Chapel: Sample Codes

Outline

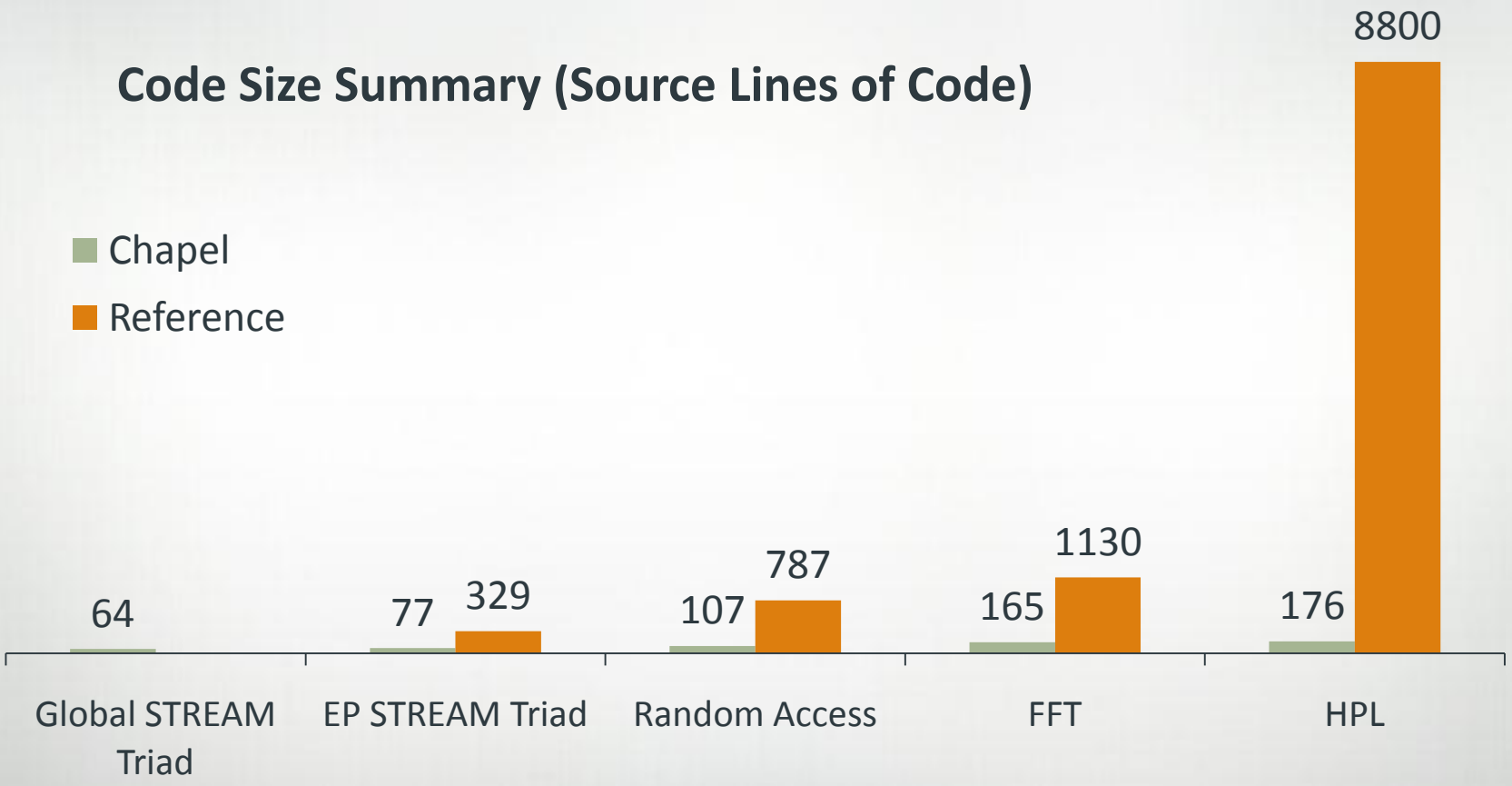
- **STREAM and RA HPC Challenge Benchmarks**
  - simple, regular 1D computations
  - results from SC '09 competition
- **AMR Computations**
  - hierarchical, regular computation
- **SSCA #2**
  - unstructured graph computation

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 may care:**
  - provides an alternative to the top-500's focus on peak performance
- **Recent Class 2 Winners:**

2008: *performance:* IBM (UPC/X10)
productive: Cray (Chapel), IBM (UPC/X10), Mathworks (Matlab)
2009: *performance:* IBM (UPC+X10)
elegance: Cray (Chapel)

Chapel Implementation Characteristics



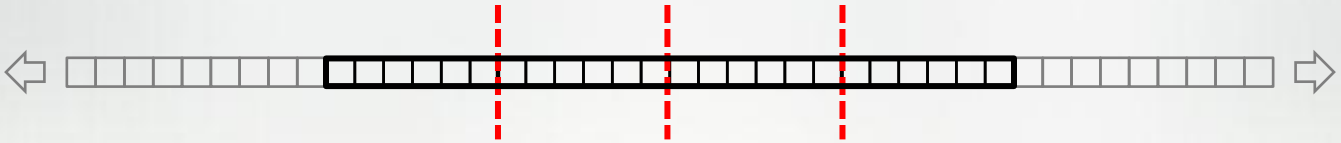
HPC Challenge: Chapel Entries (2008-2009)

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

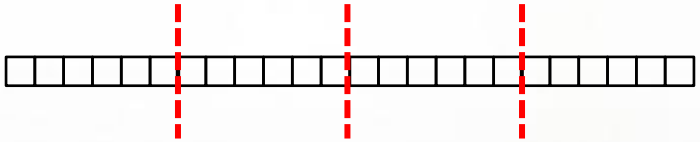
All timings on ORNL Cray XT4:

- 4 cores/node
- 8 GB/node
- no use of library routines

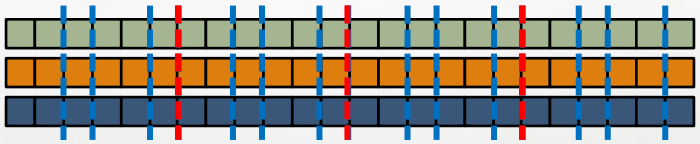
Global STREAM Triad in Chapel (Excerpts)



```
const ProblemSpace: domain(1, int(64))
                    dmapped Block([1..m])
                  = [1..m];
```



```
var A, B, C: [ProblemSpace] real;
```



```
forall (a,b,c) in (A,B,C) do
  a = b + alpha * c;
```

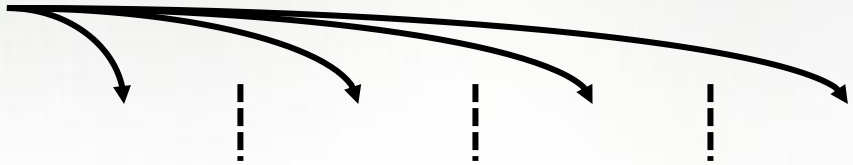
This loop should eventually be written:

```
A = B + alpha * C;
```

(and can be today, but performance is worse)

EP STREAM Triad in Chapel (Excerpts)

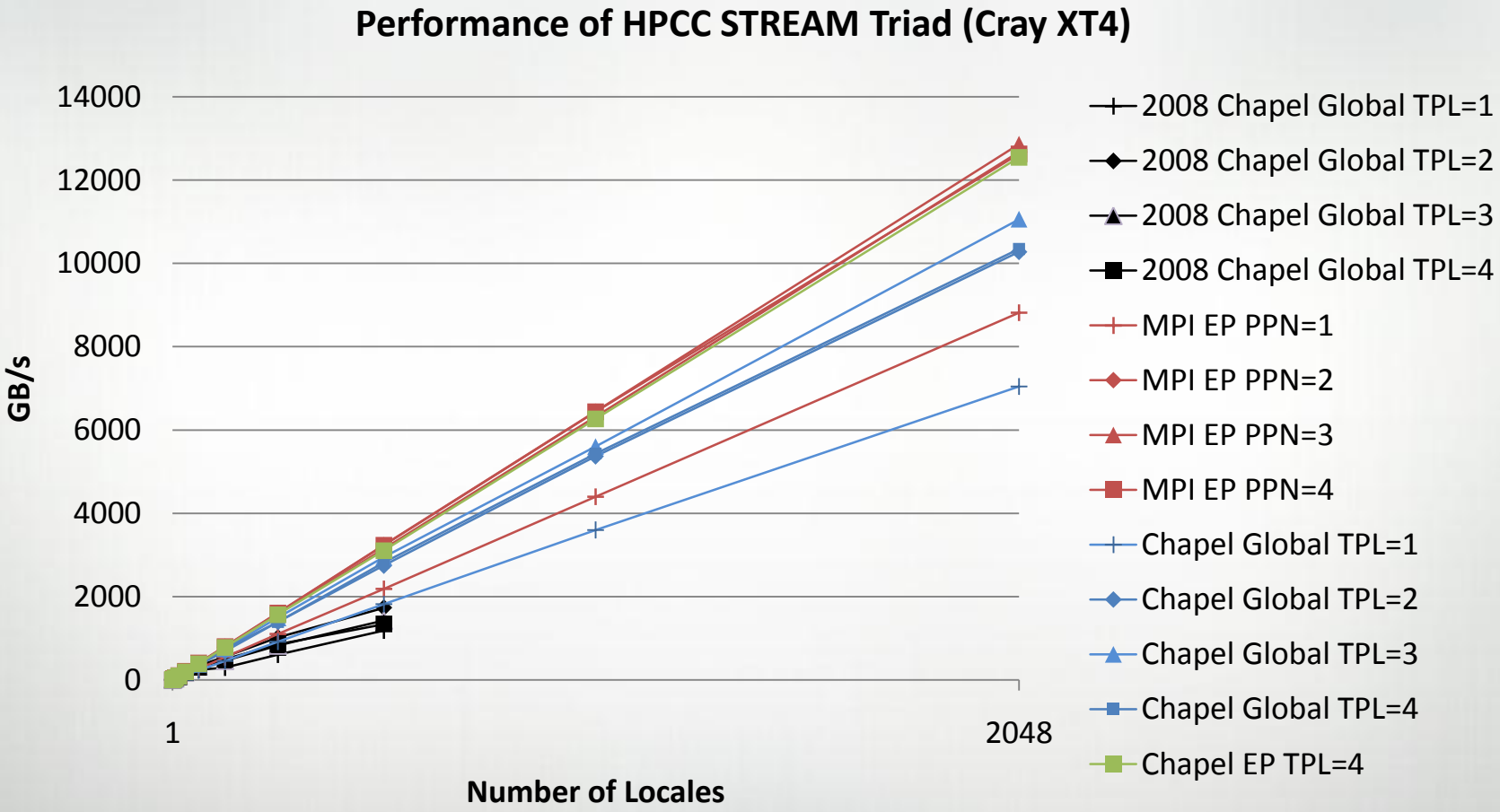
```
coforall loc in Locales do
  on loc {
```



```
  local {
    var A, B, C: [1..m] real;
```

```
    forall (a,b,c) in (A,B,C) do
      a = b + alpha * c;
  }
}
```

STREAM Triad Performance



Global Random Access in Chapel (Excerpts)

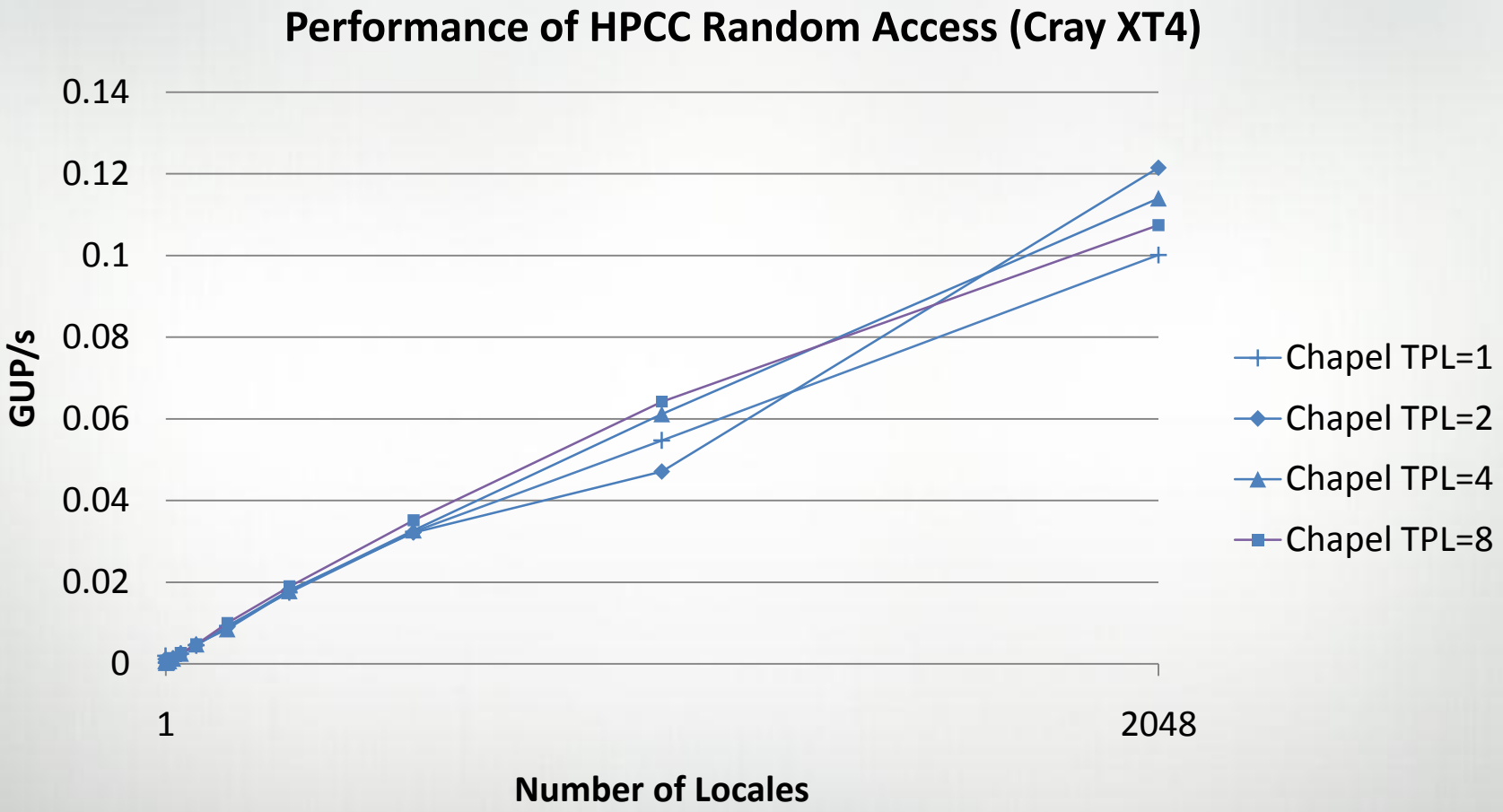
```
const TableDist = new dmap(new Block([0..m-1])),
      UpdateDist = new dmap(new Block([0..N_U-1]));

const TableSpace: domain … dmapped TableDist = …,
      Updates: domain … dmapped UpdateDist = …;

var T: [TableSpace] uint(64);

forall ( ,r) in (Updates,RAStream()) do
  on TableDist.idxToLocale(r & indexMask) {
    const myR = r;
    local T(myR & indexMask) ^= myR;
  }
```

This body should eventually simply be written:

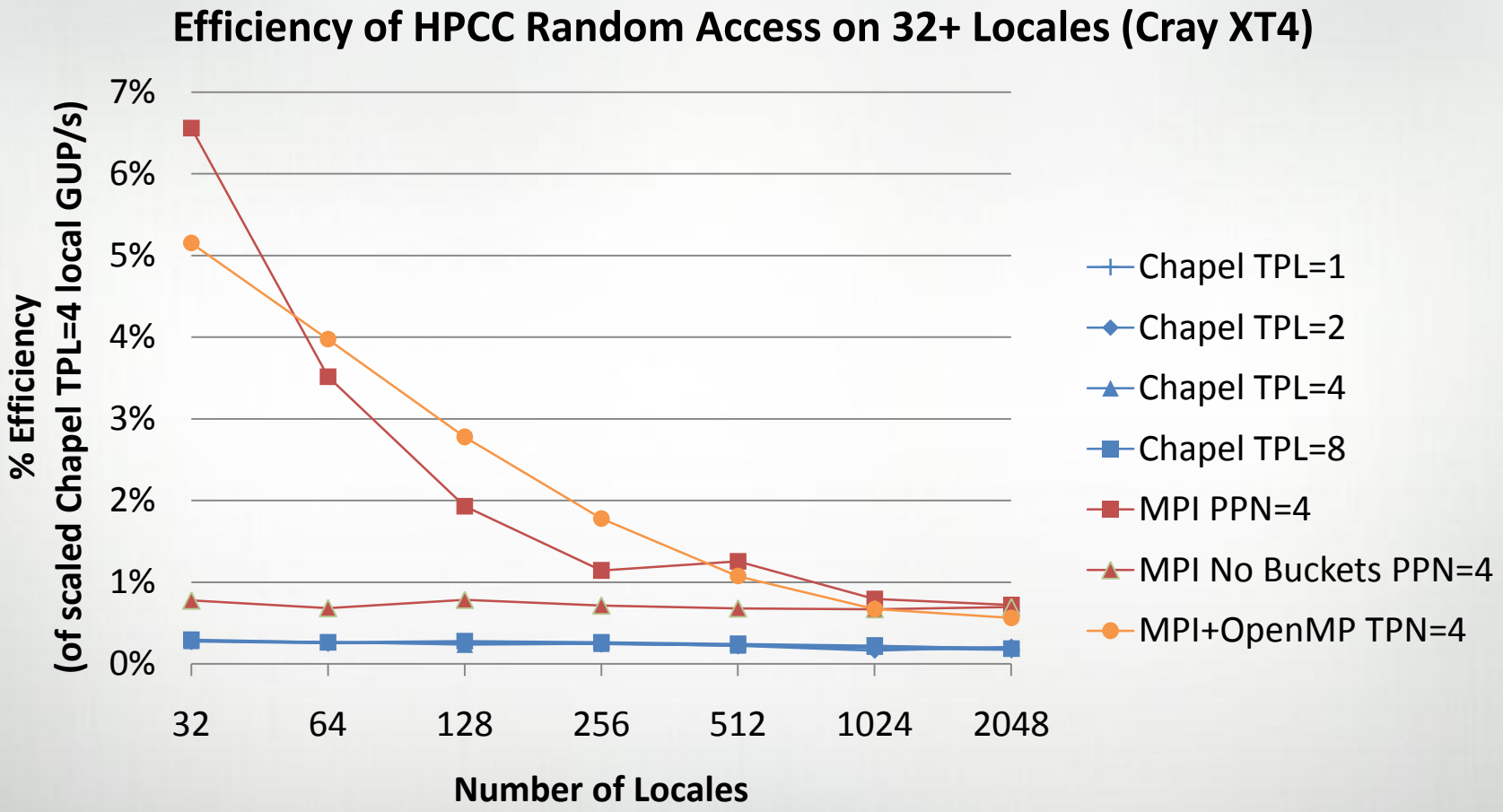
```
on T(r&indexMask) do
  T(r&indexMask) ^= r;
```

(and again, can be today, but performance is worse)

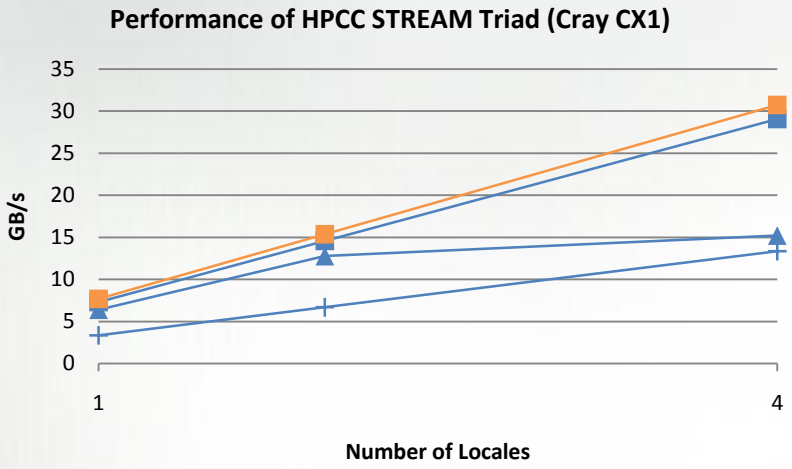
Random Access Performance

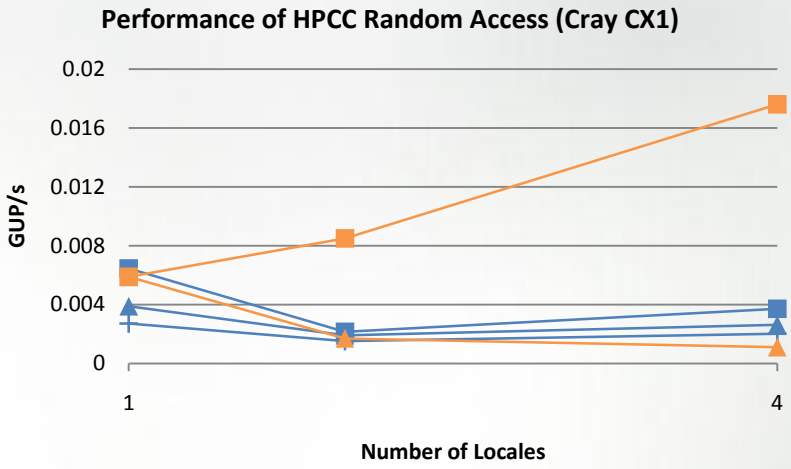


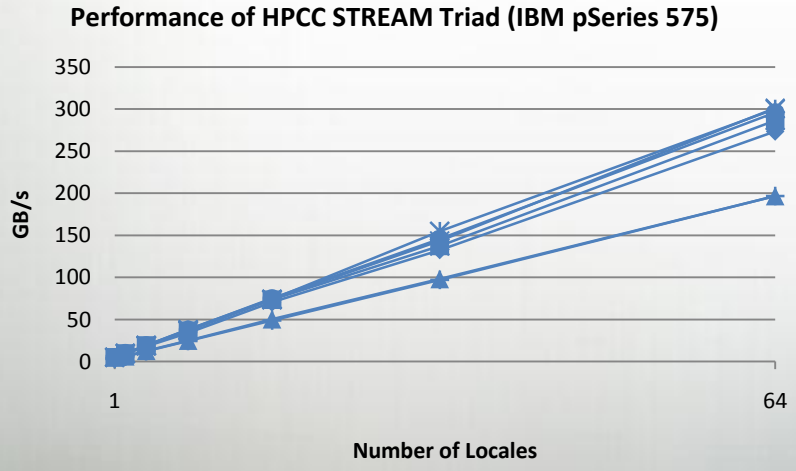
Random Access Efficiency on 32+ Nodes

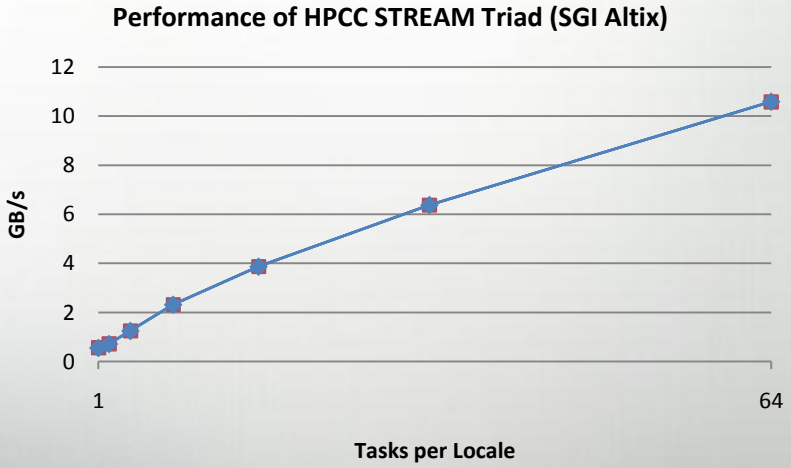


Portability Results









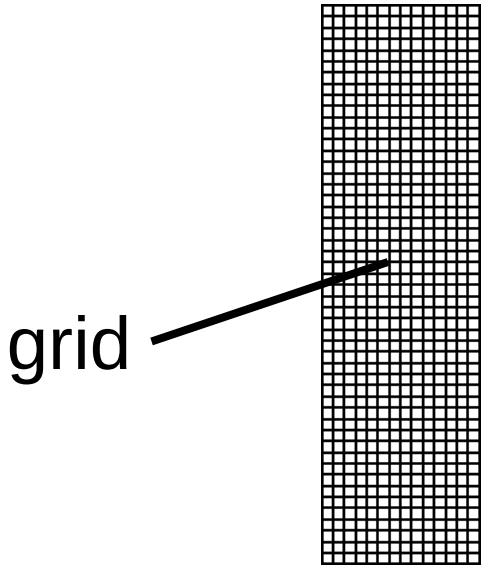
For more on the HPCC Codes

- See $CHPL_HOME/examples/hpcc in the release
- Attend the HPC Challenge BoF on Tuesday (12:15-1:15)

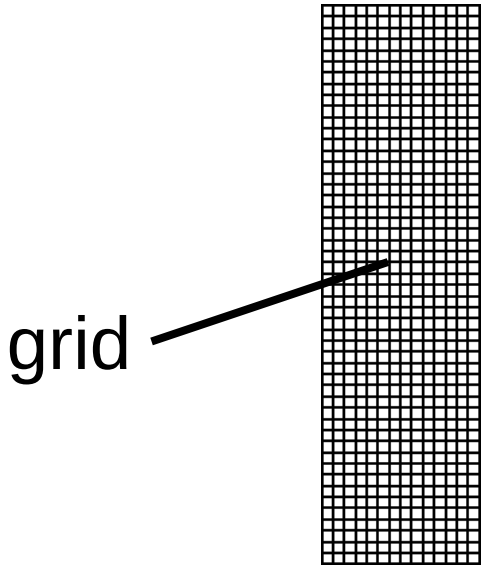
Outline

- **STREAM and RA HPC Challenge Benchmarks**
  - simple, regular 1D computations
  - results from SC '09 competition
- **AMR Computations**
  - hierarchical, regular computation
- **SSCA #2**
  - unstructured graph computation

AMR terminology



AMR terminology

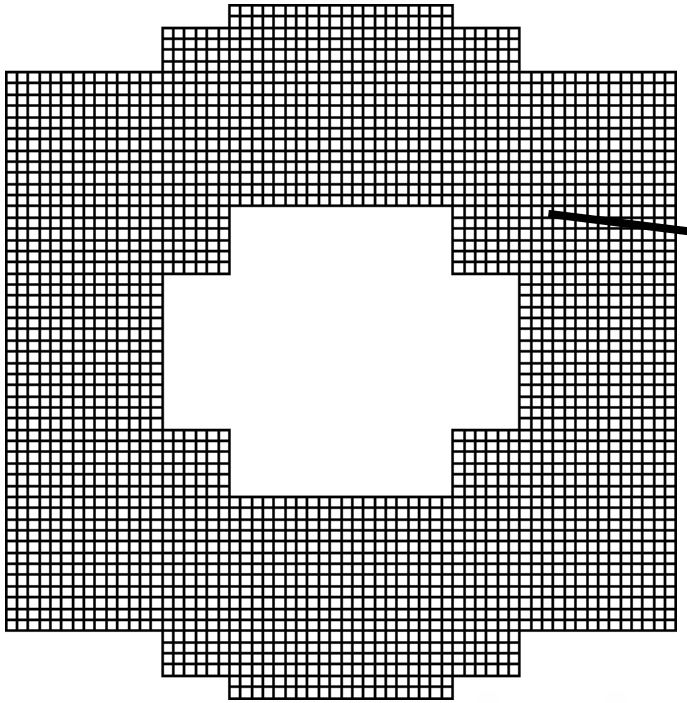


Roughly:

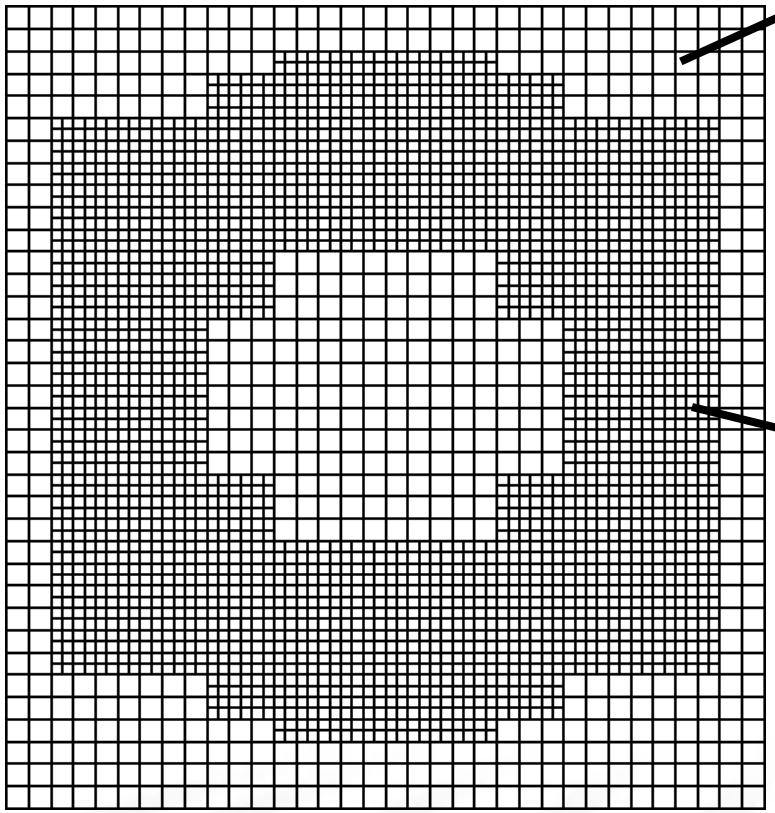
Operations on grids

Operations on rectangular (Chapel) domains

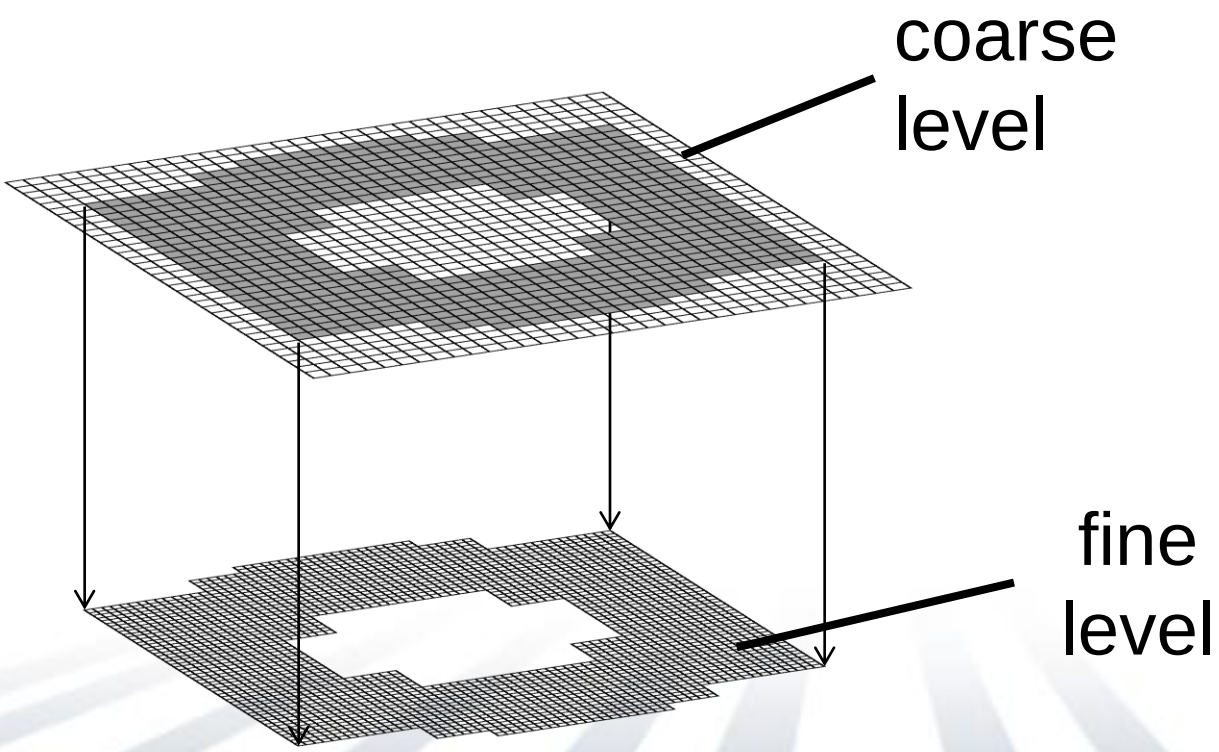
AMR terminology



AMR terminology

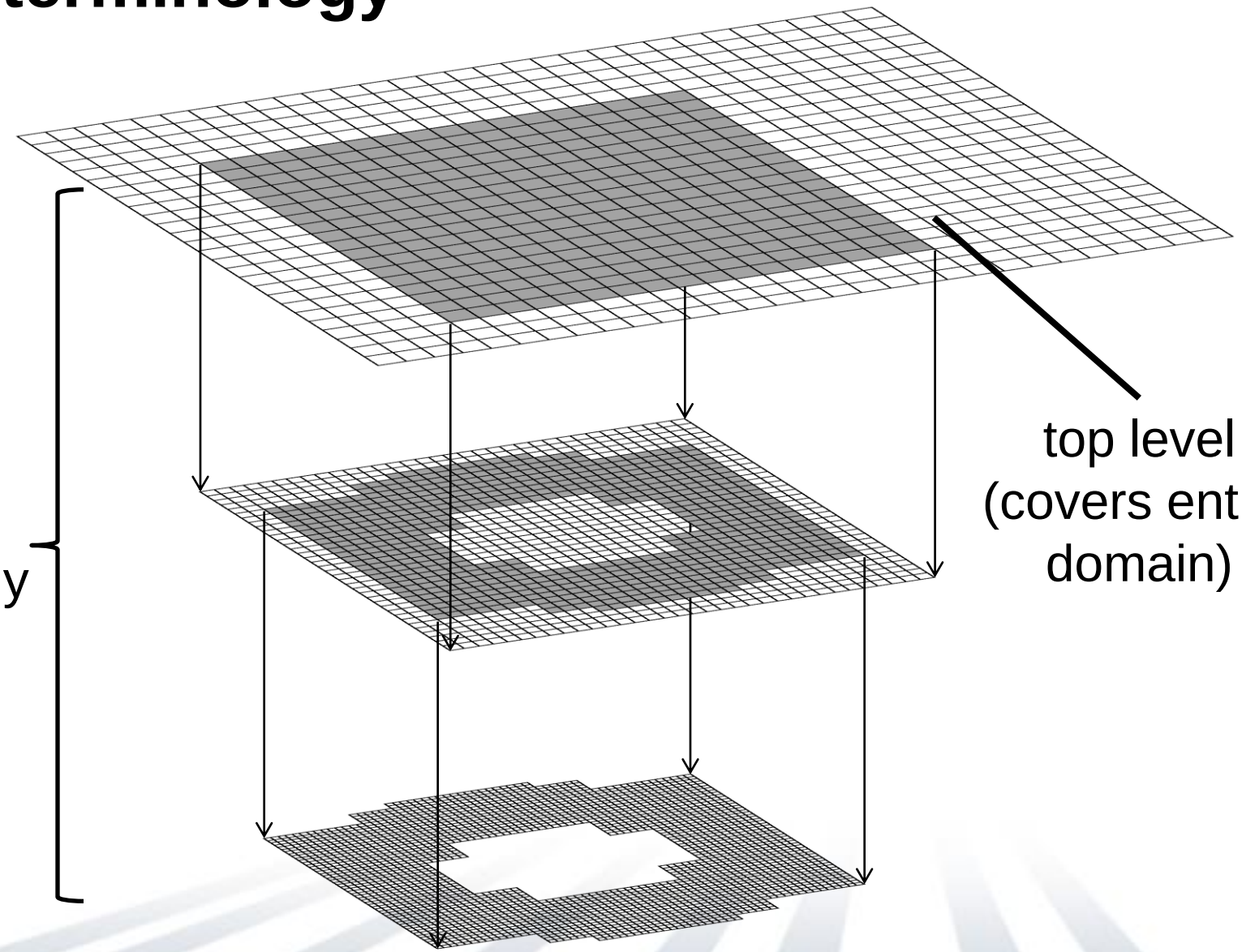


AMR terminology



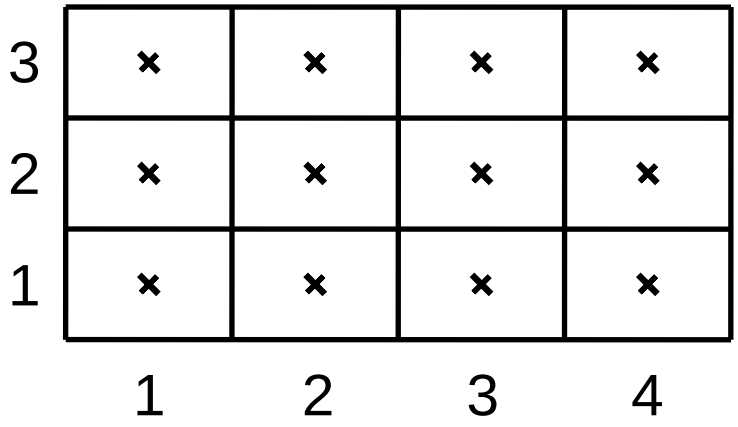
slides courtesy of Jonathan Claridge, UW AMath

AMR terminology



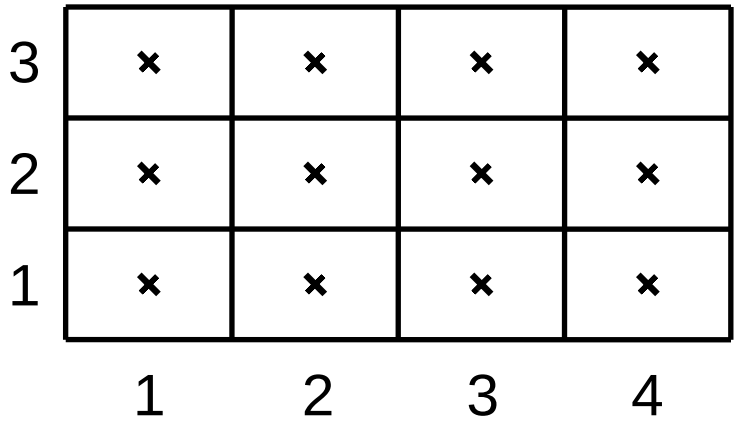
Grids: Indexing

- Conventional indexing – number grid cells sequentially



Grids: Indexing

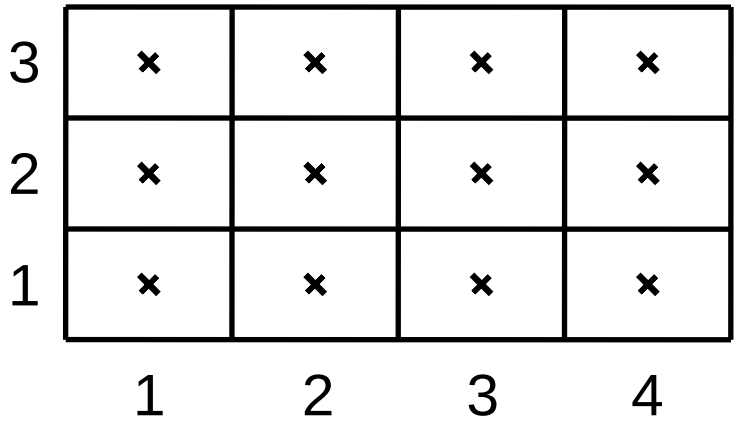
- Conventional indexing – number grid cells sequentially




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

Grids: Indexing

- Conventional indexing – number grid cells sequentially



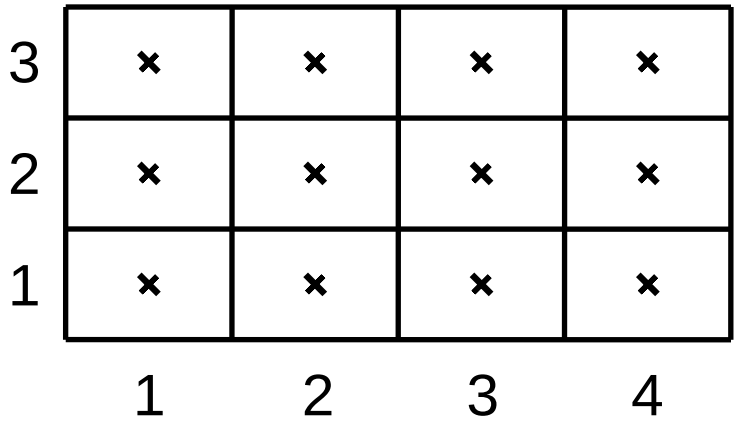

```
const cells = [1..4, 1..3];
```

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

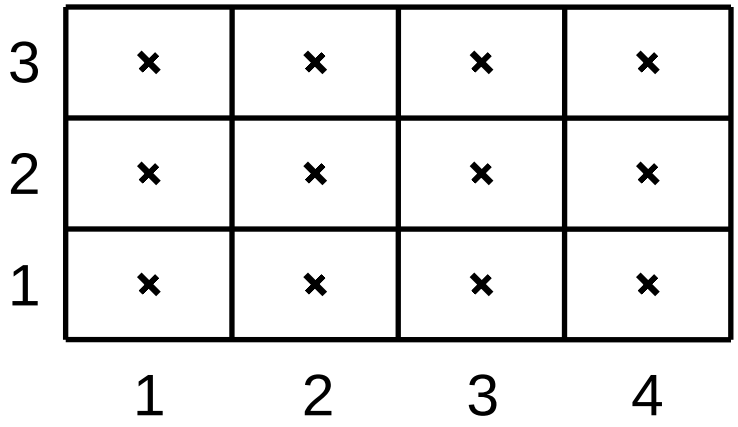



```
const cells = [1..4, 1..3];
```

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

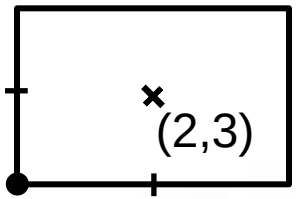
Grids: Indexing

- Conventional indexing – number grid cells sequentially



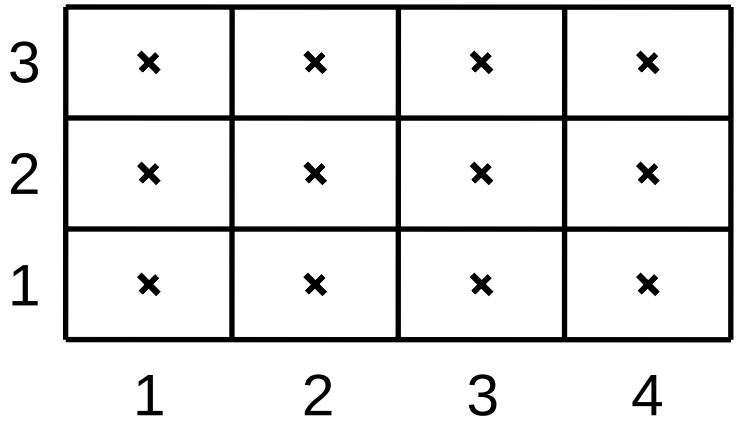

```
const cells = [1..4, 1..3];
```

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



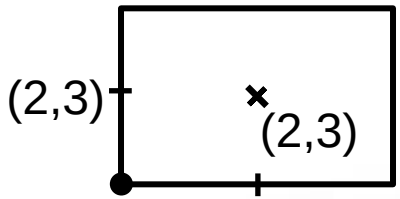
Grids: Indexing

- Conventional indexing – number grid cells sequentially



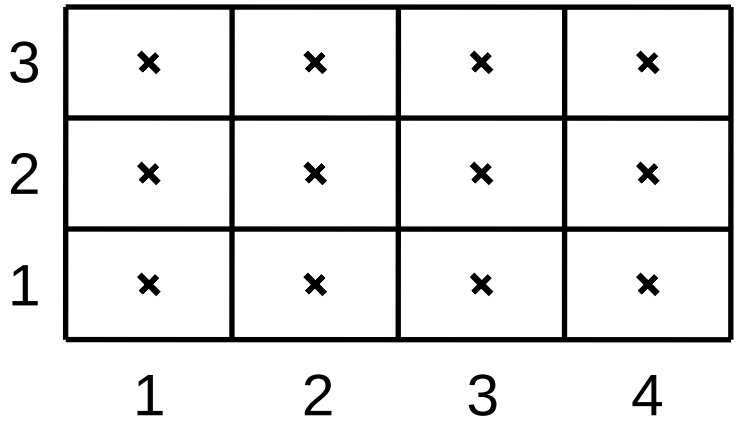

```
const cells = [1..4, 1..3];
```

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



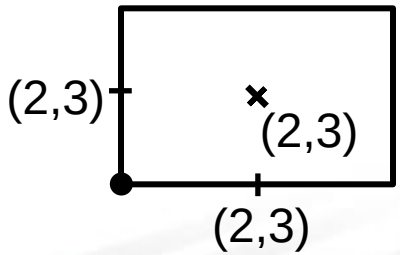
Grids: Indexing

- Conventional indexing – number grid cells sequentially



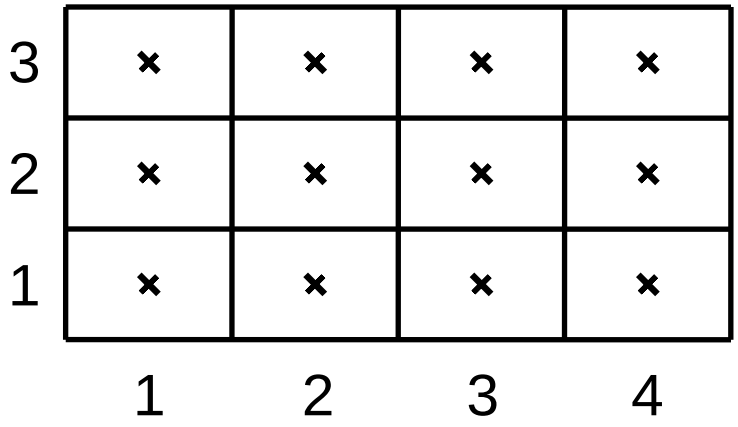

```
const cells = [1..4, 1..3];
```

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



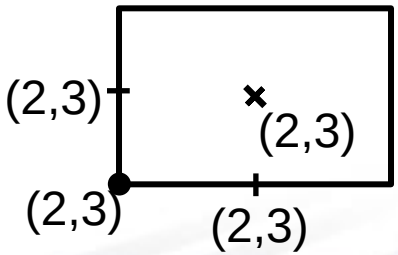
Grids: Indexing

- Conventional indexing – number grid cells sequentially



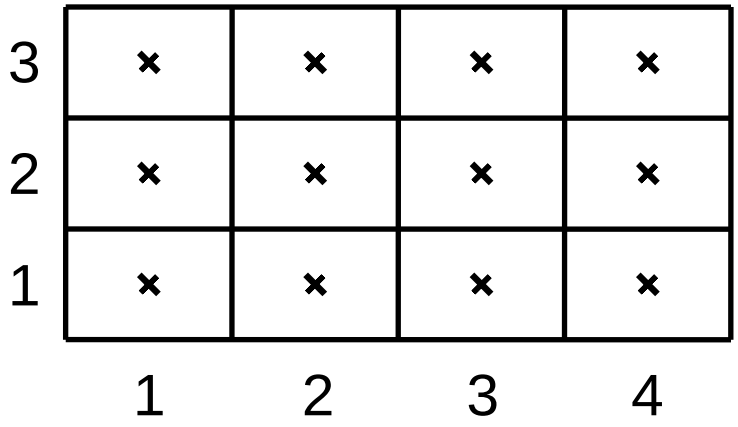

```
const cells = [1..4, 1..3];
```

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



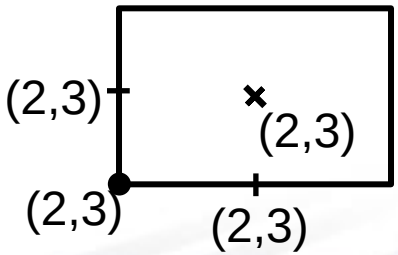
Grids: Indexing

- Conventional indexing – number grid cells sequentially




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

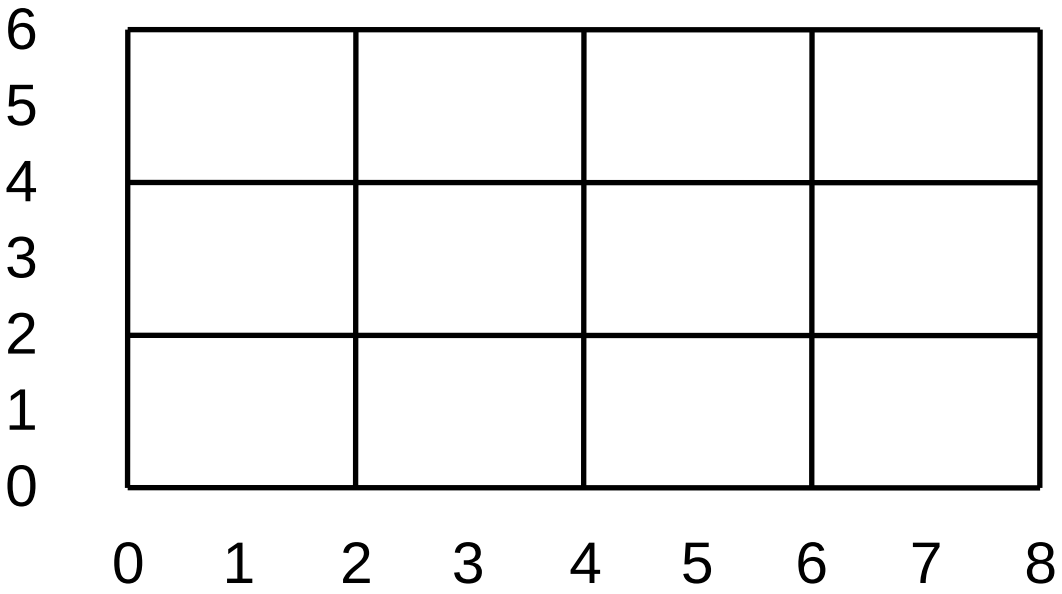
- Problem with conventional indexing: How are the interfaces indexed?
  - Usual approach: Interface has the same index as the cell above it



Many objects will have the same indices

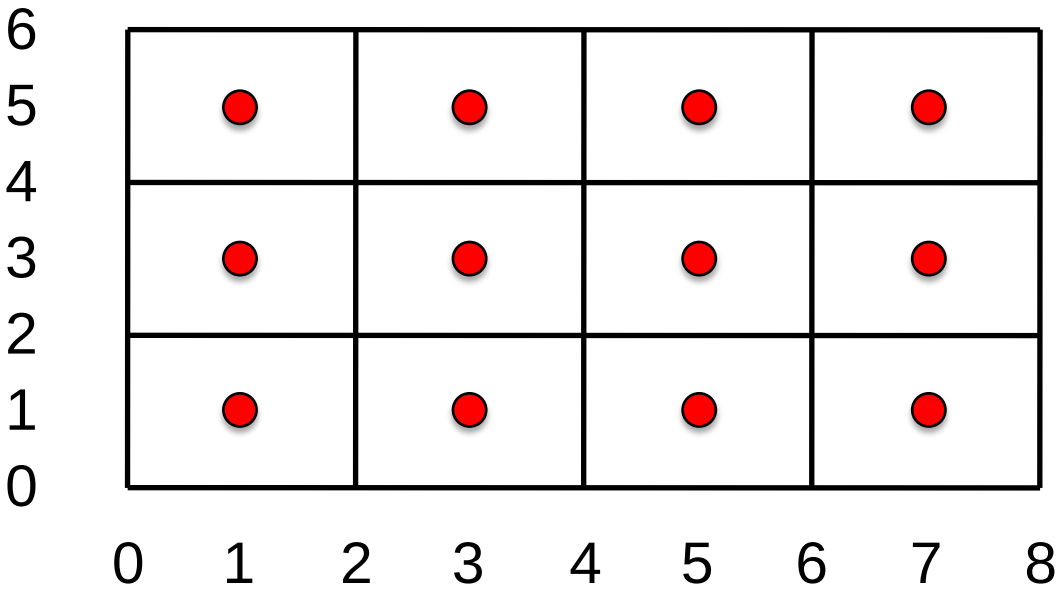
Grids: Indexing

- Modified approach – denser index space



Grids: Indexing

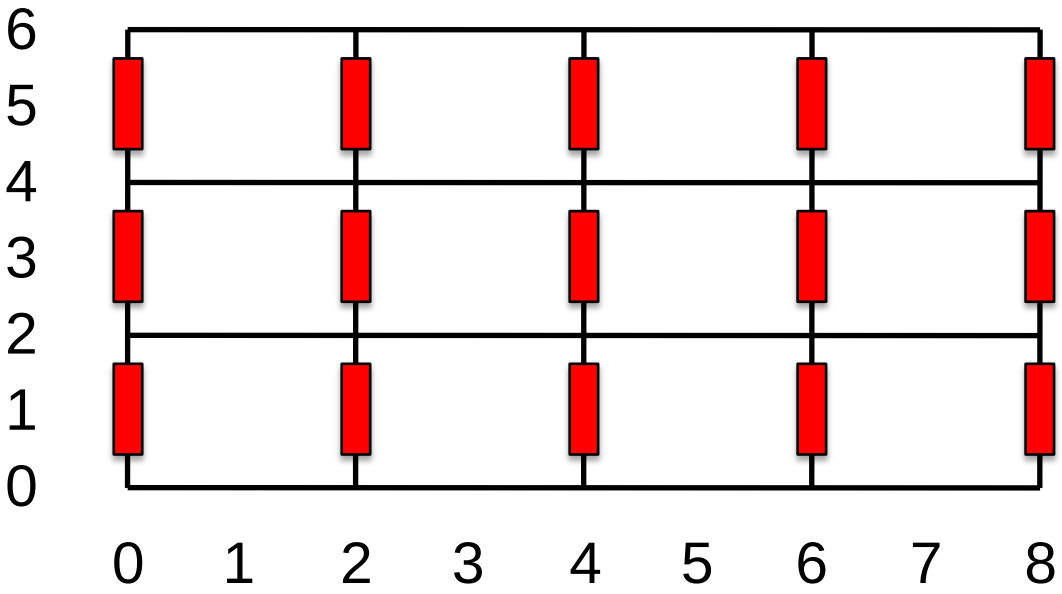
- Modified approach – denser index space




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

Grids: Indexing

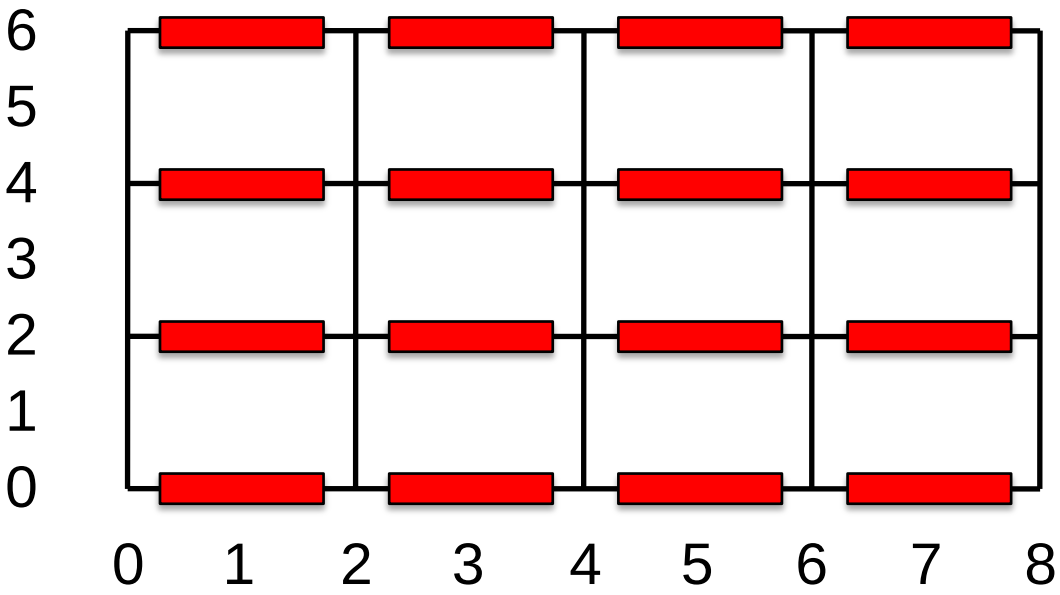
- Modified approach – denser index space




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

Grids: Indexing

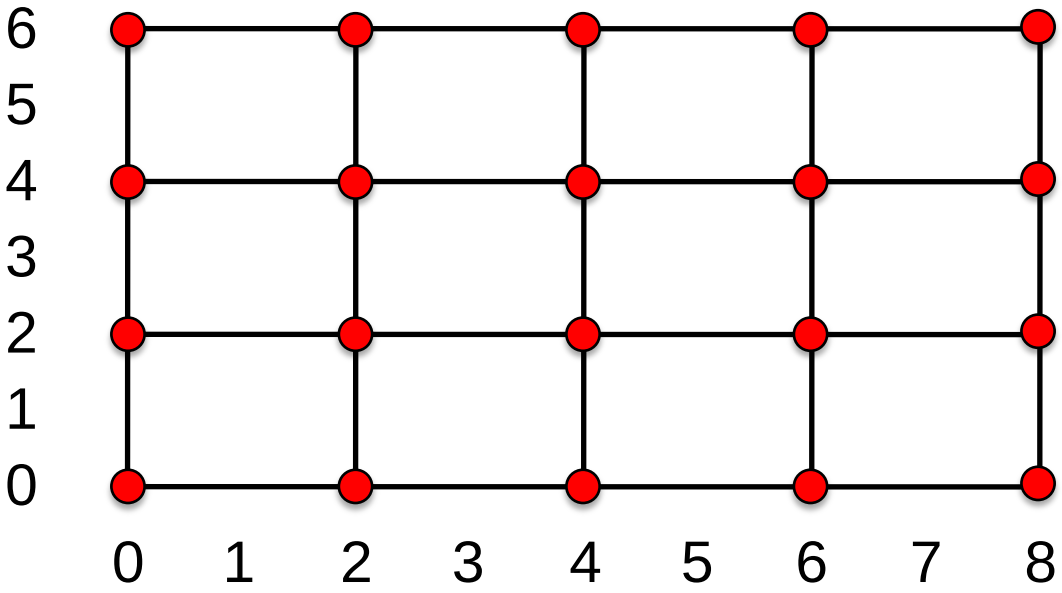
- Modified approach – denser index space




```
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

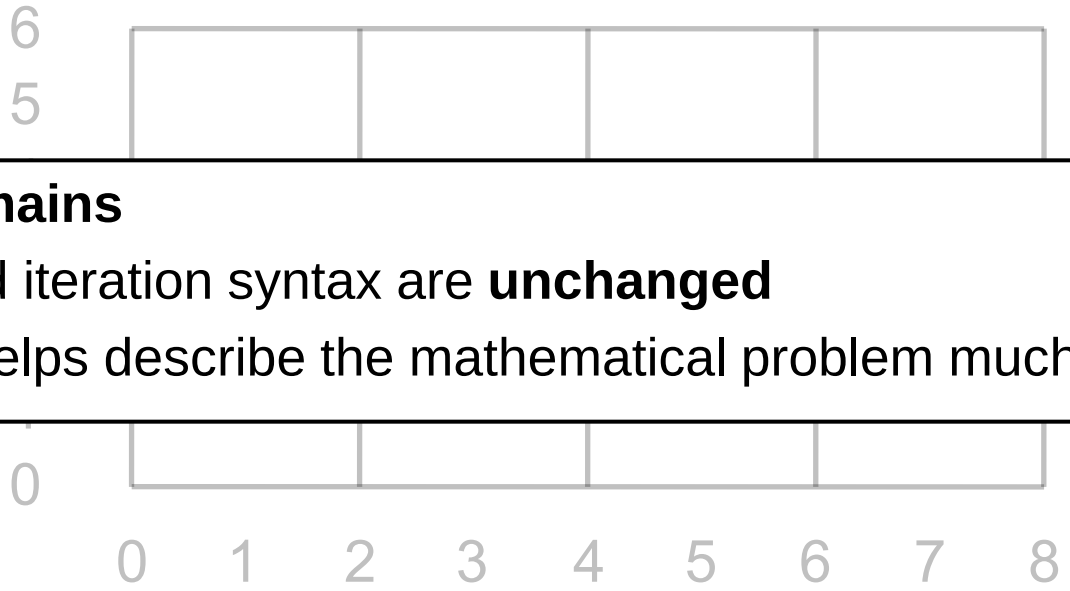
- Modified approach – denser index space




```
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



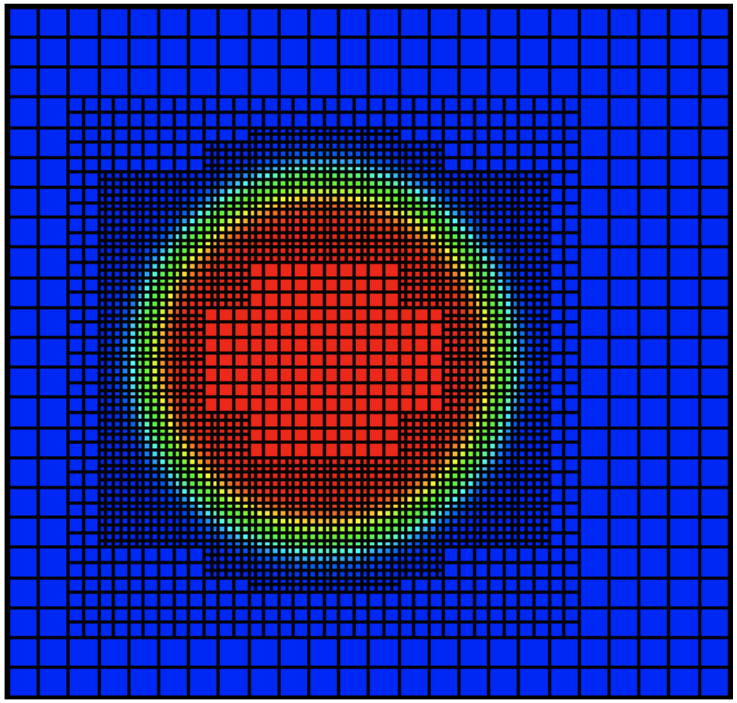
***Strided* domains**

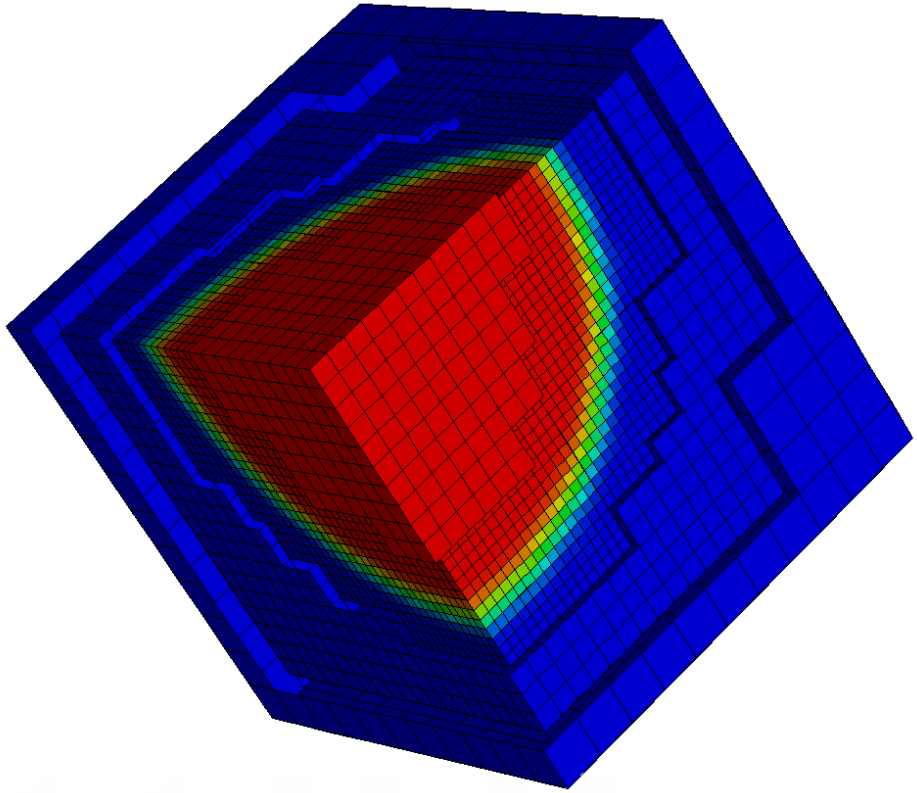
- Array and iteration syntax are **unchanged**
- Chapel helps describe the mathematical problem much more robustly

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

var cells:
cells = cel
```

Unbounded range with correct lower bound and stride

Count operator: Extracts `n_cells(d)` elements

Grids: Dimension independence

- Domain of interior cells:

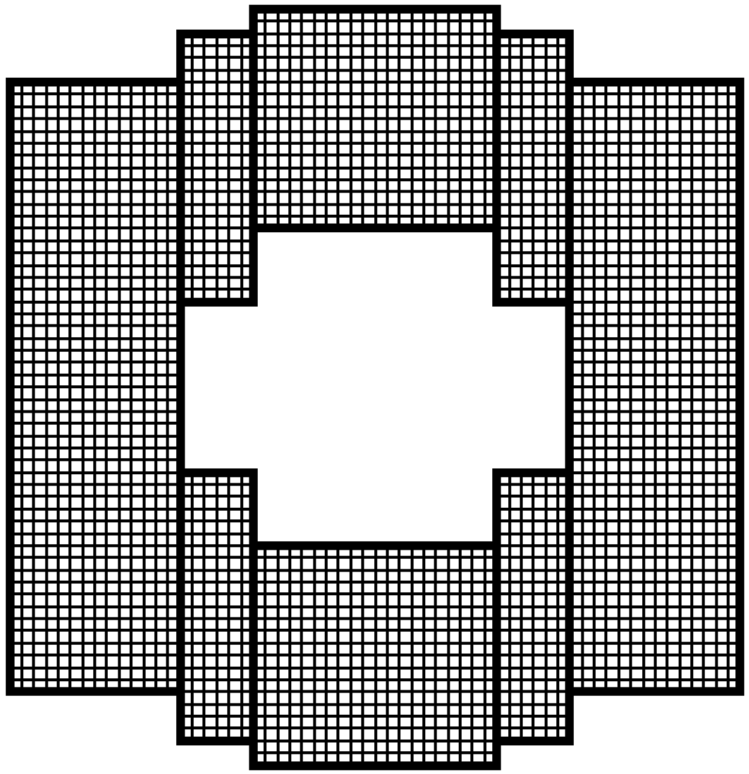
```
var subranges: dimension*range(stridable=true);

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`

Levels

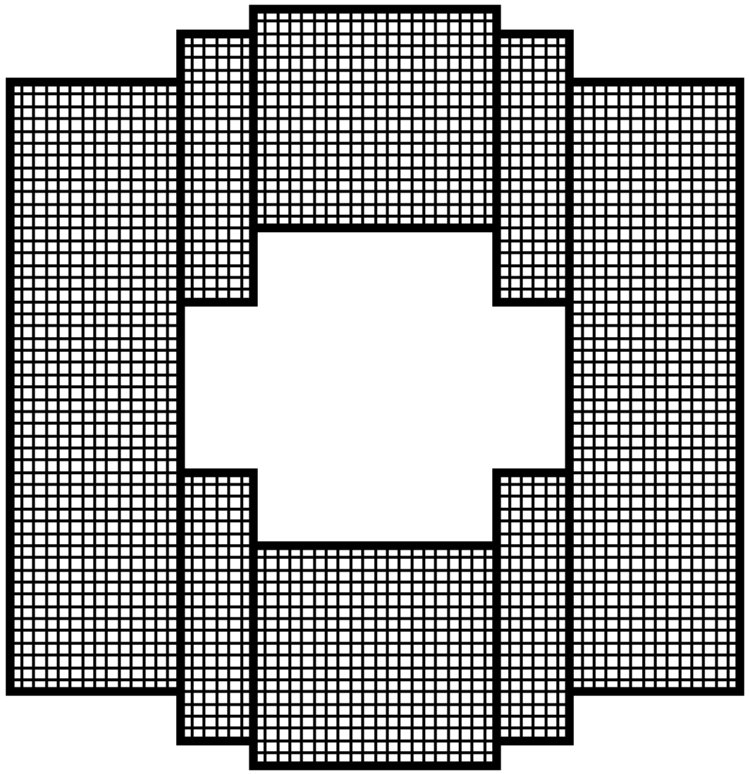
- Essentially a union of grids

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



Levels

- Essentially a union of grids



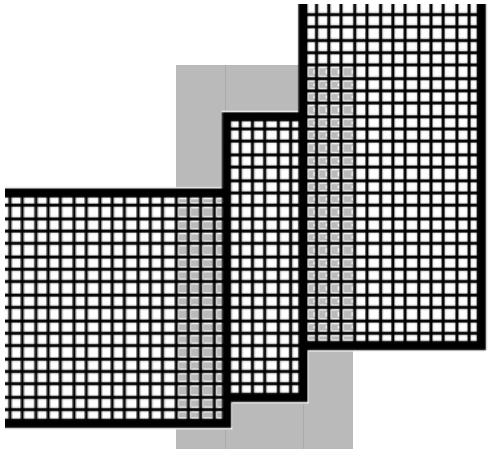
```
var grids: domain(Grid);
```

***Associative* domain**

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

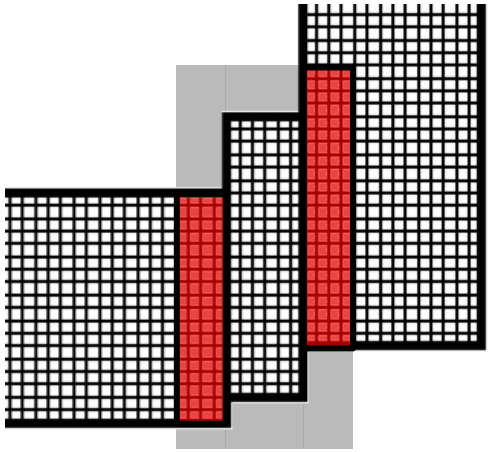
Levels: Sibling overlaps

- 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.



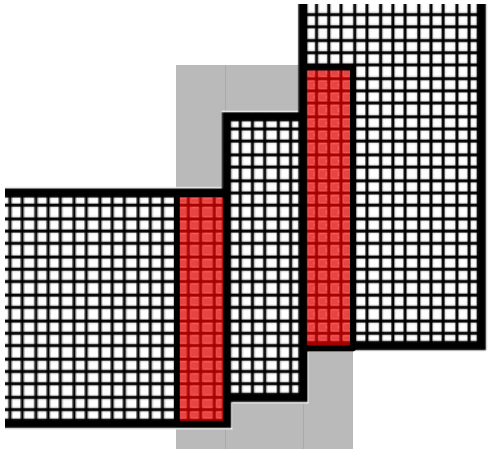
- Calculating the **overlaps** between siblings:

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

for sibling
  var overl                                  g.cells );

  if overlap                                 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.



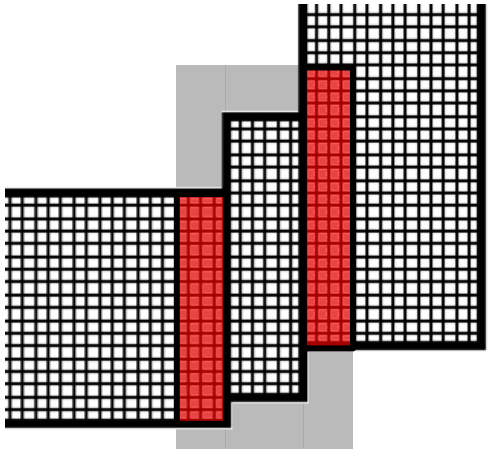
- Calculating the **overlaps** between siblings:

```
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:

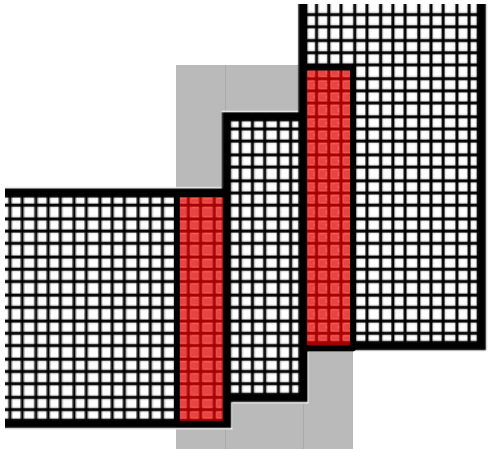
```
var neighbors: domain(Grid);
var overlaps:  [neighbors] domain(dimension,stridable=true);

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;
  }
}
```

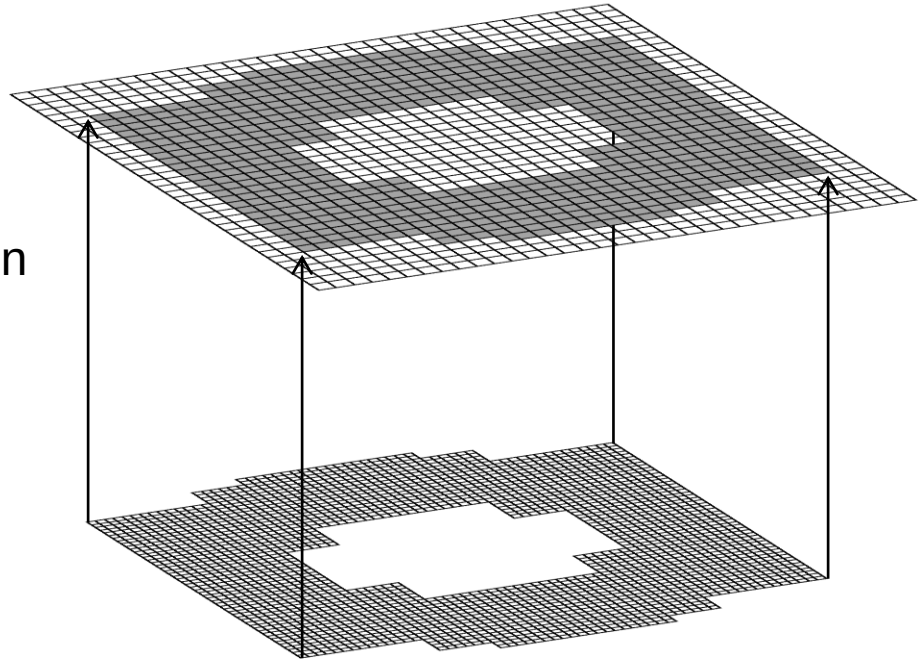
If `overlap` is nonempty, and `sibling` is distinct from this grid, then update stored data.

AMR Hierarchy: Brief overview

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

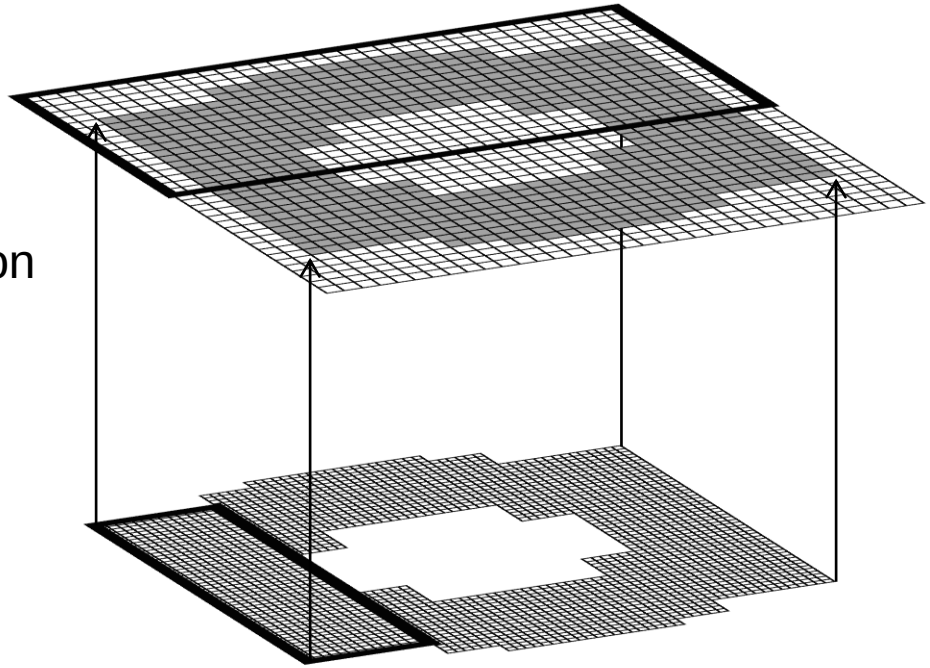
AMR Hierarchy: Brief overview

- Three major challenges
  - Data **coarsening**
  - Data **refinement**
  - **Regridding**
- Coarsening
  - Data transfer occurs on intersection of coarse grid and fine grid



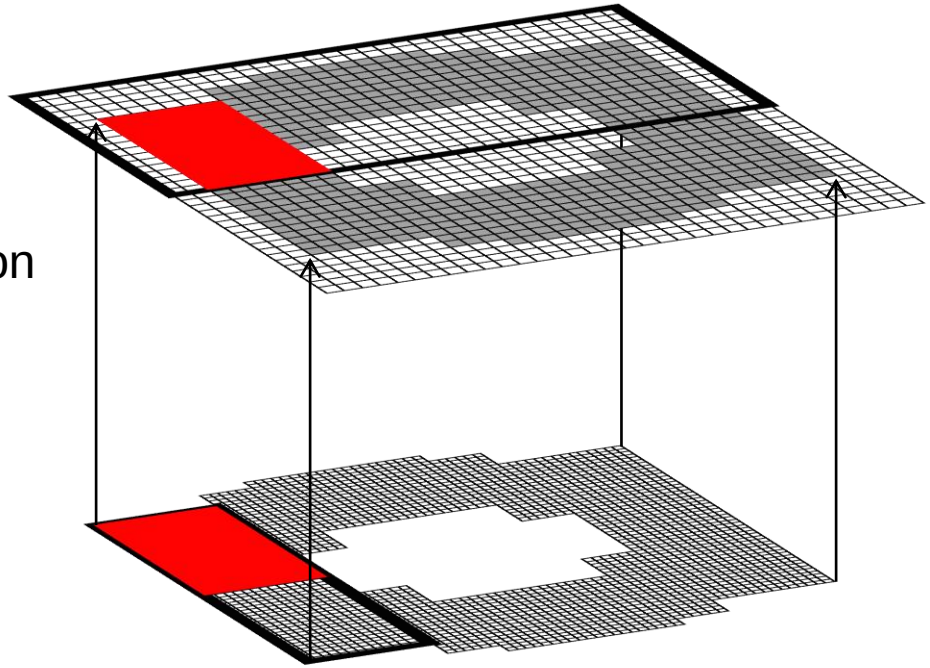
AMR Hierarchy: Brief overview

- Three major challenges
  - Data **coarsening**
  - Data **refinement**
  - **Regridding**
- Coarsening
  - Data transfer occurs on intersection of coarse grid and fine grid



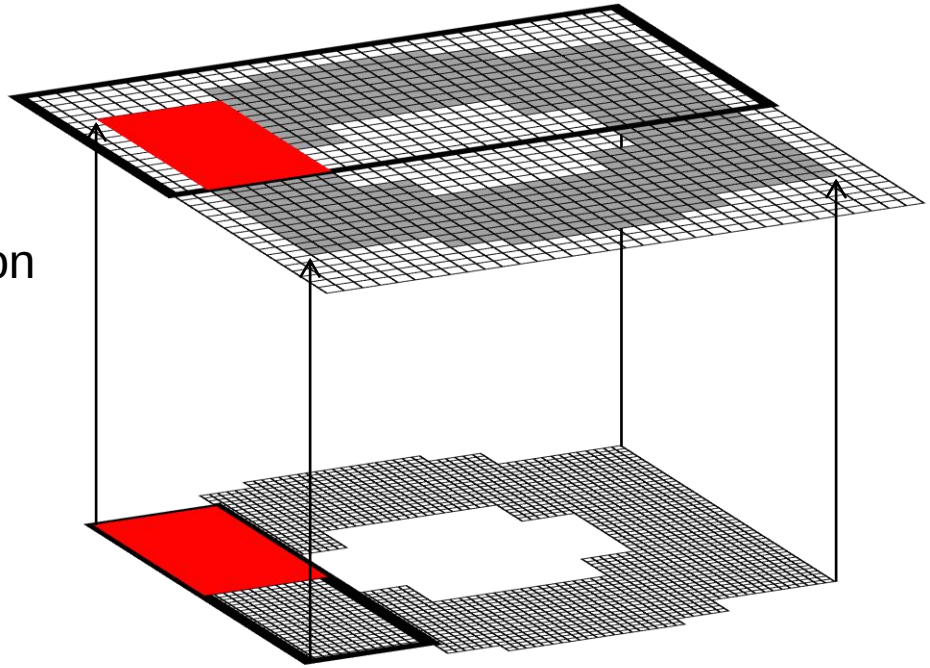
AMR Hierarchy: Brief overview

- Three major challenges
  - Data **coarsening**
  - Data **refinement**
  - **Regridding**
- Coarsening
  - Data transfer occurs on intersection of coarse grid and fine grid



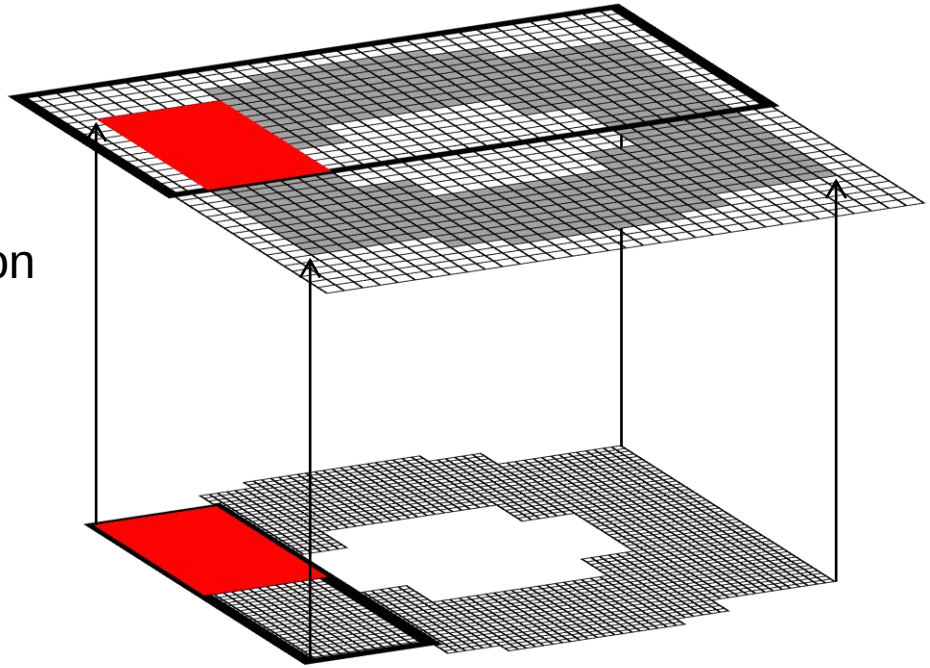
AMR Hierarchy: Brief overview

- 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

For More on AMR in Chapel

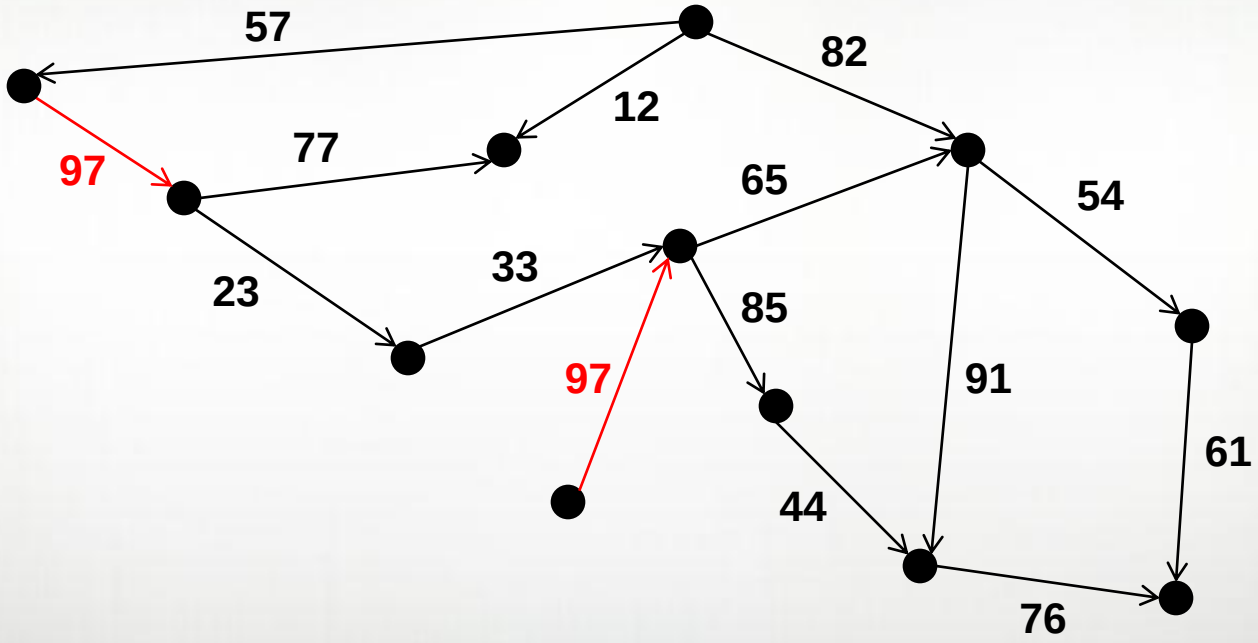
- See https://chapel.svn.sourceforge.net/svnroot/chapel/trunk/test/studies/amr/

Outline

- **STREAM and RA HPC Challenge Benchmarks**
  - simple, regular 1D computations
  - results from SC '09 competition
- **AMR Computations**
  - hierarchical, regular computation
- **SSCA #2**
  - unstructured graph computation

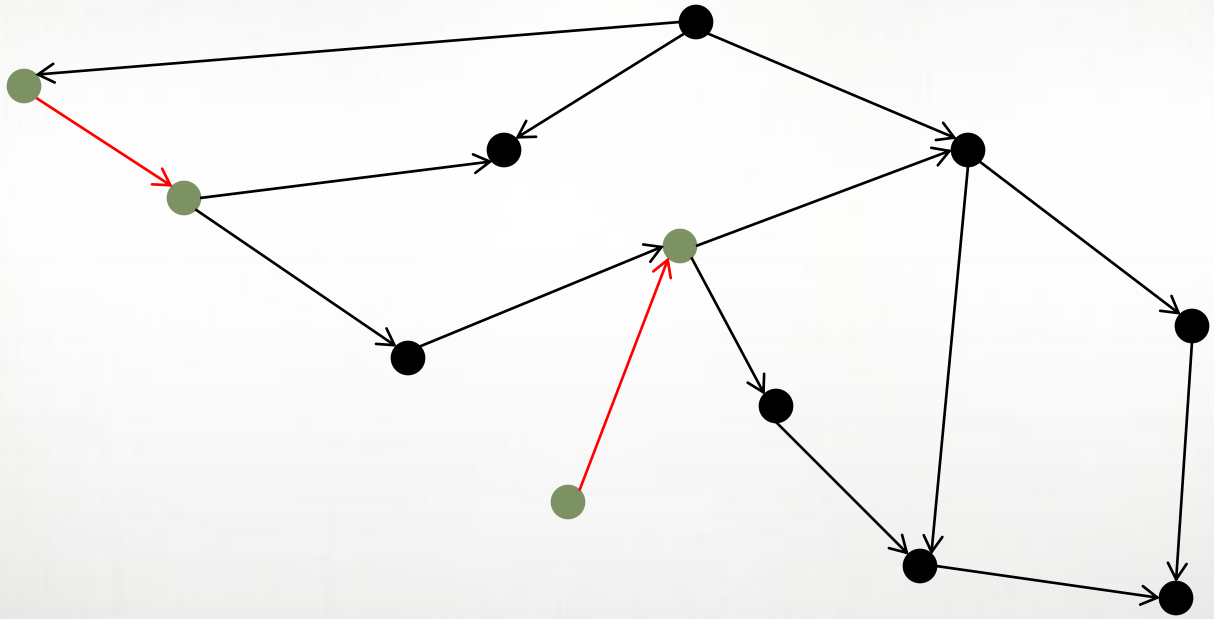
SSCA #2 Kernel 2

Given a set of heavy edges *HeavyEdges* in directed graph *G*, find sub-graphs of outgoing paths with *length* ≤ *maxPathLength*



SSCA #2 Kernel 2

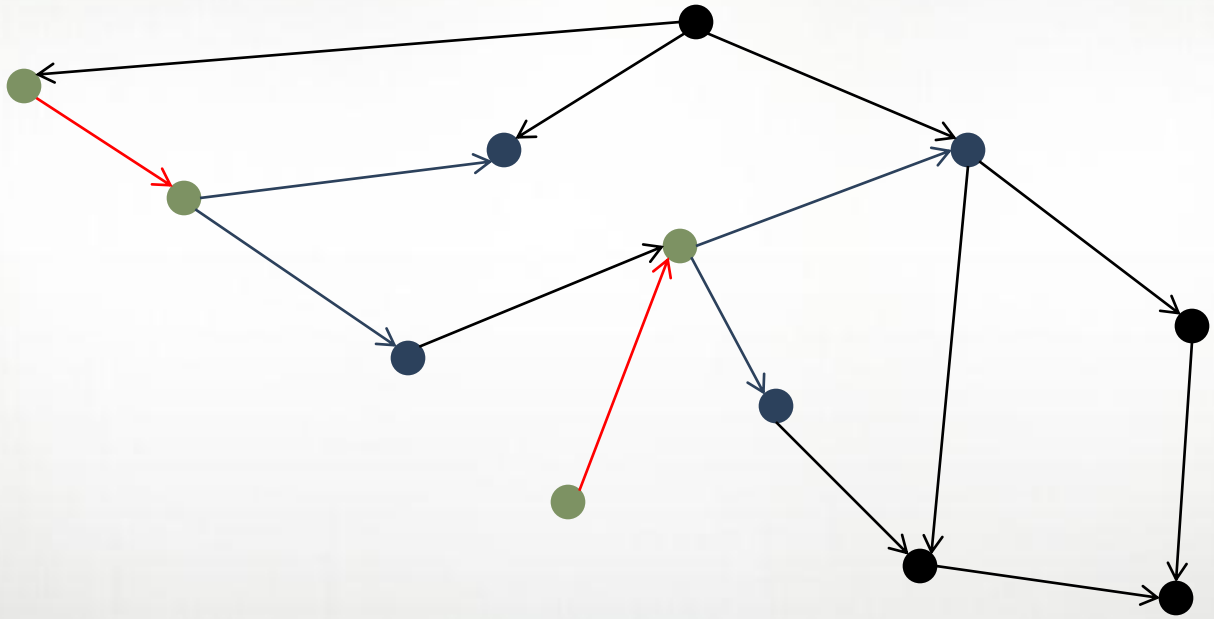
Given a set of heavy edges *HeavyEdges* in directed graph *G*, find sub-graphs of outgoing paths with *length* ≤ *maxPathLength*



maxPathLength = 0

SSCA #2 Kernel 2

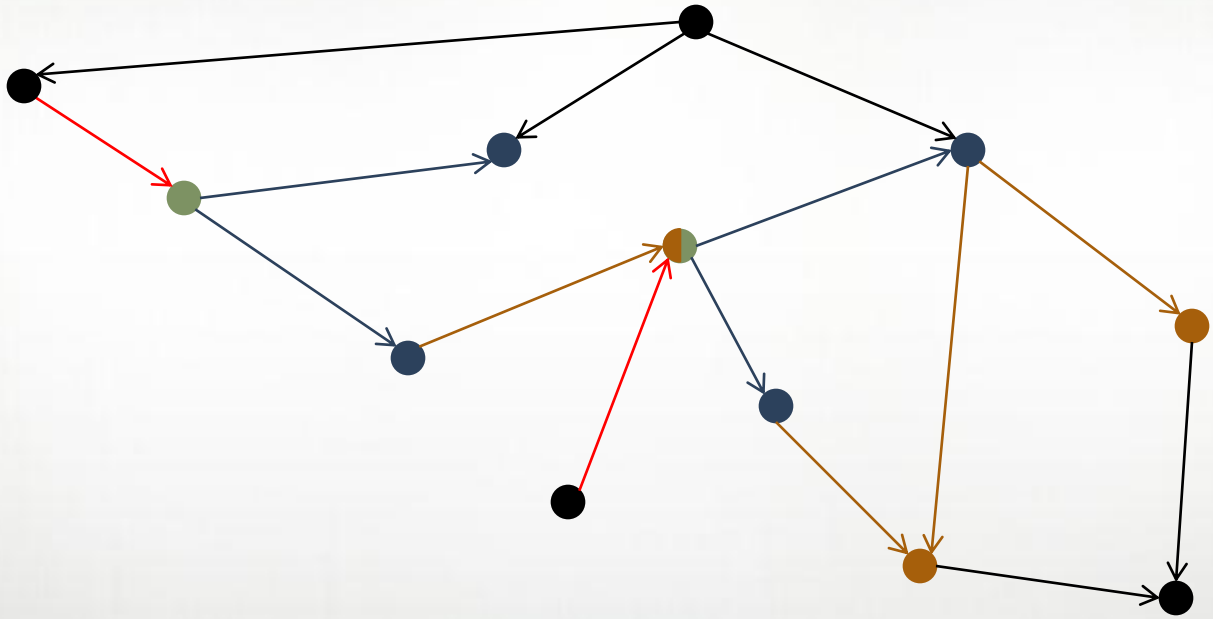
Given a set of heavy edges *HeavyEdges* in directed graph *G*, find sub-graphs of outgoing paths with *length* ≤ *maxPathLength*



maxPathLength = 0 *maxPathLength = 1*

SSCA #2 Kernel 2

Given a set of heavy edges *HeavyEdges* in directed graph *G*, find sub-graphs of outgoing paths with *length* ≤ *maxPathLength*



maxPathLength = 0 *maxPathLength = 1* *maxPathLength = 2*

SSCA #2 Kernel 2 (Code Courtesy of John Lewis)

```
proc rootedHeavySubgraphs(
       G,
       type vertexSet;
       HeavyEdges       : domain,
       HeavyEdgeSubG    : [],
       in maxPathLength: int ) {
  forall (e, subgraph) in
   (HeavyEdges, HeavyEdgeSubG) {
    const (x,y) = e;
    var ActiveLevel: vertexSet;

    ActiveLevel += y;

    subgraph.edges += e;
    subgraph.nodes += x;
    subgraph.nodes += y;
```

```
for pathLength in 1..maxPathLength {
     var NextLevel: vertexSet;
     forall v in ActiveLevel do
       forall w in G.Neighbors(v) do
         atomic {
           if !subgraph.nodes.member(w) {
             NextLevel += w;
             subgraph.nodes += w;
             subgraph.edges += (v, w);
           }
         }

     if (pathLength < maxPathLength) then
       ActiveLevel = NextLevel;
   }
 }
}
```

SSCA #2 Kernel 2 (Code Courtesy of John Lewis)

```
proc rootedHeavySubgraphs(
        G,
        type vertexSet;
        HeavyEdges      : domain,
        HeavyEdgeSubG   : [],
```

```
for pathLength in 1..maxPathLength {
    var NextLevel: vertexSet;
    forall v in ActiveLevel do
      forall w in G.Neighbors(v) do
```

Generic Implementation of Graph G

G.Vertices: A domain whose indices represent the vertices
- For toroidal graphs, a domain(*d*), so vertices are *d*-tuples
- For other graphs, a domain(1), so vertices are integers

G.Neighbors: An array over G.Vertices
- For toroidal graphs, a fixed-size array over the domain [1..2**d*]
- For other graphs…

  …an associative domain with indices of type index(G.vertices)

  …a sparse subdomain of G.Vertices

This kernel and the others are generic w.r.t. these decisions!

SSCA #2 Kernel 2 (Code Courtesy of John Lewis)

```
proc rootedHeavySubgraphs(
      G,
      type vertexSet;
```

```
for pathLength in 1..maxPathLength {
    var NextLevel: vertexSet;
    forall v in ActiveLevel do
      forall w in G.Neighbors(v) do
        atomic {
          if !subgraph.nodes.member(w) {
            NextLevel += w;
            subgraph.nodes += w;
            subgraph.edges += (v, w);
          }
        }

    if (pathLength < maxPathLength) then
      ActiveLevel = NextLevel;
```

Generic with respect to vertex sets

vertexSet: A type argument specifying how to represent vertex subsets

Requirements:

- Parallel iteration
- Ability to add members, test for membership

Options:

- An associative domain over vertices
  `domain(index(G.vertices))`
- A sparse subdomain of the vertices
  `sparse subdomain(G.vertices)`

SSCA #2 Kernel 2 (Code Courtesy of John Lewis)

```
proc rootedHeavySubgraphs(
      G,
      type vertexSet;
      HeavyEdges       : domain,
      HeavyEdgeSubG    : [],
      in maxPathLength: int ) {
  forall (e, subgraph) in
   (HeavyEdges, HeavyEdgeSubG) {
    const (x,y) = e;
    var ActiveLevel: vertexSet;

    ActiveLe

    subgraph.edges += e;
    subgraph.nodes += x;
    subgraph.nodes += y;
```

```
for pathLength in 1..maxPathLength {
    var NextLevel: vertexSet;
    forall v in ActiveLevel do
      forall w in G.Neighbors(v) do
        atomic {
          if !subgraph.nodes.member(w) {
            NextLevel += w;
            subgraph.nodes += w;
            subgraph.edges += (v, w);

    if (pathLength < maxPathLength) then
      ActiveLevel = NextLevel;
  }
 }
}
```

The same genericity applies to subgraphs

SSCA #2 Kernel 2 (Code Courtesy of John Lewis)

```
proc rootedHeavySubgraphs(
       G,
       type vertexSet;
       HeavyEdges       : domain,
       HeavyEdgeSubG    : [],
       in maxPathLength: int ) {
  forall (e, subgraph) in
   (HeavyEdges, HeavyEdgeSubG) {
    const (x,y) = e;
    var ActiveLevel: vertexSet;

    ActiveLevel += y;

    subgraph.edges += e;
    subgraph.nodes += x;
    subgraph.nodes += y;
```

```
for pathLength in 1..maxPathLength {
     var NextLevel: vertexSet;
     forall v in ActiveLevel do
       forall w in G.Neighbors(v) do
         atomic {
           if !subgraph.nodes.member(w) {
             NextLevel += w;
             subgraph.nodes += w;
             subgraph.edges += (v, w);
           }
         }

     if (pathLength < maxPathLength) then
       ActiveLevel = NextLevel;
   }
 }
}
```

SSCA #2 Kernel 2 (Code Courtesy of John Lewis)

```
proc rootedHeavySubgraphs(
       G,
       type vertexSet;
       HeavyEdges       : domain,
       HeavyEdgeSubG    : [],
       in maxPathLength: int ) {
  forall (e, subgraph) in
   (HeavyEdges, HeavyEdgeSubG) {
    const (x,y) = e;
    var ActiveLevel: vertexSet;

    ActiveLevel += y;

    subgraph.edges += e;
    subgraph.nodes += x;
    subgraph.nodes += y;
```

```
for pathLength in 1..maxPathLength {
     var NextLevel: vertexSet;
     forall v in ActiveLevel do
       forall w in G.Neighbors(v) do
         atomic {
           if !subgraph.nodes.member(w) {
             NextLevel += w;
             subgraph.nodes += w;
             subgraph.edges += (v, w);
           }
         }

     if (pathLength < maxPathLength) then
       ActiveLevel = NextLevel;
   }
 }
}
```

For more on SSCA #2

- See $CHPL_HOME/examples/ssca2 in release

Questions?

- **STREAM and RA HPC Challenge Benchmarks**
  - simple, regular 1D computations
  - results from SC '09 competition
- **AMR Computations**
  - hierarchical, regular computation
- **SSCA #2**
  - unstructured graph computation