Chapel: Status, HPCC, and SSCA #2

Sung-Eun Choi and Steve Deitz

Cray Inc.

Outline

- Status and Collaborations
  - About Chapel v1.1
  - Implementation status
  - External collaborations
- HPCC Benchmarks in Chapel as presented at SC '09
- SSCA #2 in Chapel

Chapel Version 1.1

- Features
  - Open source at http://sourceforge.net/projects/chapel/
  - Distributed under the BSD Open Source license
  - Ported to Linux/Unix, Mac, Cygwin
- Contents
  - Compiler, runtime, standard modules, third-party libraries
  - Language spec, quick reference, numerous examples
- Highlights
  - Most data-parallel operations execute in parallel
  - Improved control of data parallelism
  - Completed Block and Cyclic distributions

Implementation Status

- Language Basics
  - No support for inheritance from multiple or generic classes
  - Incomplete support for user-defined constructors
  - Incomplete support for sparse arrays and domains
  - Unchecked support for index types and sub-domains
  - No support for skyline arrays
  - No constant checking for domains, arrays, fields
  - Several internal memory leaks
- Task Parallelism
  - No support for atomic statements
  - Memory consistency model is not guaranteed

Implementation Status

- Locality and Affinity
  - String assignment across locales is by reference
- Data Parallelism
  - Promoted functions/operators do not preserve shape
  - User-defined reductions are undocumented and in flux
  - No partial scans or reductions
  - Some data parallel statements are serialized
- Distributions and Layouts
  - Distributions are limited to Block and Cyclic
  - User-defined domain maps are undocumented and in flux

Collaborations I

- **Notre Dame/ORNL** (Peter Kogge, Srinivas Sridharan, Jeff Vetter)
  Asynchronous software transactional memory over distributed memory
- **UIUC** (David Padua, Albert Sidelnik, Maria Garzaran)
  Chapel for hybrid CPU-GPU computing
- **BSC/UPC** (Alex Duran)
  Chapel over Nanos++ user-level tasking
- **U. Malaga** (Rafa Asenio, Maria Gonzales, Rafael Larossa)
  Parallel file I/O for arrays
- **OSU** (Gagan Agrawal, Bin Ren)
  User-defined reductions over FREERIDE for data intensive computing

Collaborations II

- **U. Colorado** (Jeremy Siek, Jonathan Turner)
  Interfaces and modular generics for Chapel
- **PNNL/CASS-MT** (John Feo, Daniel Chavarria)
  Hybrid computing in Chapel; Cray XMT performance tuning; ARMCI port
- **ORNL** (David Bernholdt *et al.*, Steve Poole *et al.*)
  Code studies – Fock matrices, MADNESS, Sweep3D, coupled models, ...
- **Berkeley** (Dan Bonachea, Paul Hargrove *et al.*)
  Efficient GASNet support for Chapel; collective communication
- **U. Oregon/Paratools Inc.** (Sameer Shende)
  Performance analysis with Tau

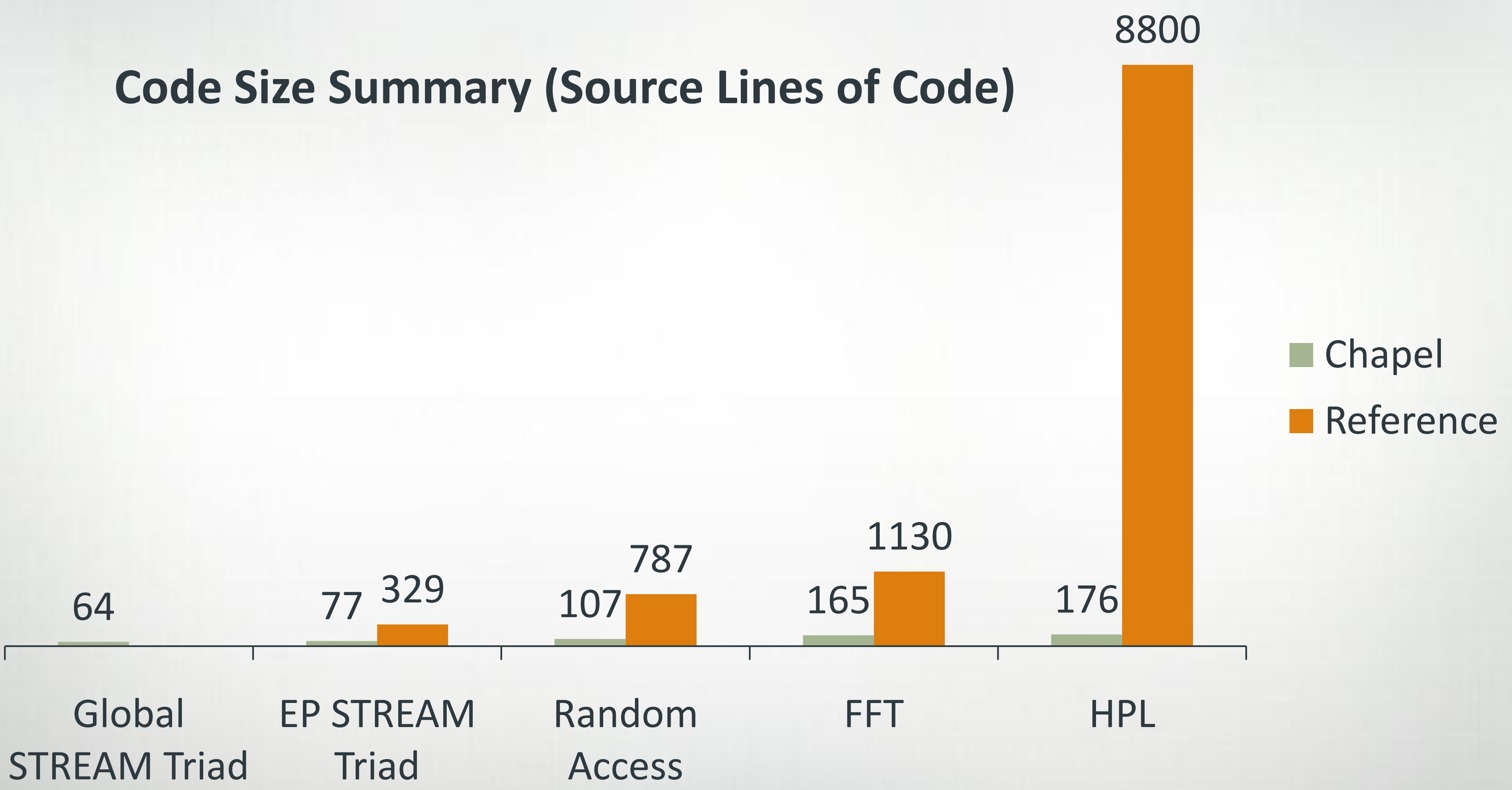
Outline

- Status and Collaborations
- HPCC Benchmarks in Chapel as presented at SC ’09
  - Code updated for Chapel v1.1
- SSCA #2 in Chapel

Highlights

- Global STREAM Triad 10.8 TB/s (6.4x over 2008)
  - Executed on 2048 nodes (up from 512 nodes in 2008)
  - Better scaling by eliminating extra communication
- EP STREAM Triad 12.2 TB/s
  - More similar to EP STREAM reference version
- Random Access 0.122 GUP/s (111x over 2008)
  - Executed on 2048 nodes (up from 64 nodes in 2008)
  - Optimized remote forks + better scaling as with STREAM
- A distributed-memory implementation of FFT
- A demonstration of portability
  - Cray XT4, Cray CX1, IBM pSeries 575, SGI Altix

Chapel Implementation Characteristics

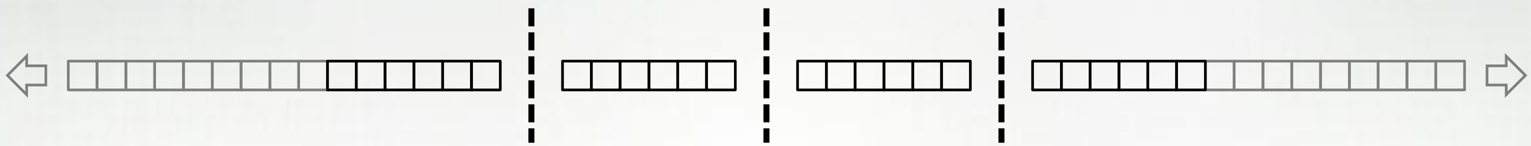


FFT and HPL in a Nutshell

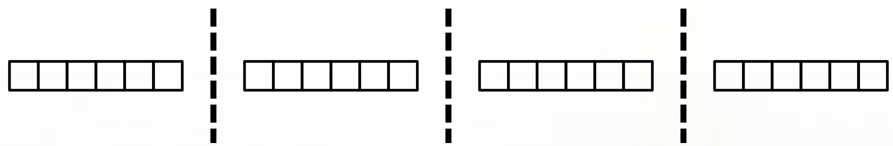
- FFT
  - Uses both Block and Cyclic distributions
  - Butterfly-patterned accesses are completely local
    - Communication with nearby neighbors is local with Block
    - Communication with far off neighbors is local with Cyclic
  - Executes on distributed memory, but is slow
- HPL
  - Implementation is ready for BlockCyclic distribution
  - Executes on single locale only, but is multi-threaded

Global STREAM Triad in Chapel (Excerpts)

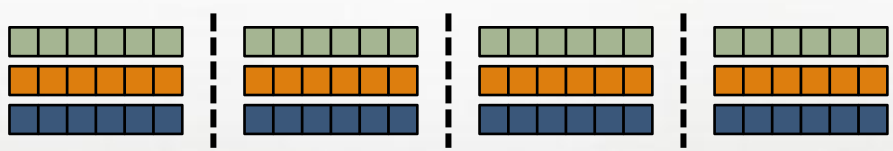
```
const BlockDist = new dmap(new Block([1..m]));
```



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



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

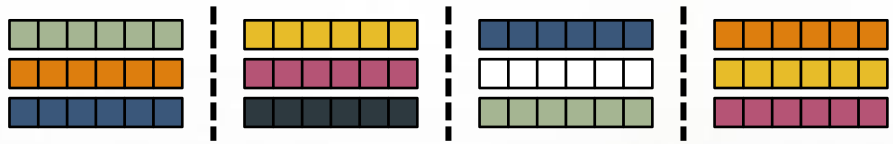


```
forall (a,b,c) in (A,B,C) do
  a = b + alpha * c;
```

EP STREAM Triad in Chapel (Excerpts)

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

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



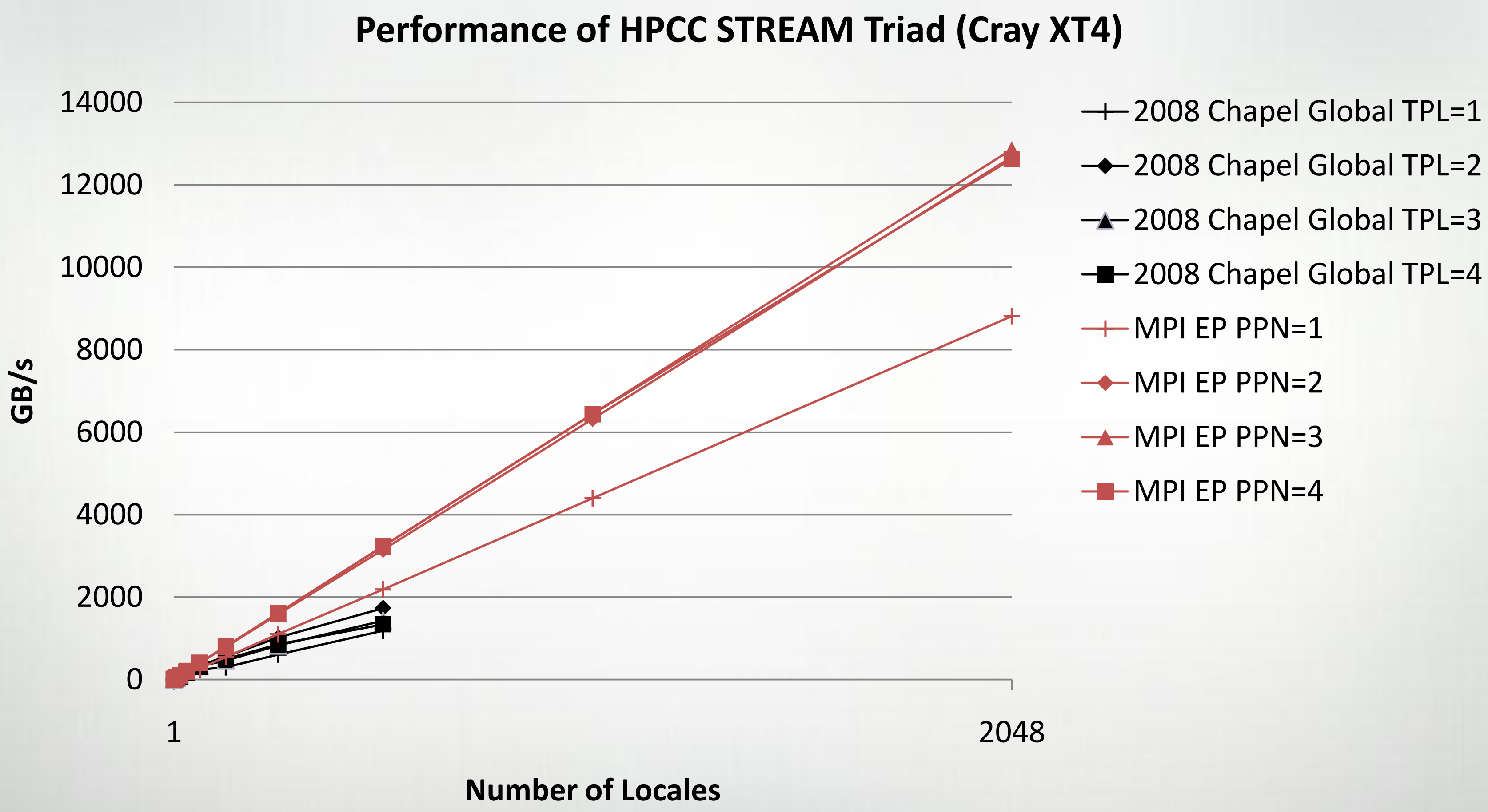
```
    forall (a,b,c) in (A,B,C) do
      a = b + alpha * c;
  }
}
```

Experimental Setup

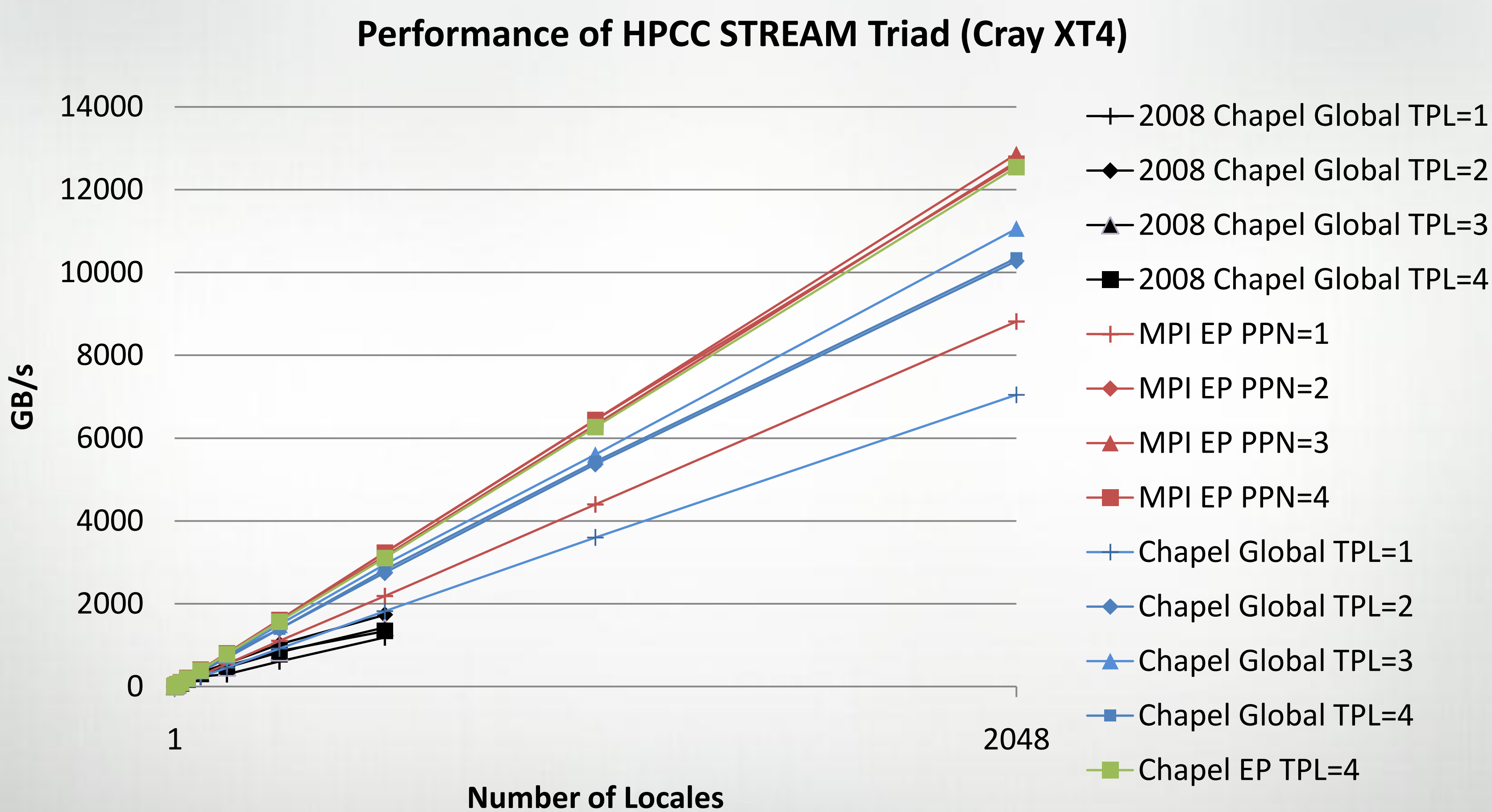
Machine Characteristics	
Model	Cray XT4
Location	ORNL
Nodes	7832
Processor	2.1 GHz Quadcore AMD Opteron
Memory	8 GB per node

Benchmark Parameters	
STREAM Triad Memory	Least value greater than 25% of memory
Random Access Memory	Least power of two greater than 25% of memory
Random Access Updates	2^{n-10} for memory equal to 2^n

STREAM Triad Performance



STREAM Triad Performance



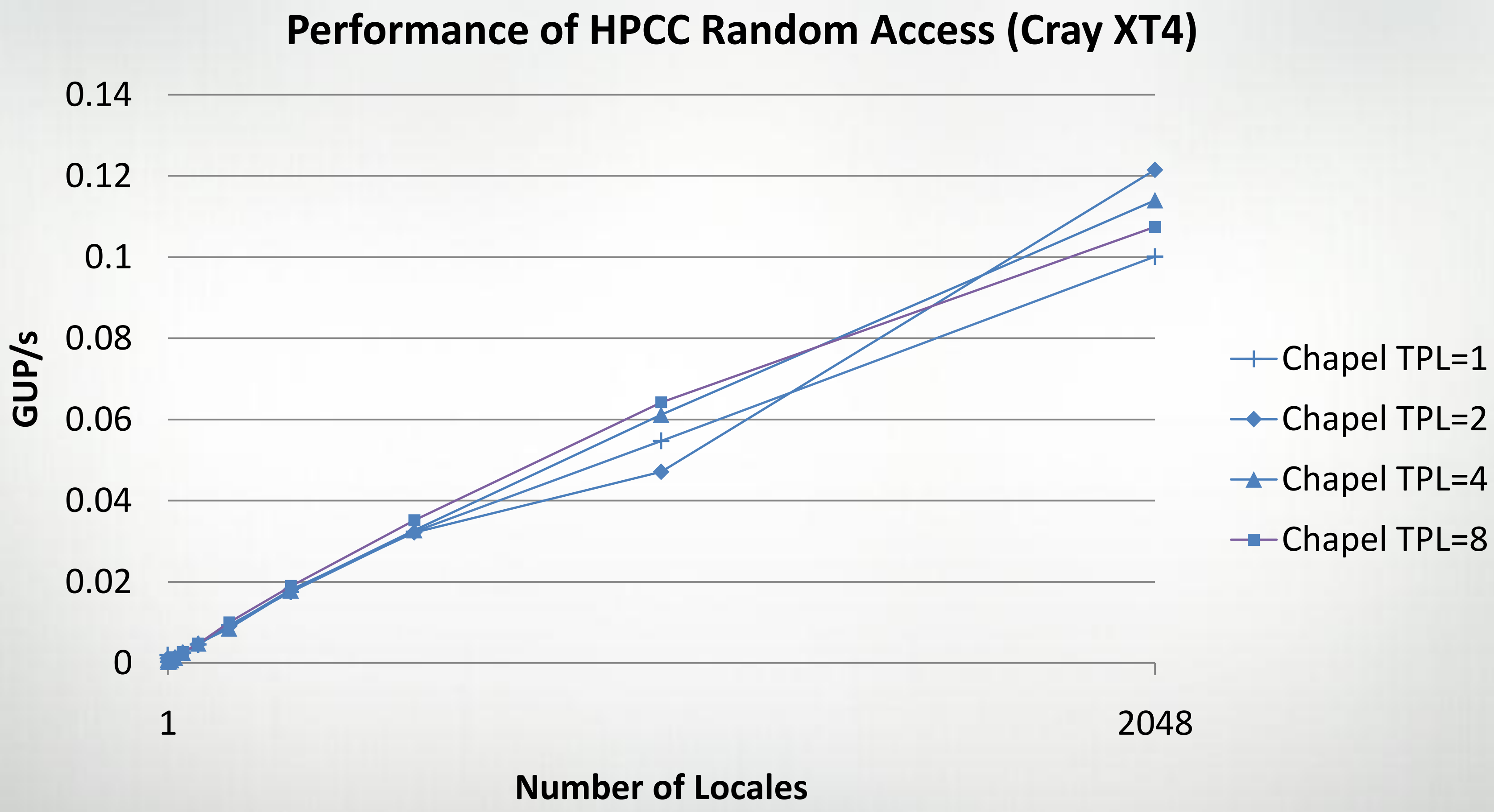
Global Random Access in Chapel (Excerpts)

```
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;
  }
```

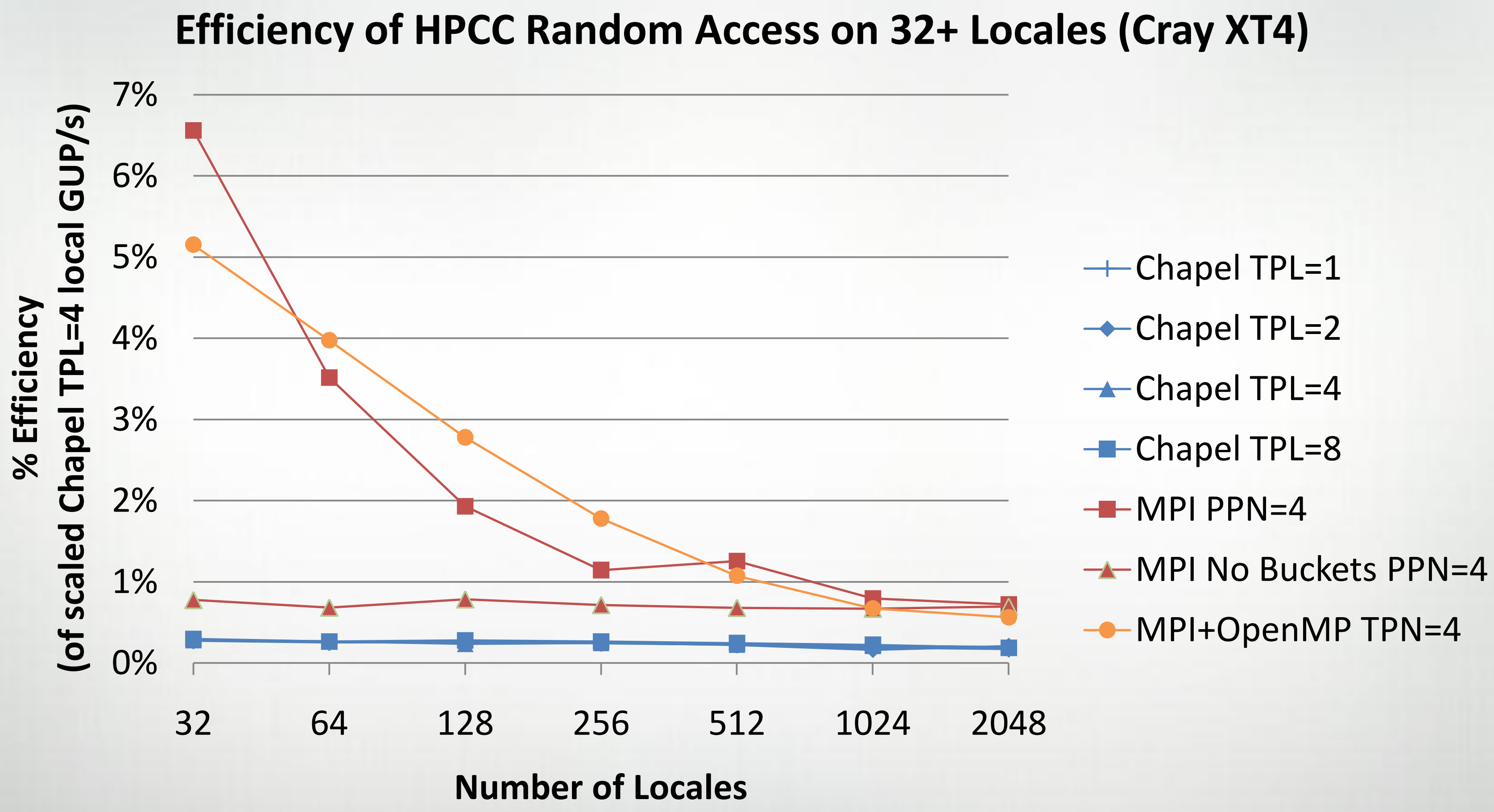
More elegant on-block

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

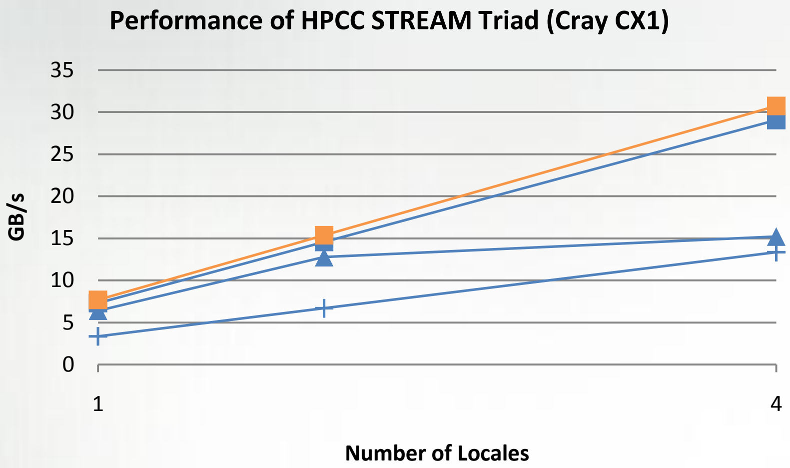
Random Access Performance

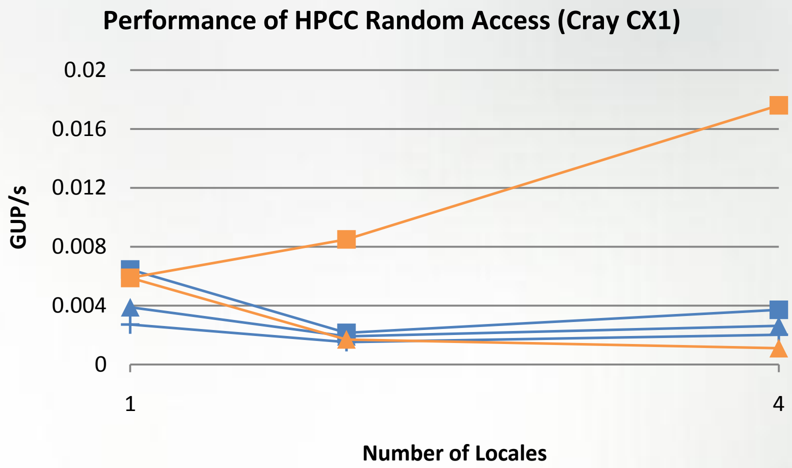


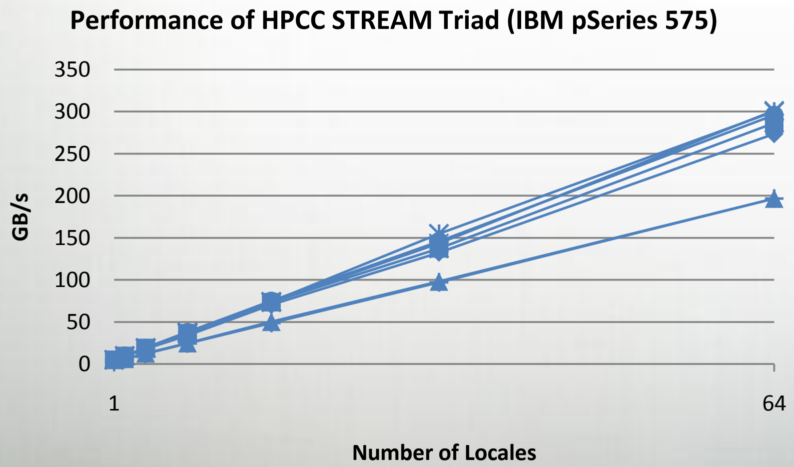
Random Access Efficiency on 32+ Nodes

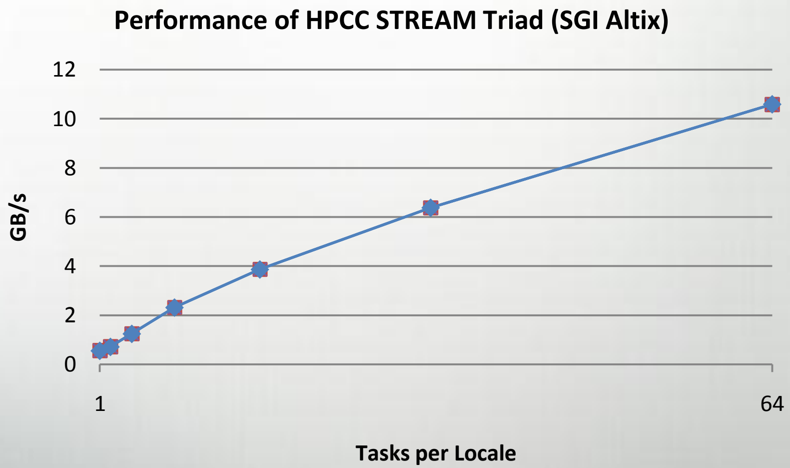


Portability Results







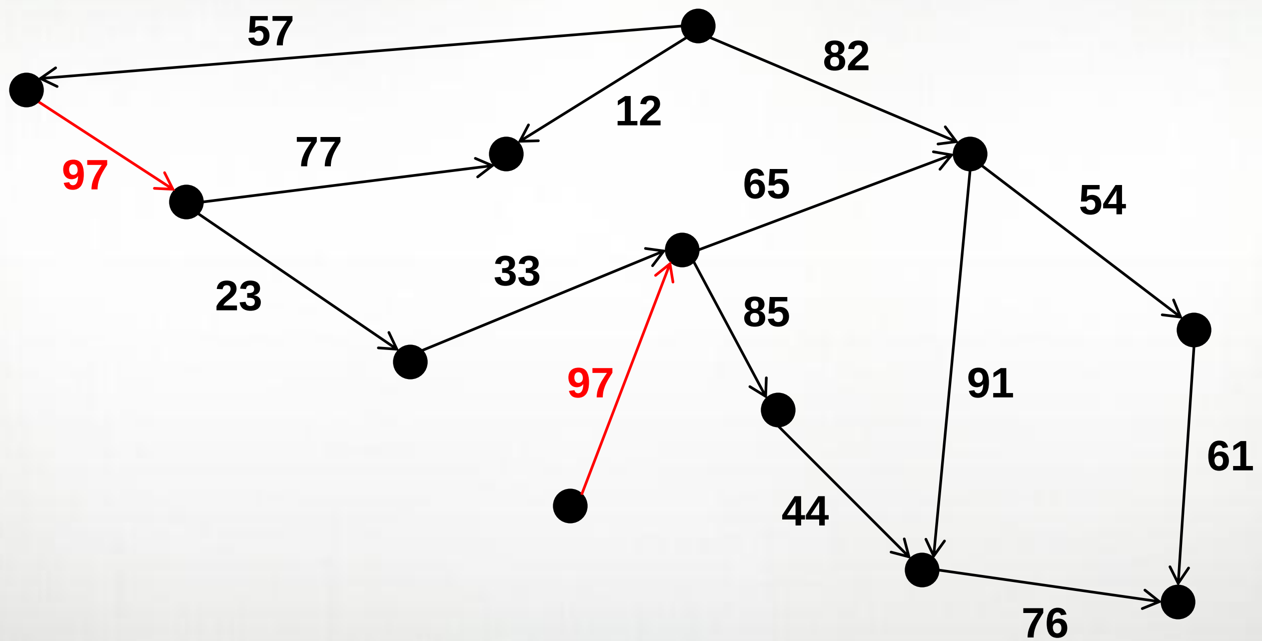


Outline

- Status and Collaborations
- HPCC Benchmarks in Chapel as presented at SC ’09
- SSCA #2 in Chapel

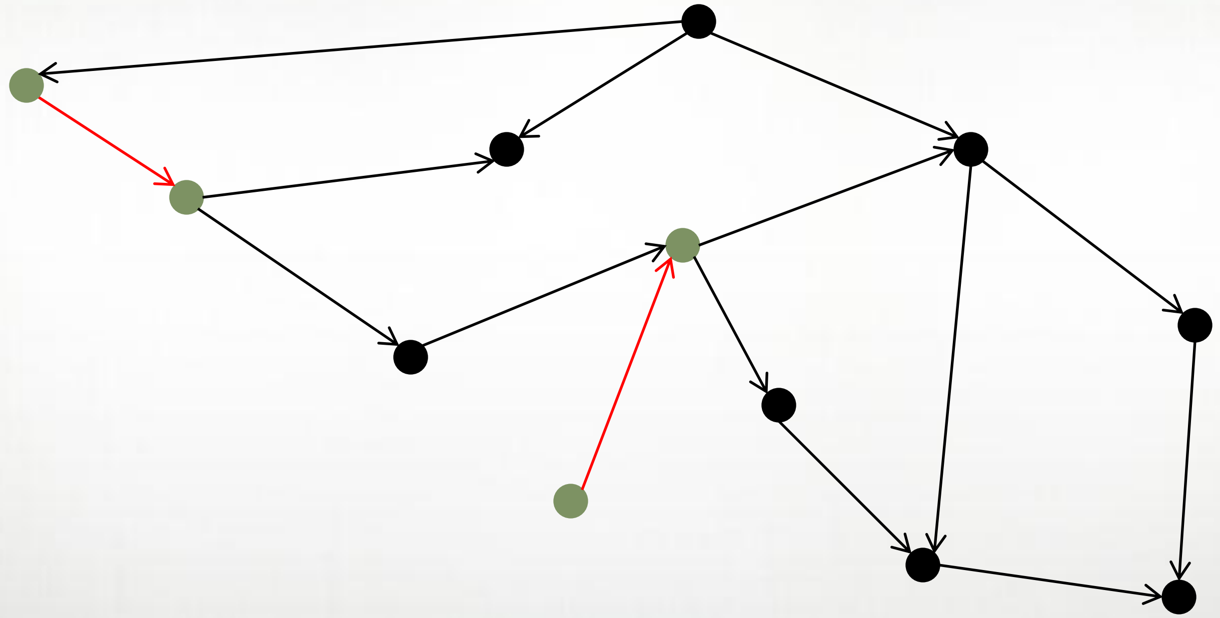
SSCA #2 Kernel 2

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



SSCA #2 Kernel 2

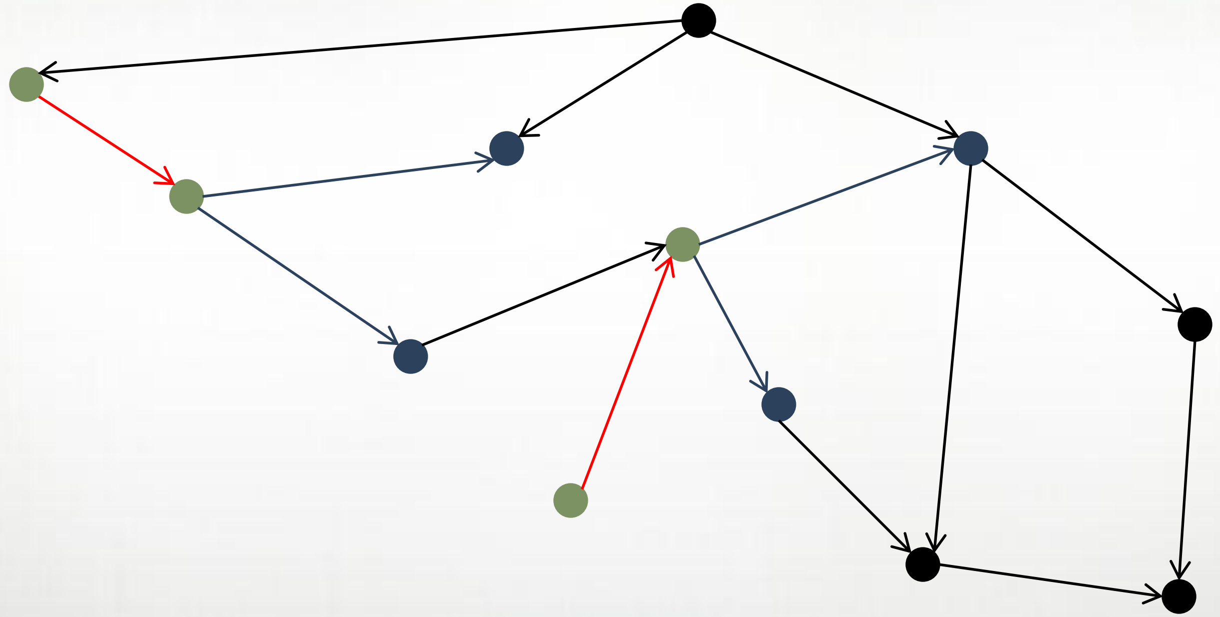
Given a set of heavy edges *HeavyEdges* in directed graph *G*, find sub-graphs of outgoing paths with *length* ≤ *maxPathLength*



maxPathLength = 0

SSCA #2 Kernel 2

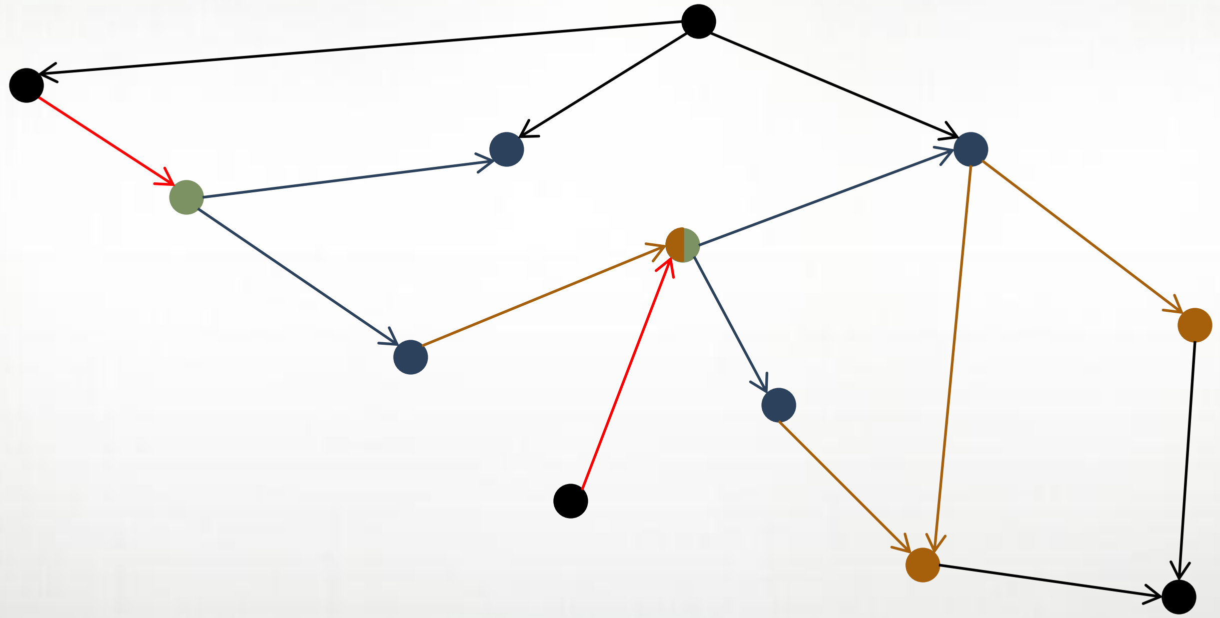
Given a set of heavy edges *HeavyEdges* in directed graph *G*, find sub-graphs of outgoing paths with *length* ≤ *maxPathLength*



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

.nodes.member(w) {
+= w;
odes += w;
dges += (v, w);

axPathLength) then
xtLevel;
```

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);
          }
        }

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

<u>Generic with respect to vertex sets</u>

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)`

Questions?

- Status and Collaborations
  - About Chapel v1.1
  - Implementation status
  - External collaborations
- HPCC Benchmarks in Chapel as presented at SC ’09
- SSCA #2 in Chapel