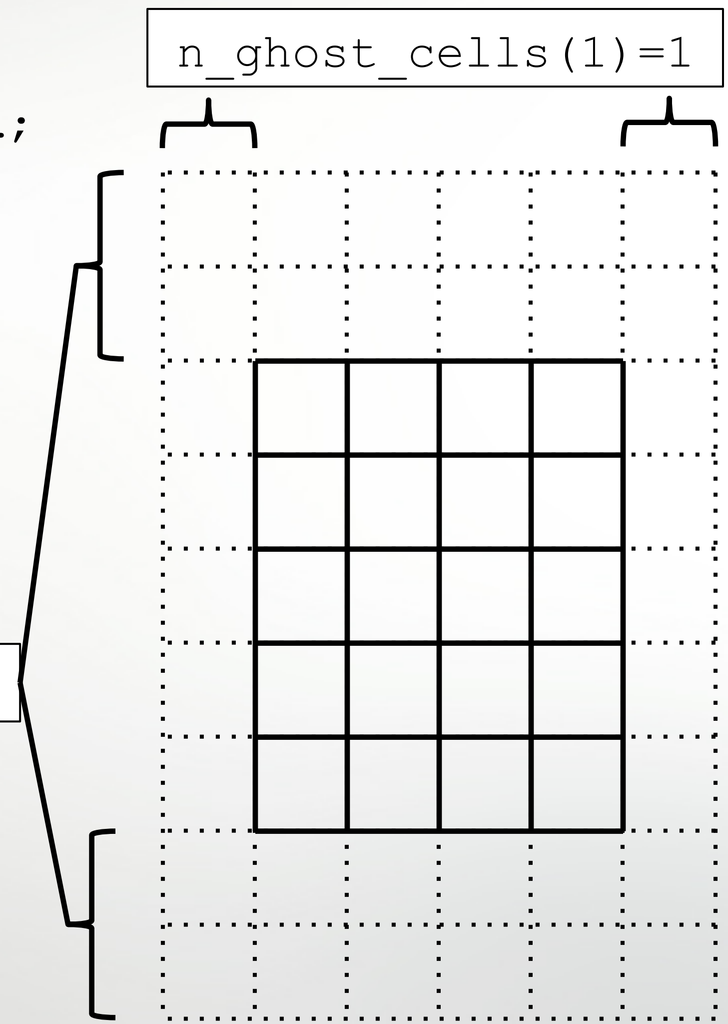
```
const x_low, x_high: dimension*real;

const i_low, i_high: dimension*int;

const n_cells: dimension*int;
```



Grids

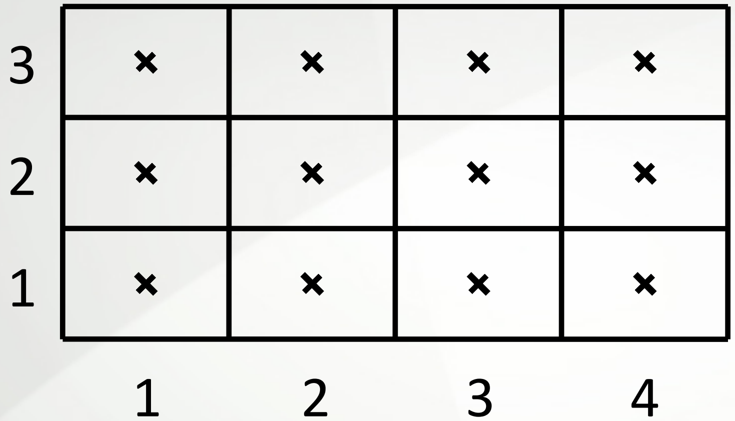
Basic geometry described by tuples

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

const i_low, i_high: dimension*int;

const n_cells: dimension*int;

const n_ghost_cells: dimension*int;
```



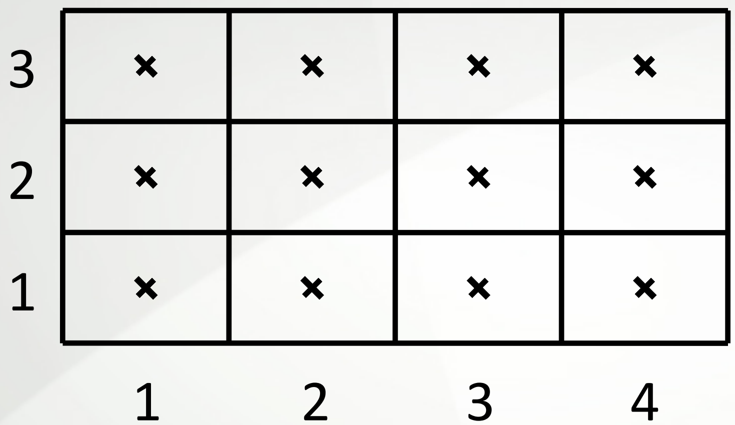
Grids: Indexing

Conventional approach – number the cells sequentially



Grids: Indexing

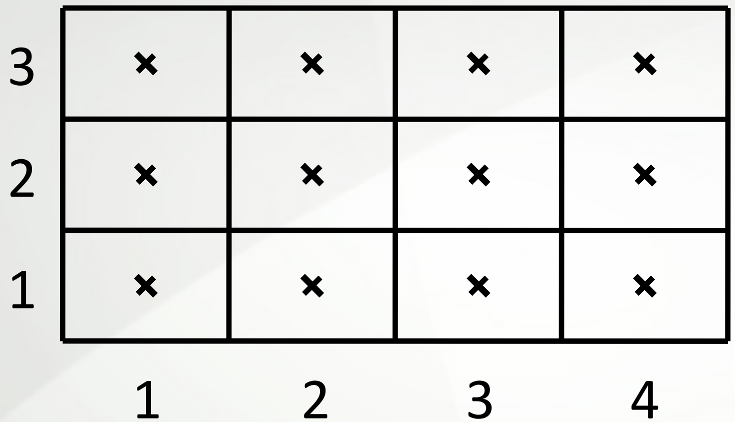
Conventional approach – number the cells sequentially




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

Grids: Indexing

Conventional approach – number the cells sequentially




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

Arithmetic domain: Multidimensional index space

- Supports storage:

```
var my_array: [cells] real;
```

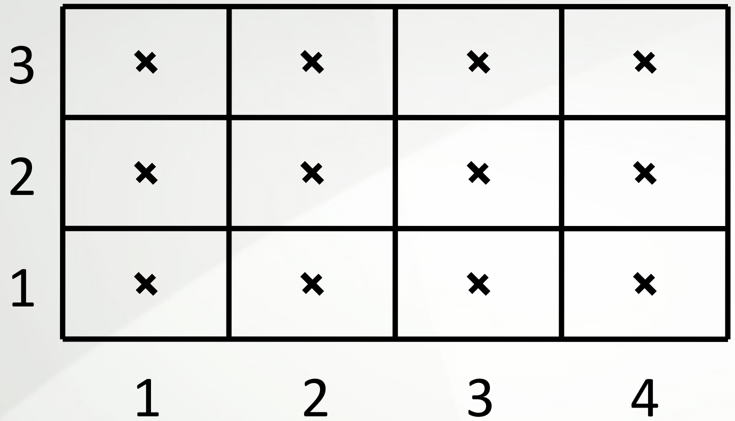
- Supports (parallel) iteration:

```
for(all) cell in cells do ...
```

- The reason Chapel is so useful for AMR

Grids: Indexing

Conventional approach – number the cells sequentially



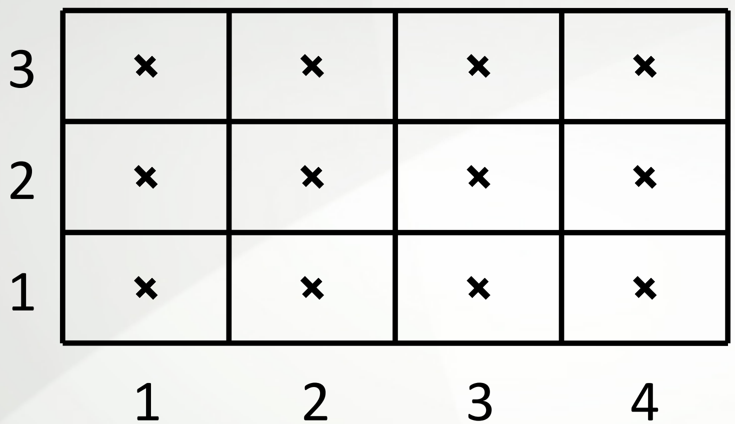

```
const cells = [1..4, 1..3];
```

Problem with conventional indexing:

- How are interfaces and vertices indexed?

Grids: Indexing

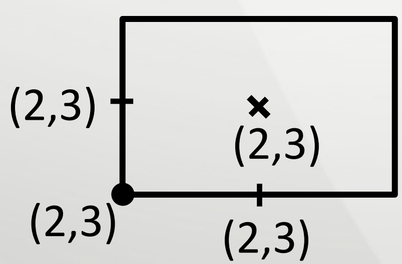
Conventional approach – number the cells sequentially




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

Problem with conventional indexing:

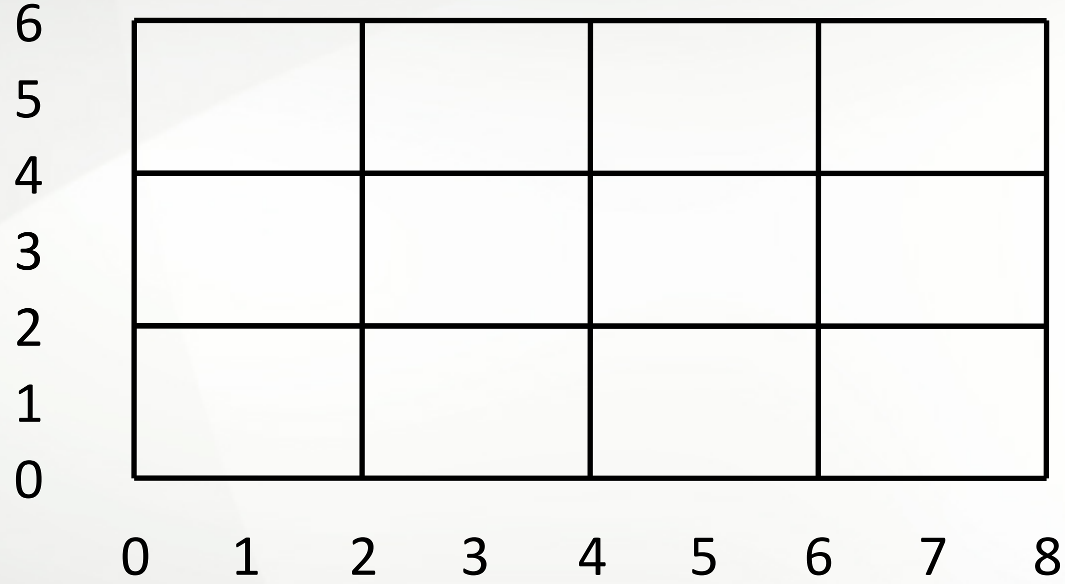
- How are interfaces and vertices indexed?



Many objects will have the same indices

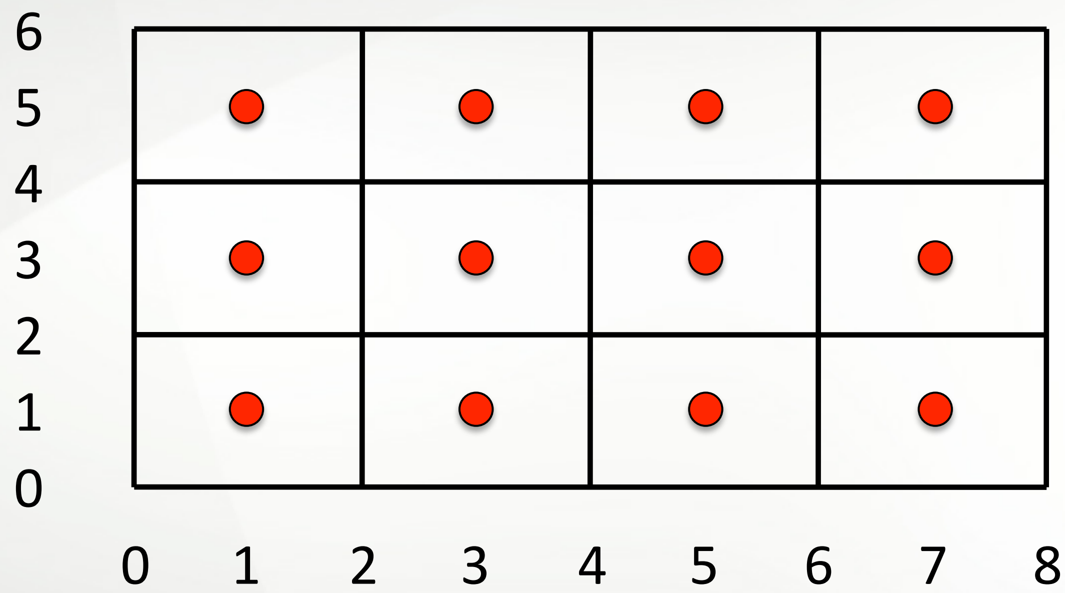
Grids: Indexing

Modified approach – denser index space



Grids: Indexing

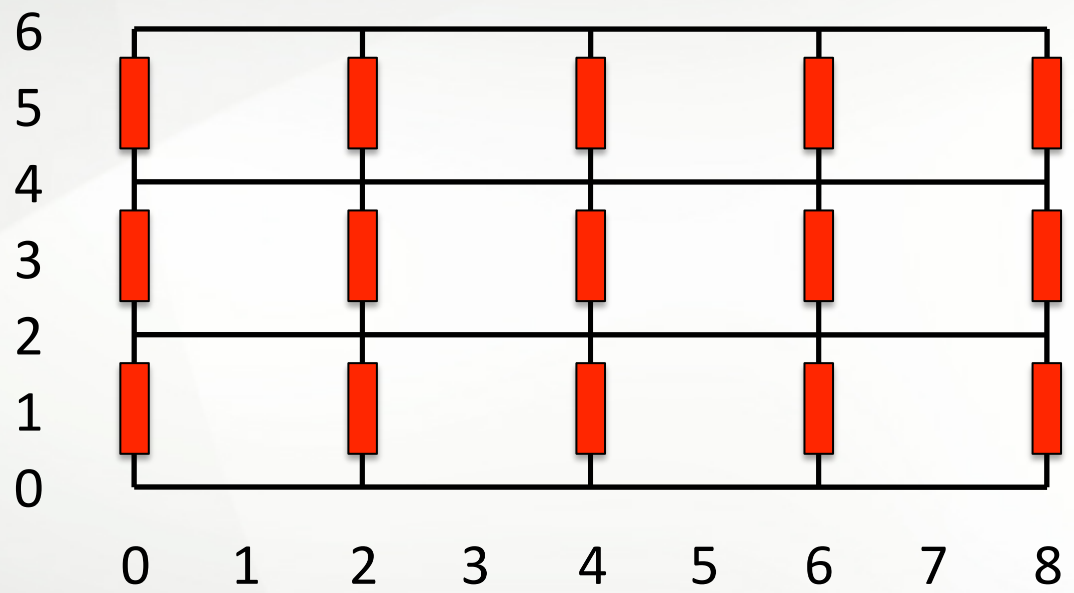
Modified approach – denser index space




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

Grids: Indexing

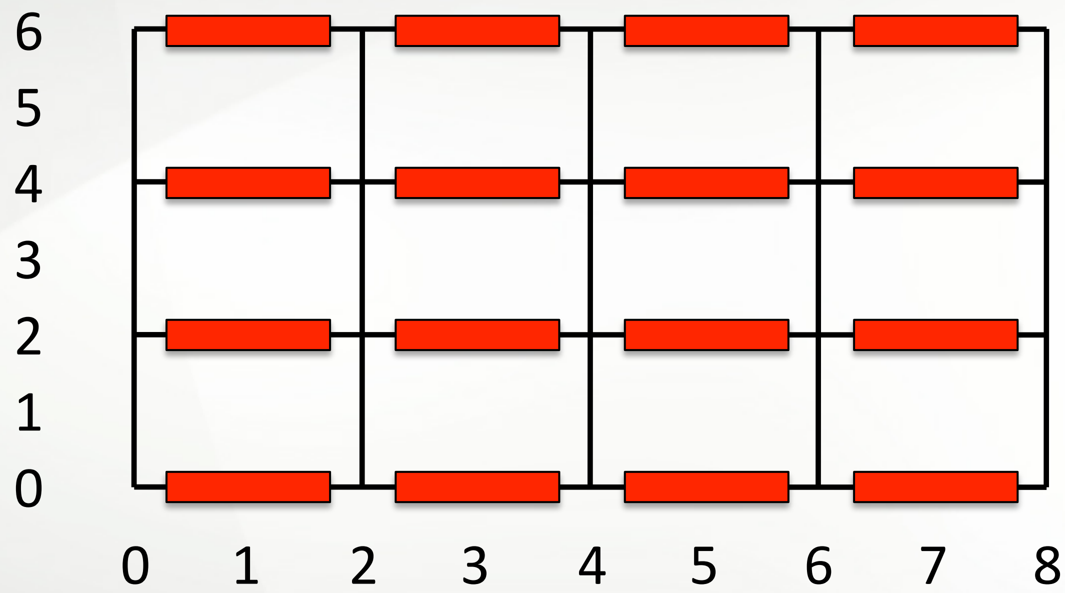
Modified approach – denser index space




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

Grids: Indexing

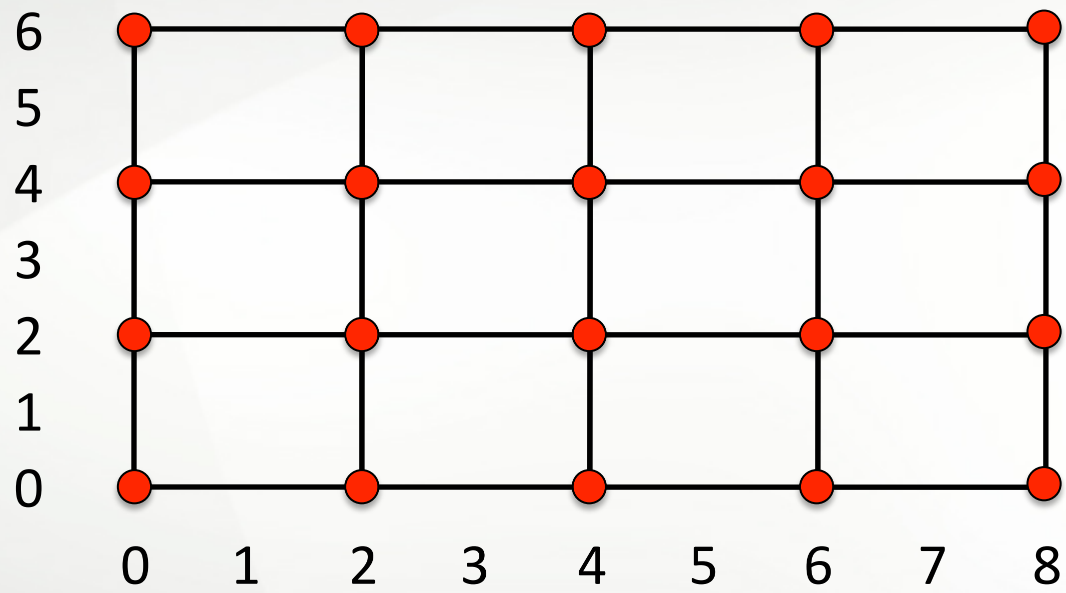
Modified approach – denser index space




```
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

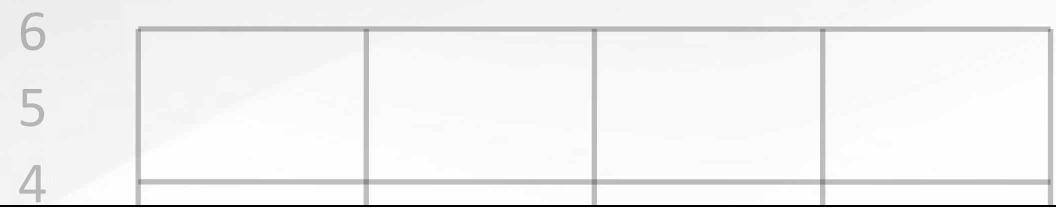
Modified approach – denser index space




```
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



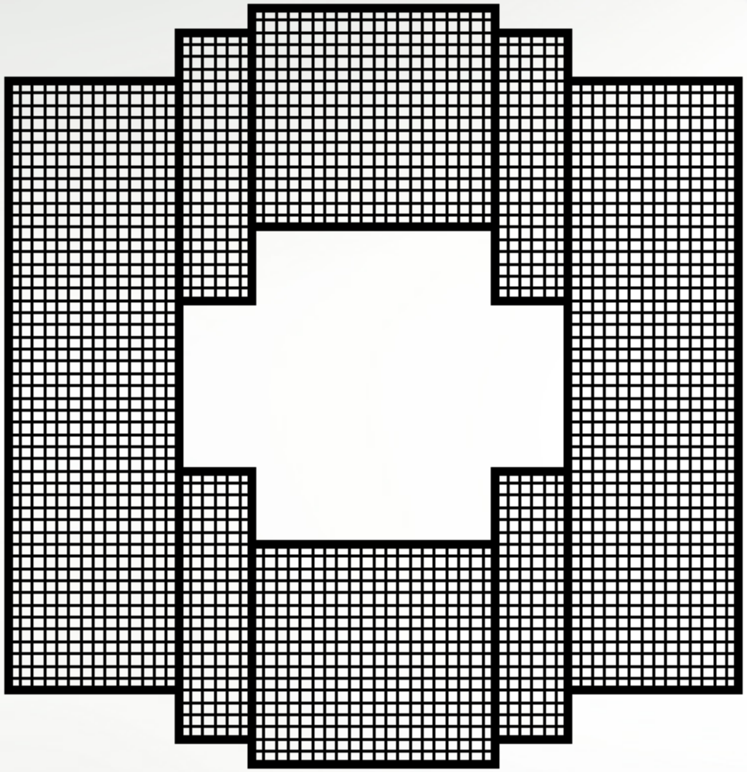
***Strided* domains**

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

```
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

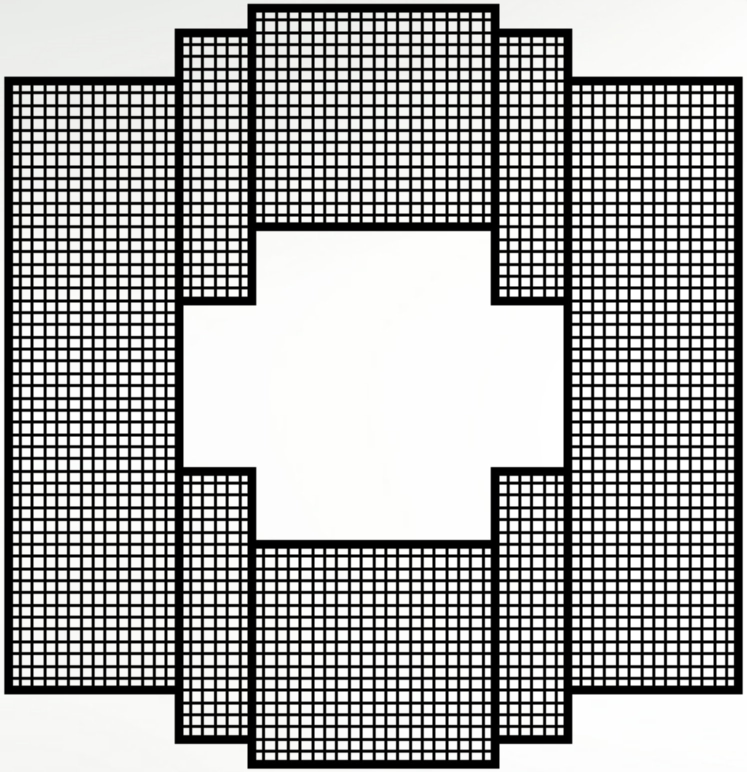
Essentially a union of grids



Levels

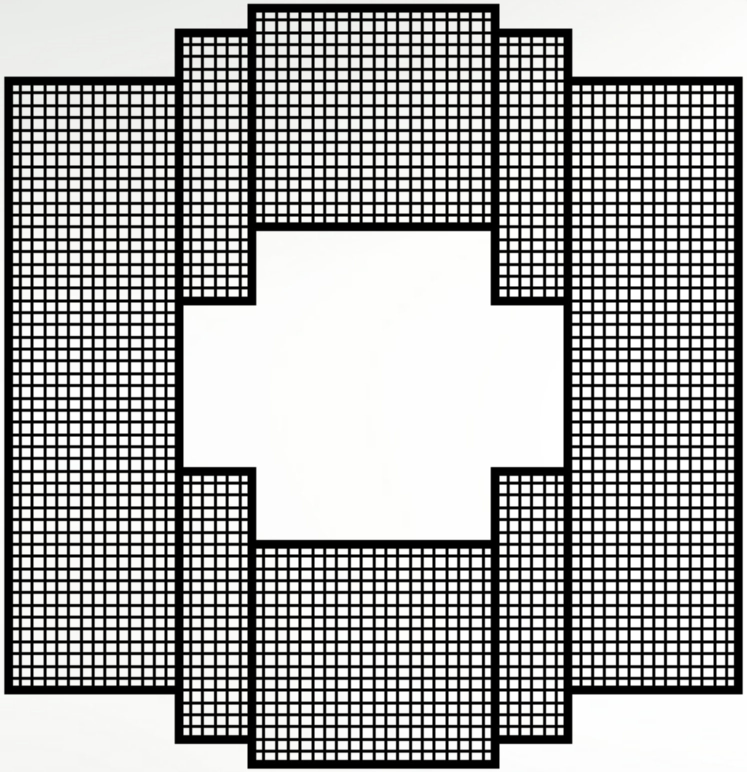
Essentially a union of grids

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



Levels

Essentially a union of grids



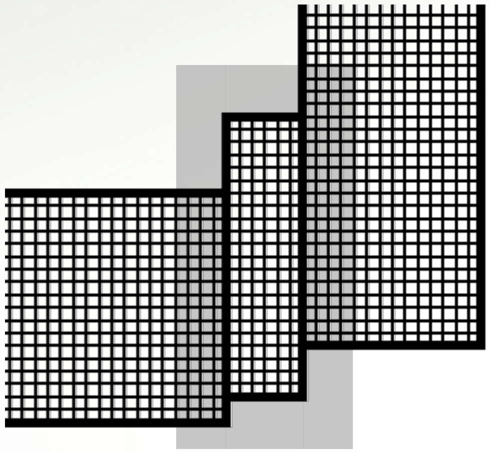
```
var grids: domain(Grid);
```

Associative domain

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

Levels: Overlaps

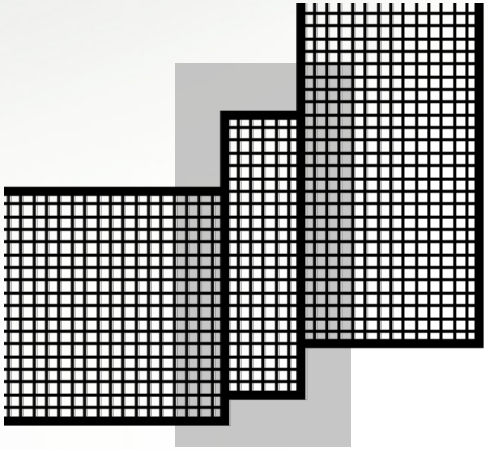
Each grid has a layer of **ghost cells** to facilitate data transfer



Levels: Overlaps

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

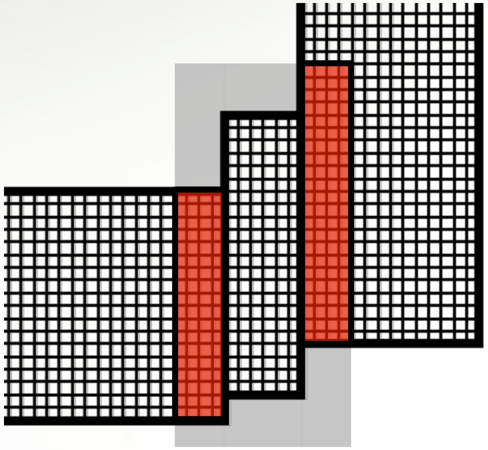
```
const extended_cells =
      cells.expand(n_ghost_cells);
```



Levels: Overlaps

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

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

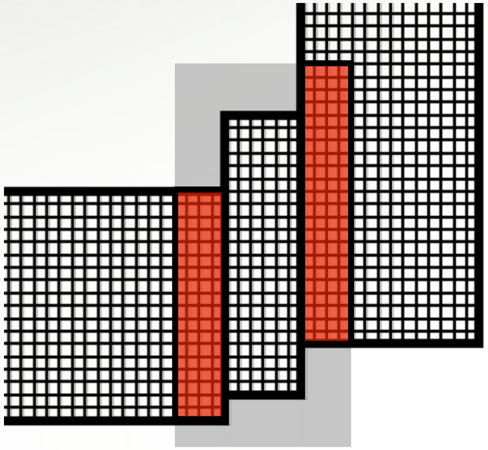


Calculating the overlaps

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

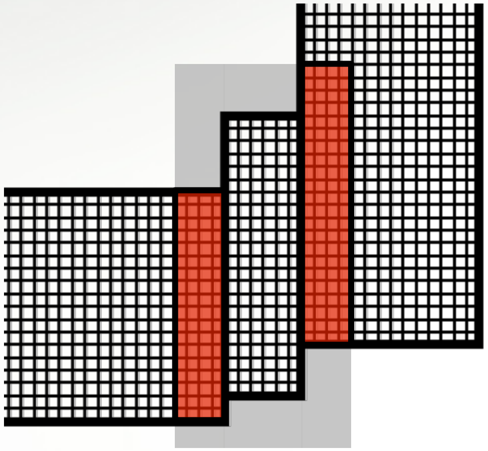
```
var neighbors: domain(Grid);
var overlap                        ension,stridable=true);

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

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



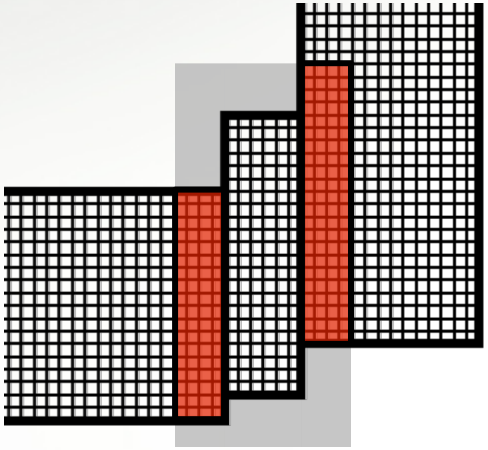
Calculating the overlaps

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

for sibling in p
  var overlap =                          ls );

  if overlap.num                          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);
```

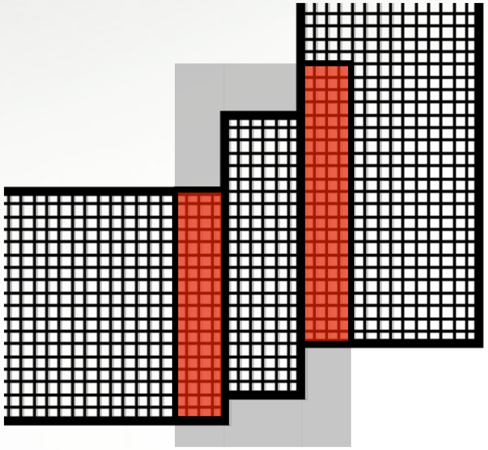
Calculating the overlaps

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

for sibling in parent_level.grids {
  var over                    ibling.cells );

  if over                    bling != 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



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

Calculating the overlaps

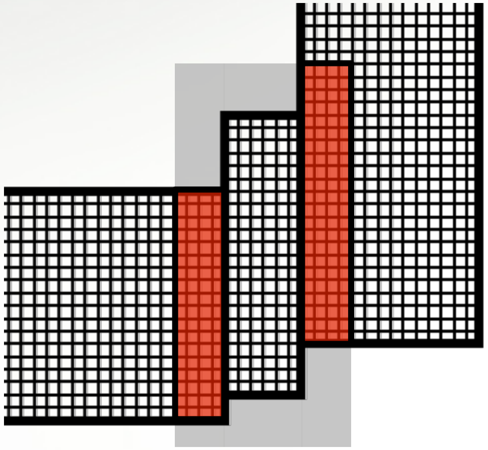
```
var neighbors: domain(Grid);
var overlaps:  [neighbors] domain(dimension,stridable=true);

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

  if overl                                his {
    neighb
    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

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



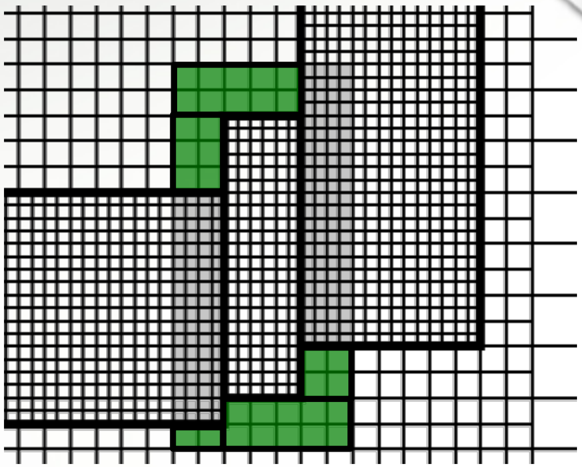
Calculating the overlaps

```
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;
  }
}
```

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

Levels: Interpolation

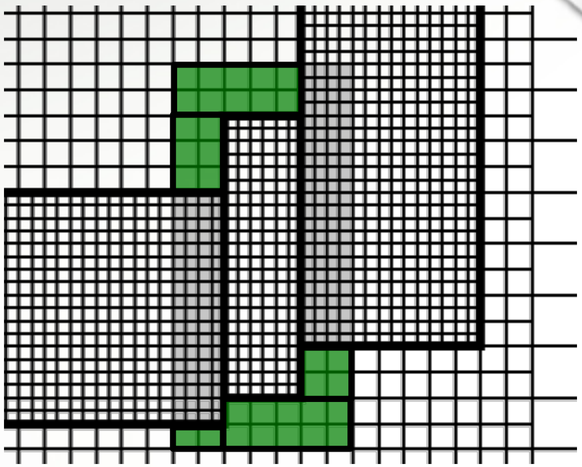
Remaining ghost cells will receive data interpolated from the coarser level



Levels: Interpolation

Remaining ghost cells will receive data interpolated from the coarser level

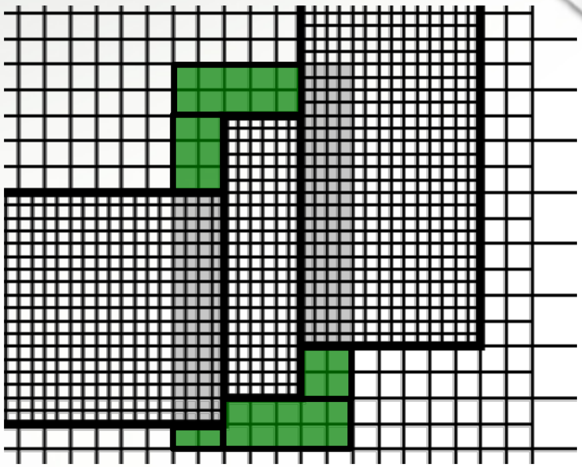
- Region is naturally described by a union of domains



Levels: Interpolation

Remaining ghost cells will receive data interpolated from the coarser level

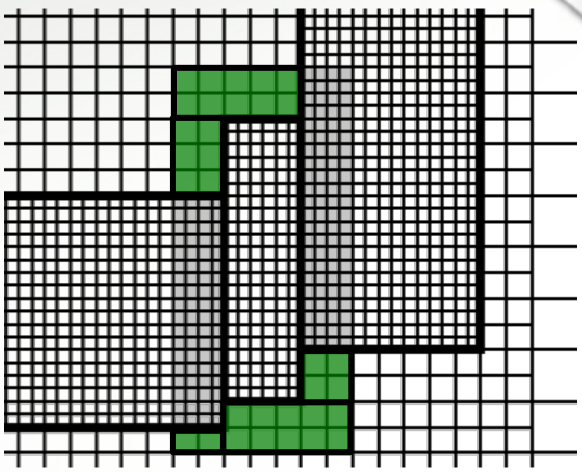
- Region is naturally described by a union of domains
- Region is naturally calculated by *subtraction* of domains



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



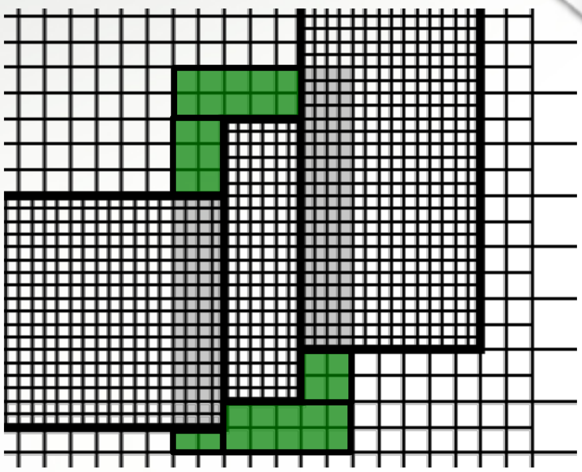
New object: **MultiDomain**

- 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

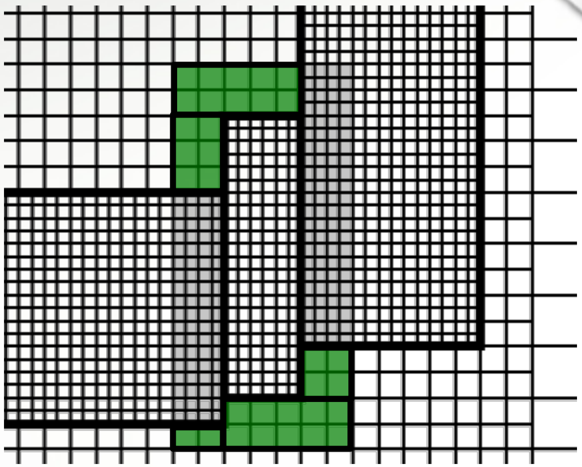


Computing the interpolation region

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

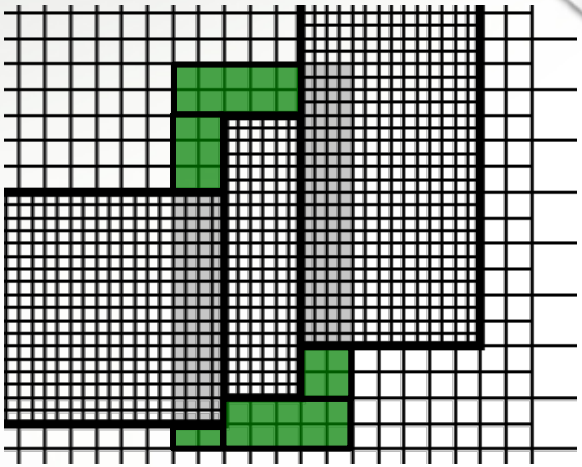
```
var interp_region =
      new MultiDomain(dimension,stridable=true);
int
int

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

Declare a new MultiDomain; initializes to empty.

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

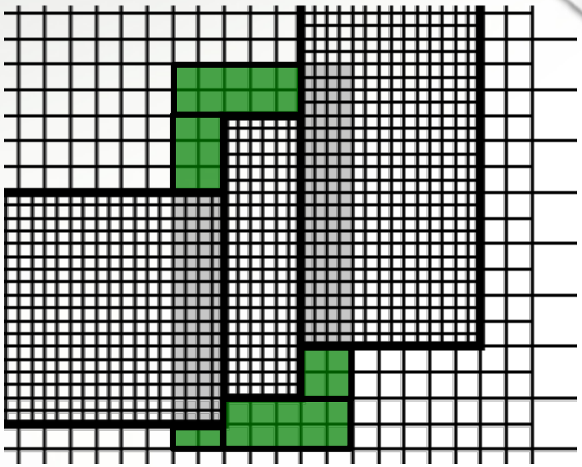
```
var interp_region =
      new MultiDomain(dimension,stridable=true);
interp_region.add( extended_cells );
interp_region.subtract( cells );

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

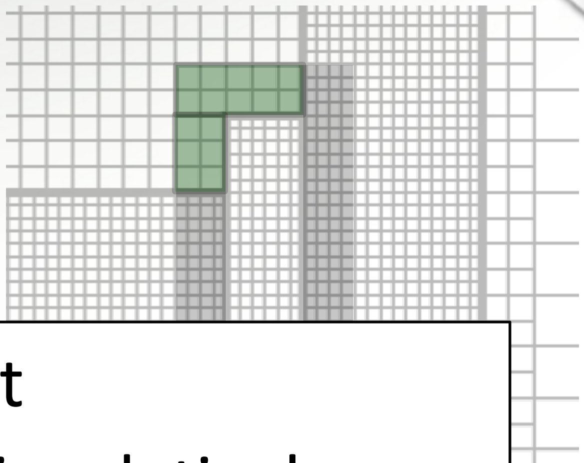
```
var interp_region =
      new MultiDomain(dimension,stridable=true);
interp_region.add( extended_cells );
interp_region.subtract( cells );

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

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

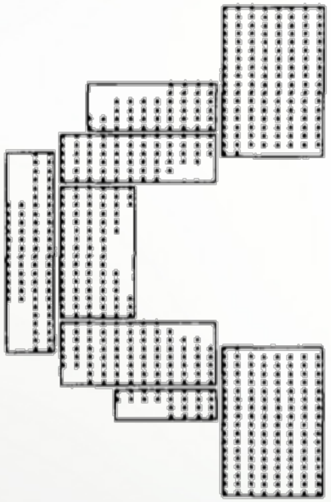


- This code is dimension-independent
- Underlying code for MultiDomains is relatively simple because domains do most of the hard work

```
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) );
```

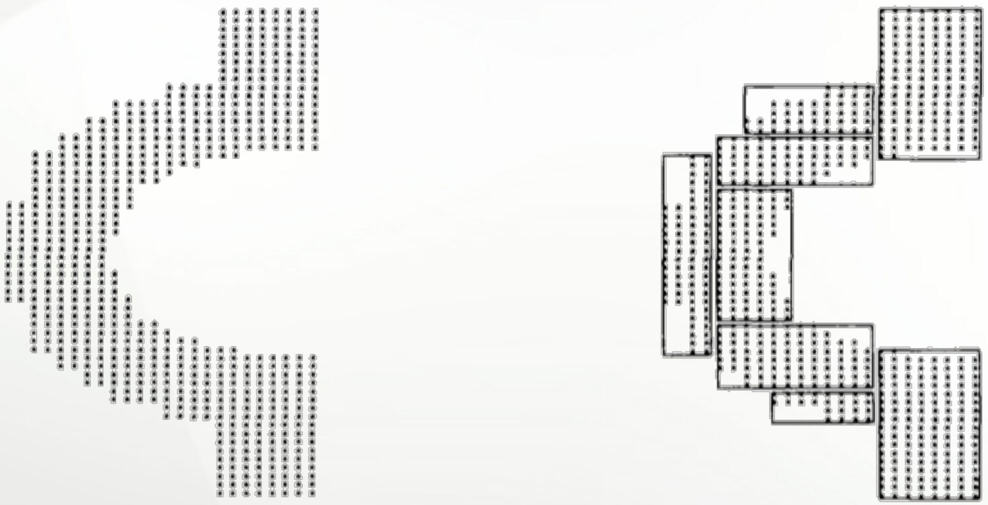
Regridding (very briefly)

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



Regridding (very briefly)

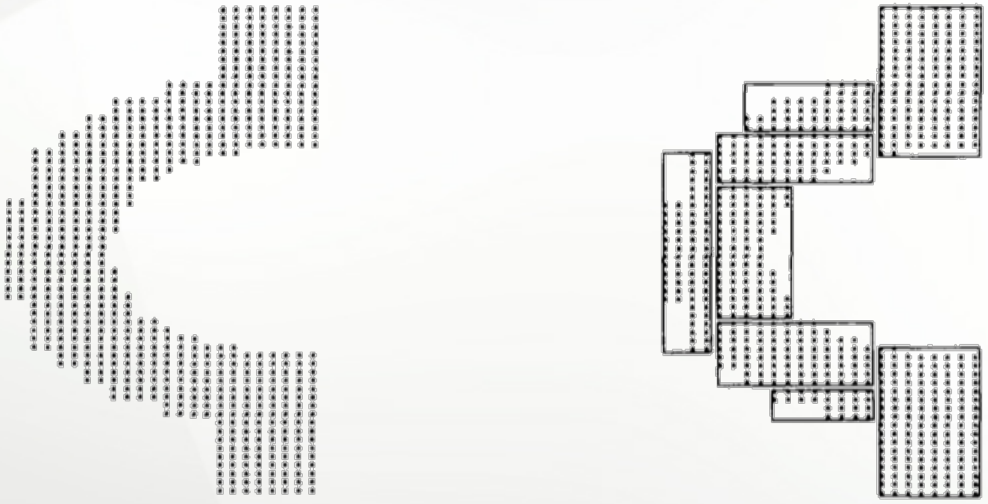
- Primarily the work of a **partitioning algorithm**: Given a set of flags, find rectangles that cover them (Berger & Rigoutsos, 1991)



- 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:

```
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 parallelism are so simple that you can almost forget about them.

- Massive reductions in code size:

Language	Parallelism	SLOC[1]	Tokens	Relative size (tokens)
Chapel (any D)	Shared mem.	1988	13783	1
Fortran (2D+3D) [2] 2D 3D	Serial	16562 8297 8265	151992 71639 80353	11.03 5.20 5.83
C/C++ (any D) [3]	Dist. mem.	40200	261427	20.22

[1] source ines of code, [2] AMRClaw, [3] Chombo BoxTools+AMRTools