Chapel HyperGraph Library (CHGL)

MARCIN ZALEWSKI[1]

LOUIS JENKINS[1], TANVEER BHUIYAN[2], SARAH HARUN[2], CHRISTOPHER LIGHTSEY[2], DAVID MENTGEN[2], SINAN AKSOY[1], TIMOTHY STAVENGER[1], HUGH MEDAL[2], CLIFF JOSLYN[1]

[1] Pacific Northwest National Laboratory, Seattle, Washington, USA.
[2] Mississippi State University, Mississippi State, Mississippi, USA.

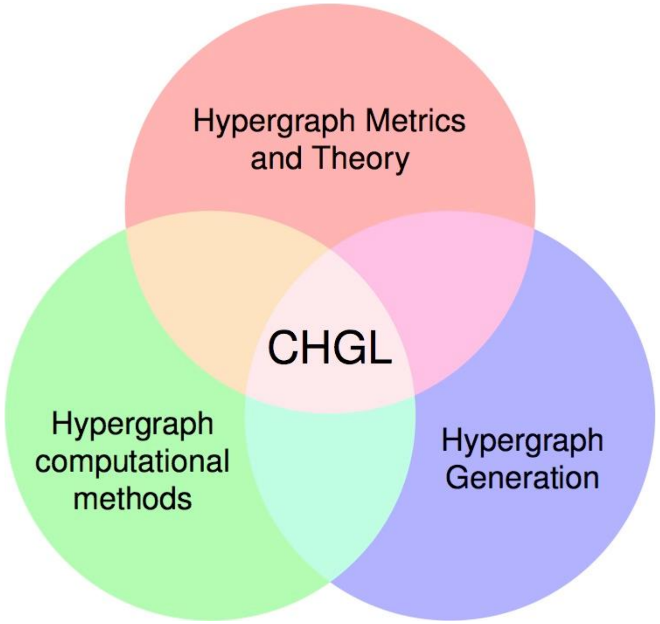
HPEC 2018

Pacific Northwest NATIONAL LABORATORY

U.S. DEPARTMENT OF ENERGY BATTELLE

What We're Trying To Do

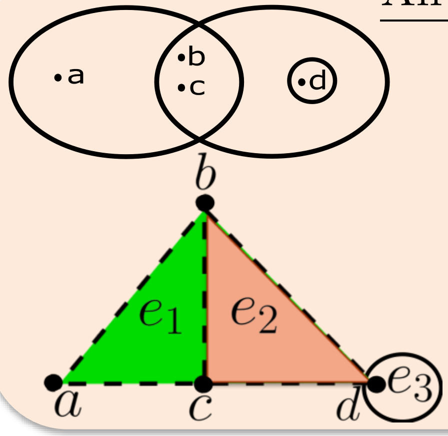
- ***Develop scalable parallel computation methodologies for complex high dimensional graphical data objects***
- **Abstract Hypergraph Analytics:**
  - Graph HPC runtime for vertex and edge centric computation extended to support hypergraphs
  - Mapping abstract hypergraph algorithms to families of efficient asynchronous parallel implementations
- **Chapel HyperGraph Library (CHGL):**
  - Hypergraph generation
    - Scalable generation algorithms that preserve key properties of hypergraphs
  - Hypergraph algorithms
    - Metrics, S-Metrics, connected components, etc.
  - Exploration of irregular applications in Chapel
  - Exploration of abstract interfaces in Chapel
  - Distributed, large-scale, and scalable out of the box
  - Contribute back to Chapel

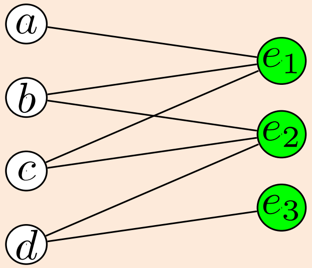


Hypergraphs

- A **hypergraph** H on a finite set of vertices V is a set $H = \{e_1, \ldots, e_m\}$ such that for $i = 1, \ldots, m$, we have $e_i \subseteq V$ and $e_i \neq \varnothing$.
  - A **graph** G is a hypergraph in which every edge has cardinality 2.
- Ex: $H = \{\underbrace{\{a, b, c\}}_{e_1}, \underbrace{\{b, c, d\}}_{e_2}, \underbrace{\{d\}}_{e_3}\}$ is a hypergraph on $V = \{a, b, c, d\}$.

All the same hypergraph...





	e_1	e_2	e_3
a	1	0	0
b	1	1	0
c	1	1	0
d	0	1	1

$$H = \{abc, bcd, d\}$$

Why Chapel

Pacific Northwest
NATIONAL LABORATORY

- Chapel...
  - Has strong HPC abstractions and language constructs
    - Data-Parallelism and Data-Driven Locality
  - Is a Partitioned Global Address Space (PGAS) language
    - But data structures provide seamless access to distributed data
  - Has a rich type system and generics
  - Offers first-class support arrays, domains, and distributions such as global-arrays
- Multiresolution Philosophy
  - High-level abstractions are implemented in terms of low-level abstractions
  - Low-level abstractions can be configured to fine-tune performance of high-level abstractions
    - Communication & Tasking Layer, Hierarchical Locale Models, global-view arrays
- Designed to work on a laptop or supercomputer
  - Chapel enables this 'out-of-the-box'
- Optimized for both shared memory and distributed memory

CRAY CHAPEL

CHGL Warmup

- Graph is created with a *distribution*
  - Can be default (local), one of the Chapel-provided distributions (Cyclic) here, or custom
  - Here, distribution is cyclic on locales 4, 6, and 8 (4..8 by 2)
- Aggregation of messages can be turned on and off
  - Adding inclusions produces small messages, so aggregation improves performance
- Types are inferred where possible
  - E.g., numVertices and numEdges are int
  - All types can and are inferred here, but they could be also specified explicitly

```
const numVertices = 1024;
const numEdges = 2048;
const domainMap = new Cyclic(
  startIdx=0, targetLocales=Locales[4..8 by 2]);
var graph = new AdjListHyperGraph(numVertices,
  numEdges, domainMap)
graph.startAggregation();
forall v in graph.getVertices() do
  forall e in graph.getEdges() do
    graph.addInclusion(v,e);
graph.stopAggregation();
```

CHGL Warmup

Pacific Northwest NATIONAL LABORATORY

- Simple task: collect all degrees
  - Create an array with the same domain as vertices
  - Iterate through the array and degrees in parallel
  - Assign the degrees to the array and reduce
- What if we just want the total number of inclusions?
  - Simple, just reduce on the fly
  - Reduction is built in and parameterized by a binary operation
  - Reduction can be used just like a variable
- What if we did something wrong?
  - Chapel allows us to explicitly signal errors
  - We provide a ”catch all” overload that produces a useful error message
  - This is simple example, but this is a general method

```
var vertexDegrees : [graph.verticesDomain()] int;
forall (degree, vertex) in zip(vertexDegrees,
  graph.getVertices()) {
    degree = graph.numNeighbors(vertex);
}
var totalVertexDegrees = + reduce vertexDegrees;
```

```
var numInclusions : int;
forall v in graph.getVertices() with
  (+ reduce numInclusions) do
    numInclusions += graph.numNeighbors(v);
```

```
inline proc numNeighbors(other) {
  compilerError("'numNeighbors(", other.type
      : string, "') is not supported...\n",
    "Require argument of type ", vDescType
      : string, " or ", eDescType : string);
}
```

CHGL Philosophy

Genericity

- Abstract interfaces that describe classes of data structures
  - Well-thought out interfaces
  - Durable
  - Minimal
  - Performance guarantees
- Reusable algorithms
  - Write once
  - Use with many data structures
  - Avoid implementation details

Performance

- Enable performance at scale
  - Distributed memory
  - Scalability
- Rely on Chapel for the basics
- Design efficient data structures and algorithms
  - Efficient but elegant
  - Explore what is possible today
  - Low-level implementation if necessary with forward looking design

Usability

- Provide simple interfaces
- Provide multiple interfaces
- Allow customization for advanced users
- Modern feel
  - Use language features
  - Fit the expected language style
- Drive development by user expectations rather than by implementation needs

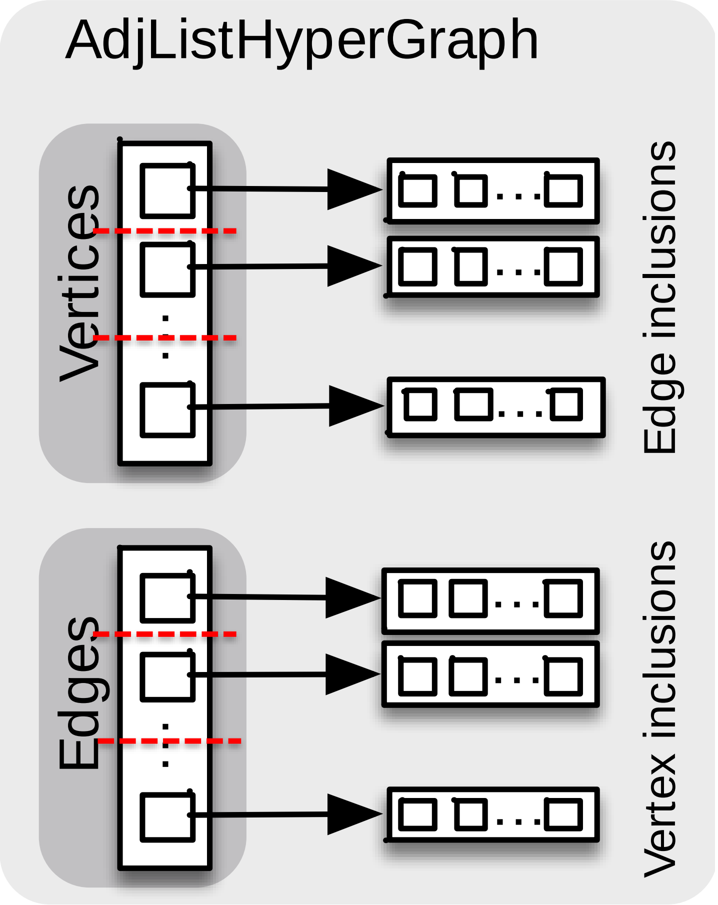
API

- CHGL: Chapel-flavored generic hypergraph interface
- Use-case driven
  - Make sure that interfaces are necessary for some algorithms
  - Do not overdevelop
- Currently used for graph generation
- This is observable interface
  - Implementation ”under the hood” may be more complex

```
iter getVertices() : vDescType;
iter getEdges() : eDescType;
proc verticesDomain : vDomainType;
proc edgesDomain : eDomainType;
proc startAggregation() : void;
proc stopAggregation() : void;
proc addInclusion(v : vDescType, e : eDescType) : void;
proc removeInclusion(v : vDescType, e : eDescType) : void;
proc hasInclusion(v : vDescType, e : eDescType) : bool;
iter neighbors(v : vDescType) : eDescType;
iter neighbors(e : eDescType) : vDescType;
proc numNeighbors(v : vDescType) : int;
proc numNeighbors(e : eDescType) : int;
```

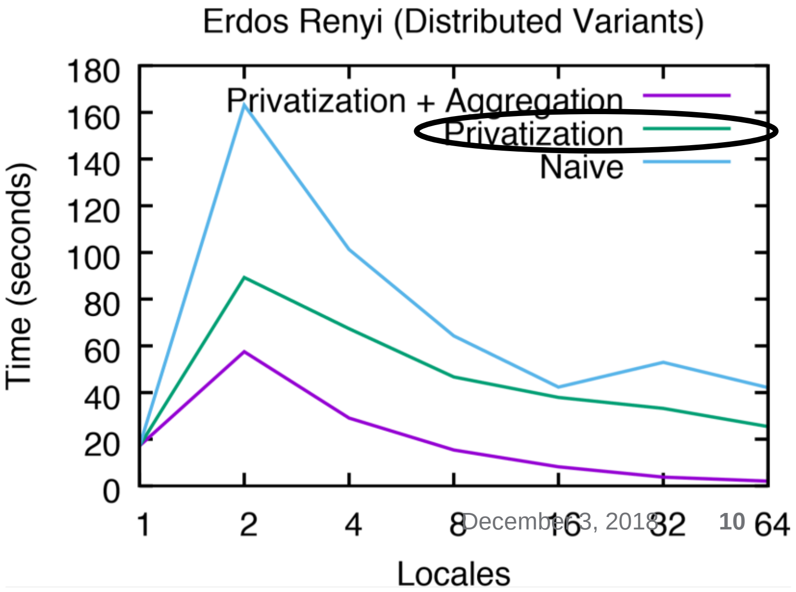
CHGL AdjListHyperGraph

- Adjacency list hypergraph
  - CSR storage for edges and vertices
  - Very much like a bipartite graph storage
- Both inner and outer containers are implemented with Chapel arrays
  - We want to reuse one of Chapel's strongest abstractions
  - We can build on distributions functionality
    - Outer lists are distributed (1D)
    - In the future, inner lists may be distributed for some vertices (1.5D)
- Currently, traversal is based on inclusions
  - We will be extending our generic interface with s-walk concepts
    - Not strictly necessary for graph generation yet



Privatization

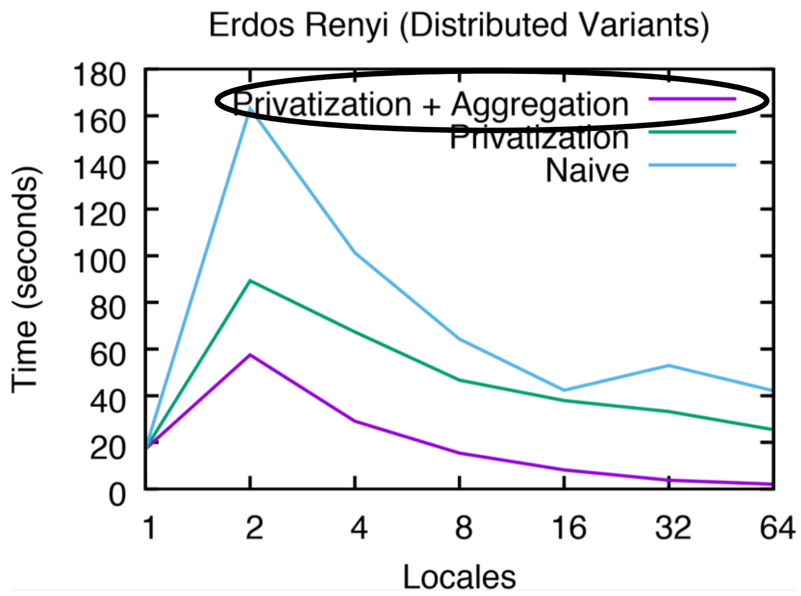
- A shallow clone of the data structure is maintained on each locale
  - All accesses to data structure are forwarded to per-locale clone
  - Clone can have locale-private decentralized data fields
  - Clone can have wide pointers to centralized data fields
- Eliminates fine-grained communication associated with accessing a remote objects
  - Lightens network bottleneck

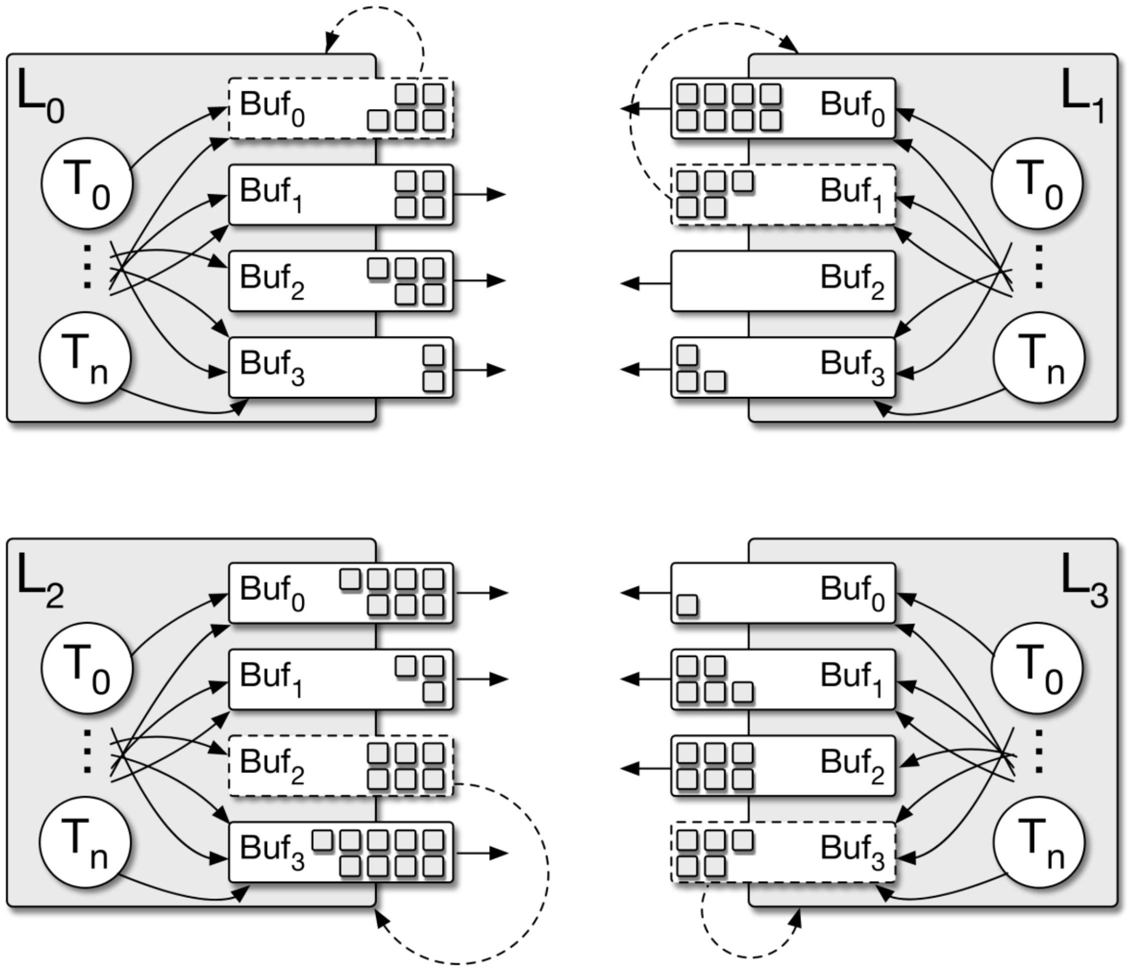



```
pragma "always RVF"
record AdjListHyperGraph {
  var instance; var pid = -1;
  proc _value {
    return chpl_getPrivatizedCopy(instance.type, pid);
  }
  proc init(numVertices = 0, numEdges = 0, map) {
    instance = new AdjListHyperGraphImpl(numVertices, numEdges, map);
    pid = instance.pid;
  }
  forwarding _value;
}
class AdjListHyperGraphImpl {
  var _vertices : [_verticesDomain] NodeData(eDescType);
  var _privatizedVertices = _vertices._value;
  proc init(numVertices = 0, numEdges = 0, map) {
    this.pid = _newPrivatizedClass(this);
  }
}
```

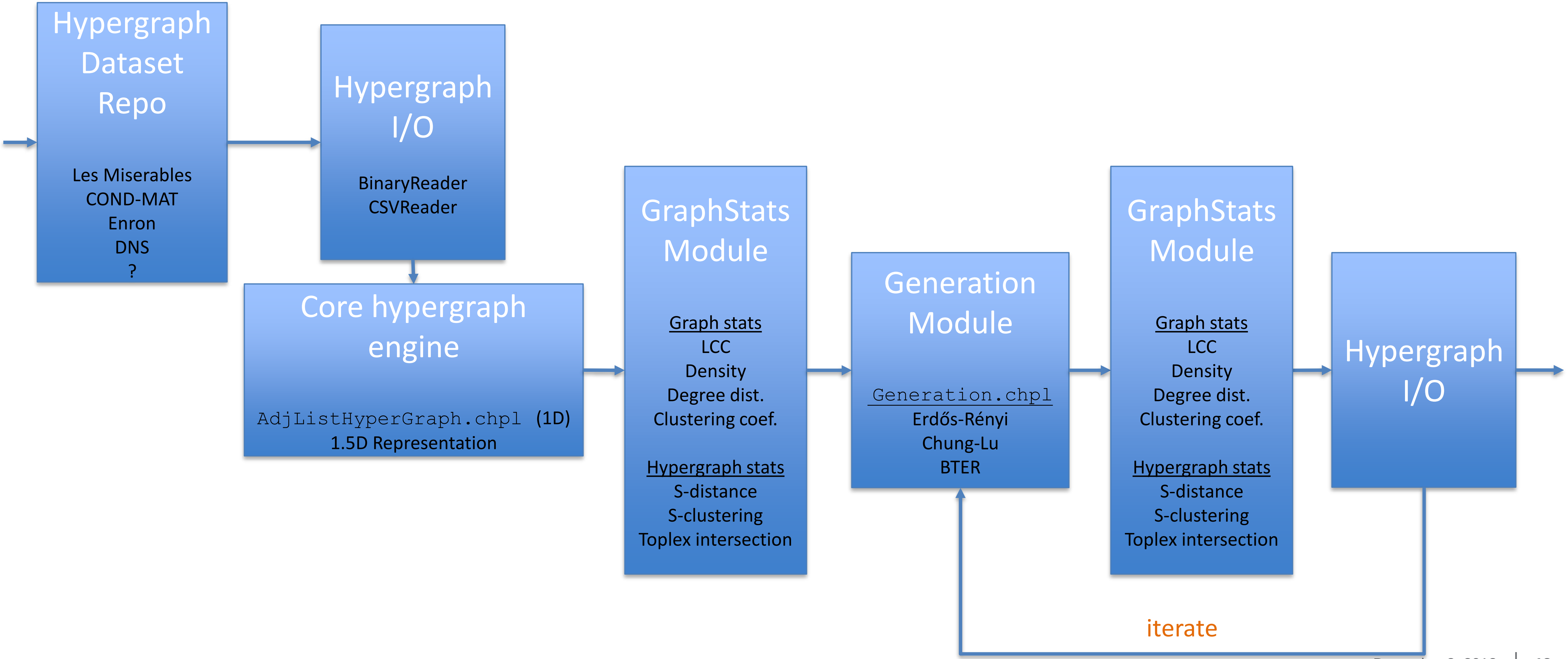
Aggregation

- Chapel Aggregation Library
  - To Appear in PAW-ATM, an SC'18 Workshop
- Each privatized instance manages its own aggregation buffer
  - Currently only used in 'addInclusion'
  - Further reduces the network bottleneck



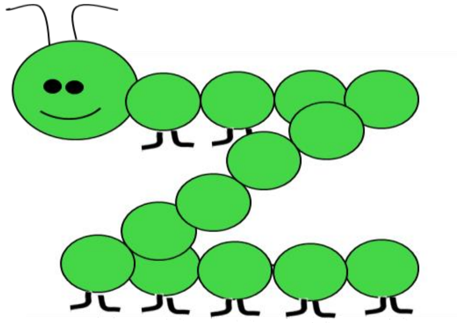


Goal: End-to-End Hypergraph Analytics Tool

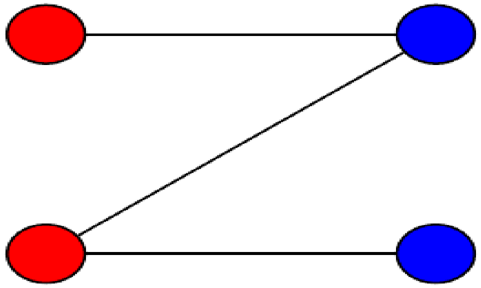


Metamorphosis Coefficient for Clustering

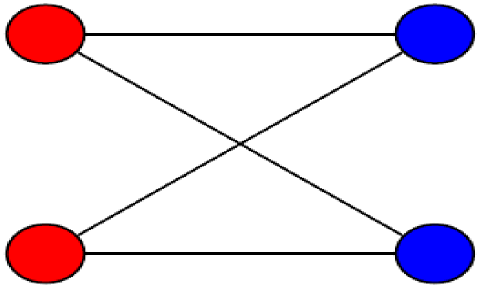
- 4-cycle = smallest units of social cohesion in a bipartite graph



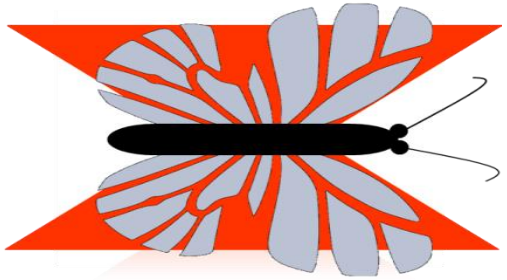
Caterpillar



3-path



4-cycle



Butterfly

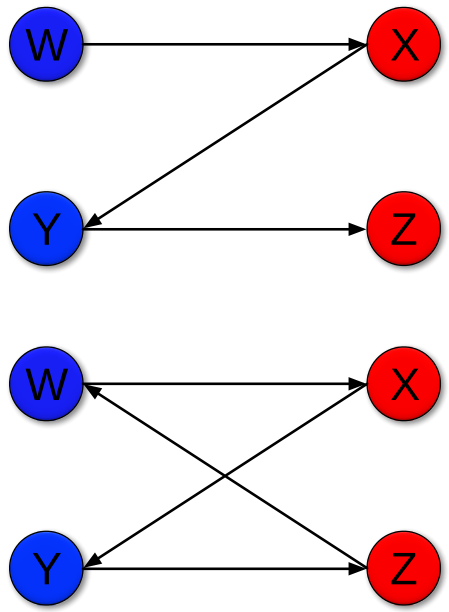
How often does a 3-path close into a 4-cycle?, i.e.
How frequently are shared affiliations repeated?

Citation: Measuring and modeling bipartite graphs with community structure, Sinan G. Aksoy Tamara G. Kolda Ali Pinar, *Journal of Complex Networks*, Volume 5, Issue 4, 1. August 2017, Pages 581–603.

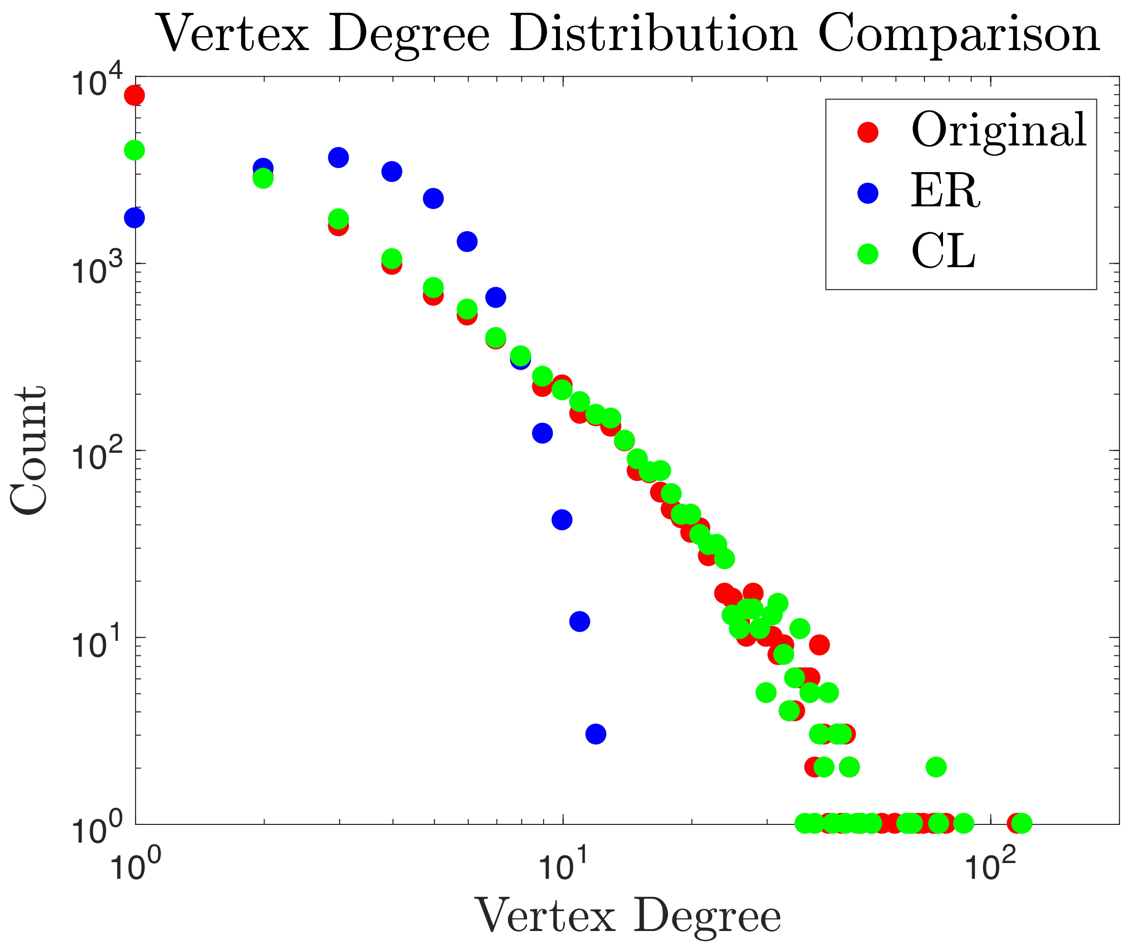
Counting Caterpillars and Butterflies

- Iterate through caterpillars and through butterflies in a hypergraph
- This code works in shared and in distributed memory
- Works for any graph

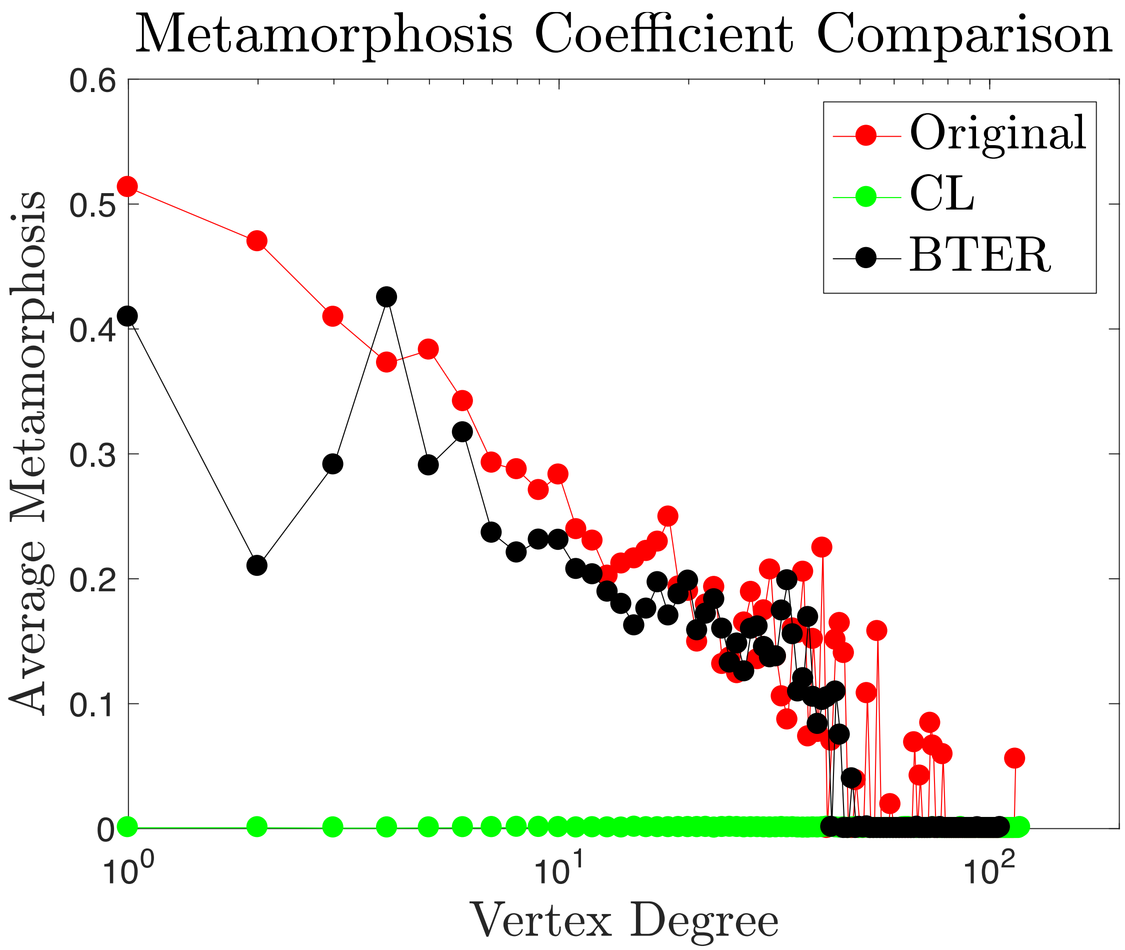
```
iter caterpillars(graph) {
    forall w in graph.getVertices() do
        forall x in graph.neighbors(w) do
            forall y in graph.neighbors(x) do
                if y != w then forall z in graph.neighbors(y) do
                    if z != x then yield (w,x,y,z);
}

iter butterflies(graph) {
    forall (w,x,y,z) in caterpillars(graph) do
        if graph.hasInclusion(w,z) then yield (w,x,y,z);
}
```



Evaluation

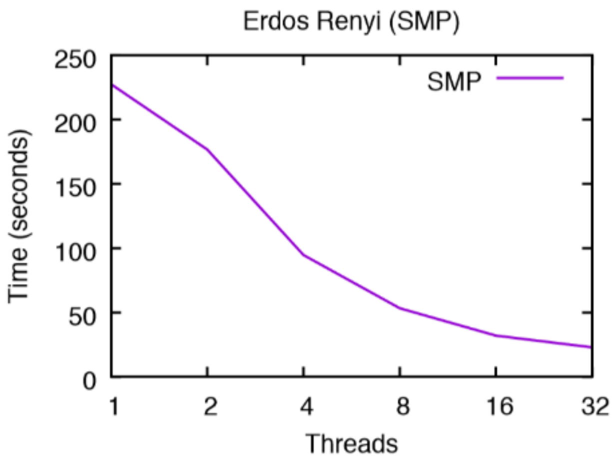


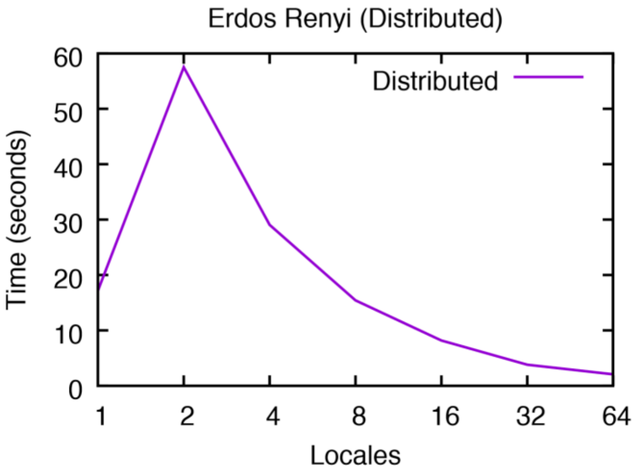
Vertex Degree Distribution Comparison
Count
Vertex Degree
Original
ER
CL

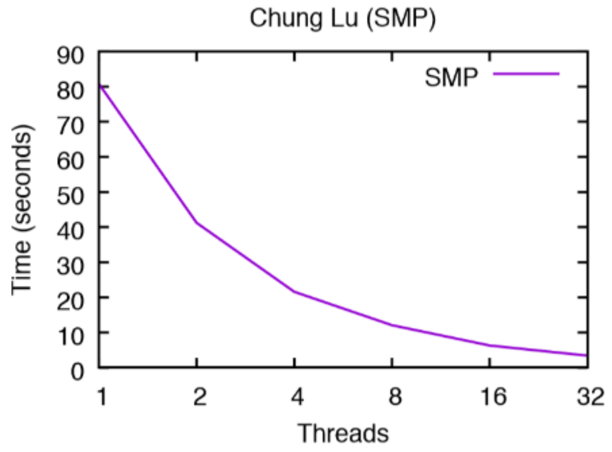


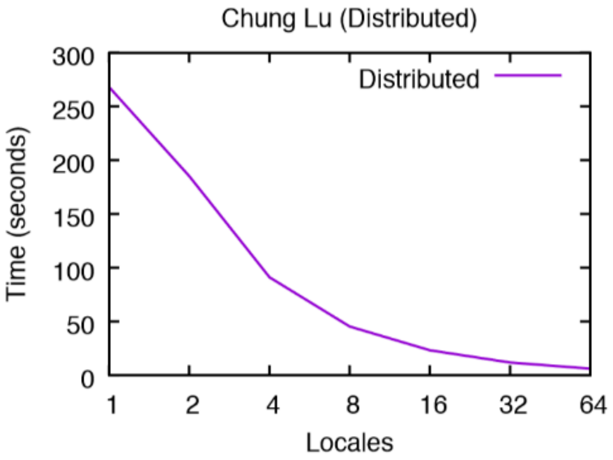
Metamorphosis Coefficient Comparison
Average Metamorphosis
Vertex Degree
Original
CL
BTER

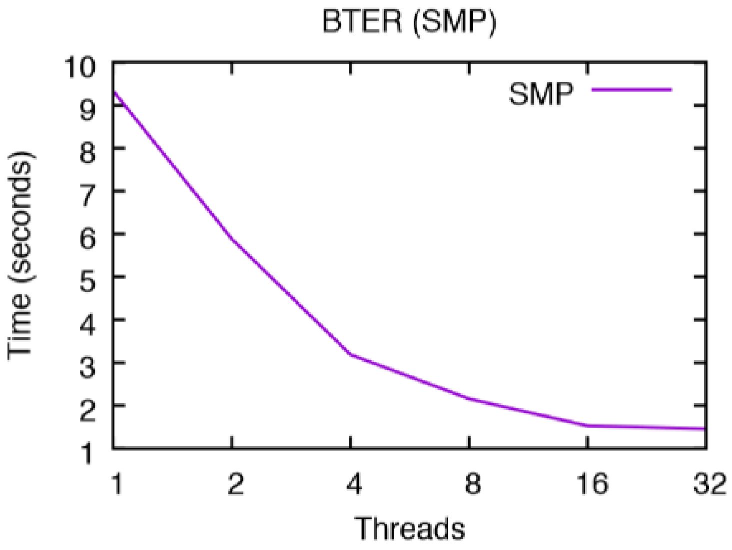
Performance





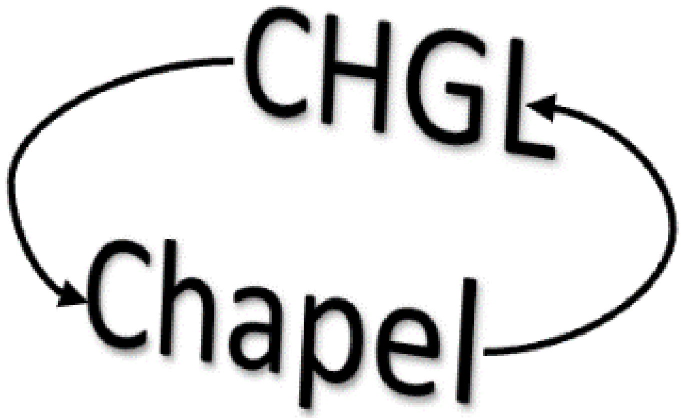






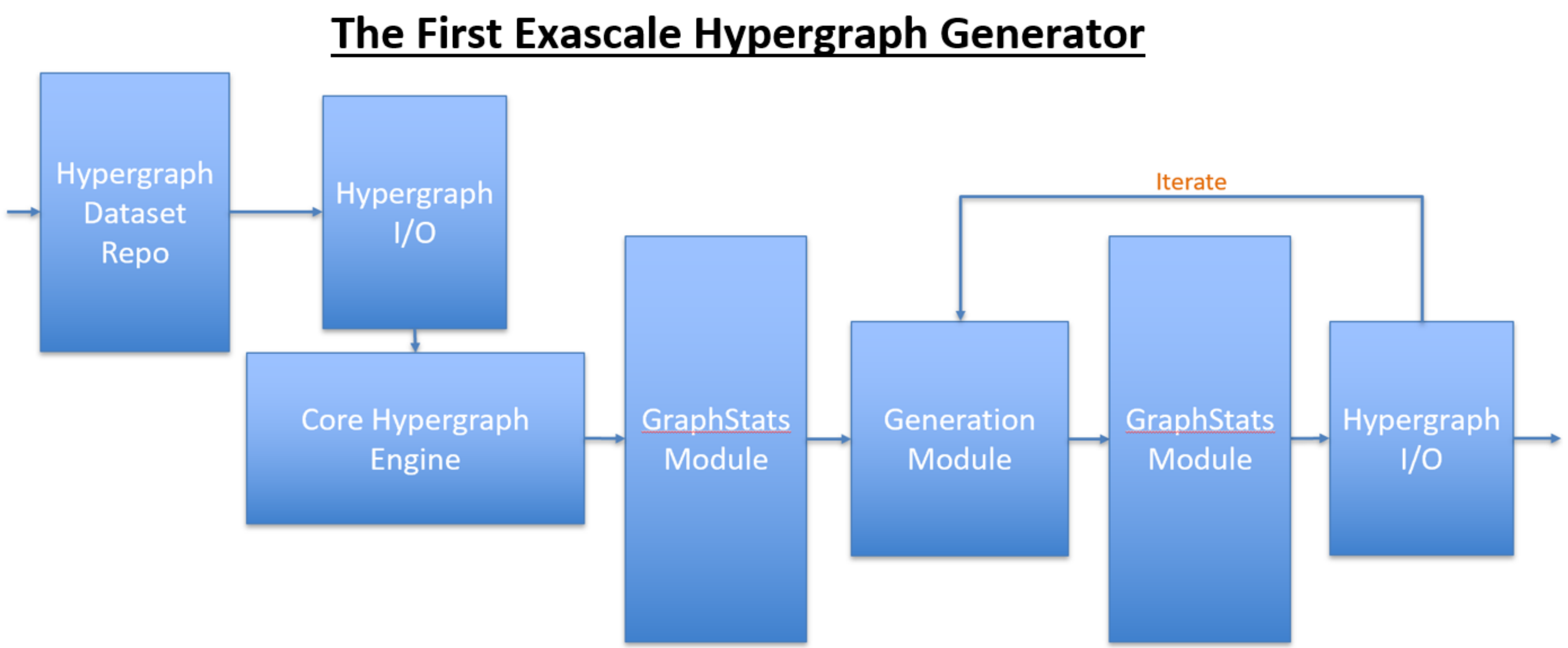
Conclusions

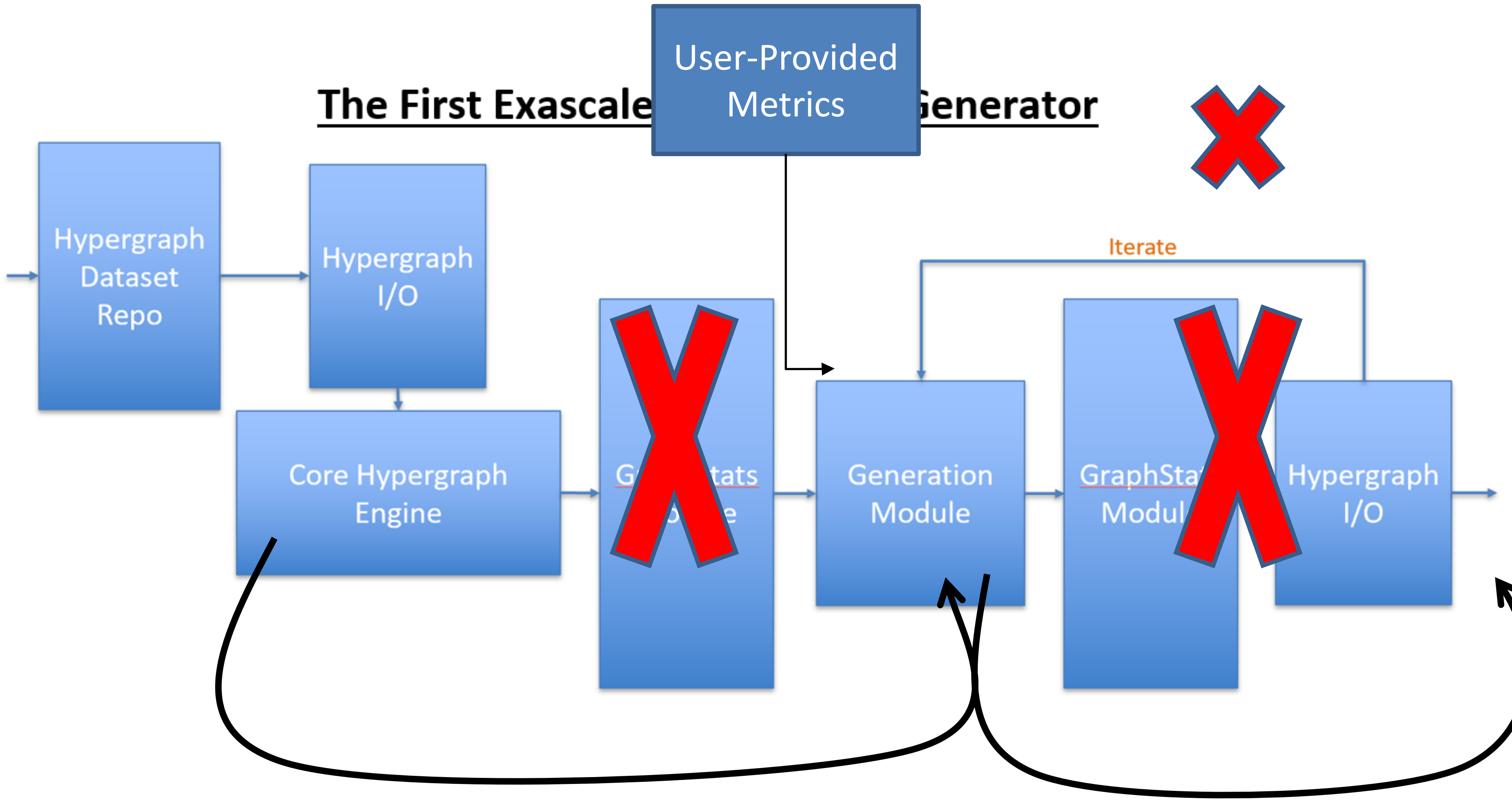
- One of the few software packages specifically targeted at hypergraphs
- Provides a good initial set of methods and data structures
  - 1-D distributed hypergraph
  - Hypergraph metrics
  - 3 hypergraph generation algorithms
- Generic design: high-level, conceptual, write once
- Ease of use is one of the main goals
- Efficient
  - Privatization, aggregation, other low level features
- Collaboration between PNNL and Cray
  - Chapel is not designed for irregular algorithms
  - Chapel improves as CHGL exposes flaws
    - CHGL improves as Chapel improves
  - Many issues opened in the Chapel issue tracker



Thank You!

https://github.com/pnnl/chgl


The First Exascale Hypergraph Generator
Hypergraph Dataset Repo
Hypergraph I/O
Core Hypergraph Engine
GraphStats Module
Generation Module
GraphStats Module
Hypergraph I/O
Iterate



User-Provided Metrics
The First Exascale
Generator
Hypergraph Dataset Repo
Hypergraph I/O
Core Hypergraph Engine
tats
Iterate
Generation Module
GraphSta
Modul
Hypergraph I/O