CRAY
THE SUPERCOMPUTER COMPANY

Chapel: Sample Codes

Outline

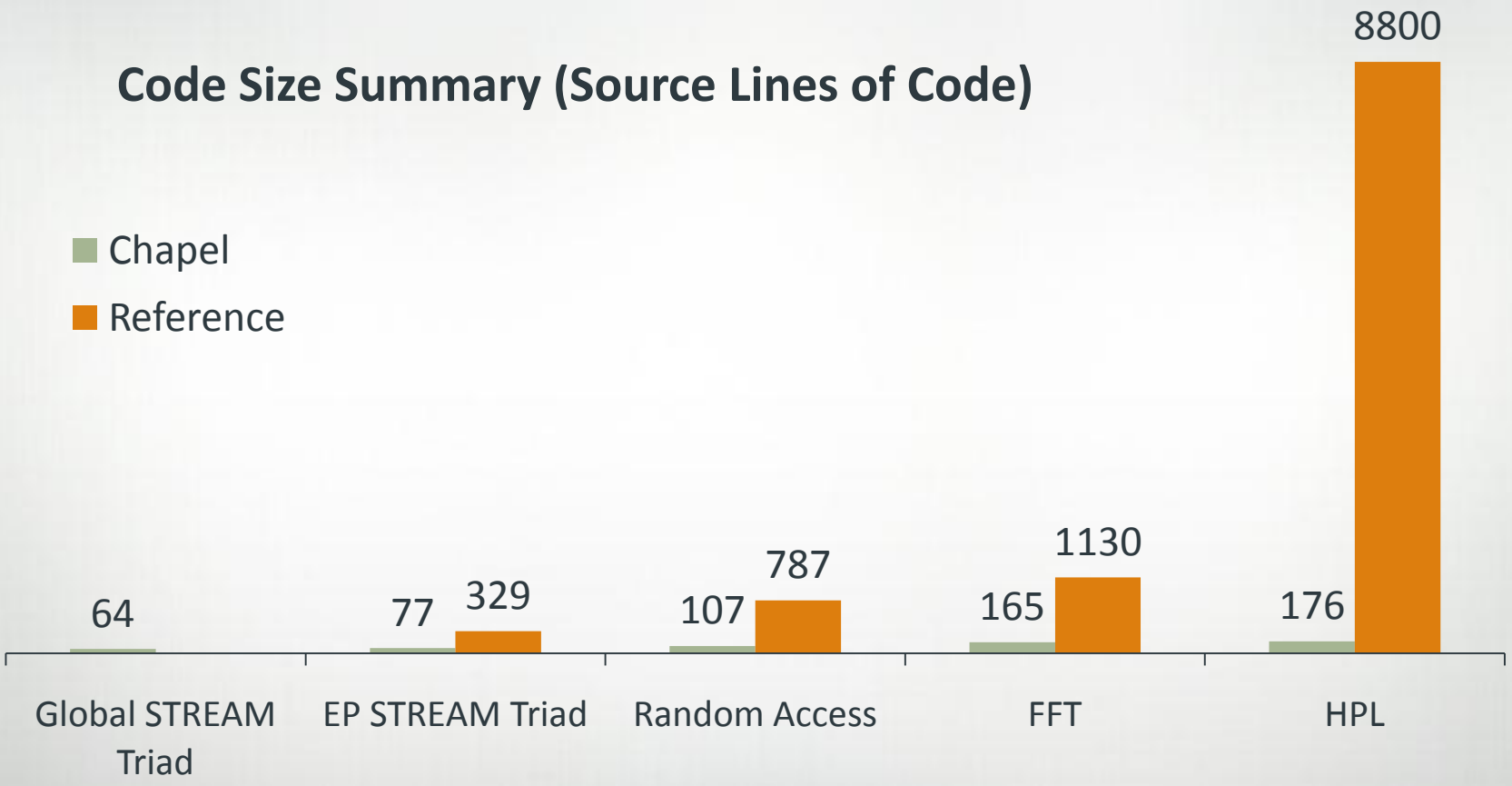
- **STREAM and RA HPC Challenge Benchmarks**
  - simple, regular 1D computations
  - results from SC '09 competition
- **AMR Computations**
  - hierarchical, regular computation
- **SSCA #2**
  - unstructured graph computation

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



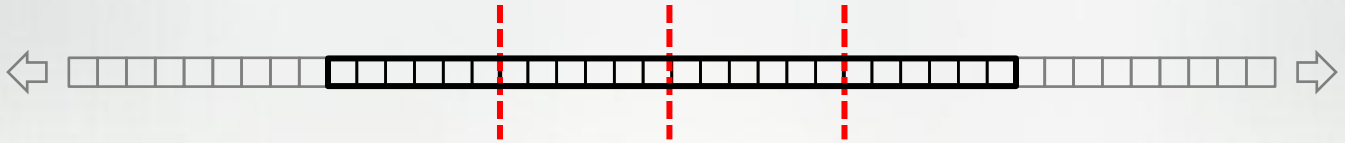
HPC Challenge: Chapel Entries (2008-2009)

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

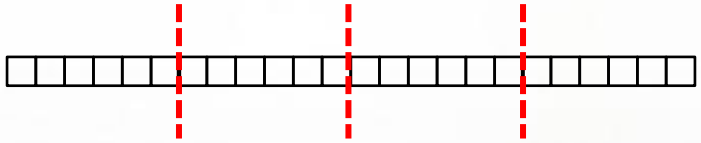
All timings on ORNL Cray XT4:

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

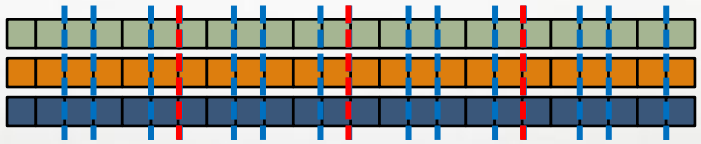
Global STREAM Triad in Chapel (Excerpts)



```
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;
```

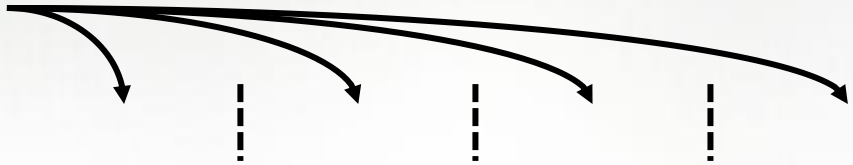
This loop should eventually be written:

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

(and can be today, but performance is worse)

EP STREAM Triad in Chapel (Excerpts)

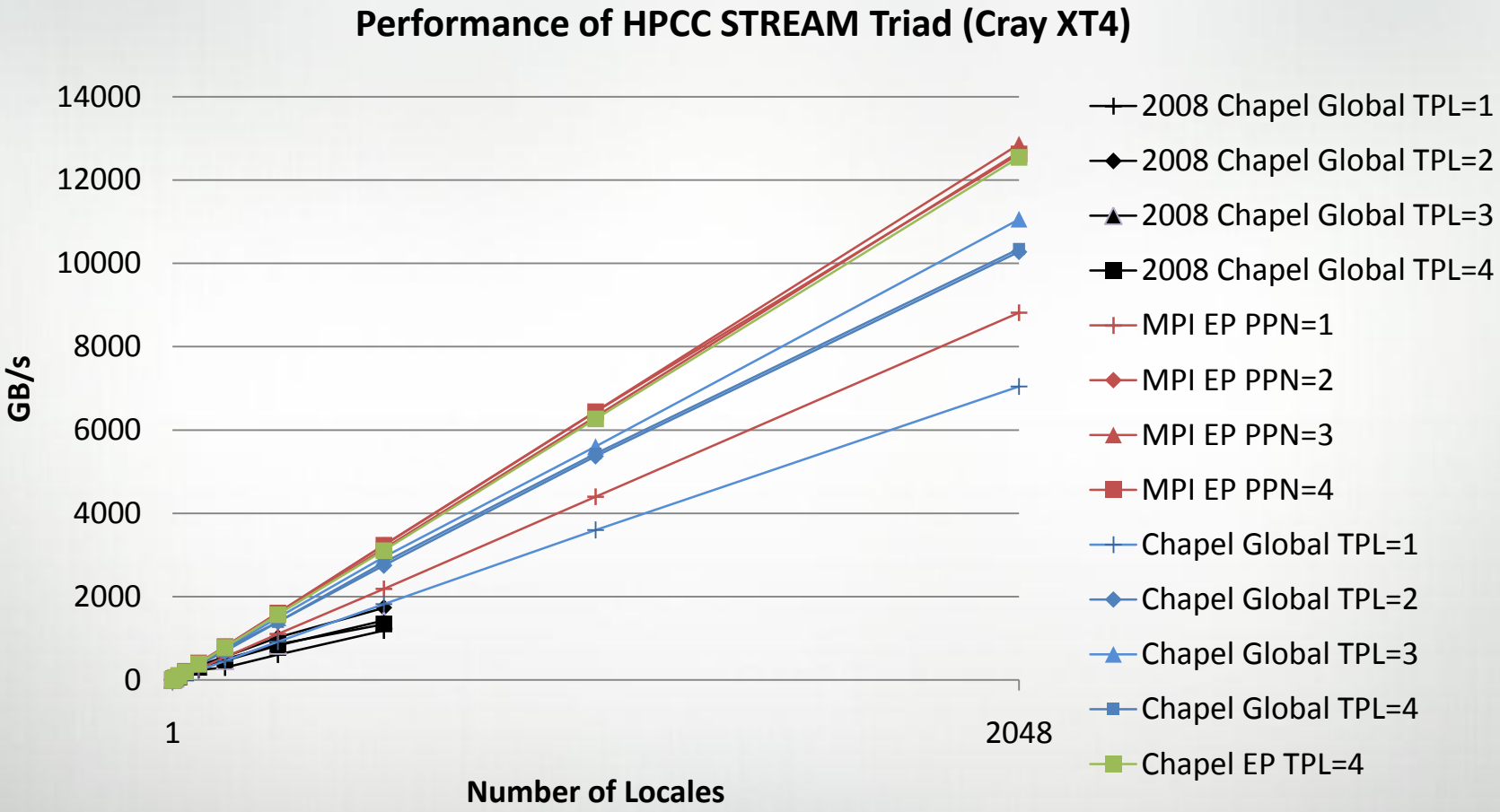
```
coforall loc in Locales do
  on loc {
```



```
  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)

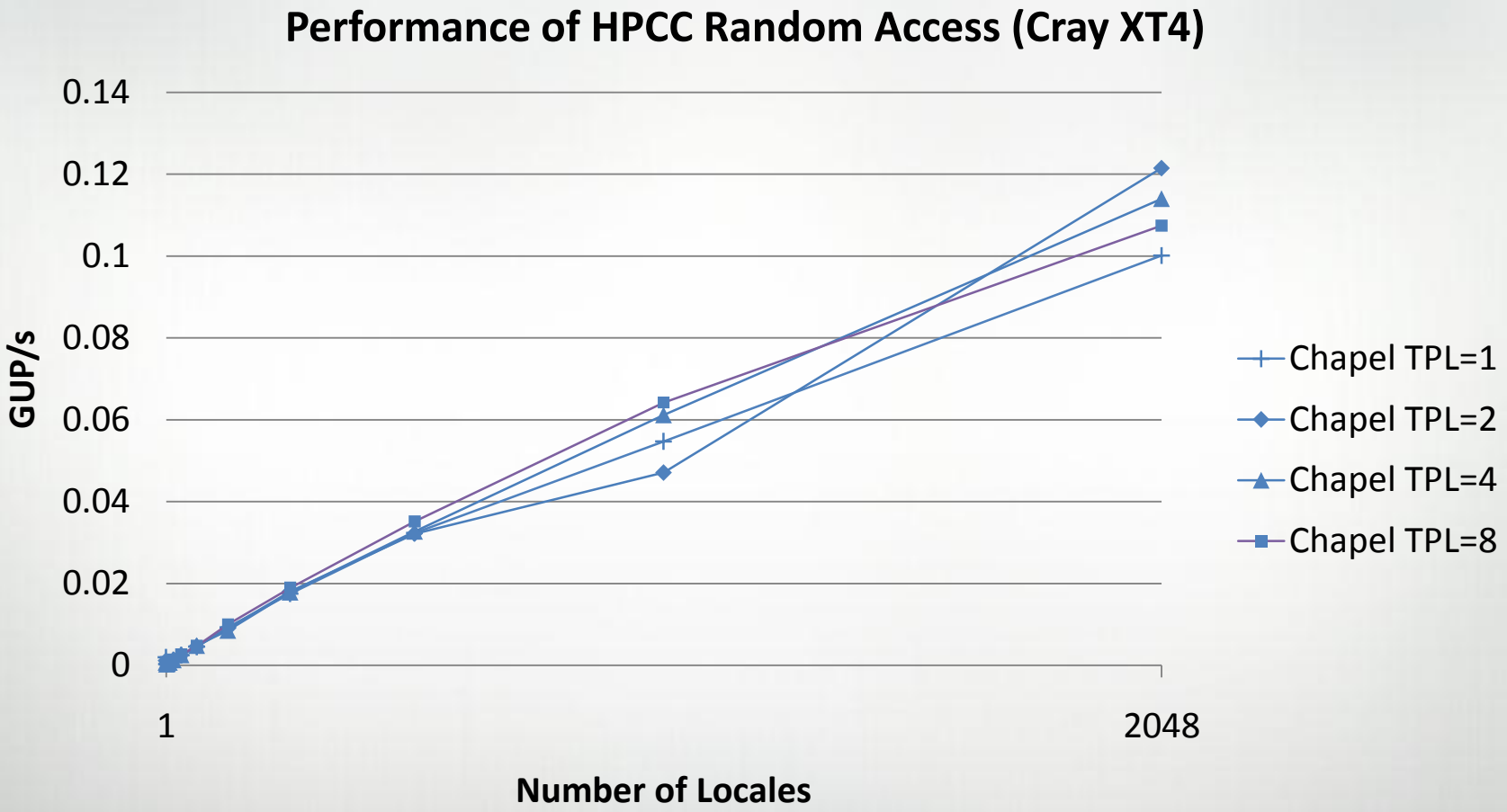
```
const TableDist = new dmap(new Block([0..m-1])),
      UpdateDist = new dmap(new Block([0..N_U-1]));

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:

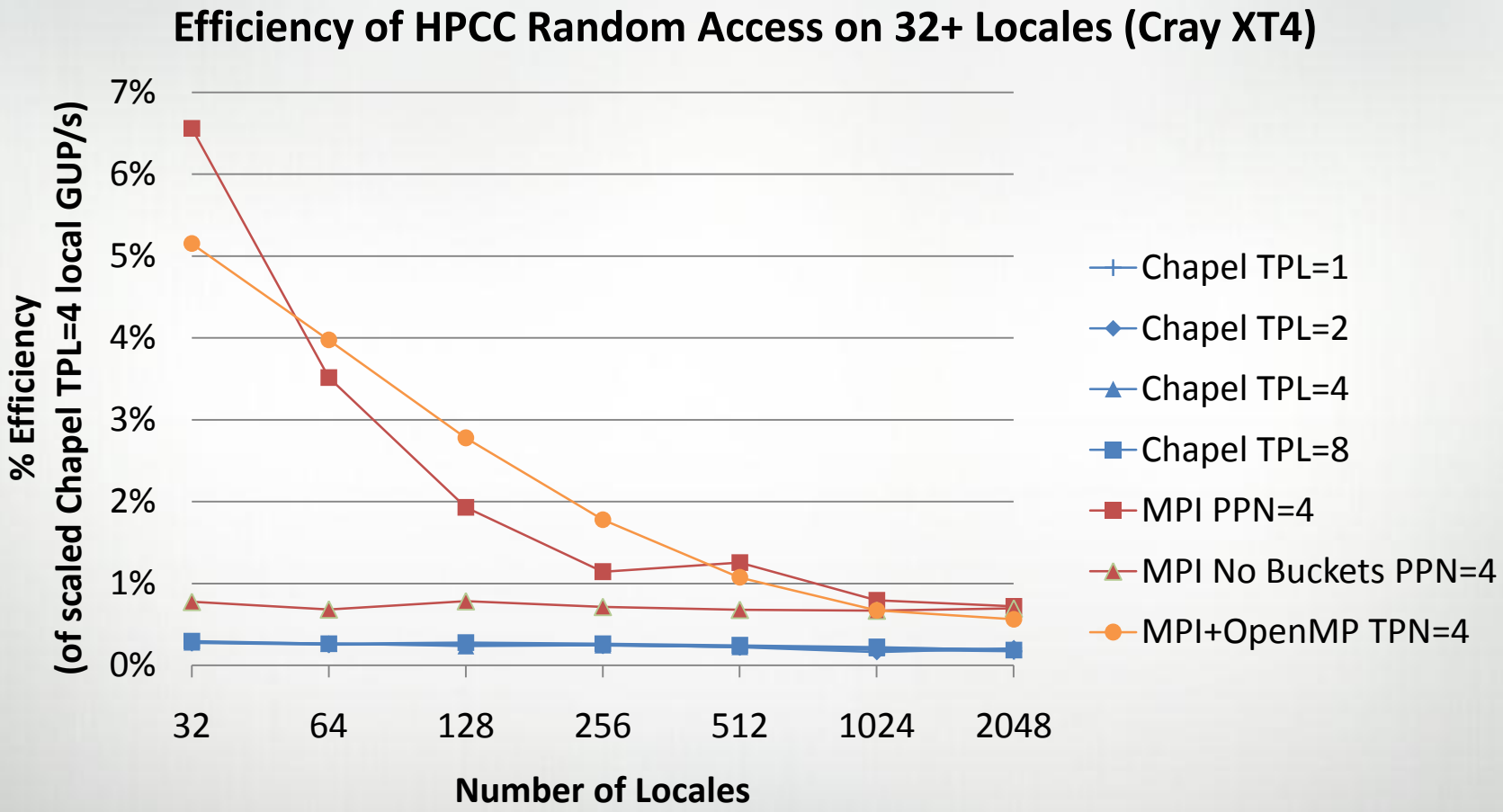
```
on T(r&indexMask) do
  T(r&indexMask) ^= r;
```

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

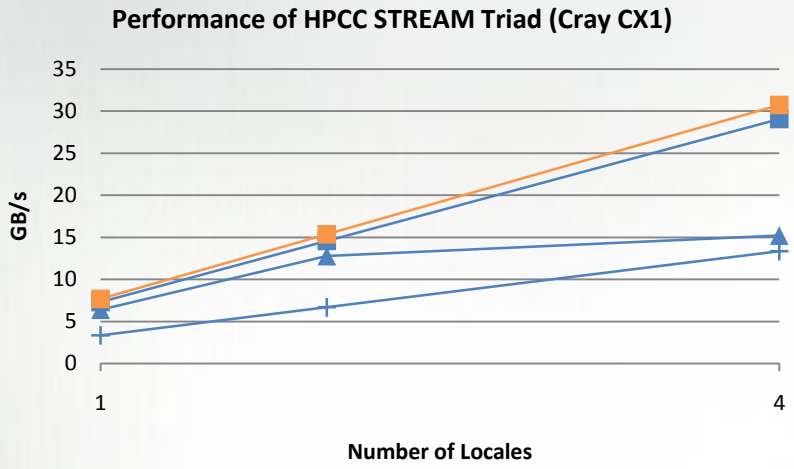
Random Access Performance

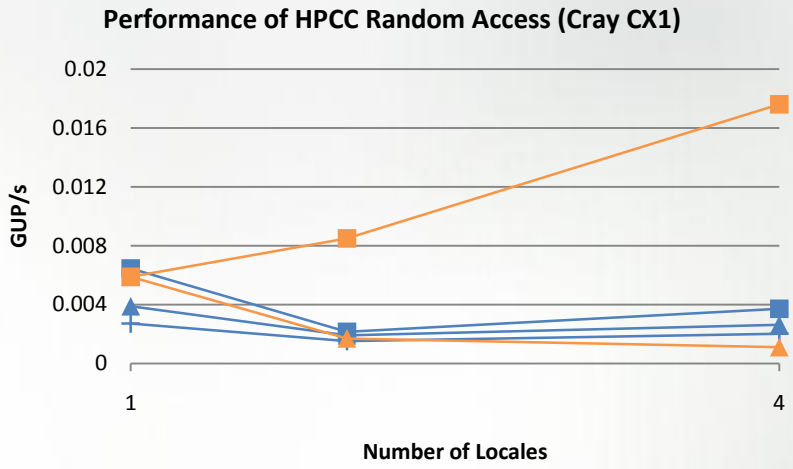


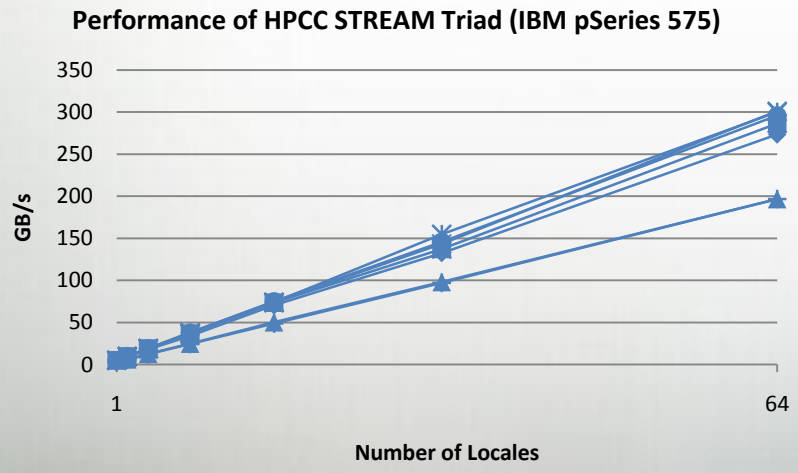
Random Access Efficiency on 32+ Nodes

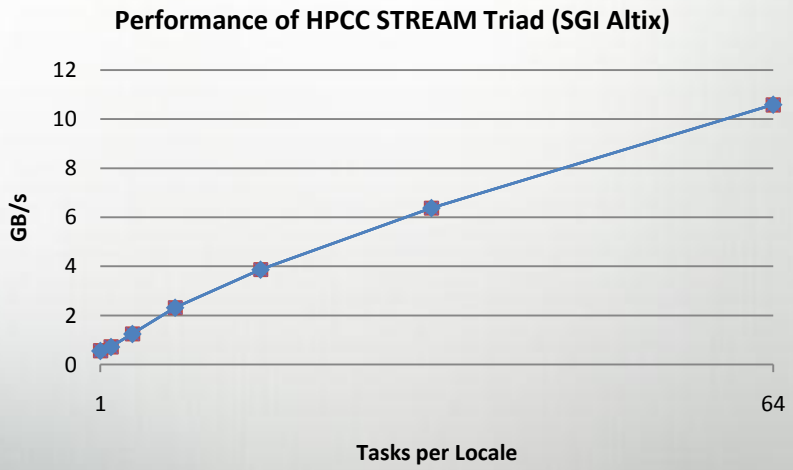


Portability Results









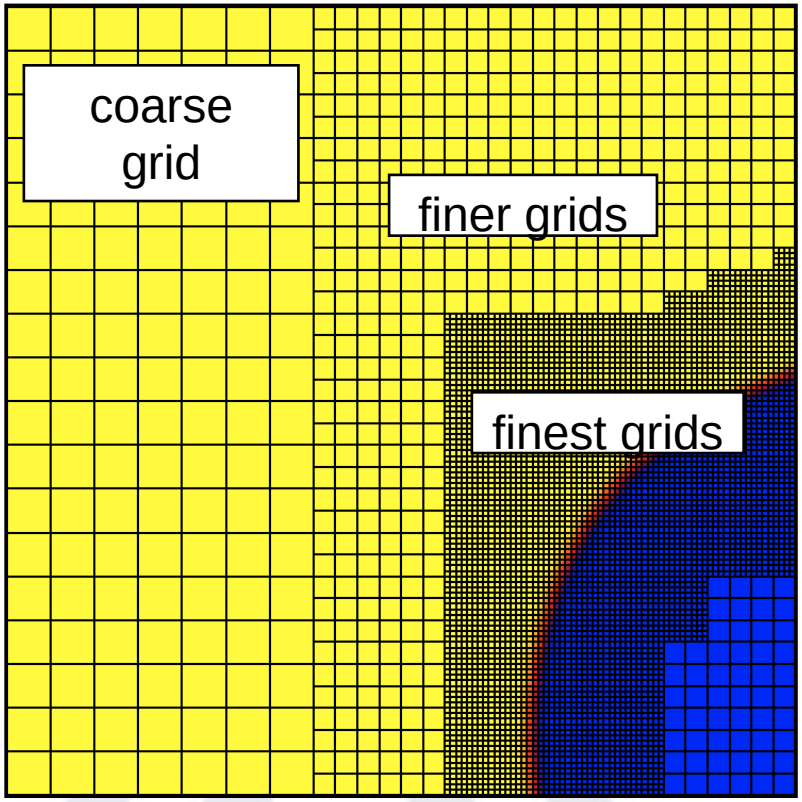
Outline

- **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

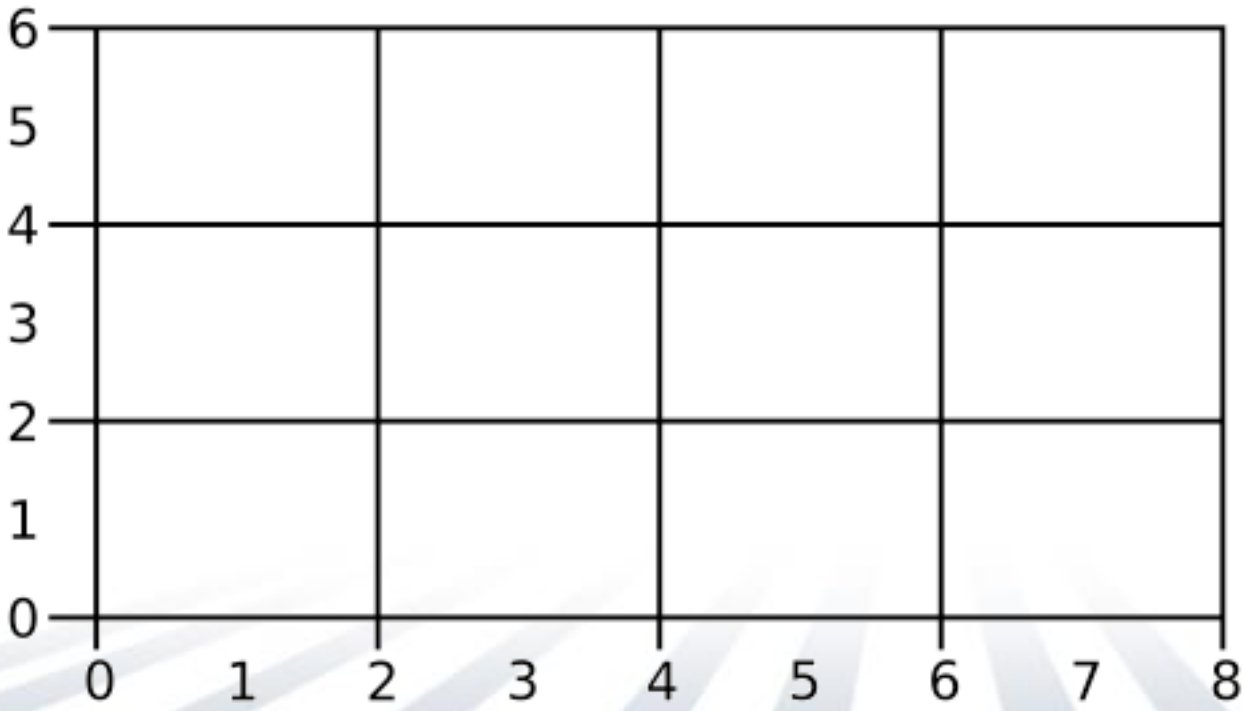
Adaptive Mesh Refinement in Chapel

What's so great about domains?

- Ability to reason about unions of rectangular index spaces (as unions of domains)
- Trivial shared-memory parallelism; easy access to distributed parallelism with distributions
- Fewer nested loops, and no bounds to mess up
- Striding allows much better description of grids (vertices, edges, cell centers)
- **Dimension-independent code**

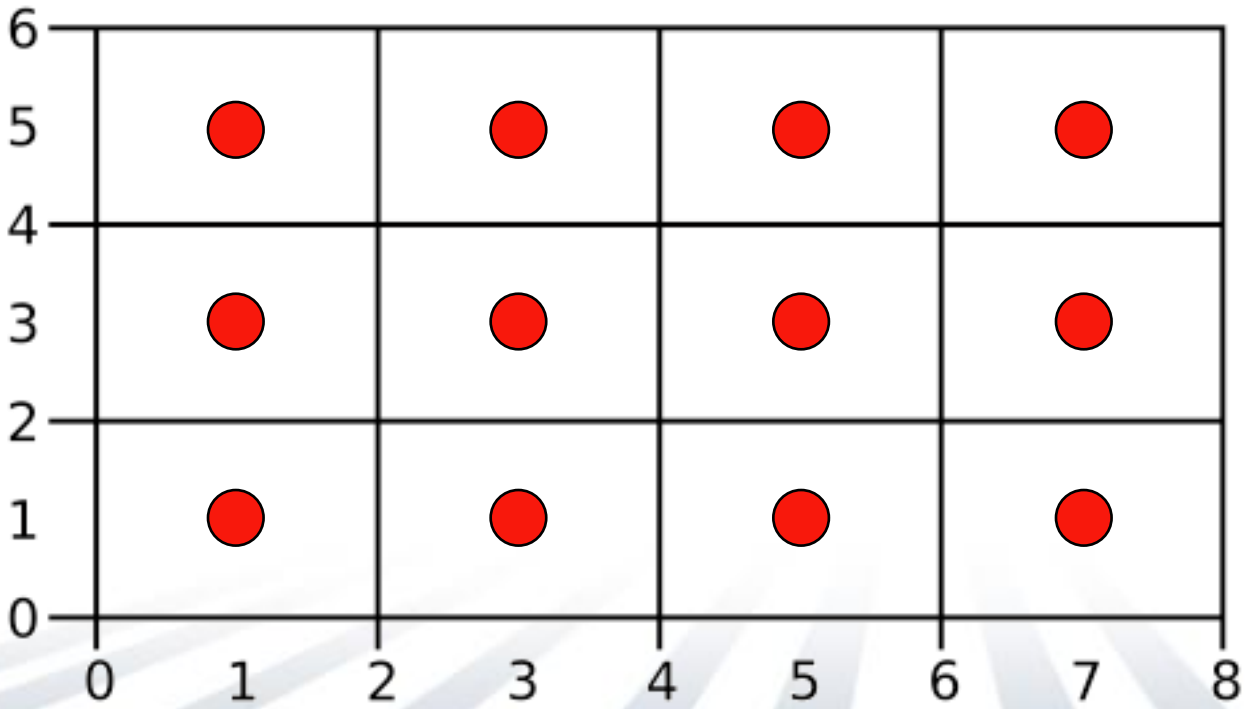


Strided grid description



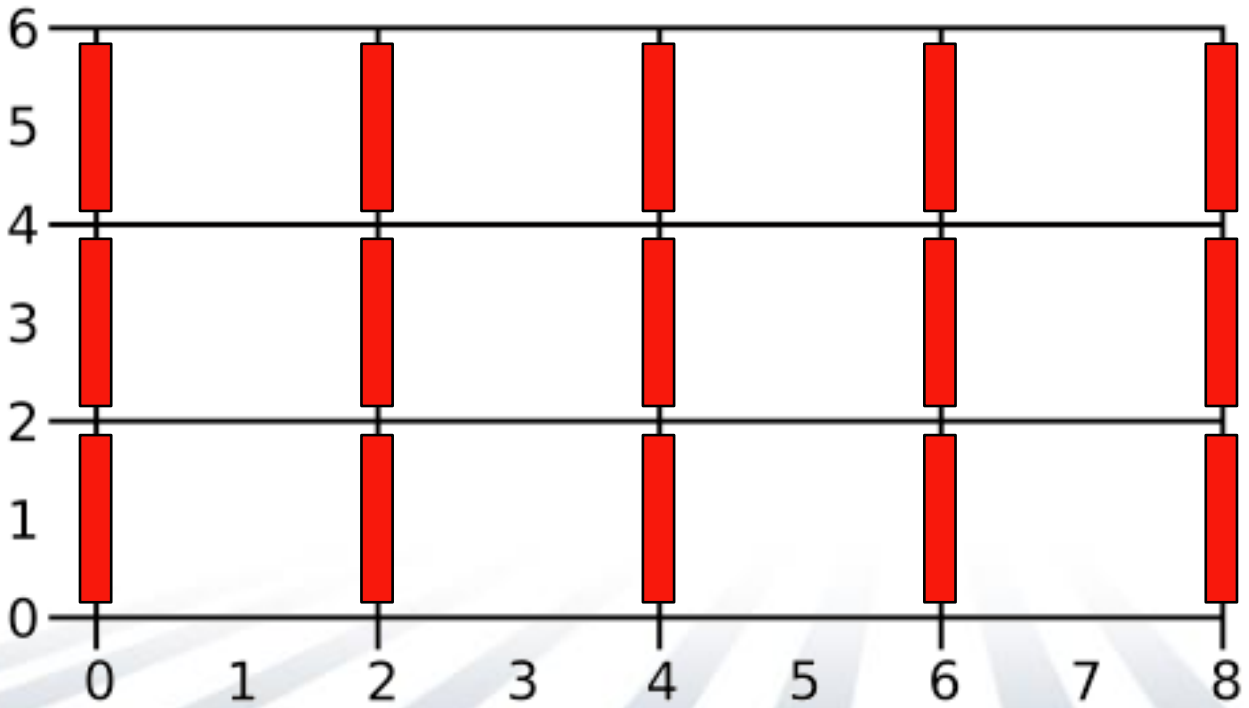
Strided grid description

```
var cell_centers      = [1..7 by 2, 1..5 by 2];
```



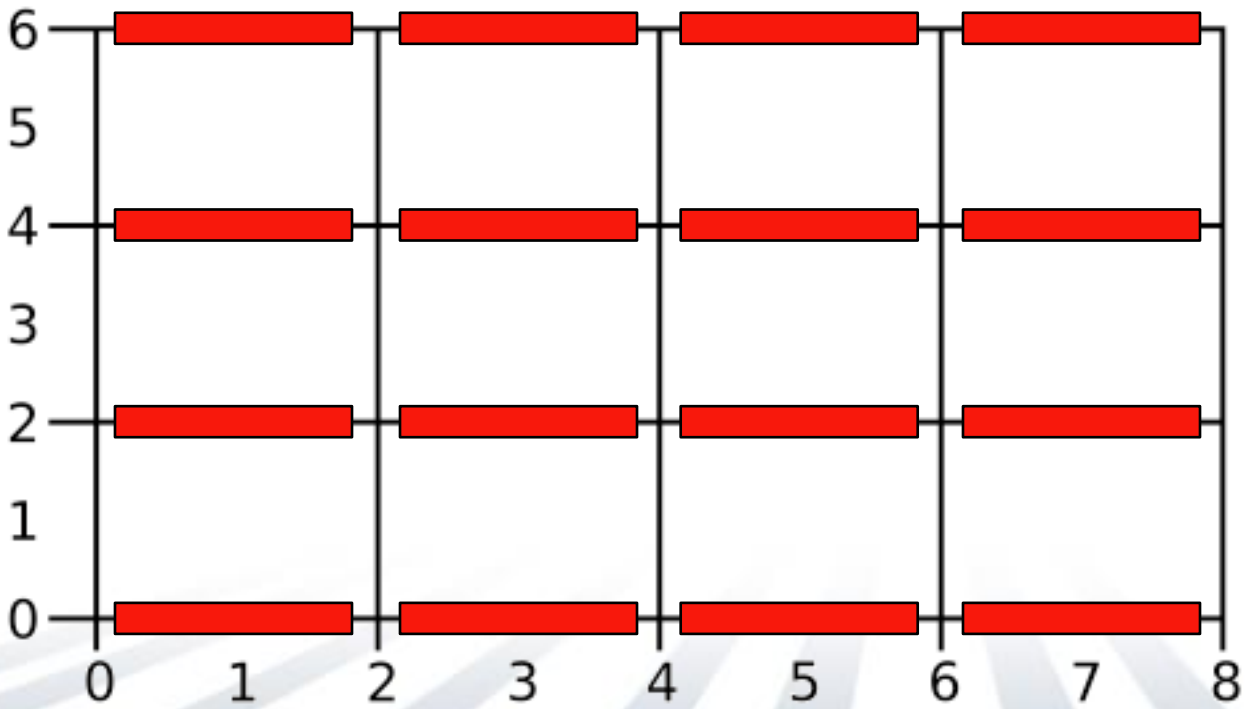
Strided grid description

```
var cell_centers    = [1..7 by 2, 1..5 by 2];
var vertical_edges  = [0..8 by 2, 1..5 by 2];
```



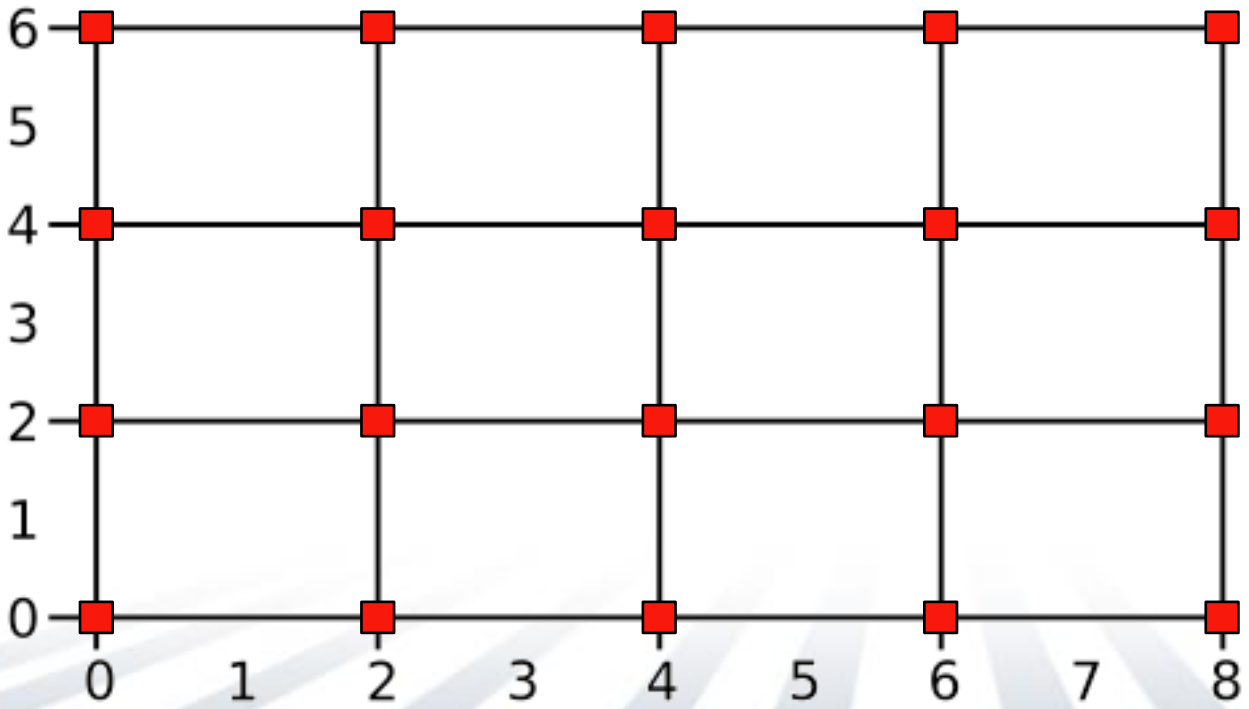
Strided grid description

```
var cell_centers    = [1..7 by 2, 1..5 by 2];
var vertical_edges  = [0..8 by 2, 1..5 by 2];
var horizontal_edges = [1..7 by 2, 0..6 by 2];
```



Strided grid description

```
var cell_centers    = [1..7 by 2, 1..5 by 2];
var vertical_edges  = [0..8 by 2, 1..5 by 2];
var horizontal_edges = [1..7 by 2, 0..6 by 2];
var vertices        = [0..8 by 2, 0..6 by 2];
```



Dimension-free stencils

Tasks:

1. Create an *N*-dimensional grid.
2. Evaluate the function

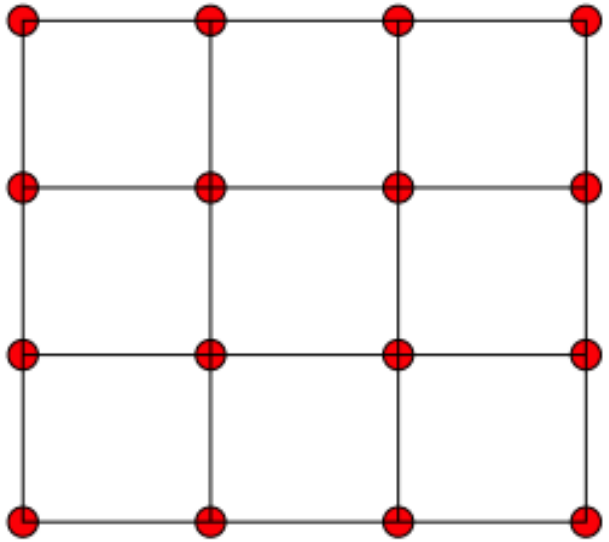
   $$f(x_1, x_2, \ldots, x_N) = \sin(x_1)\sin(x_2)\cdots\sin(x_N)$$

   on the grid.
3. Approximate the Laplacian,

   $$\Delta f = f_{x_1 x_1} + f_{x_2 x_2} + \cdots + f_{x_N x_N}$$

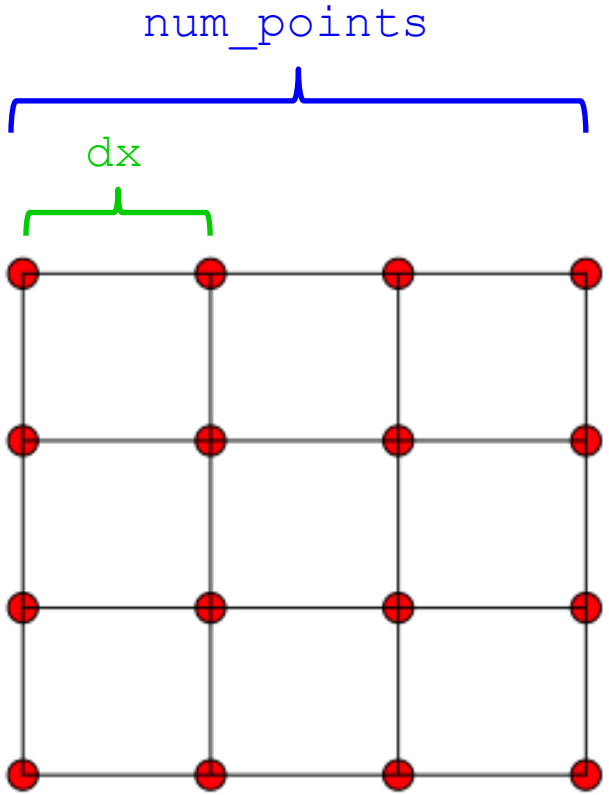
1. Create an *N*-dimensional grid

```
config param N: int = 2;
const dimensions = [1..N];
```



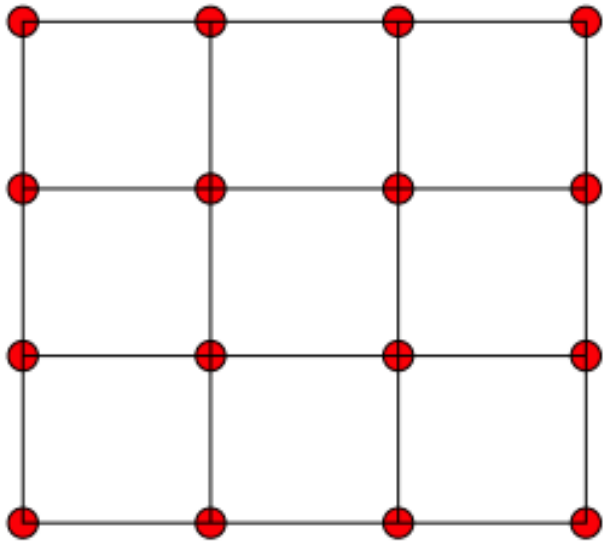
1. Create an *N*-dimensional grid

```
config param N: int = 2;
const dimensions = [1..N];

config const num_points = 20;
const dx = 1.0 / (num_points-1);
```



1. Create an *N*-dimensional grid

```
config param N: int = 2;
const dimensions = [1..N];

config const num_points = 20;
const dx = 1.0 / (num_points-1);
```



```
var grid_points: domain(N);

var ranges: N*range;
for d in dimensions do
  ranges(d) = 1..num_points;

grid_points = ranges;
```

2. Evaluate the function

$$f(x_1, x_2, \ldots, x_N) = \sin(x_1)\sin(x_2)\cdots\sin(x_N)$$

```
var f: [grid_points] real = 1.0;

forall point in points {
  for d in dimensions do
    f(point) *= sin( (point(d)-1)*dx );
}
```

2. Evaluate the function

$$f(x_1, x_2, \ldots, x_N) = \sin(x_1)\sin(x_2)\cdots\sin(x_N)$$

```
var f: [grid_points] real = 1.0;

forall point in points {
  for d in dimensions do
    f(point) *= sin( (point(d)-1)*dx );
}
```

Calculates real coordinate x_d

3. Approximate the Laplacian,

$$\Delta f = f_{x_1 x_1} + f_{x_2 x_2} + \cdots + f_{x_N x_N}$$

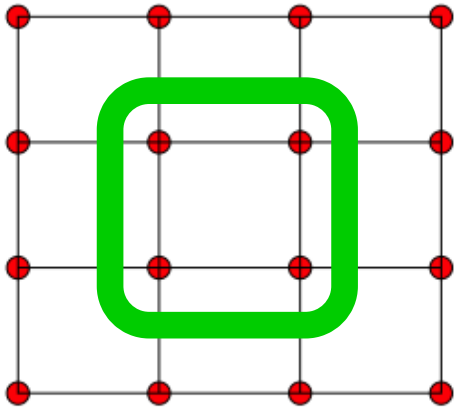
```
var interior_points = grid_points.expand(-1);
var laplacian: [interior_points] real;
```

3. Approximate the Laplacian,

$$\Delta f = f_{x_1 x_1} + f_{x_2 x_2} + \cdots + f_{x_N x_N}$$

```
var interior_points = grid_points.expand(-1);
var laplacian: [interior_points] real;
```

Laplacian is only defined on the **interior** of the grid



3. Approximate the Laplacian,

$$\Delta f = f_{x_1 x_1} + f_{x_2 x_2} + \cdots + f_{x_N x_N}$$

```
var interior_points = grid_points.expand(-1);
var laplacian: [interior_points] real;


forall point in interior_points {
  var shift: N*int;

  for d in dimensions {
    shift(d) = 1;
    laplacian(point) += (    f(point+shift)
                         - 2*f(point)
                         +   f(point-shift)
                        )/ dx**2;
    shift(d) = 0;
  }
}
```

3. Approximate the Laplacian,

$$\Delta f = f_{x_1 x_1} + f_{x_2 x_2} + \cdots + f_{x_N x_N}$$

```
var interior_points = grid_points.expand(-1);
var laplacian: [interior_points] real;


forall point in interior_points {
  var shift: N*int;

  for d in dimensions {
    shift(d) = 1;
    laplacian(point) += (    f(point+shift)
                         - 2*f(point)
                         +   f(point-shift)
                        )/ dx**2;
    shift(d) = 0;
  }
}
```

Approximates $f_{x_d x_d}$

3. Approximate the Laplacian,

$$\Delta f = f_{x_1 x_1} + f_{x_2 x_2} + \cdots + f_{x_N x_N}$$

```
var interior_points = grid_points.expand(-1);
var laplacian: [interior_points] real;

forall point in interior_points {
  var shift: N*int;

  for d in dimensions {
    shift(d) = 1;
    laplacian(point) += (    f(point+shift)
                         - 2*f(point)
                         +   f(point-shift)
                        )/ dx**2;
    shift(d) = 0;
  }
}
```

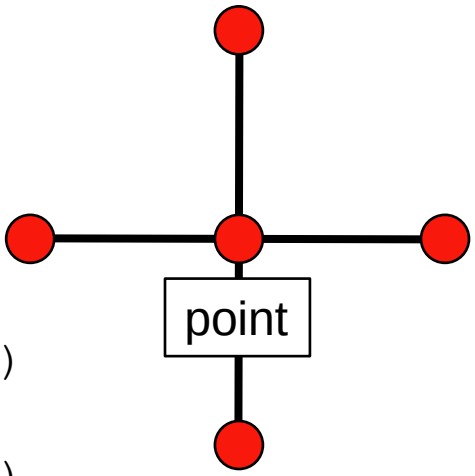
Translates in dimension `d`

3. Approximate the Laplacian,

$$\Delta f = f_{x_1 x_1} + f_{x_2 x_2} + \cdots + f_{x_N x_N}$$

```
var interior_points = grid_points.expand(-1);
var laplacian: [interior_points] real;

forall point in interior_points {
  var shift: N*int;

  for d in dimensions {
    shift(d) = 1;
    laplacian(point) += (    f(point+shift)
                         - 2*f(point)
                         +   f(point-shift)
                        )/ dx**2;
    shift(d) = 0;
  }
}
```

Translates in dimension `d`

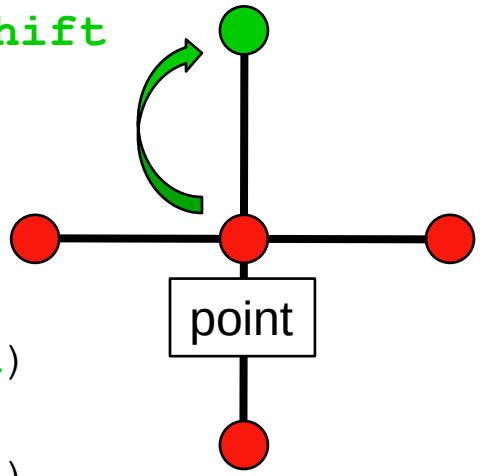


3. Approximate the Laplacian,

$$\Delta f = f_{x_1 x_1} + f_{x_2 x_2} + \cdots + f_{x_N x_N}$$

```
var interior_points = grid_points.expand(-1);
var laplacian: [interior_points] real;

forall point in interior_points {
  var shift: N*int;

  for d in dimensions {
    shift(d) = 1;
    laplacian(point) += (    f(point+shift)
                         - 2*f(point)
                         +   f(point-shift)
                        )/ dx**2;
    shift(d) = 0;
  }
}
```

Translates in dimension `d`

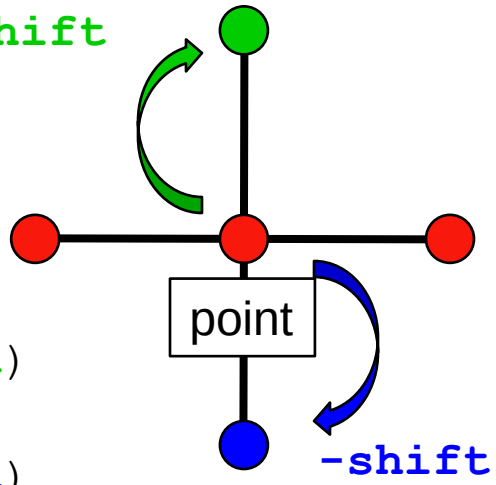


3. Approximate the Laplacian,

$$\Delta f = f_{x_1 x_1} + f_{x_2 x_2} + \cdots + f_{x_N x_N}$$

```
var interior_points = grid_points.expand(-1);
var laplacian: [interior_points] real;

forall point in interior_points {
  var shift: N*int;

  for d in dimensions {
    shift(d) = 1;
    laplacian(point) += (    f(point+shift)
                         - 2*f(point)
                         +   f(point-shift)
                        )/ dx**2;
    shift(d) = 0;
  }
}
```

Translates in dimension `d`

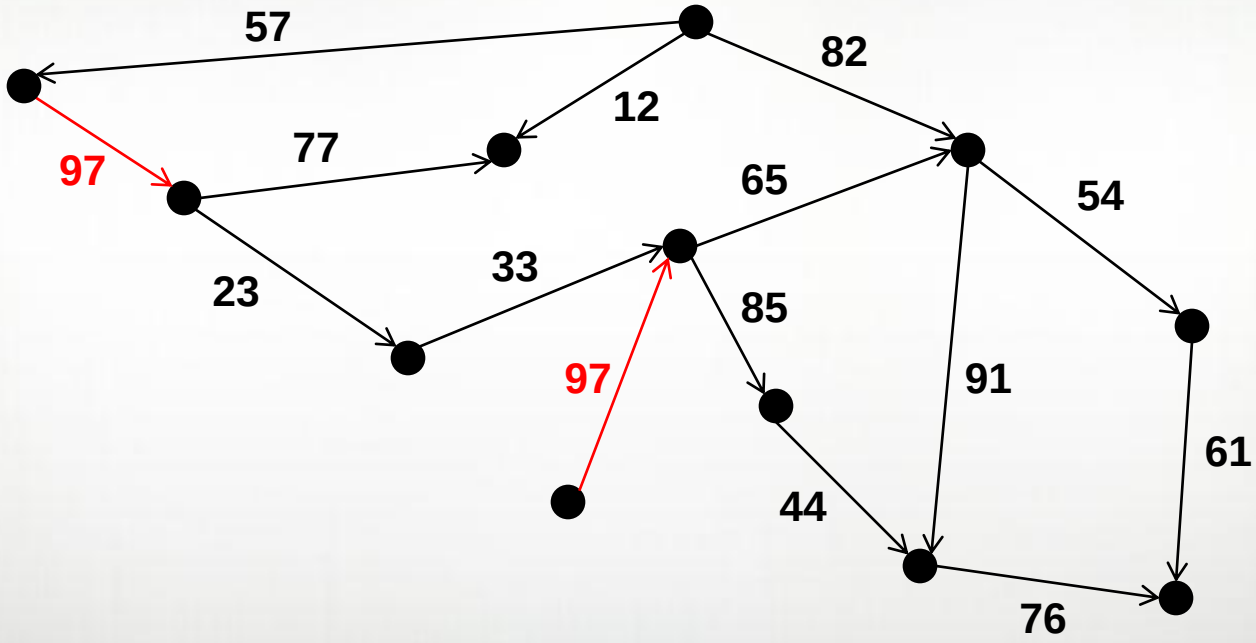


Outline

- **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

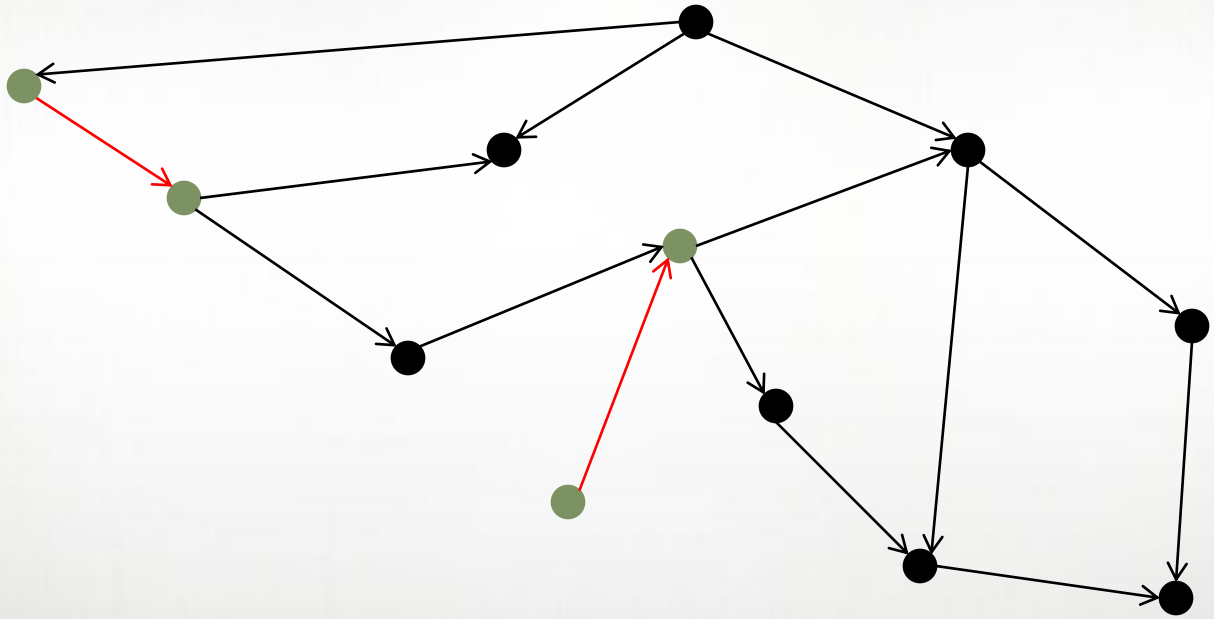
SSCA #2 Kernel 2

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



SSCA #2 Kernel 2

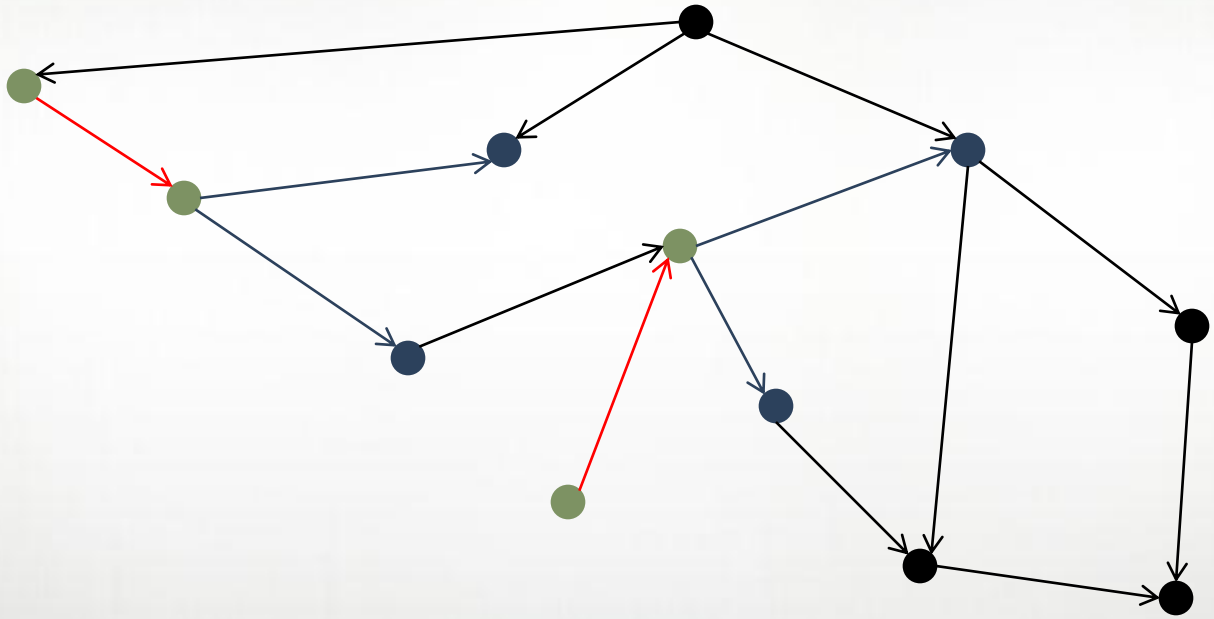
Given a set of heavy edges *HeavyEdges* in directed graph *G*, find sub-graphs of outgoing paths with *length* ≤ *maxPathLength*



maxPathLength = 0

SSCA #2 Kernel 2

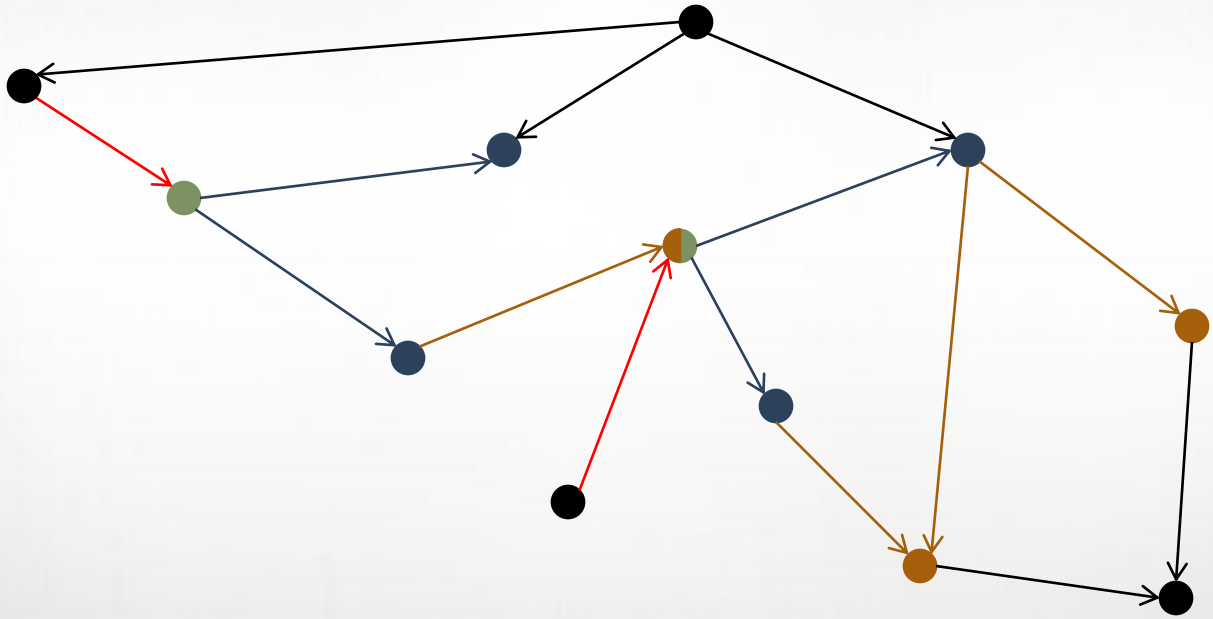
Given a set of heavy edges *HeavyEdges* in directed graph *G*, find sub-graphs of outgoing paths with *length* ≤ *maxPathLength*



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)

```
def 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)

```
def 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)

```
def 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)

```
def 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)

```
def 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;
   }
 }
}
```

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