Pacific Northwest
NATIONAL LABORATORY

Computing Hypergraph Homology in Chapel

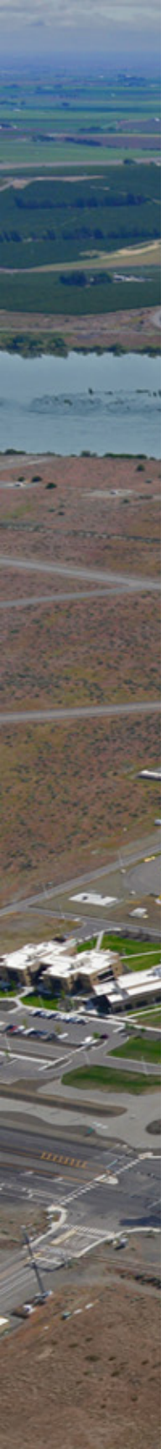
Jesun Firoz (PNNL)

Louis Jenkins (U Rochester),Cliff Joslyn (PNNL),
Brenda Praggastis (PNNL), Emilie Purvine (PNNL),
Mark Raugas (PNNL)

7th annual Chapel Implementers and Users Workshop (CHIUW 2020)

U.S. DEPARTMENT OF ENERGY BATTELLE

PNNL is operated by Battelle for the U.S. Department of Energy



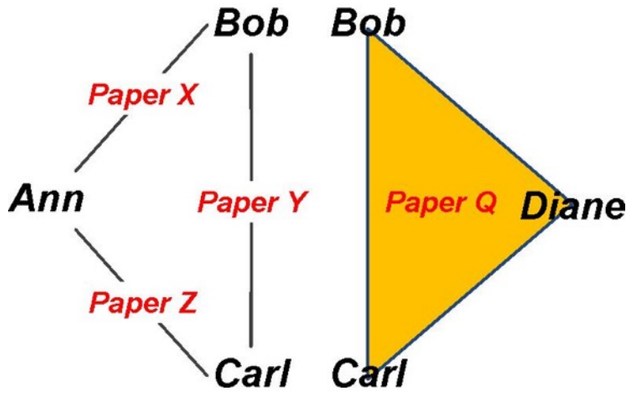
Pacific Northwest NATIONAL LABORATORY

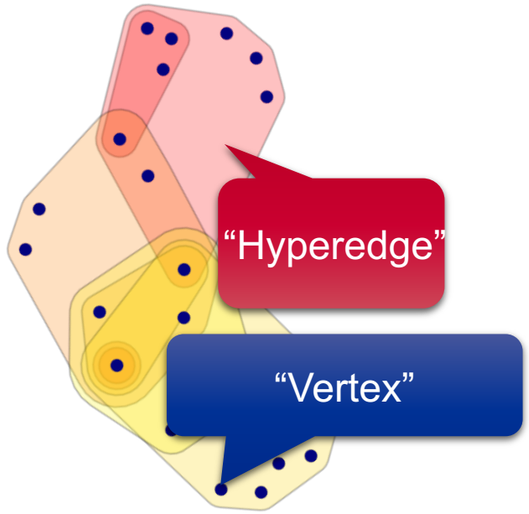
Outline

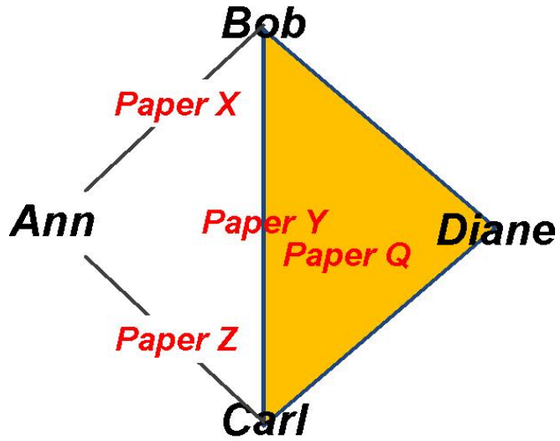
- Hypergraphs
- Hypergraph Homology Computation
  - Motivation
- Linear algebra operations: Chapel vs Python
- Hypergraph Homology Computation in Chapel
  - Two main steps
  - Optimizations
  - Experimental results

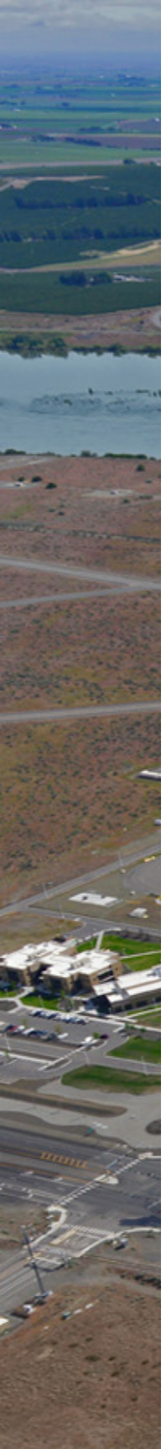
Hypergraphs

- **Graphs** model distributed interactions among entities
  - Questions of connectivity, reachability, density, clustering
- ***But multi-way interactions cannot be modeled natively by graphs!***
  - **Requires higher coding strategies:** Property graphs, reification, bipartite structures
- **Multi-Dimensional Graph:** Hypergraph





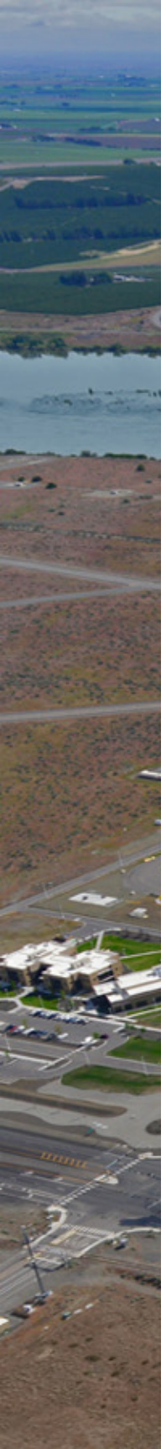




Pacific Northwest NATIONAL LABORATORY

Hypergraph Homology Computation in Chapel

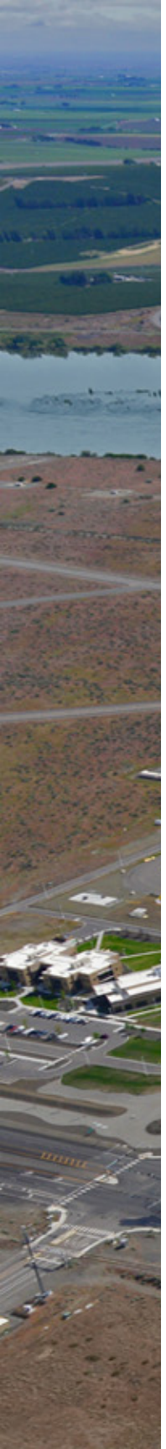
- Motivation:
  - Learn about missing data and their surrounding datasets.
  - Datasets and ASCs can be huge.
  - Leverage Chapel's parallel and distributed execution capability to run at scale.
- Two main steps:
  - Computing the boundary matrices
  - Computing the Smith Normal form of a matrix.
- Heavily involves matrix and linear algebra operations.



Pacific Northwest NATIONAL LABORATORY

Homology Computation in Chapel (Cont.)

- Matrices are represented as 2D arrays in Chapel.
- Array-centric operations include:
  - Row and column interchange,
  - Pivot calculation,
  - Addition of rows and columns,
  - Rank computation,
  - Multiplication, and
  - Slicing
- In Chapel, such operations are very concise and more intuitive than in Python.
- While Chapel's linear algebra library performs various operations on matrices (such as singular value decomposition, LU decomposition etc.), our operations are in the Z_2 field, so we wrote our own.



Linear Algebra Operations: Chapel vs Python

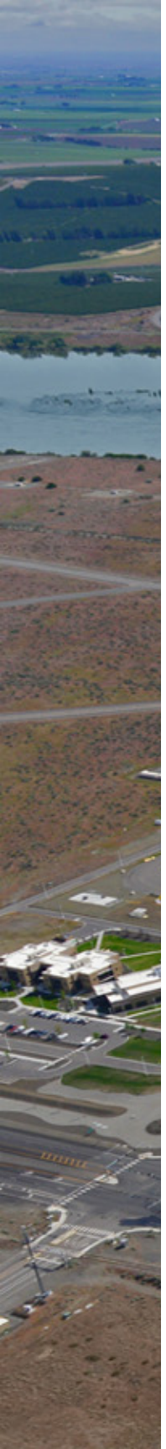
Chapel

```
1  proc swap_rows(i,j,M) {
2    var N = M;
3    N[i, ..] <=> N[j, ..];
4    return N;
5  }
6  proc add_to_row(M,i,j,ri=1,rj=1) {
7    var N = M;
8    N[i, ..]  = (ri * N[i, ..] + rj * N[j, ..]) % 2;
9    return N;
10 }
11 proc calculateRank(M) return + reduce
12   [i in M.domain.dim(2)] (max reduce M[.., i]);
```

Python (Numpy/sequential)

```
1  def rowswap(i,j,M):
2  N = copy.deepcopy(M)
3      N[i] = M[j]
4      N[j] = M[i]
5      return N
6  def add_to_row(M,i,j,ri=1,rj=1,mod=2):
7      N = copy.deepcopy(M)
8      N[i] = np.mod(ri*N[i] + rj*N[j],[mod])
9      return N
10 def calculaterank(M):
11   return np.sum(M)
```

Reduction intent: For each column, see if at least one 1, **in parallel**



Pacific Northwest NATIONAL LABORATORY

Linear Algebra Operations: Chapel vs Python

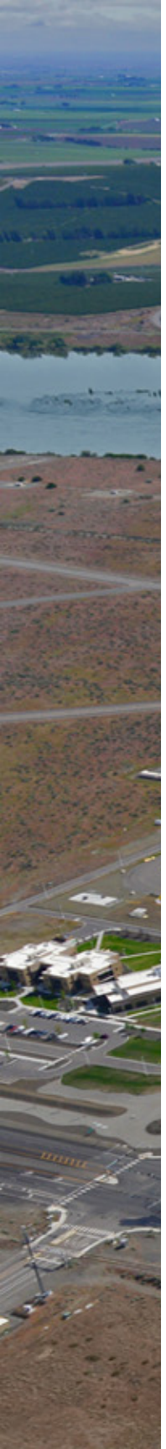
Chapel

```chapel
proc matmultmod (M, N, mod =2) {
  var C : [M.domain.dim(1), N.domain.dim(2)] int;
  forall (i,j) in C.domain {
    C[i,j] = (+ reduce (M[i, M.domain.dim(2)]
     * N[M.domain.dim(2), j])) % 2;
  return C;
}
```

Parallel matrix multiplication

Python (Numpy/sequential)

```python
def modmult(M,N,mod=2):
  return np.mod(np.matmul(M,N),mod)
```



Pacific Northwest
NATIONAL LABORATORY

Homology Computation in Chapel

Step 1. Boundary Matrix Computation: What We Will Do

- For example: Given a hypergraph with one hyperedge with four vertices:

```
1 [{ 0  1  2  3 }]
```

- Generate the ASCs (powerset of the hyperedge) and accumulate the k-cells in the corresponding bins:

```
1 2 -> [{ 0  1  2 }, { 0  1  3 }, { 0  2  3 }, { 1  2  3 }]
2 1 -> [{ 0  1 }, { 0  2 }, { 0  3 }, { 1  2 }, { 1  3 }, { 2  3 }]
3 0 -> [{ 0 }, { 1 }, { 2 }, { 3 }]
4 3 -> [{ 0  1  2  3 }]
```

0-cells

- Compute the boundary maps:

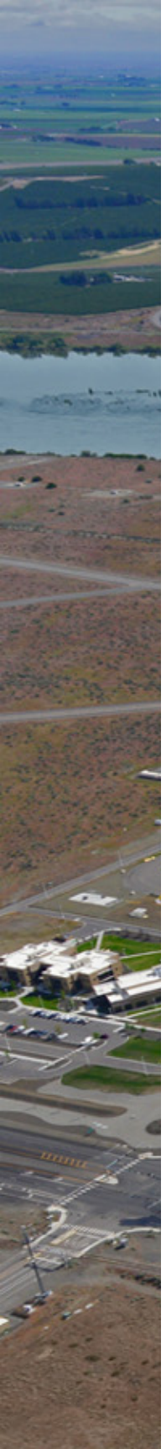
1-cells

0-cells

	1	2	3	4	5	6
1	1	1	1	0	0	0
2	1	0	0	1	1	0
3	0	1	0	1	0	1
4	0	0	1	0	1	1

	1	2	3	4
1	1	1	0	0
2	1	0	1	0
3	0	1	1	0
4	1	0	0	1
5	0	1	0	1
6	0	0	1	1

	1
1	1
2	1
3	1
4	1



Boundary Matrix Computation
Step 1a: Computing Abstract Simplicial Cell Complexes (ASCs)

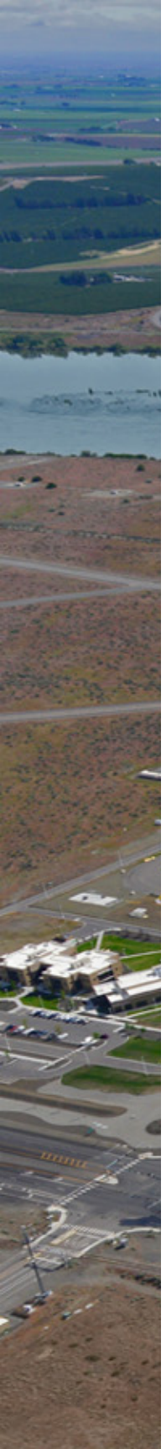
Maintain one set per task per locale for gathering the ASCs

```
1 var cellSets : [0..#numLocales, 0..#here.maxTaskPar] set(Cell);
```

Each edge computes the ASCs in parallel

```
1 var taskIdCounts : [0..#numLocales] atomic int;
2 forall e in hypergraph.getEdges()
3   with (var tid : int = taskIdCounts[here.id].fetchAdd(1)) {
4   var vertices = hypergraph.incidence(e);
5   ref tmp = vertices[1..#vertices.size];
6   var verticesInEdge : [1..#vertices.size] int = tmp.id;
7   processCell(new Cell(verticesInEdge), cellSets[here.id, tid]);
8 }
```

Generate power set/ASCs for the hyperedge



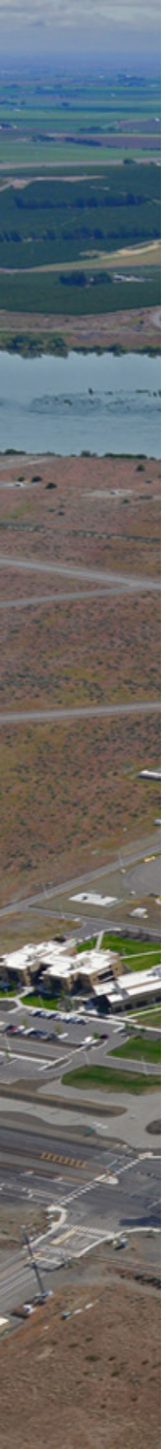
Boundary Matrix Computation
Step 1b: Combining K-cells

```
1 var cellSets : [0..#numLocales, 0..#here.maxTaskPar] set(Cell);
```

Hashed distribution that maps the distributed associative domains of cells to target locales

```
1 var cellSet : domain(Cell, parSafe=true) dmapped Hashed(idxType=Cell);
2 for cset in cellSets {
3   forall cell in cset with (ref cellSet) {
4     cellSet += cell;
5   }
```

Adding cells to the domain



Boundary Matrix Computation
Step 1b: Binning all K-cells

Bin cells based on their sizes

Thread-safe map

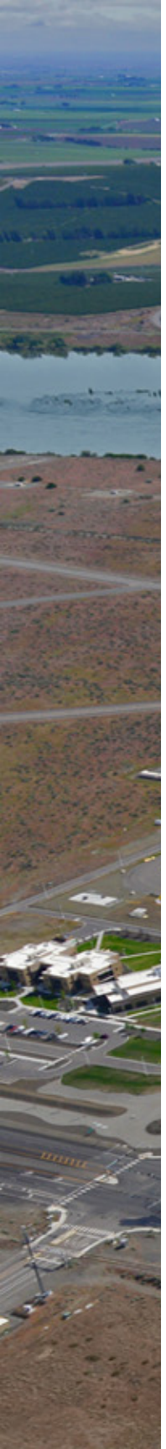
```
1 var kCellMap = new map(int, list(Cell, parSafe=true), parSafe=true);
2 forall cell in cellSet {
3   kCellMap[cell.size - 1].append(cell);
4 }
```

Potential bottleneck: merge is happening on one Locale.

```
1 forall (_kCellsArray, kCellKey) in zip(kCellsArrayMap, kCellKeys){
2   _kCellsArray = new owned kCellsArray(kCellMap[kCellKey].size);
3   _kCellsArray.A = kCellMap[kCellKey].toArray();
4   sort(_kCellsArray.A, comparator=absComparator);
5 }
```

Sort each bin in parallel

Customized comparator for sorting

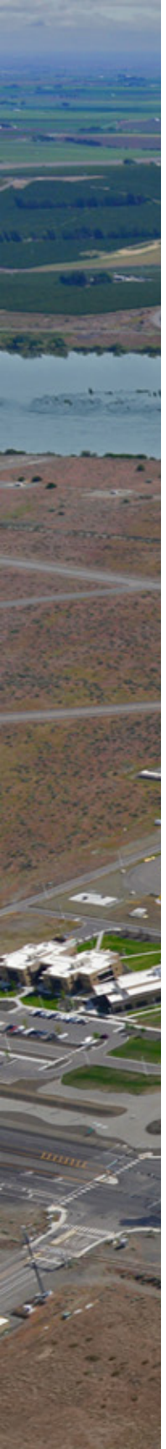


Pacific Northwest NATIONAL LABORATORY

Boundary Matrix Computation: Comparator for Sorting Bins

- Customized comparator for sorting

```
1 proc Comparator.compare(a : Cell, b : Cell) : int {
2   assert(a.size == b.size);
3   for (_a, _b) in zip(a,b) {
4     if (_a > _b) then return 1;
5     else if (_a < _b) then return -1;
6   }
7   return 0;
8 }
```



Pacific Northwest NATIONAL LABORATORY

Boundary Matrix Computation: What We Have Done So Far

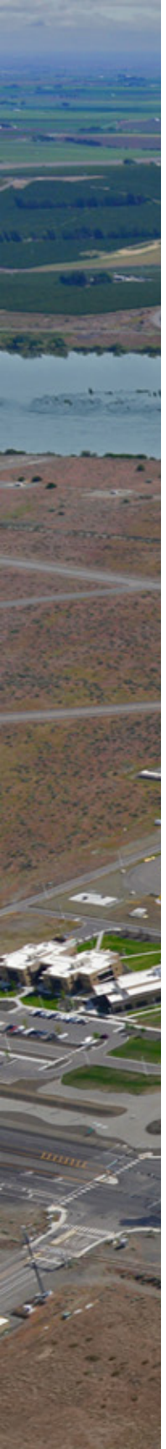
- For example: Given a hypergraph with one hyperedge with four vertices:

```
1 [{ 0  1  2  3 }]
```

- Generate the ASCs (powerset of the hyperedge) and accumulate the k-cells in the corresponding bins:

```
1 2 -> [{ 0  1  2 }, { 0  1  3 }, { 0  2  3 }, { 1  2  3 }]
2 1 -> [{ 0  1 }, { 0  2 }, { 0  3 }, { 1  2 }, { 1  3 }, { 2  3 }]
3 0 -> [{ 0 }, { 1 }, { 2 }, { 3 }]
4 3 -> [{ 0  1  2  3 }]
```

0-cells



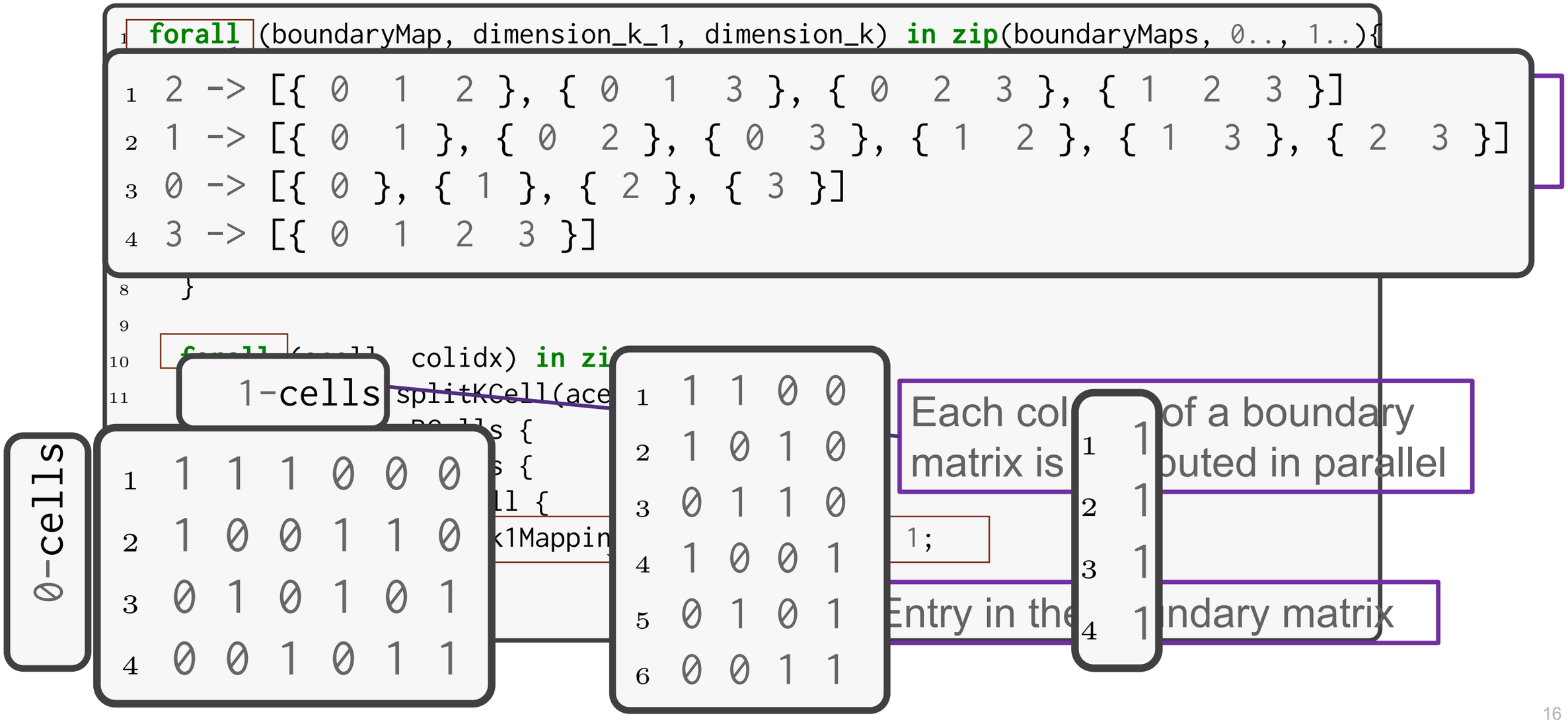
Pacific Northwest NATIONAL LABORATORY

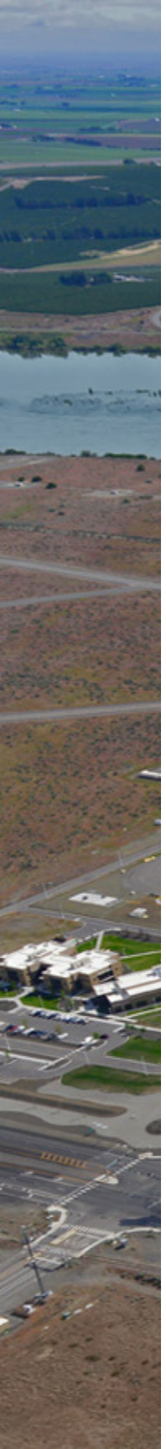
Boundary Matrix Data Structure

```
1 class Matrix {
2   var N : int;
3   var M : int;
4   var D = {1..N, 1..M} dmapped Block(boundingBox = {1..N, 1..M});
5   var matrix : [D] int;
6   proc init(_N: int, _M:int) {
7     N = _N;
8     M = _M;
9 }
```

Block-distributed 2D array

Boundary Matrix Computation





Boundary Matrix Computation: What we have done so far

- Given the bins:

```
2 -> [{ 0  1  2 }, { 0  1  3 }, { 0  2  3 }, { 1  2  3 }]
1 -> [{ 0  1 }, { 0  2 }, { 0  3 }, { 1  2 }, { 1  3 }, { 2  3 }]
0 -> [{ 0 }, { 1 }, { 2 }, { 3 }]
3 -> [{ 0  1  2  3 }]
```

- Computed the boundary maps:

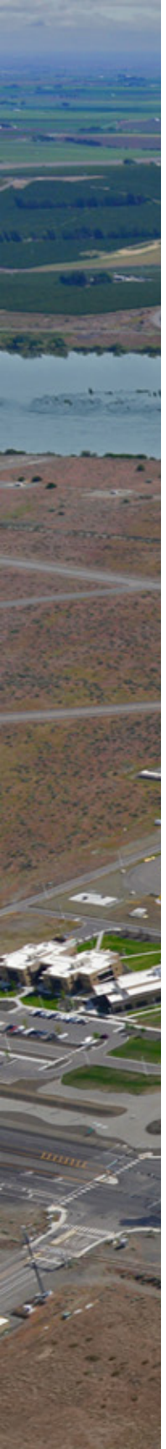
1-cells

0-cells

1	1	1	1	0	0	0
2	1	0	0	1	1	0
3	0	1	0	1	0	1
4	0	0	1	0	1	1

1	1	1	0	0
2	1	0	1	0
3	0	1	1	0
4	1	0	0	1
5	0	1	0	1
6	0	0	1	1

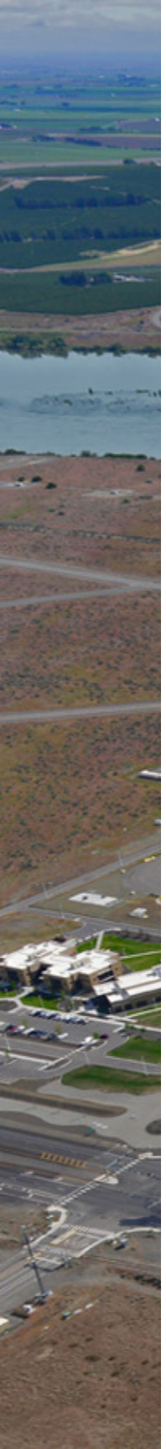
1	1
2	1
3	1
4	1



Pacific Northwest NATIONAL LABORATORY

Computing the Smith Normal Form: Recall the Linear Algebra Operations

```
1  proc swap_rows(i, j, M) {
2    var N = M;
3    N[i, ..] <=> N[j, ..];
4    return N;}
5  proc swap_columns(i, j, M) {
6    var N = M;
7    N[.., i] <=> N[.., j];
8    return N;}
9  proc add_to_row(M, i, j, ri = 1, rj = 1, mod = 2) {
10   var N = M;
11   N[i, ..]  = (ri * N[i, ..] + rj * N[j, ..]) % mod;
12   return N;}
13 proc add_to_column(M, i, j, ci = 1, cj = 1, mod = 2) {
14   var N = M;
15   N[.., i]  = (ci * N[.., i] + cj * N[..,j]) % mod;
16   return N;}
```



Pacific Northwest NATIONAL LABORATORY

Computing the Smith Normal Form: Recall the Linear Algebra Operations (Cont.)

```
1 proc matmultmod (M, N, mod =2) {
2   var C : [M.domain.dim(1), N.domain.dim(2)] int;
3   forall (i,j) in C.domain {
4     C[i,j] = (+ reduce (M[i, M.domain.dim(2)]
5       * N[M.domain.dim(2), j])) % 2;
6   return C;
7 }
```

Each entry in the resultant matrix is computed in parallel

Reduction intent: doing partial multiplications in parallel and then combining the partial sums

Putting it all together for Computing Smith Normal Form with the Basic LA Operations: LbR=S

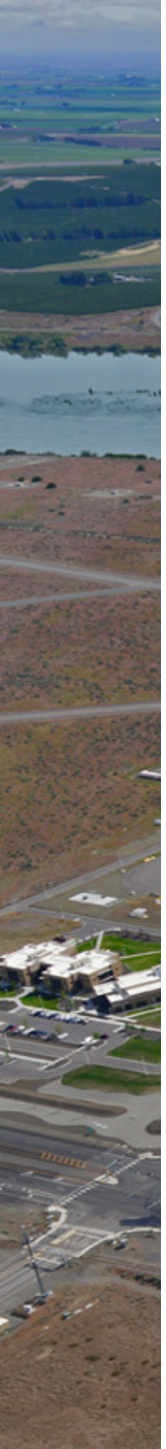
Swapping rows/Columns

```
1  proc smithNormalForm(b) {
2    var IL = IdentityMatrix(dimL);
3    for s in 1..minDim {
4      var pivot = _get_next_pivot(S,s);
5      var rdx : int, cdx : int;
6      (rdx, cdx) = pivot;
7      if (rdx > s) {
8        S = swap_rows(s, rdx, S);
9        L = swap_rows(s, rdx, L);
10       var tmp = swap_rows(s, rdx, IL);
11       var LM = new unmanaged Matrix2D(tmp.domain.high(1),
12                   tmp.domain.high(2));
13       LM._arr = tmp;
14       Linv.append(LM);
15     }
16     if (cdx > s) {
17       S = swap_columns(s, cdx, S);
18       R = swap_columns(s, cdx, R);
19       var tmp = swap_columns(s, cdx, IR);
20       var RM = new unmanaged Matrix2D(tmp.domain.high(1),
21                   tmp.domain.high(2));
22       RM._arr = tmp;
23       Rinv.append(RM);
24     }
```

Potential Bottleneck !!

```
1  proc smithNormalForm(b) {
2  //cont..
3    var row_indices = [idx in 1..dimL]
4           if (idx != s && S(idx,s) == 1) then idx;
5    for rdx in row_indices {
6      S = add_to_row(S, rdx, s);
7      L = add_to_row(L, rdx, s);
8      var tmp = add_to_row(IL, rdx, s);
9      var LM = new unmanaged Matrix2D(tmp.domain.high(1),
10                  tmp.domain.high(2));
11     LM._arr = tmp;
12     Linv.append(LM);
13   }
14   var column_indices = [jdx in 1..dimR]
15          if (jdx != s && S(s,jdx) == 1) then jdx;
16   for (jdx,cdx) in zip(1..,column_indices) {
17     S = add_to_column(S, cdx, s);
18     R = add_to_column(R, cdx, s);
19     var tmp = add_to_column(IR, cdx, s);
20     var RM = new unmanaged Matrix2D(tmp.domain.high(1),
21                  tmp.domain.high(2));
22     RM._arr = tmp;
23     Rinv.append(RM);
24   }
25 }
26 var LinvF = matmulreduce(Linv);
27 var RinvF = matmulreduce(Rinv, true, 2);
28 return (L,R,S,LinvF,RinvF);}
```

Adding to rows/Columns



Pacific Northwest NATIONAL LABORATORY

Opportunities for parallelization

- Straightforward: Compute SNF of all the boundary matrices in parallel.

```
1 var computedMatrices = smithNormalForm(boundaryMaps[1].matrix);
```

- Interesting case: Multiplying a list of matrices in parallel.

```
1  type listType = list(unmanaged Matrix?, true);
2  proc matmulreduce(arr : listType, reverse = false, mod = 2) {
3    var PD = arr[if reverse then arr.size else 1].D;
4    var P : [PD] int;
5    P = arr(1).matrix;
6    for i in 2..arr.size {
7      ref temp = matmultmod(P, arr(i).matrix);
8      PD = temp.domain;
9      P = temp;
10 }}
```

List of matrices to be multiplied

Pairwise matrix multiplication

- Reminder: What are these matrices? All the intermediate transformations done on a boundary matrix
  - Depending on the number of transformation, list-size may vary

Profiling Result

chplvis

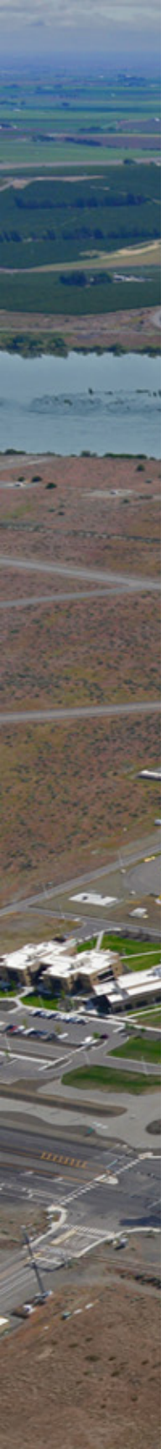
file: SNF

Clock	Internal Function Name	File:line
418.9617	wrapcoforall_fn_chpl154	betti_number_calculator_parallel.chpl:516
31.4294	wrapcobegin_fn31	modules/internal/ChapelArray.chpl:302
30.6619	wrapcobegin_fn34	modules/internal/ChapelArray.chpl:305
27.5633	wrapcoforall_fn_chpl8	modules/dists/BlockDist.chpl:890
27.3298	wrapcoforall_fn_chpl5	modules/dists/BlockDist.chpl:884
26.8153	wrapcoforall_fn_chpl9	modules/dists/BlockDist.chpl:898
15.5744	wrapcoforall_fn_chpl157	betti_number_calculator_parallel.chpl:517
0.0505	wrapcoforall_fn_chpl159	betti_number_calculator_parallel.chpl:547
0.0338	wrapcoforall_fn40	modules/internal/ChapelArray.chpl:4171
0.0228	wrapcoforall_fn30	modules/internal/ChapelArray.chpl:4171

```
proc matmultmod (M, N, mod =2) {
  var CD = {M.domain.dim(1), N.domain.dim(2)} dmapped Block(boundingBox = {M.domain.dim(1), N.domain.dim(2)});
  var C : [CD] int;
  forall (i,j) in C.domain {
    C[i,j] = (+ reduce (M[i, M.domain.dim(2)] * N[M.domain.dim(2), j])) % 2;
  }
  return C;
}
```

- Matrix multiplication is the bottleneck.
- Shared-memory vs distributed 2D array →lots of associated overhead

- In one of the test cases, with 20 vertices and 20 hyperedges, on one compute node/ 1 locale with InfiniBand interconnect and 20-cores, sequential execution took 25s vs distributed-memory 2D version took ~2400s !!



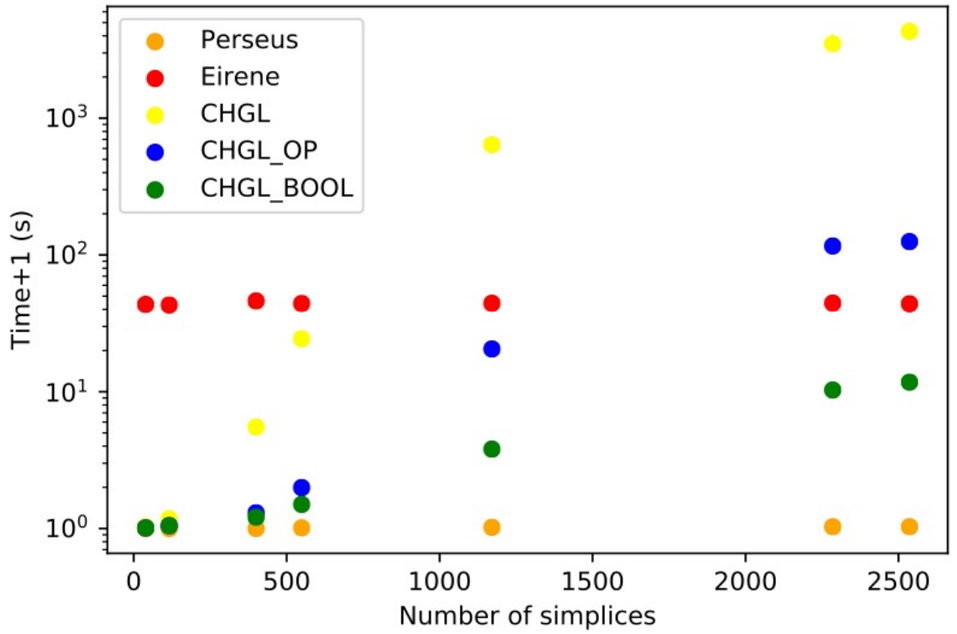
Pacific Northwest NATIONAL LABORATORY

Algorithmic Optimizations

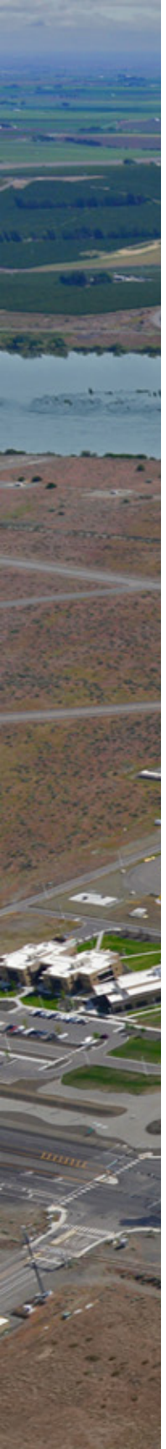
- Optimization 1: Reformulated the calculation of Smith Normal Form
  - Eliminate the requirement of performing explicit matrix-multiplications.
  - Instead, in-place modification of the matrices are made on-the-fly to find the invertible matrices.
- Optimization 2: Since all of our computations are done in the Z_2 field,
  - Opted for boolean datatype and operations for boundary matrices instead of integer matrices.
  - Accordingly, some of the matrix operations changed: for example add_to_row operation on Boolean matrix is elementwise xor.

Experimental Result

- Platform: A single compute node with a 20-core Intel Xeon processor and 132GB memory.
- Chapel version 1.20
- Compared with two other homology packages: Perseus (writtenin C++) and Eirene (written in Julia).



Legends:
CHGL: with explicit matmult
CHGL_OP: No explicit matmult
CHGL_BOOL: Boolean datatype



Pacific Northwest NATIONAL LABORATORY

Conclusion and Future Plan

- Implemented parallel/distributed homology computation in CHGL
  - Computing the boundary matrix
  - Computing the Smith normal form (SNF) of matrices
  - Computing the reduced row Echelon form (Not shown here, similar to SNF)
- Linear algebra operations in the homology computation in Chapel
  - Concise
  - Intuitive
  - Parallel and distributed
- Bottlenecks:
  - Overhead of shared-memory vs block-distributed 2D arrays.
  - Matrix multiplication is one of the main time-consuming kernel
  - Multiplications of list of matrices need to be done in parallel.
- **Moving forward:**
  - Scalable and efficient Parallel/distributed implementation (Ongoing) and scaling test

Pacific Northwest
NATIONAL LABORATORY

Thank you