APPLIED MATHEMATICS
UNIVERSITY *of* WASHINGTON

Adaptive Mesh Refinement in Chapel
Part I: Hard problems made easy

Jonathan Claridge

University of Washington

March 2, 2011

DARPA

HPCS

CRAY
THE SUPERCOMPUTER COMPANY

Overview of two talks

- This talk:
  - Several AMR challenges that Chapel makes easy
- Next talk:
  - A difficult part of AMR that Chapel sets us up to solve

What is adaptive mesh refinement (AMR)?

- Method for solving partial differential equations (PDEs) in which resolution is adaptively increased near “interesting” features

Movie omitted to reduce file size

Development overview

- Developed working, dimension-independent AMR infrastructure in just under 4 months, beginning with no Chapel experience

Development overview

- Developed working, dimension-independent AMR infrastructure in just under 4 months, beginning with no Chapel experience
- Chapel made many challenges of AMR easy with little-to-no additional infrastructure required, while providing a large head start on the really hard parts

Development overview

- Developed working, dimension-independent AMR infrastructure in just under 4 months, beginning with no Chapel experience
- Chapel made many challenges of AMR easy with little-to-no additional infrastructure required, while providing a large head start on the really hard parts
- Code size compares very favorably to existing AMR frameworks -- but keep in mind that the Chapel version is a "minimal" implementation!

Language	Parallelism	SLOC[1]	Tokens	Relative size (tokens)
C++ (D≤6)[3]	Dist. mem.	40200	261427	100%
Fortran (2D+3D)[2] 2D 3D	Serial	16562 8297 8265	151992 71639 80353	58% 27% 31%

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

Development overview

- Developed working, dimension-independent AMR infrastructure in just under 4 months, beginning with no Chapel experience
- Chapel made many challenges of AMR easy with little-to-no additional infrastructure required, while providing a large head start on the really hard parts
- Code size compares very favorably to existing AMR frameworks -- but keep in mind that the Chapel version is a "minimal" implementation!

Language	Parallelism	SLOC[1]	Tokens	Relative size (tokens)
C++ (D≤6) [3]	Dist. mem.	40200	261427	100%
Fortran (2D+3D) [2]	Serial	16562	151992	58%
2D		8297	71639	27%
3D		8265	80353	31%
Chapel (any D)	Shared mem.	1988	13783	**5%**

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

Development overview

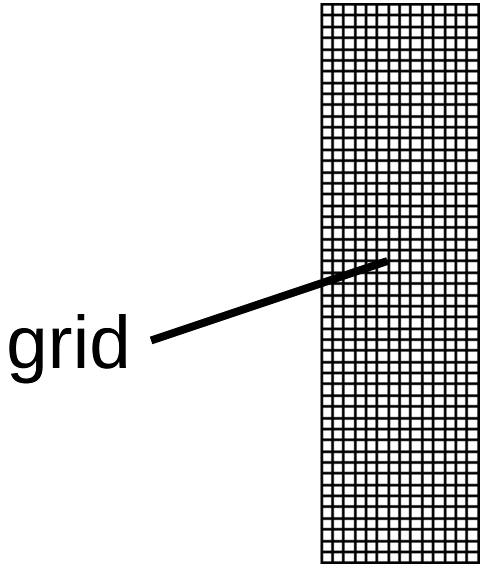
- Developed working, dimension-independent AMR infrastructure in just under 4 months, beginning with no Chapel experience
- Chapel made many challenges of AMR easy with little-to-no additional infrastructure required, while providing a large head start on the really hard parts
- Code size compares very favorably to existing AMR frameworks -- but keep in mind that the Chapel version is a “minimal” implementation!

Language	Parallelism	SLOC[1]	Tokens	Relative size (tokens)
C++ (D≤6) [3]	Dist. mem.	40200	261427	100%
Fortran (2D+3D) [2]	Serial	16562	151992	58%
2D		8297	71639	27%
3D		8265	80353	31%
Chapel (any D)	**Shared mem.**	1988	13783	**5%**

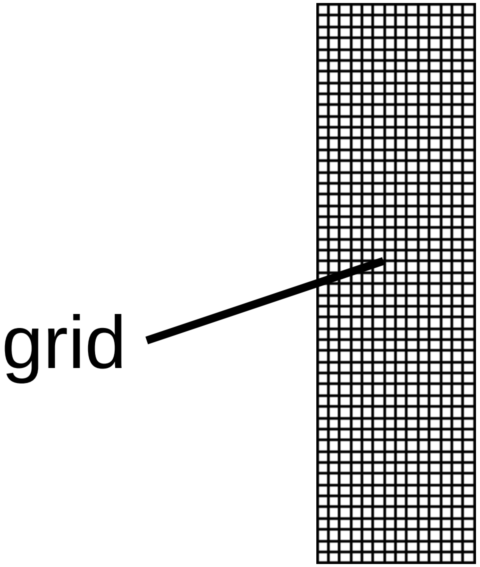
Reflects limitations of developer time, not Chapel itself

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

AMR terminology



AMR terminology



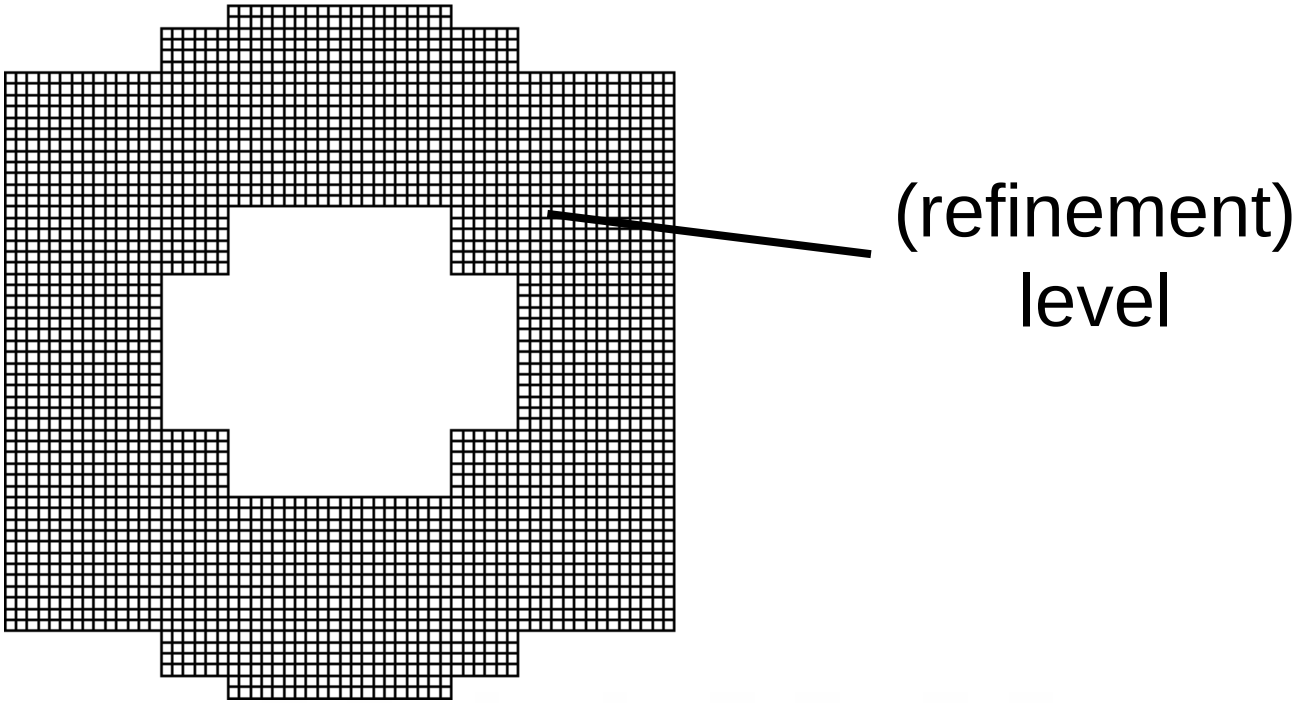
Roughly:

Operations on grids

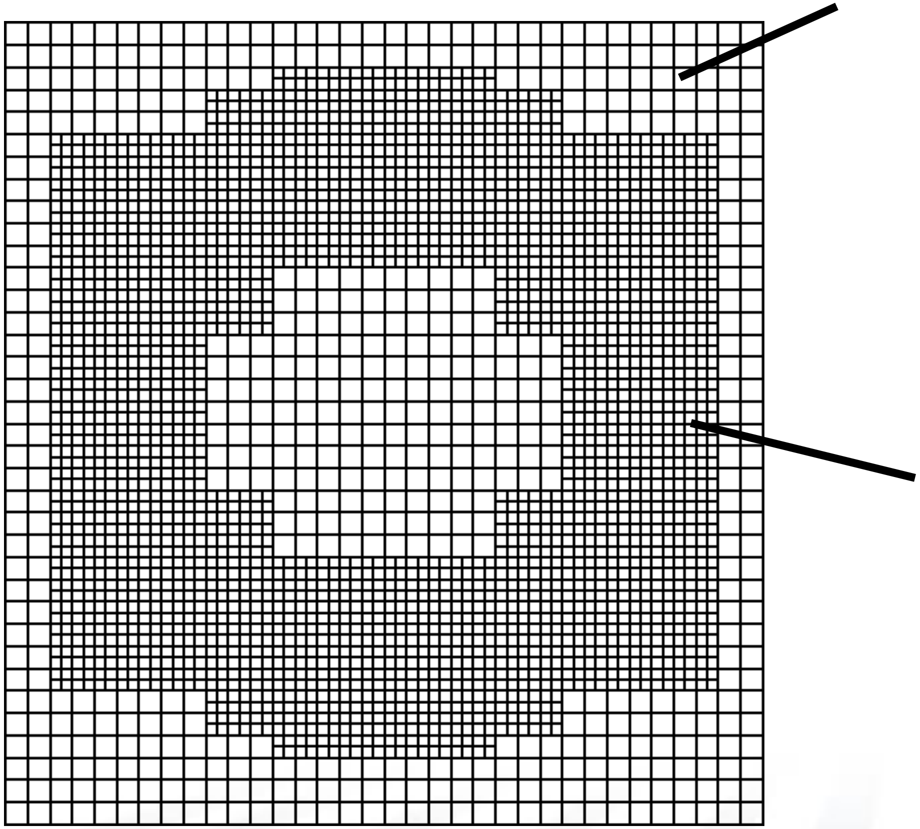
↕

Operations on rectangular (Chapel) domains

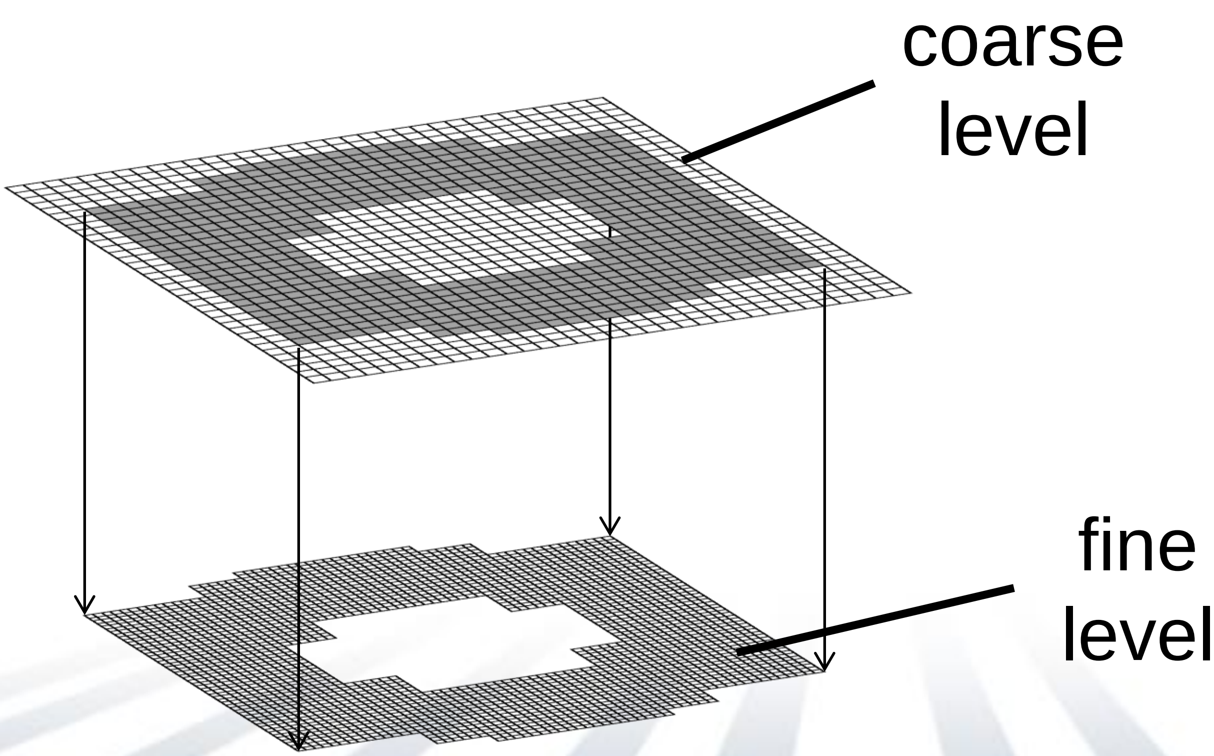
AMR terminology



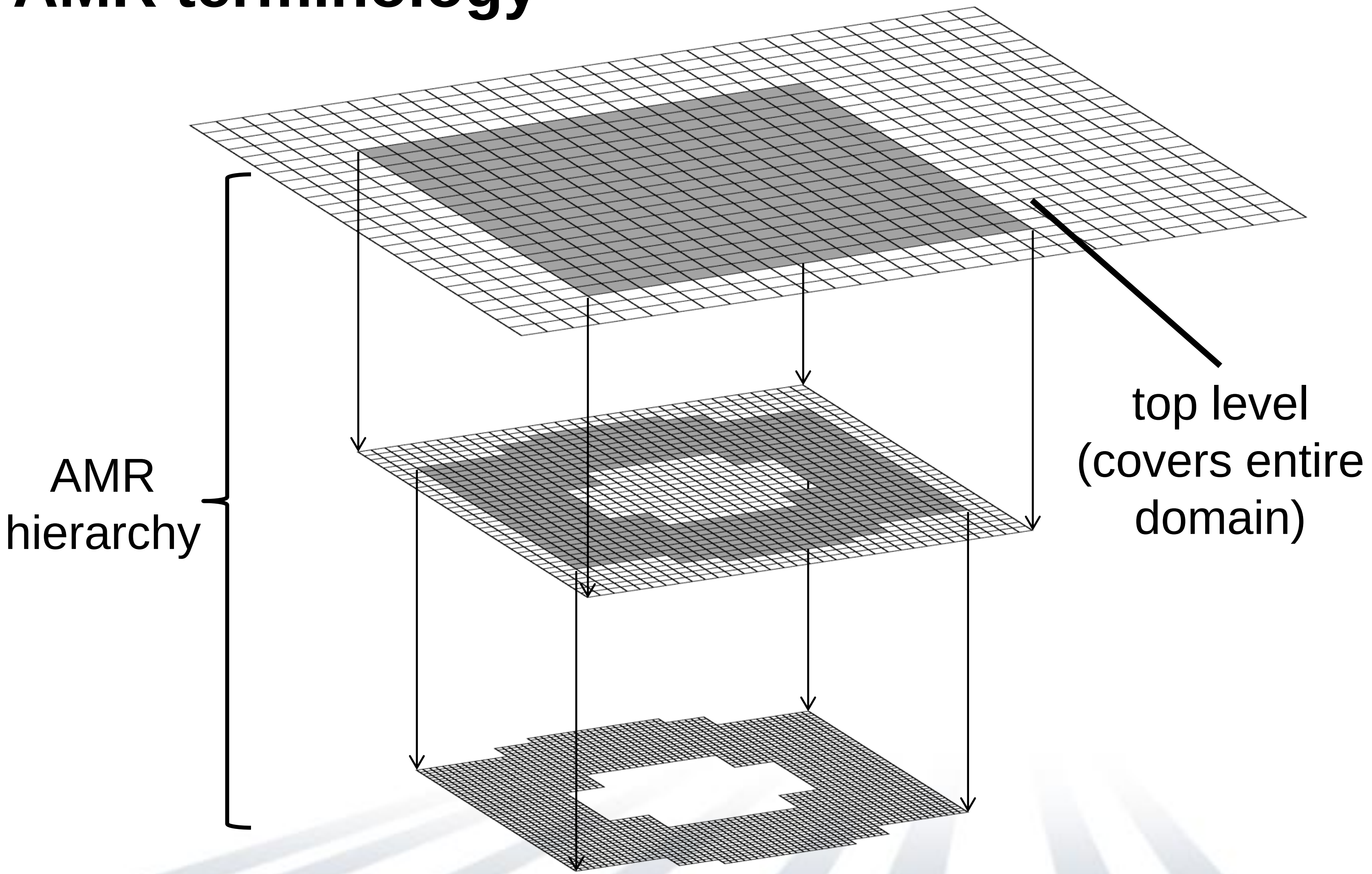
AMR terminology



AMR terminology

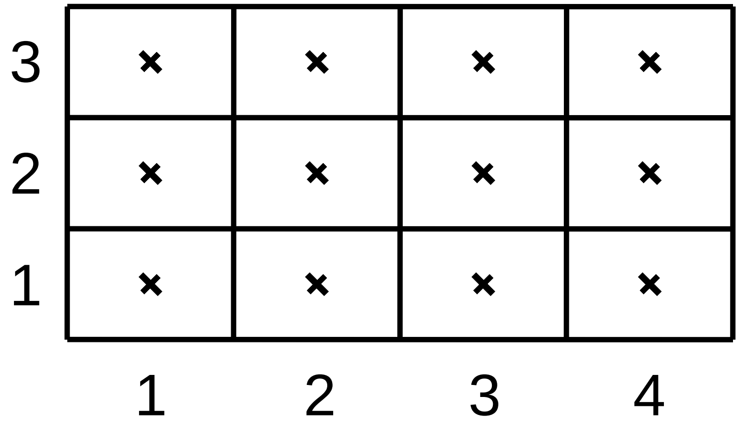


AMR terminology



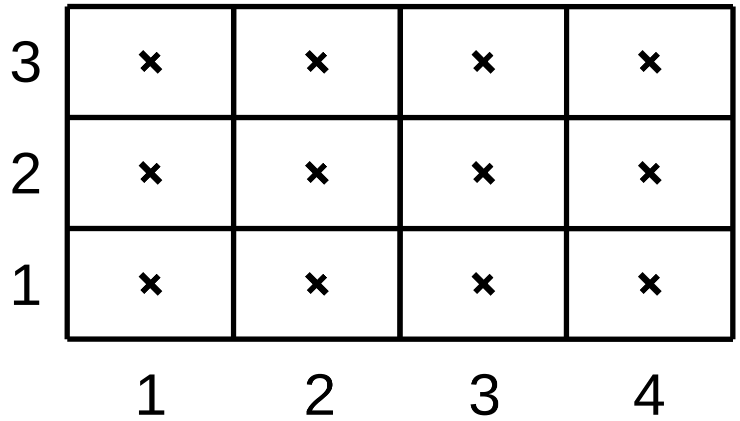
Grids: Indexing

- Conventional indexing – number grid cells sequentially



Grids: Indexing

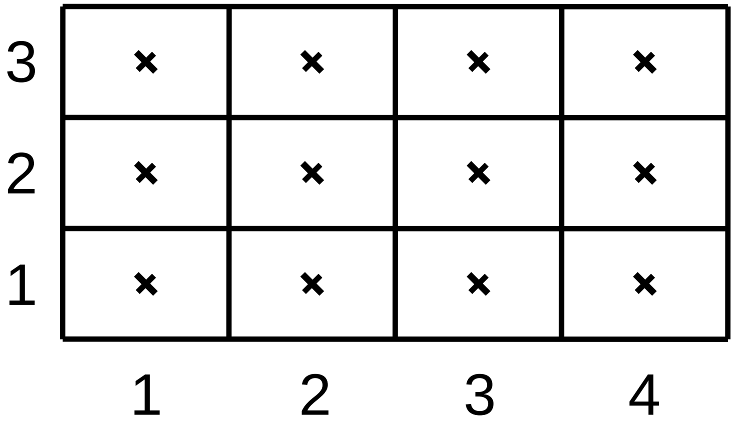
- Conventional indexing – number grid cells sequentially




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

Grids: Indexing

- Conventional indexing – number grid cells sequentially



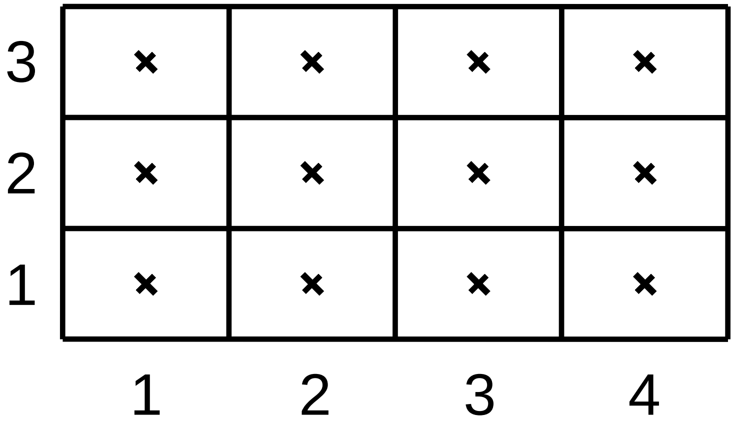

```
const cells = [1..4, 1..3];
```

Rectangular domain: Multidimensional index space

- Supports storage:
  var my_array: [cells] real;
- Supports (parallel) iteration:
  for(all) cell **in** cells **do** …

Grids: Indexing

- Conventional indexing – number grid cells sequentially

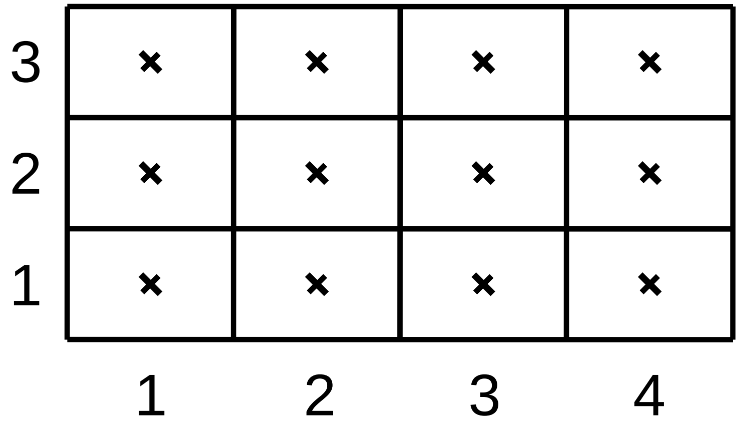



```
const cells = [1..4, 1..3];
```

- Problem with conventional indexing: How are the interfaces indexed?
  - Usual approach: Interface has the same index as the cell above it

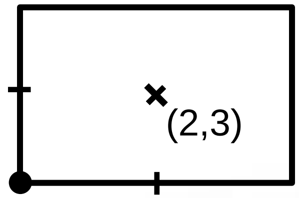
Grids: Indexing

- Conventional indexing – number grid cells sequentially



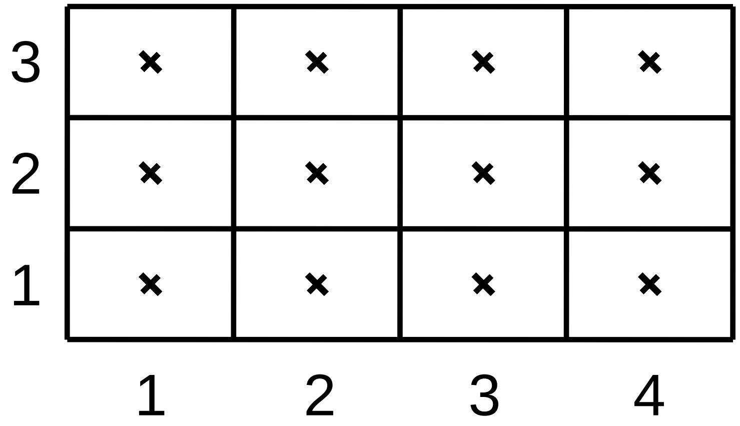

```
const cells = [1..4, 1..3];
```

- Problem with conventional indexing: How are the interfaces indexed?
  - Usual approach: Interface has the same index as the cell above it



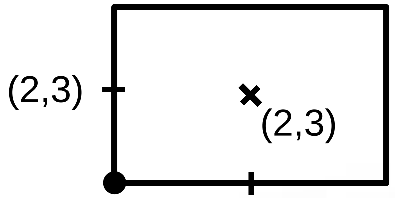
Grids: Indexing

- Conventional indexing – number grid cells sequentially



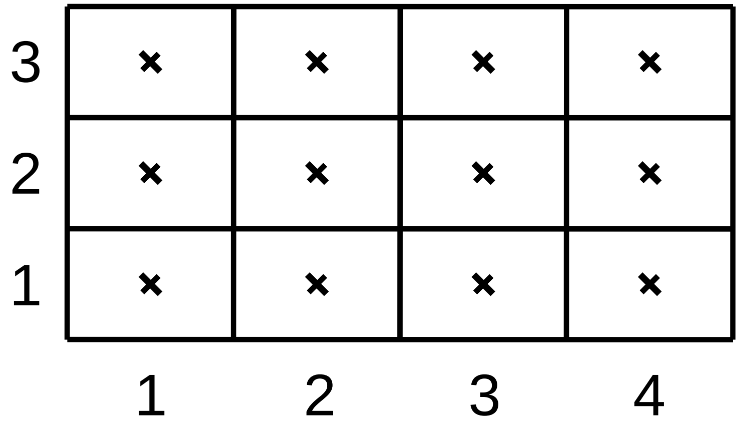

```
const cells = [1..4, 1..3];
```

- Problem with conventional indexing: How are the interfaces indexed?
  - Usual approach: Interface has the same index as the cell above it



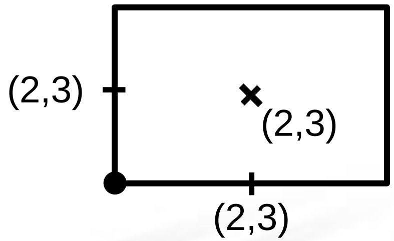
Grids: Indexing

- Conventional indexing – number grid cells sequentially



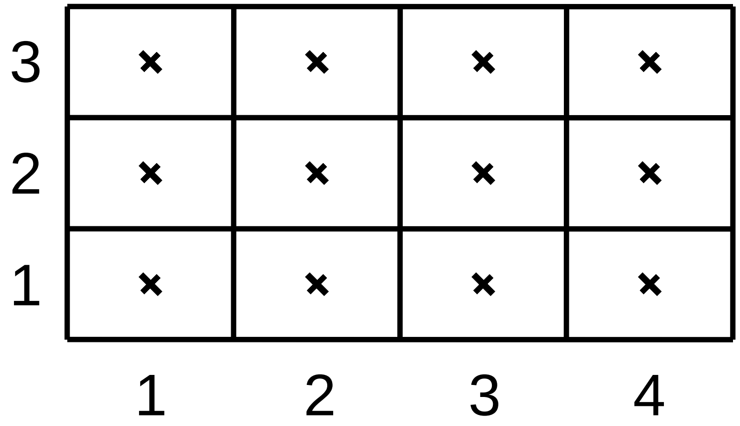

```
const cells = [1..4, 1..3];
```

- Problem with conventional indexing: How are the interfaces indexed?
  - Usual approach: Interface has the same index as the cell above it



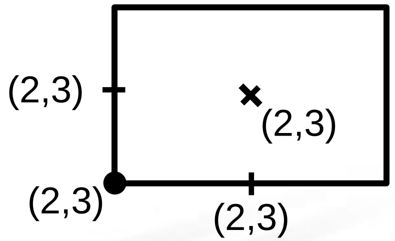
Grids: Indexing

- Conventional indexing – number grid cells sequentially



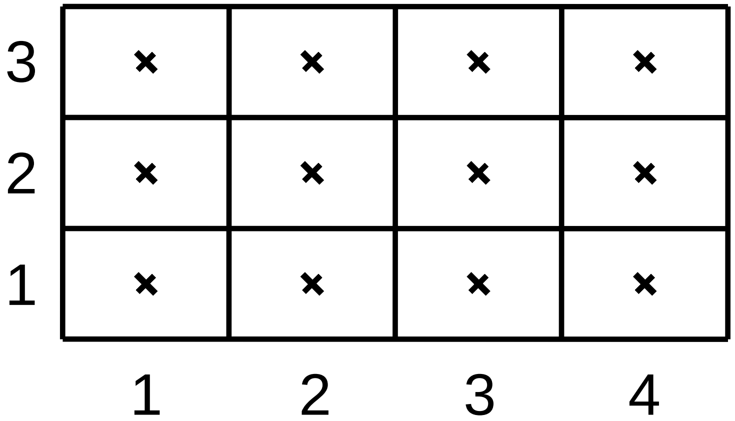

```
const cells = [1..4, 1..3];
```

- Problem with conventional indexing: How are the interfaces indexed?
  - Usual approach: Interface has the same index as the cell above it



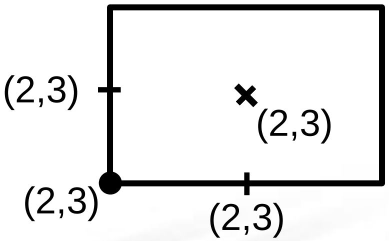
Grids: Indexing

- Conventional indexing – number grid cells sequentially




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

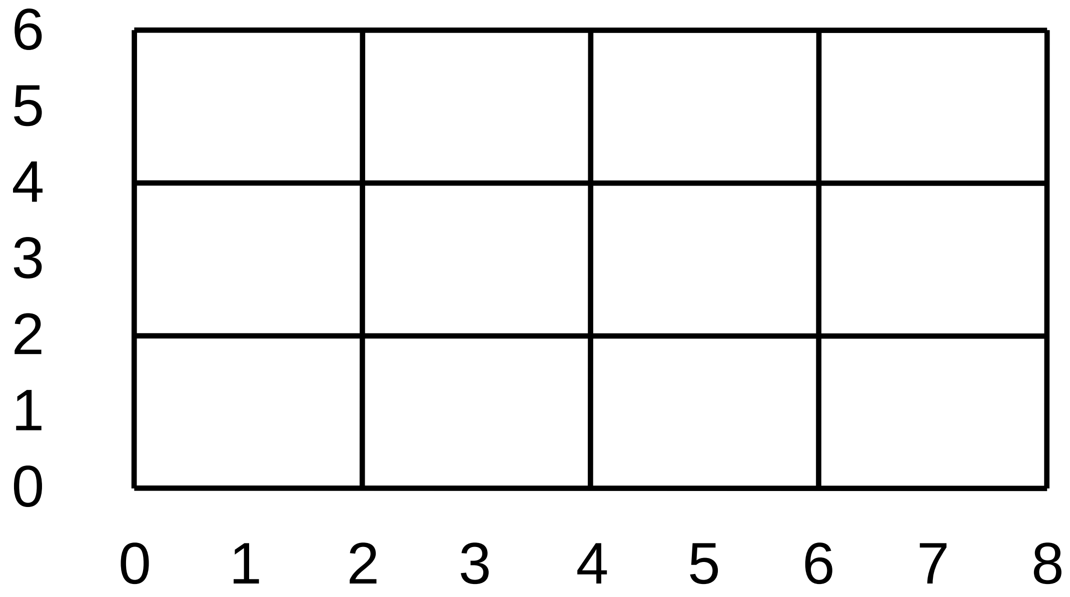
- Problem with conventional indexing: How are the interfaces indexed?
  - Usual approach: Interface has the same index as the cell above it



Many objects will have the same indices

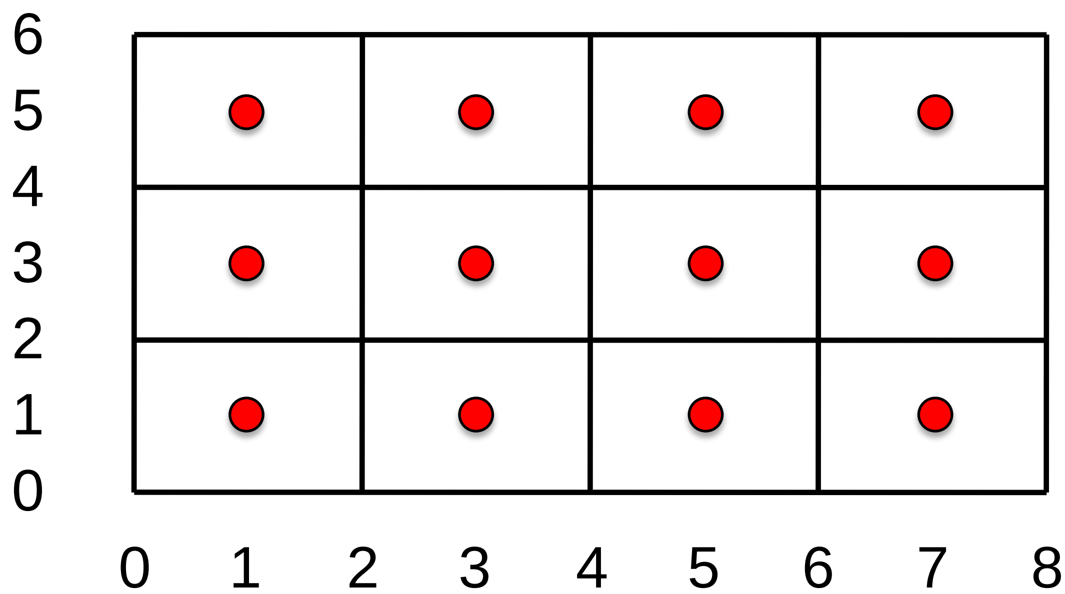
Grids: Indexing

- Modified approach – denser index space



Grids: Indexing

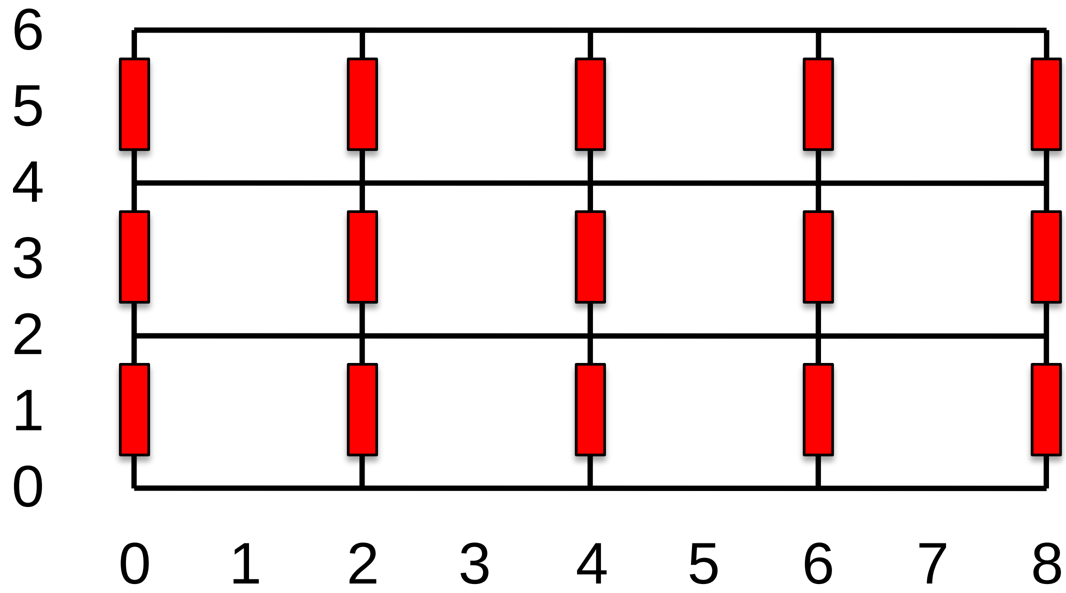
- Modified approach – denser index space




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

Grids: Indexing

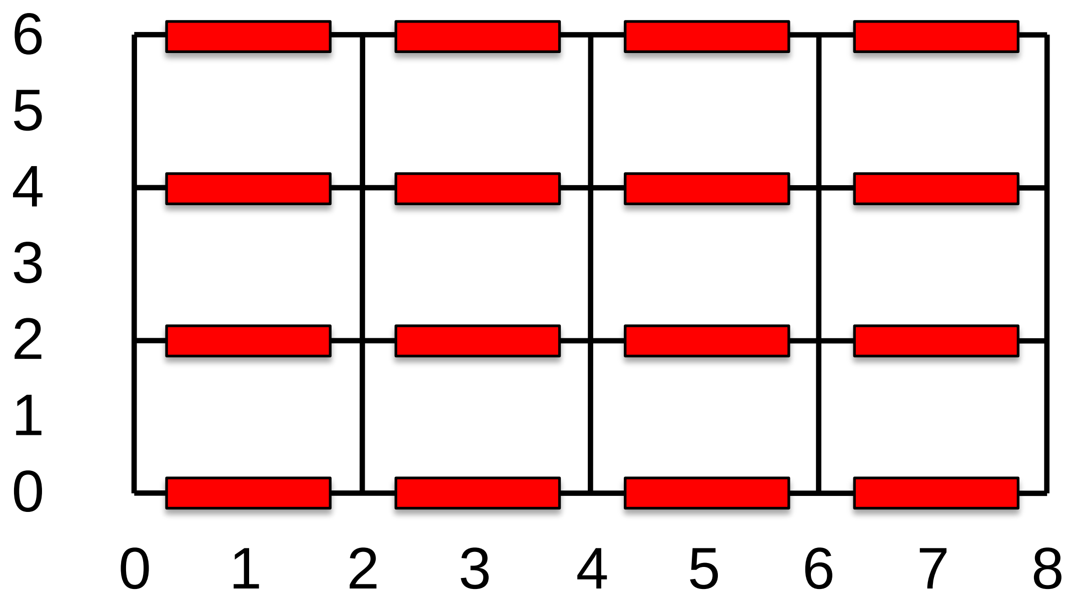
- Modified approach – denser index space




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

Grids: Indexing

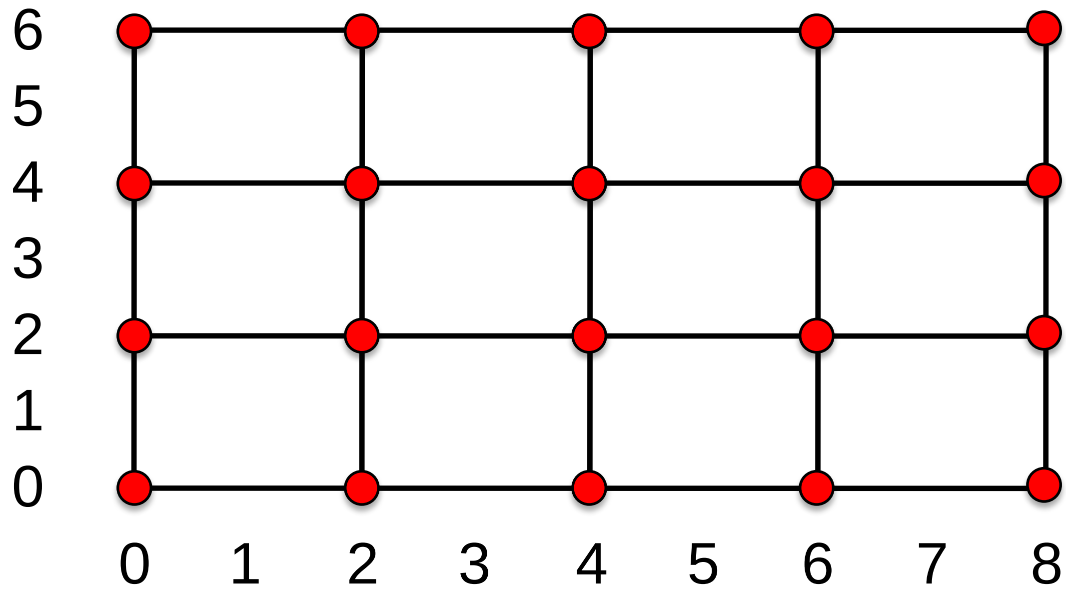
- Modified approach – denser index space




```
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

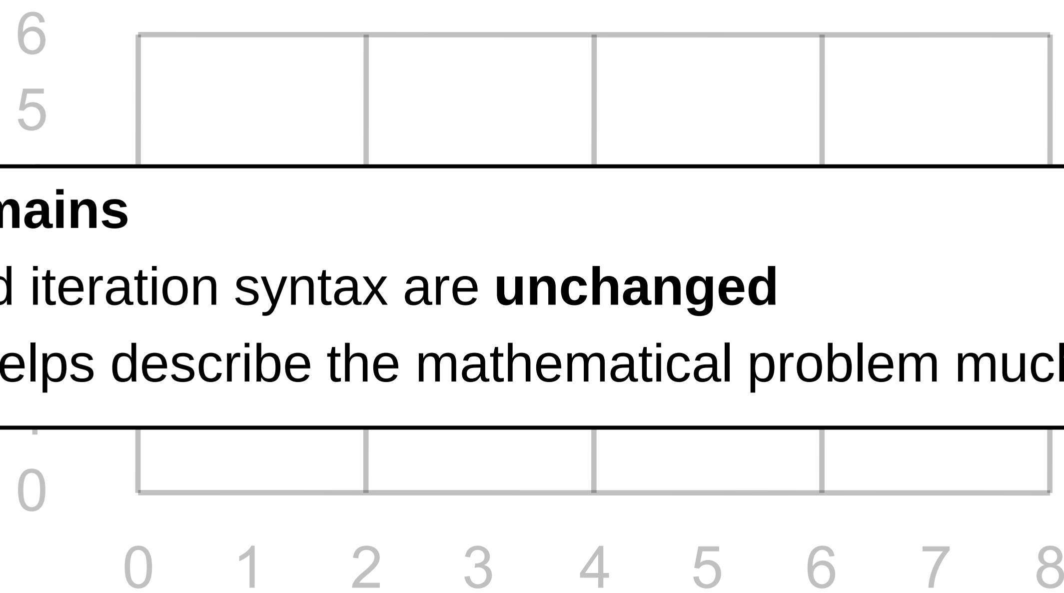
- Modified approach – denser index space




```
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



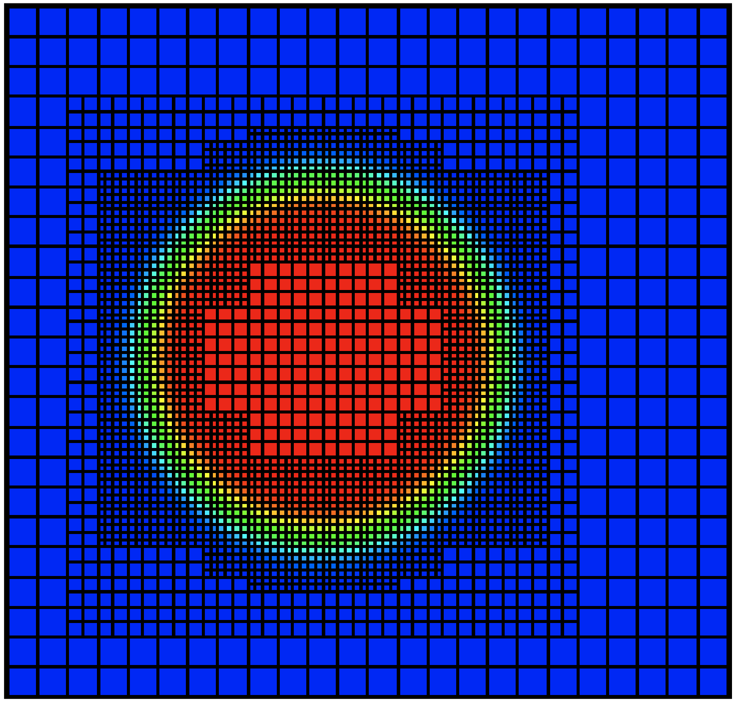
***Strided* domains**

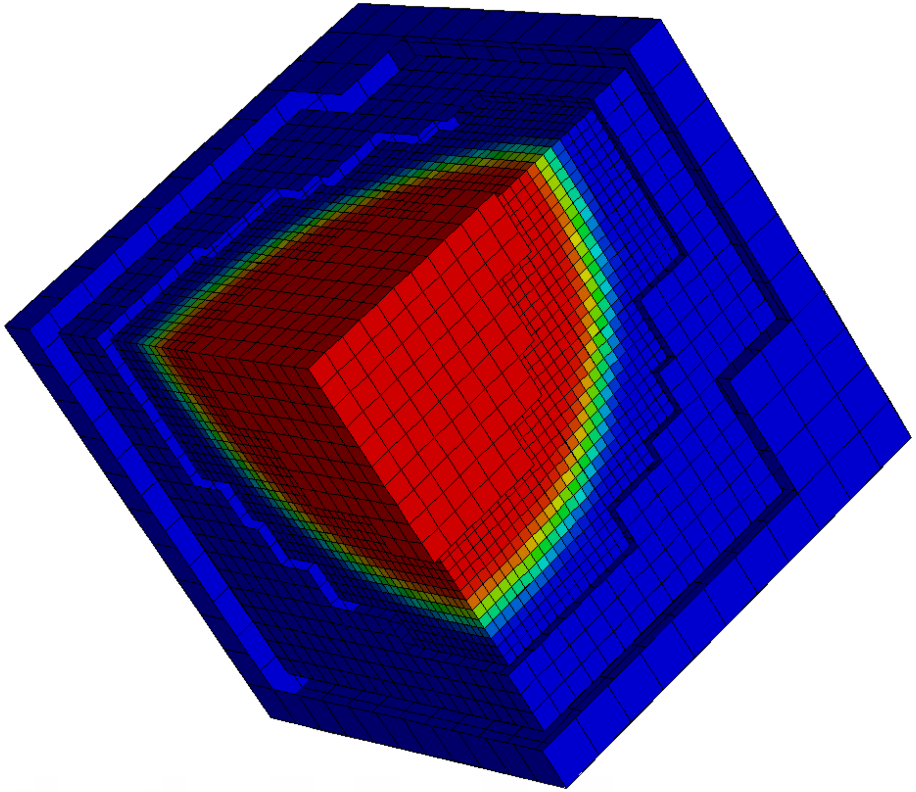
- Array and iteration syntax are **unchanged**
- Chapel helps describe the mathematical problem much more robustly

```
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: Dimension independence

- Use the same code to produce results in 2D, 3D, 6D, 17D…





Grids: Dimension independence

- Goal: Use rank-independent domain construction to define a grid of arbitrary spatial dimension

Grids: Dimension independence

- Goal: Use rank-independent domain construction to define a grid of arbitrary spatial dimension
- Begin by setting
  ```
  config param dimension: int;
  ```
  Specifies a compile-time constant (`param`) that may be specified at the command line (`config`).

Grids: Dimension independence

- Goal: Use rank-independent domain construction to define a grid of arbitrary spatial dimension

- Begin by setting

```
config param dimension: int;
```

  Specifies a compile-time constant (**param**) that may be specified at the command line (**config**).

- A grid is defined by:

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

Coordinate bounds

Coordinate bounds

Width of ghost cell layer

Lower index bound

Grids: Dimension independence

- Goal: Use rank-independent domain construction to define a grid of arbitrary spatial dimension

- Begin by setting

```
config param dimension: int;
```

  Specifies a compile-time constant (**param**) that may be specified at the command line (**config**).

- A grid is defined by:

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

Coordinate bounds

Coordinate bounds

Width of ghost cell layer

Lower index bound

Types `dimension*real` and `dimension*int` are tuples, a native type.

Grids: Dimension independence

- Domain of interior cells:

```
var subranges: dimension*range(stridable=true);

for d in 1
  subrange                                               _cells(d);

var cells: domain(dimension, stridable=true);
cells = subranges;
```

Temporary variable to store sub-ranges of the domain as they are defined

Grids: Dimension independence

- Domain of interior cells:

```
var subranges: dimension*range(stridable=true);

for d in 1..dimension do
  subranges(d) = (i_low(d)+1 .. by 2) #n_cells(d);

var cel                                              ;
cells =
```

Assign subranges in each dimension; this is the only place that the dimensions are unrolled

Grids: Dimension independence

- Domain of interior cells:

```
var subranges: dimension*range(stridable=true);

for d in 1..dimension do
  subranges(d) = (i_low(d)+1 .. by 2) #n_cells(d);

var cells: ...ble=
cells = cel...
```

Unbounded range with correct lower bound and stride

Count operator: Extracts `n_cells(d)` elements

Grids: Dimension independence

- Domain of interior cells:

```
var subranges: dimension*range(stridable=true);

for d in 1..dimension do
  subranges(d) = (i_low(d)+1 .. by 2) #n_cells(d);

var cells: domain(dimension, stridable=true);
cells = subranges;
```

Define the domain `cells`

Grids: Dimension independence

- Domain of interior cells:

```
var subranges: dimension*range(stridable=true);

for d in 1..dimension do
  subranges(d) = (i_low(d)+1 .. by 2) #n_cells(d);

var cells: domain(dimension, stridable=true);
cells = subranges;
```

- Domain of all cells, including ghost cells (spatial variables will be defined here):

```
var extended_cells = cells.expand(2*ghost_layer_width);
```

Cell centers are two indices apart

Grids: Dimension independence

- Domain of interior cells:

```
var subranges: dimension*range(stridable=true);

for d in 1..dimension do
  subranges(d) = (i_low(d)+1 .. by 2) #n_cells(d);

var cells: domain(dimension, stridable=true);
cells = subranges;
```

- Domain of all cells, including ghost cells (spatial variables will be defined here):

```
var extended_cells = cells.expand(2*ghost_layer_width);
```

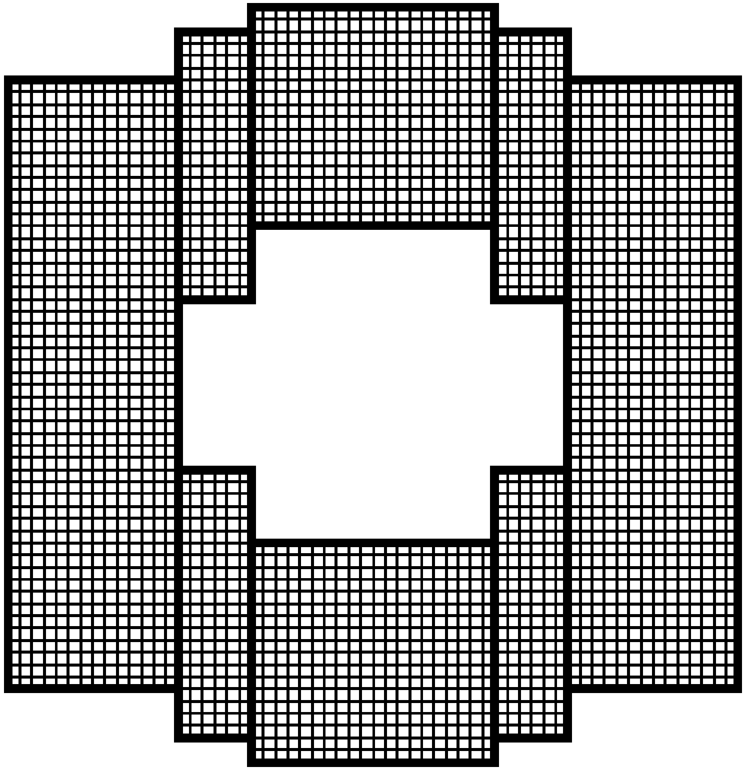
Cell centers are two indices apart

- Array declarations are automatically rank-independent:

```
var spatial_variable: [extended_cells] real;
```

Levels

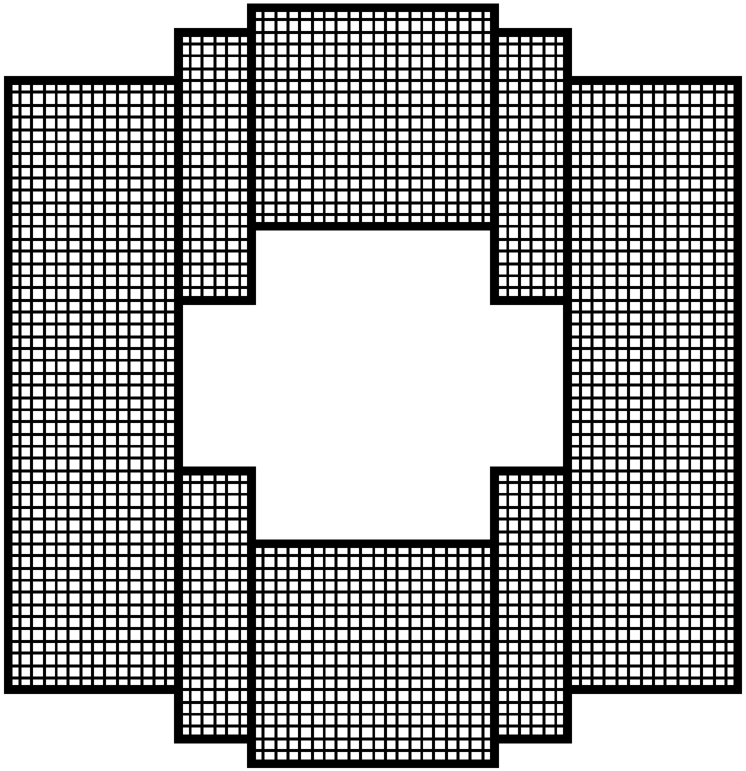
- Essentially a union of grids



Levels

- Essentially a union of grids

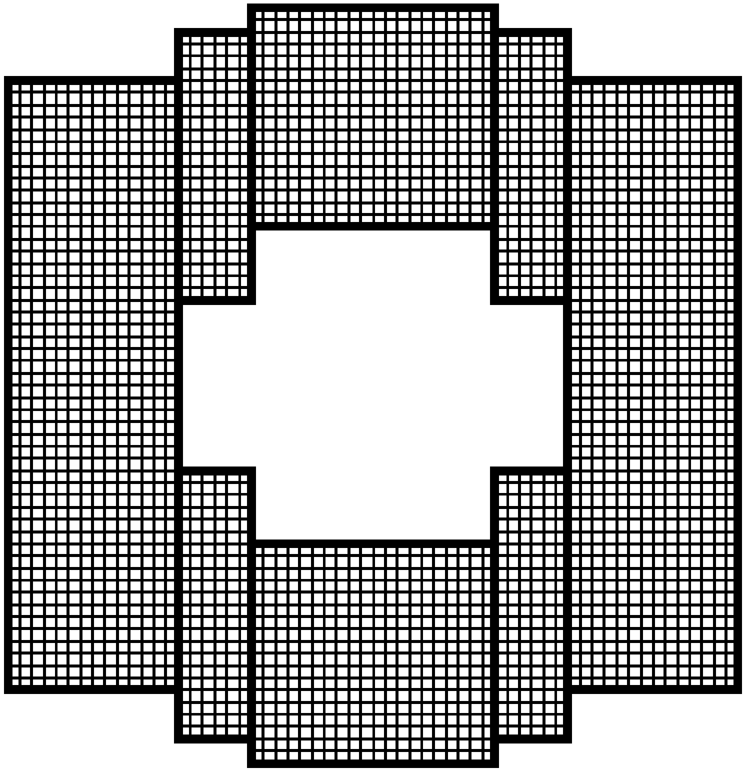
```
var grids: domain(Grid);
```



Levels

- Essentially a union of grids

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

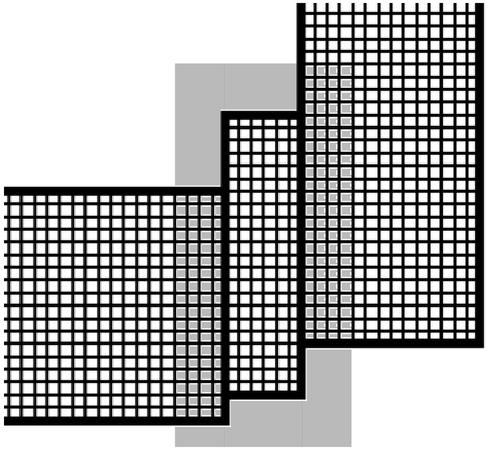


Associative domain

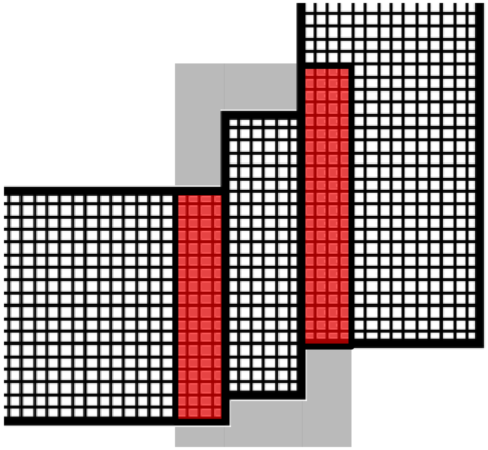
- List of indices of *any* type
- Array and iteration syntax are **unchanged**

Levels: Sibling overlaps

- A grid's layer of ghost cells will, in general, overlap some of its siblings. Data will be copied into these overlapped ghost cells prior to mathematical operations.



Levels: Sibling overlaps

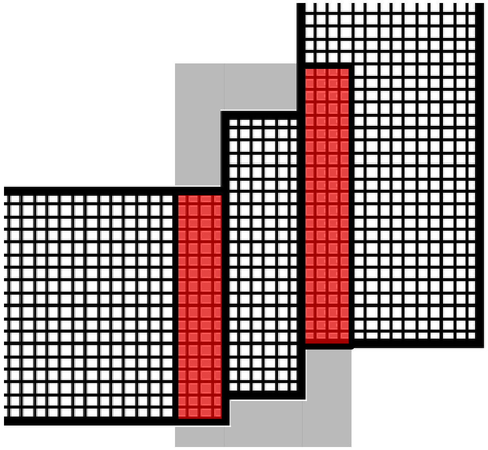


- A grid's layer of ghost cells will, in general, overlap some of its siblings. Data will be copied into these overlapped ghost cells prior to mathematical operations.

- Calculating the **overlaps** between siblings:

Levels: Sibling overlaps

- A grid's layer of ghost cells will, in general, overlap some of its siblings. Data will be copied into these overlapped ghost cells prior to mathematical operations.

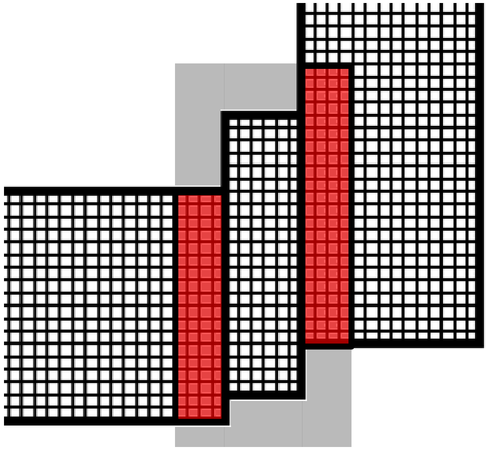


- Calculating the **overlaps** between siblings:

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

for sibl
  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: Sibling overlaps



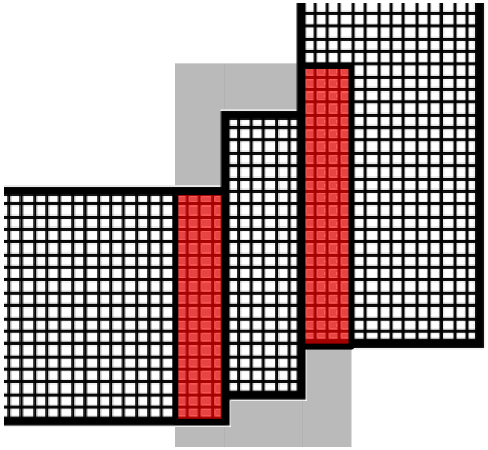
- A grid's layer of ghost cells will, in general, overlap some of its siblings. Data will be copied into these overlapped ghost cells prior to mathematical operations.

- Calculating the **overlaps** between siblings:

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

for sibling …
  var overl… g.cells );

  if overla… g != this {
    neighbors.add(sibling);
    overlaps(sibling) = overlap;
  }
}
```

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

Levels: Sibling overlaps



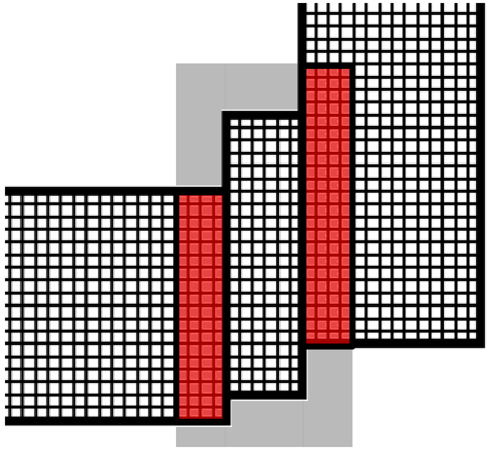
- A grid's layer of ghost cells will, in general, overlap some of its siblings. Data will be copied into these overlapped ghost cells prior to mathematical operations.

- Calculating the **overlaps** between siblings:

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

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

  if overl                   bling != this {
    neighb
    overlaps(sibling) = overlap;
  }
}
```

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

Levels: Sibling overlaps



- A grid's layer of ghost cells will, in general, overlap some of its siblings. Data will be copied into these overlapped ghost cells prior to mathematical operations.

- Calculating the **overlaps** between siblings:

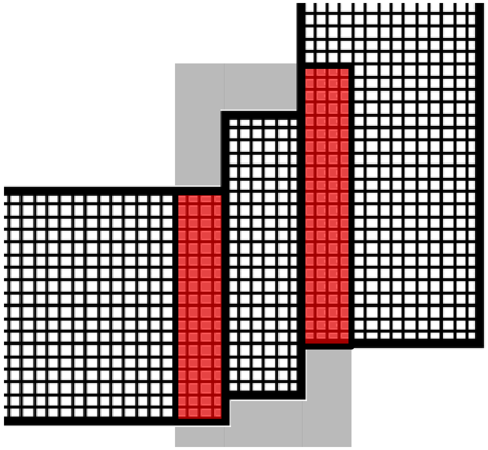
```
var neighbors: domain(Grid);
var overlaps:  [neighbors] domain(dimension,stridable=true);

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

Computes intersection of the domains `extended_cells` and `sibling.cells`.

Take a moment to appreciate what this calculation would look like without domains!

Levels: Sibling overlaps

- A grid's layer of ghost cells will, in general, overlap some of its siblings. Data will be copied into these overlapped ghost cells prior to mathematical operations.



- Calculating the **overlaps** between siblings:

```
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.

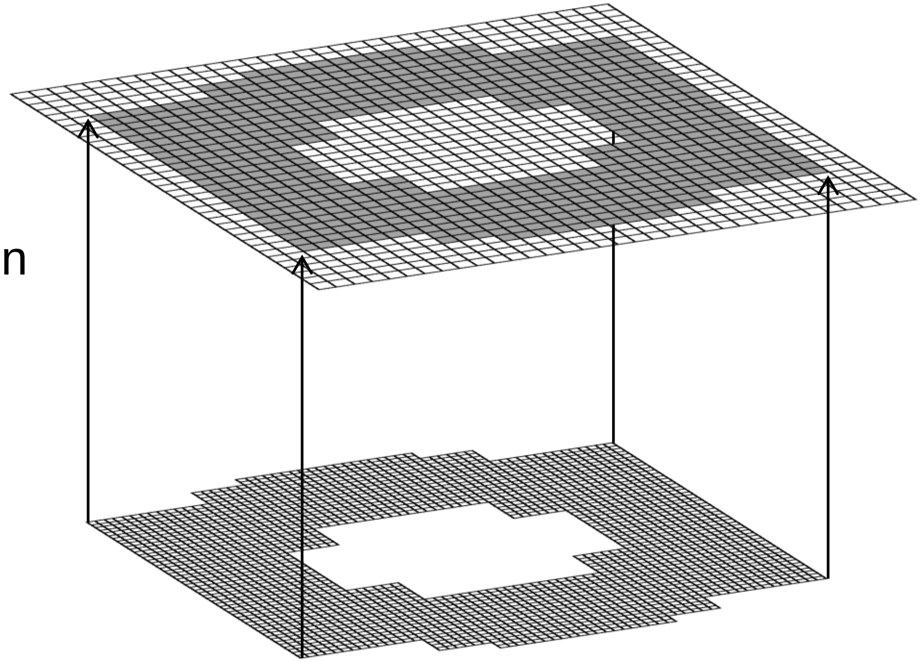
AMR Hierarchy: Brief overview

- Three major challenges
  - Data **coarsening**
  - Data **refinement**
  - **Regridding**

AMR Hierarchy: Brief overview

- Three major challenges
  - Data **coarsening**
  - Data **refinement**
  - **Regridding**

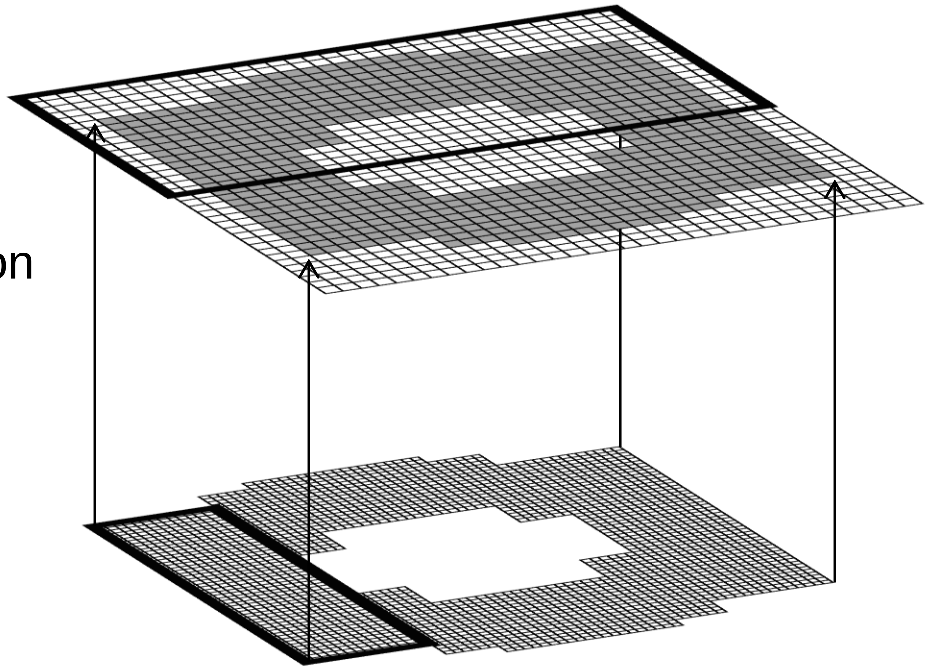
- Coarsening
  - Data transfer occurs on intersection of coarse grid and fine grid



AMR Hierarchy: Brief overview

- Three major challenges
  - Data **coarsening**
  - Data **refinement**
  - **Regridding**

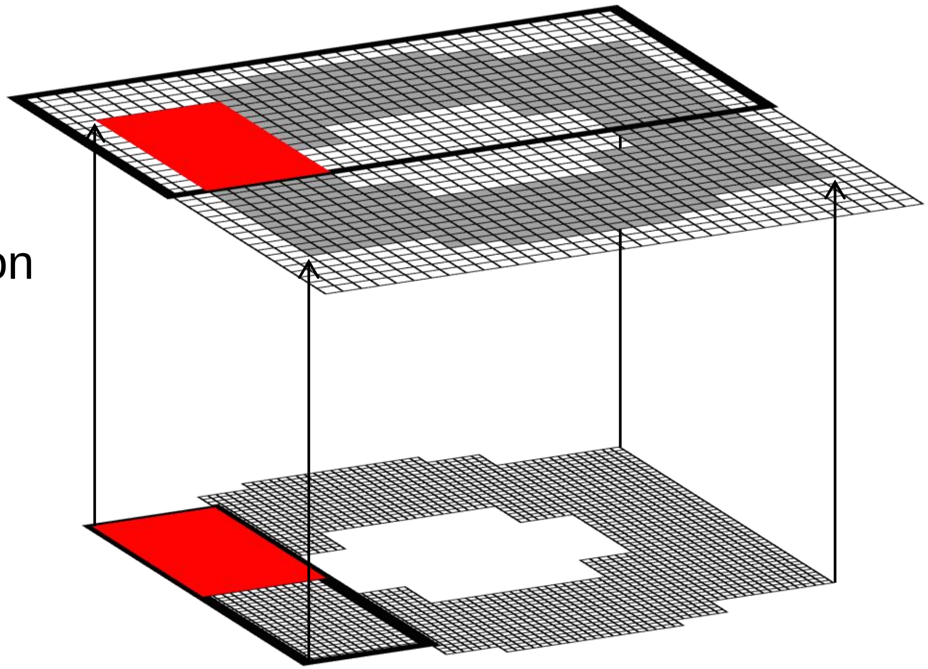
- Coarsening
  - Data transfer occurs on intersection of coarse grid and fine grid



AMR Hierarchy: Brief overview

- Three major challenges
  - Data **coarsening**
  - Data **refinement**
  - **Regridding**

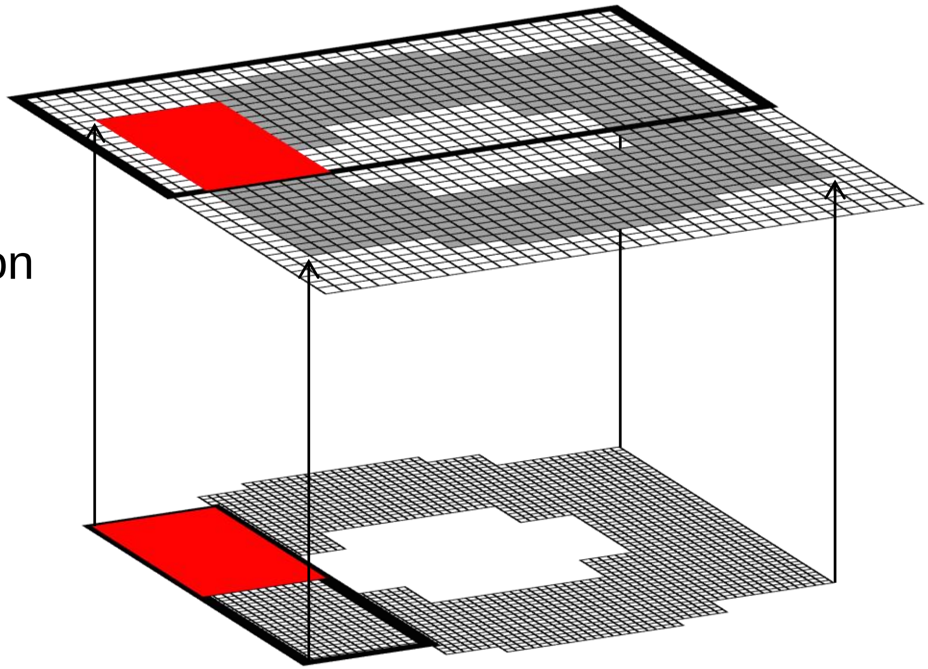
- Coarsening
  - Data transfer occurs on intersection of coarse grid and fine grid



AMR Hierarchy: Brief overview

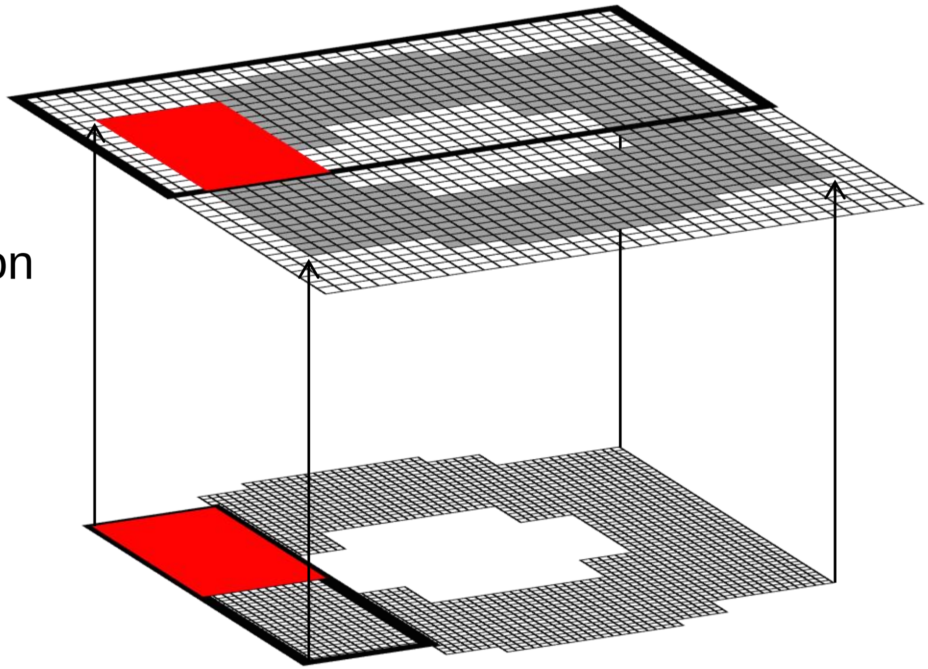
- Three major challenges
  - Data **coarsening**
  - Data **refinement**
  - **Regridding**

- Coarsening
  - Data transfer occurs on intersection of coarse grid and fine grid
  - Region is rectangular – transfer is relatively easy



AMR Hierarchy: Brief overview

- Three major challenges
  - Data **coarsening**
  - Data **refinement**
  - **Regridding**



- Coarsening
  - Data transfer occurs on intersection of coarse grid and fine grid
  - Region is rectangular – transfer is relatively easy

- Refinement and regridding
  - Involve unions and subtractions of rectangles
  - Much harder; subject of next talk

Conclusion

- Chapel domains make many fundamental AMR calculations very easy, even in a dimension-independent setting

Conclusion

- Chapel domains make many fundamental AMR calculations very easy, even in a dimension-independent setting
- Rectangular domains and associative domains are both very important

Conclusion

- Chapel domains make many fundamental AMR calculations very easy, even in a dimension-independent setting
- Rectangular domains and associative domains are both very important
- Haven't discussed objects for data storage, but Chapel's link between domains and arrays makes them easy to define and use

Conclusion

- Chapel domains make many fundamental AMR calculations very easy, even in a dimension-independent setting
- Rectangular domains and associative domains are both very important
- Haven't discussed objects for data storage, but Chapel's link between domains and arrays makes them easy to define and use
- A recap of code size, now that you've seen some of the interesting parts:

Language	Parallelism	SLOC[1]	Tokens	Relative size (tokens)
C++ ($D \leq 6$) [3]	Dist. mem.	40200	261427	100%
Fortran (2D+3D) [2]	Serial	16562	151992	58%
2D		8297	71639	27%
3D		8265	80353	31%
Chapel (any D)	Shared mem.	1988	13783	**5%**

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

Thank You.

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency.

Questions?