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)
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
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.
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 $\mathbb{Z}_2$ field, so we wrote our own.
Linear Algebra Operations: Chapel vs Python

**Chapel**

```chapel
proc swap_rows(i,j,M) {
    var N = M;
    N[i, ..] <=> N[j, ..];
    return N;
}
proc add_to_row(M,i,j,ri=1,rj=1) {
    var N = M;
    N[i, ..] = (ri * N[i, ..] + rj * N[j, ..]) % 2;
    return N;
}
proc calculateRank(M) return + reduce
    [i in M.domain.dim(2)] (max reduce M[.., i]);
```

**Python (Numpy/sequential)**

```python
def rowswap(i,j,M):
    N = copy.deepcopy(M)
    N[i] = M[j]
    N[j] = M[i]
    return N

def add_to_row(M,i,j,ri=1,rj=1,mod=2):
    N = copy.deepcopy(M)
    N[i] = np.mod(ri*N[i] + rj*N[j],[mod])
    return N

def calculateRank(M):
    return np.sum(M)
```

Reduction intent: For each column, see if at least one 1, in parallel.
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;
}
```

**Python (Numpy/sequential)**

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

Parallel matrix multiplication
Homology Computation in Chapel
Step 1. Boundary Matrix Computation: What We Will Do

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

  \[ 1 \rightarrow \{ \{ 0 \ 1 \ 2 \ 3 \}\} \]

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

  \[
  \begin{align*}
  &1 \rightarrow \{ \{ 0 \ 1 \ 2 \}, \{ 0 \ 1 \ 3 \}, \{ 0 \ 2 \ 3 \}, \{ 1 \ 2 \ 3 \}\} \\
  &2 \rightarrow \{ \{ 0 \ 1 \}, \{ 0 \ 2 \}, \{ 0 \ 3 \}, \{ 1 \ 2 \}, \{ 1 \ 3 \}, \{ 2 \ 3 \}\} \\
  &0 \rightarrow \{ \{ 0 \}, \{ 1 \}, \{ 2 \}, \{ 3 \}\} \\
  &3 \rightarrow \{ \{ 0 \ 1 \ 2 \ 3 \}\} \\
  &4 \rightarrow \{ \{ 0 \\} \}
  \end{align*}
  \]

- Compute the boundary maps:
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

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

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

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

Adding cells to the domain
Boundary Matrix Computation
Step 1b: Binning all K-cells

Bin cells based on their sizes

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

Thread-safe map

Potential bottleneck: merge is happening on one Locale.

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

Sort each bin in parallel

Customized comparator for sorting
Boundary Matrix Computation: Comparator for Sorting Bins

- Customized comparator for sorting

```plaintext
proc Comparator.compare(a : Cell, b : Cell) : int {
    assert(a.size == b.size);
    for (_, _) in zip(a, b) {
        if (_, _) > (_, _) then return 1;
        else if (_, _) < (_, _) then return -1;
    }
    return 0;
}
```
Boundary Matrix Computation: What We Have Done So Far

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

\[
1 \Rightarrow \{\{0, 1, 2, 3\}\}
\]

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

\[
\begin{align*}
1 & \rightarrow \{\{0, 1, 2\}, \{0, 1, 3\}, \{0, 2, 3\}, \{1, 2, 3\}\} \\
2 & \rightarrow \{\{0, 1\}, \{0, 2\}, \{0, 3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}\} \\
3 & \rightarrow \{\{0\}, \{1\}, \{2\}, \{3\}\} \\
4 & \rightarrow \{\{0, 1, 2, 3\}\}
\end{align*}
\]
Boundary Matrix Data Structure

class Matrix {
  var N : int;
  var M : int;
  var D = {1..N, 1..M} dmapped Block(boundingBox = {1..N, 1..M});
  var matrix : [D] int;
  proc init(_N: int, _M: int) {
    N = _N;
    M = _M;
  }
}
Boundary Matrix Computation

```plaintext
forall (boundaryMap, dimension_k_1, dimension_k) in zip(boundaryMaps, 0..1)
{
    var ACells = kCellsArrayMap[dimension_k].A;
    var BCells = kCellsArrayMap[dimension_k_1].A;

    // Mappings for permutation to index...
    var k1Mapping : map(
        false,
        Cell,
        int);

    for (k1Cell, idx) in zip(kCellMap[dimension_k_1], 1..1)
    {
        k1Mapping[k1Cell] = idx;
    }

   forall (acell, colidx) in zip(ACells, 1..1)
    {
        var perms = splitKCell(acell);

        for bcell in BCells
        {
            for cell in perms
            {
                if bcell = cell
                {
                    boundaryMap[k1Mapping[cell], colidx] = 1;
                    break;
                }
            }
        }
    }
}
```

Each column of a boundary matrix is computed in parallel.

Entry in the boundary matrix.
Boundary Matrix Computation: What we have done so far

• Given the bins:

1. $2 \rightarrow \{\{0, 1, 2\}, \{0, 1, 3\}, \{0, 2, 3\}, \{1, 2, 3\}\}$
2. $1 \rightarrow \{\{0, 1\}, \{0, 2\}, \{0, 3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}\}$
3. $0 \rightarrow \{\{0\}, \{1\}, \{2\}, \{3\}\}$
4. $3 \rightarrow \{\{0, 1, 2, 3\}\}$

• Computed the boundary maps:

$$
\begin{array}{cccccc}
1 & 1 & 1 & 0 & 0 & 0 \\
2 & 1 & 0 & 0 & 1 & 1 \\
3 & 0 & 1 & 0 & 1 & 0 \\
4 & 0 & 0 & 1 & 0 & 1 \\
\end{array}
$$

$$
\begin{array}{cccccc}
1 & 1 & 1 & 0 & 0 & 0 \\
2 & 1 & 0 & 1 & 0 & 0 \\
3 & 0 & 1 & 1 & 0 & 0 \\
4 & 1 & 0 & 0 & 1 & 0 \\
5 & 0 & 1 & 0 & 1 & 0 \\
6 & 0 & 0 & 1 & 1 & 1 \\
\end{array}
$$
Computing the Smith Normal Form: Recall the Linear Algebra Operations

```plaintext
proc swap_rows(i, j, M) {
  var N = M;
  N[i, ..] <=> N[j, ..];
  return N;
}
proc swap_columns(i, j, M) {
  var N = M;
  N[.., i] <=> N[.., j];
  return N;
}
proc add_to_row(M, i, j, ri = 1, rj = 1, mod = 2) {
  var N = M;
  N[i, ..] = (ri * N[i, ..] + rj * N[j, ..]) % mod;
  return N;
}
proc add_to_column(M, i, j, ci = 1, cj = 1, mod = 2) {
  var N = M;
  N[.., i] = (ci * N[.., i] + cj * N[..,j]) % mod;
  return N;
}
```
Computing the Smith Normal Form: Recall the Linear Algebra Operations (Cont.)

```plaintext
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;
}
```

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

```plaintext
proc smithNormalForm(b) {
    var IL = IdentityMatrix(dimL);
    for s in 1..minDim {
        var pivot = _get_next_pivot(S,s);
        var rdx : int, cdx : int;
        (rdx, cdx) = pivot;
        if (rdx > s) {
            S = swap_rows(s, rdx, S);
            L = swap_rows(s, rdx, L);
            var tmp = swap_rows(s, rdx, IL);
            var LM = new unmanaged Matrix2D(tmp.domain.high(1),
                                            tmp.domain.high(2));
            LM._arr = tmp;
            Linv.append(LM);
        }
        if (cdx > s) {
            S = swap_columns(s, cdx, S);
            R = swap_columns(s, cdx, R);
            var tmp = swap_columns(s, cdx, IR);
            var RM = new unmanaged Matrix2D(tmp.domain.high(1),
                                            tmp.domain.high(2));
            RM._arr = tmp;
            Rinv.append(RM);
        }
    }
    var row_indices = [idx in 1..dimL] if (idx != s && S(idx,s) == 1) then idx;
    for rdx in row_indices {
        S = add_to_row(S, rdx, s);
        L = add_to_row(L, rdx, s);
        var tmp = add_to_row(IL, rdx, s);
        var LM = new unmanaged Matrix2D(tmp.domain.high(1),
                                        tmp.domain.high(2));
        LM._arr = tmp;
        Linv.append(LM);
    }
    var column_indices = [jdx in 1..dimR] if (jdx != s && S(s,jdx) == 1) then jdx;
    for (jdx,cdx) in zip(1..column_indices) {
        S = add_to_column(S, cdx, s);
        R = add_to_column(R, cdx, s);
        var tmp = add_to_column(IR, cdx, s);
        var RM = new unmanaged Matrix2D(tmp.domain.high(1),
                                        tmp.domain.high(2));
        RM._arr = tmp;
        Rinv.append(RM);
    }
    var LinvF = matmulreduce(Linv);
    var RinvF = matmulreduce(Rinv, true, 2);
    return (L,R,S,LinvF,RinvF);
}
```

Potential Bottleneck!!
Opportunities for parallelization

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

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

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

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

• 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

- 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!!
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 $\mathbb{Z}_2$ 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 (written in C++) and Eirene (written in Julia).

![Graph showing performance comparison]

Legends:
- CHGL: with explicit matmult
- CHGL_OP: No explicit matmult
- CHGL_BOOL: Boolean datatype
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
Thank you