PaToH: **Pa**rtitioning **To**ol for **H**ypergraphs

Ümit V. Çatalyürek
Department of Biomedical Informatics
The Ohio State University
Columbus, OH 43210
umit@gatech.edu

Cevdet Aykanat
Computer Engineering Department
Bilkent University
Ankara, 06533 Turkey
aykanat@cs.bilkent.edu.tr

November, 1999
Revised: June, 2009
Revised: May 2010
Revised: March 2011

For additional information and documents on PaToH
http://cc.gatech.edu/~umit/software.html

Contents

1 Introduction

Hypergraph partitioning has been an important problem widely encountered in VLSI layout design [?]. Recent works have introduced new application areas, including one-dimensional and two-dimensional partitioning of sparse matrices for parallel sparse-matrix vector multiplication [?, ?, ?, ?, ?, ?], sparse matrix reordering [?, ?], permuting sparse rectangular matrices into singly-bordered block-diagonal form for parallel solution of LP problems [?]. The hypergraph partitioning problem can be defined as the task of dividing a hypergraph into two or more roughly equal sized parts such that a cost function on the hyperedges connecting vertices in different parts is minimized.

Kernighan-Lin (KL) based heuristics are widely used for graph/hypergraph partitioning because of their short run-times and good quality results. KL algorithm is an iterative improvement heuristic originally proposed for bipartitioning [?]. This algorithm became the basis for most of the subsequent partitioning algorithms. KL algorithm, starting from an initial bipartition, performs a number of passes until it finds a locally minimum partition. Each pass consists of a sequence of vertex swaps. The same swap strategy was applied to hypergraph partitioning problem by Schweikert-Kernighan [?]. Fiduccia-Mattheyses (FM) [?] introduced a faster implementation of KL algorithm for hypergraph partitioning. They proposed vertex move concept instead of vertex swap. This modification as well as proper data structures, e.g., bucket lists, reduced the time complexity of a single pass of KL algorithm to linear in the size of the graph and the hypergraph. Here, *size* refers to the number of edges and pins in a graph and hypergraph, respectively. Krishnamurthy [?] added to FM algorithm a look-ahead ability, which helps to break ties better in selecting a vertex to move. In FM-based algorithms, a vertex is locked as soon as it is moved in a pass, and it remains locked until the end of the pass. Hoffman [?], and Dasdan and Aykanat [?] introduced the dynamic locking approach to relax this restrictive locking mechanism.

The performance of KLFM algorithms deteriorates for large and too sparse graphs/hypergraphs. Here, sparsity of graphs and hypergraphs refer to their average vertex degrees. Furthermore, the solution quality of FM is not *stable* (*predictable*), i.e., average FM solution is significantly worse than the best FM solution, which is a common weakness of move-based iterative improvement approaches. Random multi-start approach is used in VLSI layout design to alleviate this problem by running FM algorithm many times starting from random initial partitions to return the best solution found [?]. However, this approach may not be viable in other applications because of high partitioning overhead. Most users will rely on one run of the partitioning heuristic, so that the quality of the partitioning tool depends equally on the worst and average partitionings than on just the best partitioning.

These considerations have motivated the *two–level* application of FM in hypergraph partitioning. In this approach, a clustering is performed on the original hypergraph $\mathcal{H}_0$ to induce a coarser hypergraph $\mathcal{H}_1$. Clustering corresponds to coalescing highly interacting vertices to supernodes as a preprocessing to FM. Then, FM is run on $\mathcal{H}_1$ to find a bipartition Π_1, and this bipartition is projected back to a bipartition Π_0 of $\mathcal{H}_0$. Finally, FM is re-run on $\mathcal{H}_0$ using Π_0 as an initial solution. Recently, the two–level approach has been extended to *multilevel* approaches [?, ?, ?] leading to fast and successful graph partitioning tools Chaco [?], MeTiS [?], WGPP [?] and reordering tools BEND [?], oMeTiS [?], and ordering code of WGPP [?]. We exploit the successful multilevel methodology to develop a new multilevel hypergraph partitioning tool, called PaToH (PaToH: **Pa**rtitioning **To**ols for **H**ypergraphs).

2 Preliminaries

2.1 Hypergraph Partitioning

A hypergraph $\mathcal{H}=(\mathcal{V},\mathcal{N})$ is defined as a set of vertices $\mathcal{V}$ and a set of nets (hyperedges) $\mathcal{N}$ among those vertices. Every net $n \in \mathcal{N}$ is a subset of vertices, i.e., $n \subseteq \mathcal{V}$. The vertices in a net n are called its *pins* and denoted as $pins[n]$. The size of a net is equal to the number of its pins, i.e., $s_n = |pins[n]|$. The set of nets connected to a vertex v is denoted as $nets[v]$. The degree of a vertex is equal to the number of nets it is connected to, i.e., $d_v = |nets[v]|$. Graph is a special instance of hypergraph such that each net has exactly two pins. Weights and costs can be respectively associated with vertices and nets of a hypergraphs. Let $w[v]$ and $c[v]$ denote the weight of vertex $v \in \mathcal{V}$ and the cost of net $n \in \mathcal{N}$.

$\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ is a *K-way partition* of $\mathcal{H}$ if the following conditions hold:

- each part $\mathcal{V}_k$ is a nonempty subset of $\mathcal{V}$, i.e., $\mathcal{V}_k \subseteq \mathcal{V}$ and $\mathcal{V}_k \neq \emptyset$ for $1 \leq k \leq K$,
- parts are pairwise disjoint, i.e., $\mathcal{V}_k \cap \mathcal{V}_\ell = \emptyset$ for all $1 \leq k < \ell \leq K$
- union of K parts is equal to $\mathcal{V}$, i.e., $\bigcup_{k=1}^{K} \mathcal{V}_k = \mathcal{V}$.

In a partition Π of $\mathcal{H}$, a net that has at least one pin (vertex) in a part is said to *connect* that part. *Connectivity set* Λ_n of a net n is defined as the set of parts connected by n. *Connectivity* $\lambda_n = |\Lambda_n|$ of a net n denotes the number of parts connected by n. A net n is said to be *cut* if it connects more than one part (i.e. $\lambda_n > 1$), and *uncut* otherwise (i.e. $\lambda_n = 1$). The cut and uncut nets are also referred to as *external* and *internal* nets, respectively. In a partition Π of $\mathcal{H}$, a vertex is said to be a *boundary* vertex if it is incident to a cut net. A K-way partition is also called a *multiway* partition if $K > 2$ and a *bipartition* if $K = 2$. A partition is said to be balanced if each part $\mathcal{V}_k$ satisfies the *balance criterion*

$$W_k \leq W_{avg}(1+\varepsilon), \quad for\ k = 1, 2, \ldots, K. \tag{1}$$

In (**??**), weight W_k of a part $\mathcal{V}_k$ is defined as the sum of the weights of the vertices in that part, i.e.,

$$W_k = \sum_{v \in \mathcal{V}_k} w[v], \tag{2}$$

W_{avg} denotes the weight of each part under the perfect load balance condition, i.e.,

$$W_{avg} = (\sum_{v \in \mathcal{V}} w[v])/K, \tag{3}$$

and ε represents the predetermined maximum imbalance ratio allowed.

The set of external nets of a partition Π is denoted as $\mathcal{N}_E$. There are various [**?**, **?**] *cutsize* definitions for representing the cost $\chi(\Pi)$ of a partition Π. Two relevant definitions are:

$$(a) \quad \chi(\Pi) = \sum_{n \in \mathcal{N}_E} c[n] \qquad and \qquad (b) \quad \chi(\Pi) = \sum_{n \in \mathcal{N}_E} c[n](\lambda_n - 1). \tag{4}$$

In (**??**.a), the cutsize is equal to the sum of the costs of the cut nets. In (**??**.b), each cut net n contributes $c[n](\lambda_n - 1)$ to the cutsize. The cutsize metrics given in (**??**.a) and (**??**.b) will be

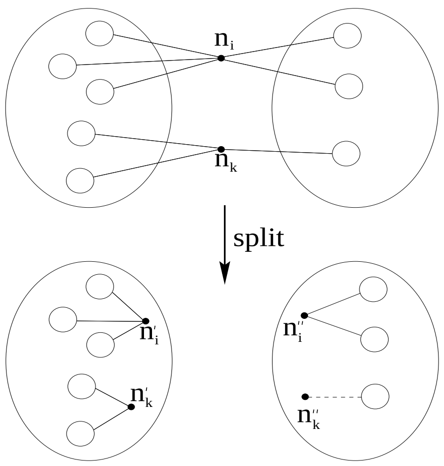

Figure 1: Cut-net splitting during recursive bisection.

referred to here as *cut-net* and *connectivity* metrics, respectively. The hypergraph partitioning problem can be defined as the task of dividing a hypergraph into two or more parts such that the cutsize is minimized, while a given balance criterion (**??**) among part weights is maintained. The hypergraph partitioning problem is known to be NP-hard [**?**].

2.2 Recursive Bisection

The K-way graph/hypergraph partitioning problem is usually solved by recursive bisection. In this scheme, first a 2-way partition of $\mathcal{H}$ is obtained, and then this bipartition is further partitioned in a recursive manner. After $\lg_2 K$ phases, hypergraph $\mathcal{H}$ is partitioned into K parts. PaToH achieves K-way hypergraph partitioning by recursive bisection for any K value. That is, K is not restricted to be a power of 2.

The cutsize metrics given in (**??**) need special attention in K-way hypergraph partitioning by recursive bisection. Note that these two metrics become equivalent in hypergraph bisection. Consider a bipartition $\mathcal{V}_\mathcal{A}$ and $\mathcal{V}_\mathcal{B}$ of $\mathcal{V}$ obtained after a bisection step. It is clear that $\mathcal{V}_\mathcal{A}$ and $\mathcal{V}_\mathcal{B}$ and the internal nets of parts $\mathcal{A}$ and $\mathcal{B}$ will become the vertex and net sets of $\mathcal{H}_\mathcal{A}$ and $\mathcal{H}_\mathcal{B}$, respectively, for the following recursive bisection steps. Note that each cut net of this bipartition already contributes 1 (assuming unit cost nets) to the total cutsize of the final K-way partition to be obtained by further recursive bisections. Since each cut net will remain to be a cut net in the final K-way partition, all cut nets of this bipartition are discarded in the cut-net metric (**??**.a). However, in the connectivity metric (**??**.b), the further recursive bisections of $\mathcal{V}_\mathcal{A}$ and $\mathcal{V}_\mathcal{B}$ may increase the connectivity of these cut nets. Hence, after every hypergraph bisection step, each cut net n_i is split into two pin-wise disjoint nets $n' = pins[n] \bigcap \mathcal{V}_\mathcal{A}$ and $n'' = pins[n] \bigcap \mathcal{V}_\mathcal{B}$, and then these two nets are added to the net lists of $\mathcal{H}_\mathcal{A}$ and $\mathcal{H}_\mathcal{B}$ if $|n'| > 1$ and $|n''| > 1$, respectively. Note that the single-pin nets are discarded during the split operation since such nets cannot contribute to the cutsize in the following recursive bisection steps. Thus, the total cutsize according to (**??**.b) will become equal to the sum of the number of cut nets at every bisection step by using the above cut-net split method. Figure **??** illustrates two cut nets n_i and n_k in a bipartition, and their splits into nets n'_i, n''_i and n'_k, n''_k, respectively. Note that net n''_k becomes a single-pin net and it is discarded.

3 Multilevel Hypergraph Bisection

The multilevel hypergraph bisection algorithm used in PaToH consists of three phases: *coarsening*, *initial partitioning*, and *uncoarsening*. In the first phase, a bottom-up multilevel clustering is successively applied starting from the original graph by adopting various heuristics until number of vertices in the coarsened graph reduces below a predetermined threshold value. In the second phase, the coarsest graph is bipartitioned using various bottom-up heuristics. In the third phase, partition found in the second phase is successively projected back towards the original graph by refining the projected partitions on intermediate level uncoarser graphs using various top-down iterative improvement heuristics. The following sections briefly summarize these three phases. Although PaToH works on weighted nets, we will assume unit cost nets both for the sake of simplicity of presentation.

3.1 Coarsening Phase

In this phase, the given hypergraph $\mathcal{H} = \mathcal{H}_0 = (\mathcal{V}_0, \mathcal{N}_0)$ is coarsened into a sequence of smaller hypergraphs $\mathcal{H}_1 = (\mathcal{V}_1, \mathcal{N}_1)$, $\mathcal{H}_2 = (\mathcal{V}_2, \mathcal{N}_2)$, $\ldots$, $\mathcal{H}_m = (\mathcal{V}_m, \mathcal{N}_m)$ satisfying $|\mathcal{V}_0| > |\mathcal{V}_1| > |\mathcal{V}_2| > \ldots > |\mathcal{V}_m|$. This coarsening is achieved by coalescing disjoint subsets of vertices of hypergraph $\mathcal{H}_i$ into *multinodes* such that each multinode in $\mathcal{H}_i$ forms a single vertex of $\mathcal{H}_{i+1}$. The weight of each vertex of $\mathcal{H}_{i+1}$ becomes equal to the sum of its constituent vertices of the respective multinode in $\mathcal{H}_i$. The net set of each vertex of $\mathcal{H}_{i+1}$ becomes equal to the union of the net sets of the constituent vertices of the respective multinode in $\mathcal{H}_i$. Here, multiple pins of a net $n \in \mathcal{N}_i$ in a multinode cluster of $\mathcal{H}_i$ are contracted to a single pin of the respective net $n' \in \mathcal{N}_{i+1}$ of $\mathcal{H}_{i+1}$. Furthermore, the single-pin nets obtained during this contraction are discarded. Note that such single-pin nets correspond to the internal nets of the clustering performed on $\mathcal{H}_i$. The coarsening phase terminates when the number of vertices in the coarsened hypergraph reduces below pre-determined number.

Clustering approaches can be classified as *agglomerative* and *hierarchical*. In the agglomerative clustering, new clusters are formed one at a time, whereas in the hierarchical clustering several new clusters may be formed simultaneously. In PaToH, we have implemented both randomized matching–based hierarchical clustering schemes and randomized hierarchic–agglomerative clustering schemes . The former and latter approaches will be abbreviated as matching–based clustering and agglomerative clustering, respectively.

The matching-based clustering works as follows. Vertices of $\mathcal{H}_i$ are visited in a random order. If a vertex $u \in \mathcal{V}_i$ has not been matched yet, one of its unmatched *adjacent* vertices is selected according to a criterion. If such a vertex v exists, we merge the matched pair u and v into a cluster. If there is no unmatched adjacent vertex of u, then vertex u remains unmatched, i.e., u remains as a singleton cluster. Here, two vertices u and v are said to be adjacent if they share at least one net, i.e., $nets[u] \cap nets[v] \neq \emptyset$.

The matching-based clustering allows the clustering of only pairs of vertices in a level. In order to enable the clustering of more than two vertices at each level, we have implemented a randomized agglomerative clustering approach. In this scheme, each vertex u is assumed to constitute a singleton cluster $C_u = \{u\}$ at the beginning of each coarsening level. Then, vertices are visited in a random order. If a vertex u has already been clustered (i.e. $|C_u| > 1$) it is not considered for being the source of a new clustering. However, an unclustered vertex u can choose to join a multinode cluster as well as a singleton cluster. That is, all adjacent vertices of an unclustered vertex u are considered for selection according to a criterion. The selection of a vertex v adjacent to u corresponds to including vertex u to cluster C_v to grow a new multinode cluster $C_u = C_v = C_v \cup \{u\}$. Note that no singleton cluster remains at the end of this process as far as there exists no isolated vertex.

3.2 Initial Partitioning Phase

The goal in this phase is to find a bipartition on the coarsest hypergraph $\mathcal{H}_m$. In PaToH, we use *Greedy Hypergraph Growing (GHG)* algorithm for bisecting $\mathcal{H}_m$. This algorithm can be considered as an extension of the GGGP algorithm used in MeTiS to hypergraphs. In GHG, we grow a cluster around a randomly selected vertex. During the coarse of the algorithm, the selected and unselected vertices induce a bipartition on $\mathcal{H}_m$. The unselected vertices connected to the growing cluster are inserted into a priority queue according to their FM gains. Here, the gain of an unselected vertex corresponds to the decrease in the cutsize of the current bipartition if the vertex moves to the growing cluster. Then, a vertex with the highest gain is selected from the priority queue. After a vertex moves to the growing cluster, the gains of its unselected adjacent vertices which are currently in the priority queue are updated and those not in the priority queue are inserted. This cluster growing operation continues until a predetermined bipartition balance criterion is reached. As also mentioned in MeTiS, the quality of this algorithm is sensitive to the choice of the initial random vertex. Since the coarsest hypergraph $\mathcal{H}_m$ is small, we run GHG multiple times starting from different random vertices and select the best bipartition for refinement during the uncoarsening phase.

3.3 Uncoarsening Phase

At each level i (for $i = m, m{-}1, \ldots, 1$), bipartition Π_i found on $\mathcal{H}_i$ is projected back to a bipartition Π_{i-1} on $\mathcal{H}_{i-1}$. The constituent vertices of each multinode in $\mathcal{H}_{i-1}$ is assigned to the part of the respective vertex in $\mathcal{H}_i$. Obviously, Π_{i-1} of $\mathcal{H}_{i-1}$ has the same cutsize with Π_i of $\mathcal{H}_i$. Then, we refine this bipartition by running a KLFM-based iterative improvement heuristics on $\mathcal{H}_{i-1}$ starting from initial bipartition Π_{i-1}. PaToH involves a wide range of KLFM-based refinement implementations as listed in Section **??**. Here, we will only discuss the details of our Boundary FM (BFM) implementation. BFM is an FM algorithm that moves only the boundary vertices from the overloaded part to the under-loaded part, where a vertex is said to be a boundary vertex if it is connected to an at least one cut net.

BFM requires maintaining the *pin-connectivity* of each net for both initial gain computations and gain updates. The pin-connectivity $\sigma_k[n] = |n \cap \mathcal{P}_k|$ of a net n to a part $\mathcal{P}_k$ denotes the number of pins of net n that lie in part $\mathcal{P}_k$, for $k = 1, 2$. In order to avoid the scan of the pin lists of all nets, we adopt an efficient scheme to initialize the σ values for the first BFM pass in a level. It is clear that initial bipartition Π_{i-1} of $\mathcal{H}_{i-1}$ has the same cut-net set with Π_i of $\mathcal{H}_i$. Hence, we scan only the pin lists of the cut nets of Π_{i-1} to initialize their σ values. For each other net n, $\sigma_1[n]$ and $\sigma_2[n]$ values are easily initialized as $\sigma_1[n] = s_n$ and $\sigma_2[n] = 0$ if net n is internal to part $\mathcal{P}_1$, and $\sigma_1[n] = 0$ and $\sigma_2[n] = s_n$ otherwise. After initializing the gain value of each vertex v as $g[v] = -d_v$, we exploit σ values as follows. We re-scan the pin list of each external net n and update the gain value of each vertex $v \in pins[n]$ as $g[v] = g[v] + 2$ or $g[v] = g[v] + 1$ depending on whether net n is *critical* to the part containing v or not, respectively. An external net n is said to be critical to a part k if $\sigma_k[n] = 1$ so that moving the single vertex of net n that lies in that part to the other part removes net n from the cut. Note that two-pin cut nets are critical to both parts. The vertices visited while scanning the pin-lists of the external nets are identified as boundary vertices and only these vertices are inserted into the priority queue according to their computed gains.

In each pass of the BFM algorithm, a sequence of unmoved vertices with the highest gains are selected to move to the other part. As in the original FM algorithm, a vertex move necessitates gain updates of its adjacent vertices. However, in the BFM algorithm, some of the adjacent vertices of the moved vertex may not be in the priority queue, because they may not be boundary vertices before the move. Hence, such vertices which become boundary vertices after the move are inserted into the priority queue according to their updated gain values. The refinement process within a pass terminates either no *feasible* move remains or the sequence of last $max\{50,\ 0.001|\mathcal{V}_i|\}$ moves does not yield a decrease in the total cutsize. A move is said to be feasible if it does not disturb the load balance criterion (**??**) with $K=2$. At the end of a BFM pass, we have a sequence of tentative vertex moves and their respective gains. We then construct from this sequence the maximum prefix subsequence of moves with the maximum prefix sum which incurs the maximum decrease in the cutsize. The permanent realization of the moves in this maximum prefix subsequence is efficiently achieved by rolling back the remaining moves at the end of the overall sequence. The initial gain computations for the following pass in a level is achieved through this rollback. The overall refinement process in a level terminates if the maximum prefix sum of a pass is not positive.

4 Library Interface

PaToH v3.2 library interface consists of two files; a header file `patoh.h` which contains constants, structure definitions and functions proto-types, and a library file `libpatoh.a`. The hypergraph representation used by the library interface is described in Section **??**, then detail description of the functions are presented in Section **??**. The parameter structure that is used by the PaToH's recursive multilevel hypergraph partitioner is discussed in the Section **??**.

Before starting to discuss the details, lets look at a simple C program that partitions an input hypergraph using PaToH functions. The program is displayed in Figure **??**. First statement is a function call to read the input hypergraph file which is given by the first command line argument. PaToH partition functions is customizable through a set of arguments, Although user (programmer) can set each of these arguments one by one, it is a good habit to call PaToH function `PaToH_Initialize_Parameters` to set all parameters to default values. After this call, user may prefer to modify the parameters according to his/her need before calling `PaToH_Alloc`. All memory that will be used by PaToH partitioning functions is allocated by `PaToH_Alloc` function, that is, there will be no more dynamic memory allocation inside the partitioning functions. Now, we are ready to partition the hypergraph using PaToH's multilevel hypergraph partitioning functions. Call to `PaToH_Part` will partition the hypergraph and resulting partition vector, part weights and cutsize will be returned in the parameters. Here, variable `cut` will hold the cutsize of the computed partition according to cutsize definition **??**(b) since we requested to use this metric by initializing the parameters with constant `PATOH_CONPART`. User may call partitioning functions as many times as he/she wants before calling function `PaToH_Free`. There is no need to re-allocate the memory before each partitioning call, unless hypergraph is changed. However, changing the coarsening algorithm and number of parts may also require a re-allocation.

```
#include <stdio.h>
#include "patoh.h"

int main(int argc, char *argv[])
{
PaToH_Parameters args;
int             _c, _n, _nconst, *cwghts, *nwghts,
                *xpins, *pins, *partvec, cut, *partweights;


PaToH_Read_Hypergraph(argv[1], &_c, &_n, &_nconst, &cwghts, &nwghts,
                      &xpins, &pins);

printf("Hypergraph %10s -- #Cells=%6d  #Nets=%6d  #Pins=%8d #Const=%2d\n",
       argv[1], _c, _n, xpins[_n], _nconst);

PaToH_Initialize_Parameters(&args, PATOH_CONPART, PATOH_SUGPARAM_DEFAULT);

args._k = atoi(argv[2]);
partvec =  (int *) malloc(_c*sizeof(int));
partweights  = (int *) malloc(args._k*_nconst*sizeof(int));

PaToH_Alloc(&args, _c, _n, _nconst, cwghts, nwghts, xpins, pins);

PaToH_Part(&args, _c, _n, _nconst, 0, cwghts, nwghts,
           xpins, pins, NULL, partvec, partweights, &cut);

printf("%d-way cutsize is: %d\n", args._k, cut);

free(cwghts);      free(nwghts);
free(xpins);       free(pins);
free(partweights); free(partvec);

PaToH_Free();
return 0;
}
```

Figure 2: A simple C program that partitions an input hypergraph using PaToH functions

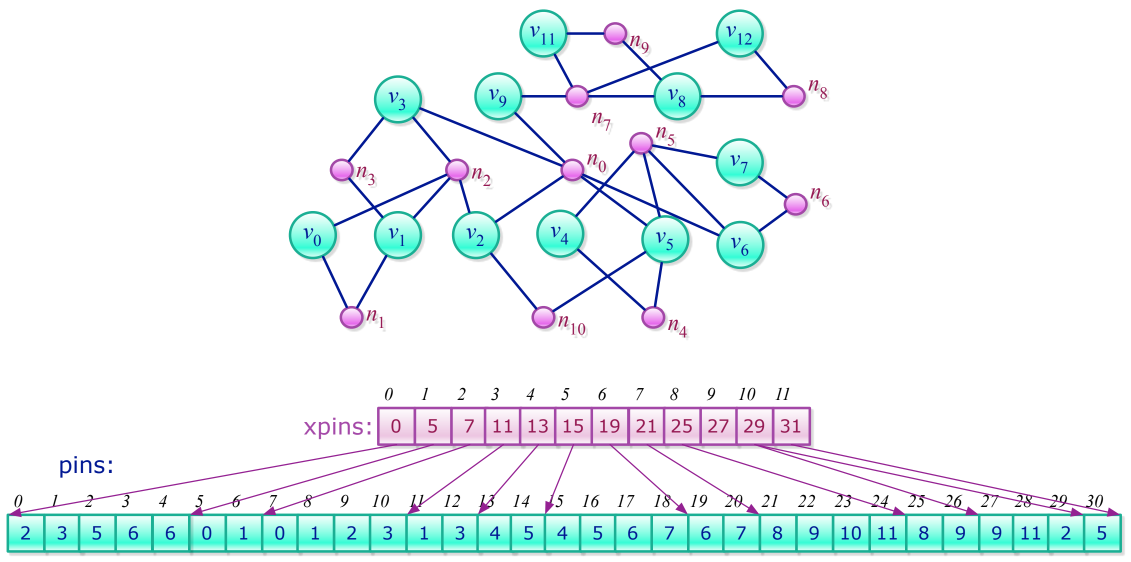

Figure 3: A hypergraph and its representation.

4.1 Hypergraph Representation

A hypergraph and its representation can be seen in Figure **??**. In the figure, large white circles are cells (vertices) of the hypergraph, and small black circles are nets. `xpins` and `pins` arrays stores, the beginning index of "pins" (cells) connected to each net, and IDs of the pins, respectively. Hence, `xpins` is an array of size number of nets plus one, and `pins` is an array of size number of pins in the hypergraph. Cells connected to net n_j are stored in `pins[xpins[j]]` through `pins[xpins[j+1]-1]`.

4.2 Functions

Current PaToH interface contains three function categories; initialization and memory functions, partitioning functions, and utility functions. Following subsections present the detailed descriptions of the functions of each category.

4.2.1 Initialization and memory functions

```
int PaToH_Initialize_Parameters(PPaToH_Parameters pargs, int cuttype,
                                int SBProbType);
```

Description:
Initializes the parameters that will be used in the partitioning to some default values according to `SBProbType` (SuggestByProblemType) parameter.

Parameters:

`pargs`	output	pointer to parameters structure described in Section **??**. The structure that is pointed by this argument will be filled by this function
`cuttype`	input	must be either `PATOH_CUTPART` for cutnet metric (Equation **??**(a)) or `PATOH_CONPART` for "Connectivity-1" metric (Equation **??**(b)).
`SBProbType`	input	Must be set to one of • `PATOH_SUGPARAM_DEFAULT`: sets the parameters to default values. • `PATOH_SUGPARAM_SPEED`: if you need faster partitionings use this. For most of the matrix partitioning problems, we observed that this setting will produce reasonably good results much faster than the default value. • `PATOH_SUGPARAM_QUALITY`: if you could afford a little bit more extra time for a little better quality (such as VLSI partitioning), use this value.

```
int PaToH_Alloc(PPaToH_Parameters pargs, int _c, int _n, int _nconst,
                int *cwghts, int *nwghts, int *xpins, int *pins);
```

Description:
Allocates the memory that will be used by partitioning algorithms.

Parameters:

`pargs`	input	pointer to parameters structure described in Section **??**. Allocation will be done using some of the parameters of this structure.
`_c`	input	number of cells of the hypergraph.
`_n`	input	number of nets of the hypergraph.
`_nconst`	input	number of constraints.
`cwghts`	input	array of size `_c`×`_nconst` that stores the weights of each cell. In multi-constraint partitioning, each cell v_i has `_nconst` weights, and they are stored in `cwghts[i*_nconst]` through `cwghts[(i+1)*_nconst-1]`
`nwghts`	input	array of size `_n` that stores the cost of each net. If hypergraph has unweighted nets, this parameter can be `NULL`.
`xpins`	input	array of size `_n+1` that stores the beginning index of cells connected to nets.
`pins`	input	array that stores the pin-lists (cell-list) of nets. Cells connected to net n_j are stored in `pins[xpins[j]]` through `pins[xpins[j+1]-1]`.

```
int PaToH_Free(void);
```

Description:
Frees the memory allocated by `PaToH_Alloc`.

4.2.2 Partitioning functions

```
int PaToH_Part(PPaToH_Parameters pargs, int _c, int _n, int _nconst, int useFixCells,
               int *cwghts, int *nwghts, int *xpins, int *pins, float *targetweights,
               int *partvec, int *partweights, int *cut);
```

Description:
Unified interface for regular, multi-constraint and fix-vertex hypergraph partitioning methods. Partitions the hypergraph into `pargs->_k` parts using recursive multilevel hypergraph bisection algorithm. If argument `useFixCells` is nonzero, some of the cells of the hypergraph may have been pre-assigned (fixed to a part). If `_nconst`$>$ 1, multi-constraint partitioning is used. Please note that multi-constraint partitioning does not work fixed cells.

Parameters:

`pargs`	input	pointer to parameters structure described in Section ??. Partitioning will use the parameters of this structure.
`_c`	input	number of cells of the hypergraph.
`_n`	input	number of nets of the hypergraph.
`_nconst`	input	number of constraints.
`useFixCells`	input	If non-zero, partitioning is done using pre-assigned cells.
`cwghts`	input	array of size `_c`$\times$`_nconst` that stores the weights of each cell. In multi-constraint partitioning, each cell v_i has `_nconst` weights, and they are store in `cwghts[i*_nconst]` through `cwghts[(i+1)*_nconst-1]`.
`nwghts`	input	array of size `_n` that stores the cost of each net.
`xpins`	input	array of size `_n+1` that stores the beginning index of pins (cells) connected to nets.
`pins`	input	array that stores the pin-lists of nets. Cells connected to net n_j are stored in `pins[xpins[j]]` through `pins[xpins[j+1]-1]`.
`partvec`	in/out	array of size `_c` that stores the part number of each cell belong to. If `useFixCells` is zero, all vertices are assumed free, otherwise this array is interpretted as follows: -1 indicates cell is free (can be assigned any part), 0 to `pargs->_k-1` indicates that cell is pre-assigned (fixed) to that part.
`targetweights`	input	array of size `pargs->_k` (or `pargs->_k`$\times$`_nconst` for multi-constraint partitioning) with target part weights. Alternatively, `targetweights[i]` should store the fraction of total weight that should be assigned to part `i`. For example, for a hypergraph with a total cell weight of 400, if user wishes to have 3-way partitioning with 25% of total weight in parts 1 and 2, and remaining 50% in part 3, he can provide either `targetweights = { 100, 100, 200}` or `targetweights = { 0.25, 0.25, 0.50}`.
`partweights`	output	array of size `pargs->_k`$\times$`_nconst` that returns the total part weight of each part.
`cut`	output	cutsize of the solution, according to the requested cutsize metric by `pargs->cuttype`.

```
int PaToH_Partition(PPaToH_Parameters pargs, int _c, int _n,
                    int *cwghts, int *nwghts, int *xpins, int *pins,
                    int *partvec, int *partweights, int *cut);
```

Description:
Deprecated: Partitions the hypergraph into `pargs->_k` parts using recursive multilevel hypergraph bisection algorithm. Please use `PaToH_Part` instead.

Parameters:

`pargs`	input	pointer to parameters structure described in Section **??**. Partitioning will use the parameters of this structure.
`_c`	input	number of cells of the hypergraph.
`_n`	input	number of nets of the hypergraph.
`cwghts`	input	array of size `_c` that stores the weight of each cell
`nwghts`	input	array of size `_n` that stores the cost of each net.
`xpins`	input	array of size `_n+1` that stores the beginning index of pins (cells) connected to nets.
`pins`	input	array that stores the pin-lists of nets. Cells connected to net n_j are stored in `pins[xpins[j]]` through `pins[xpins[j+1]-1]`.
`partvec`	output	array of size `_c` that returns the part number of each cell.
`partweights`	output	array of size `pargs->_k` that returns the total part weight of each part.
`cut`	output	cutsize of the solution, according to the requested cutsize metric by `pargs->cuttype`.

```
int PaToH_Partition_with_FixCells(PPaToH_Parameters pargs, int _c,
                        int _n, int *cwghts, int *nwghts, int *xpins, int *pins,
                        int *partvec, int *partweights, int *cut);
```

Description:
Deprecated: Partitions the hypergraph into `pargs->_k` parts using recursive multilevel hypergraph bisection algorithm. Some of the cells of the hypergraph may have been pre-assigned (fixed to a part). Please use `PaToH_Part` instead.

Parameters:

`pargs`	input	pointer to parameters structure described in Section **??**. Partitioning will use the parameters of this structure.
`_c`	input	number of cells of the hypergraph.
`_n`	input	number of nets of the hypergraph.
`cwghts`	input	array of size `_c` that stores the weight of each cell.
`nwghts`	input	array of size `_n` that stores the cost of each net.
`xpins`	input	array of size `_n+1` that stores the beginning index of pins (cells) connected to nets.
`pins`	input	array that stores the pin-lists of nets. Cells connected to net n_j are stored in `pins[xpins[j]]` through `pins[xpins[j+1]-1]`.
`partvec`	in/out	array of size `_c` that stores the part number of each cell belong to. -1 indicates cell is free (can be assigned any part), 0 to `pargs->_k-1` indicates that cell is pre-assigned (fixed) to that part.
`partweights`	output	array of size `pargs->_k` that returns the total part weight of each part.
`cut`	output	cutsize of the solution, according to the requested cutsize metric by `pargs->cuttype`.

```
int PaToH_MultiConst_Partition(PPaToH_Parameters pargs, int _c,
                               int _n, int _nconst, int *cwghts, int *xpins,
                               int *pins, int *partvec, int *partweights, int *cut);
```

Description:
Deprecated: Partitions the hypergraph into `pargs->_k` parts using multi-constraint recursive multilevel hypergraph bisection algorithm. Please note that *this call to* multi-constraint partitioning only works with unit net weights. Please use `PaToH_Part` instead.

Parameters:

`pargs`	input	pointer to parameters structure described in Section **??**. Partitioning will use the parameters of this structure.
`_c`	input	number of cells of the hypergraph.
`_n`	input	number of nets of the hypergraph.
`_nconst`	input	number of constraints.
`cwghts`	input	array of size `_c`×`_nconst` that stores the weights of each cell. In multi-constraint partitioning, each cell v_i has `_nconst` weights, and they are store in `cwghts[i*_nconst]` through `cwghts[(i+1)*_nconst-1]`
`xpins`	input	array of size `_n+1` that stores the beginning index of pins (cells) connected to nets.
`pins`	input	array that stores the pin-lists of nets. Cells connected to net n_j are stored in `pins[xpins[j]]` through `pins[xpins[j+1]-1]`.
`partvec`	output	array of size `_c` that stores the part number of each cell belongs to.
`partweights`	output	array of size `pargs->_k`×`_nconst` that returns the total part weight of each part for each constraint.
`cut`	output	cutsize of the solution, according to the requested cutsize metric by `pargs->cuttype`.

```
int PaToH_Refine_Bisec(PPaToH_Parameters pargs, int _c, int _n,
                       int *cwghts, int *nwghts, int *xpins,
                       int *pins, int *partvec, int *partweights,
                       int *cut);
```

Description:
This function given an input bipartition assignments for cells in `partvec` array, refines the bisection using one of the available refinement algorithms in PaToH. This is a single-level refinement function that could be directly used in the development of new (multilevel) partitioning methods.

Parameters:

`pargs`	input	pointer to parameters structure described in Section ??. Refinement will use the parameters of this structure.
`_c`	input	number of cells of the hypergraph.
`_n`	input	number of nets of the hypergraph.
`cwghts`	input	array of size `_c` that stores the weight of each cell
`nwghts`	input	array of size `_n` that stores the cost of each net.
`xpins`	input	array of size `_n+1` that stores the beginning index of pins (cells) connected to nets.
`pins`	input	array that stores the pin-lists of nets. Cells connected to net n_j are stored in `pins[xpins[j]]` through `pins[xpins[j+1]-1]`.
`partvec`	in/out	array of size `_c` that stores the part number of each cell (0 or 1).
`partweights`	output	array of size `pargs->_k` that returns the total part weight of each part.
`cut`	output	cutsize of the solution, according to the requested cutsize metric by `pargs->cuttype`.

4.2.3 Utility functions

```
int PaToH_Check_User_Parameters(PPaToH_Parameters pargs, int verbose);
```

Description:
Verifies some of the user parameters. Returns non-zero value indicating an error in the parameters. Please note that verification is not exhaustive, hence does not cover all possible errors. Shoud be use for simple checking. In most of the problematic cases, during partitioning PaToH tries to re-set/ignore invalid parameter values. If this is not an option, it gives an error message and exits.

Parameters:

`pargs`	input	pointer to parameters structure described in Section **??**.
`verbose`	input	If it is non-zero, PaToH display explanation warnings if it finds a problem in the parameters.

```
int PaToH_Read_Hypergraph(char *filename, int *_c, int *_n,  int *_nconst,
                          int **cwghts, int **nwghts, int **xpins, int **pins);
```

Description:
Reads a hypergraph from the given file. See Section **??** for details of the file format. PaToH allocates `cwghts, nwghts, xpins` and `pins` arrays. User should call `free()` function with those pointers when hypergraph is not needed.

Parameters:

`filename`	input	name of the file that contains the hypergraph.
`_c`	output	number of cells of the hypergraph.
`_n`	output	number of nets of the hypergraph.
`_nconst`	output	number of constraints.
`cwghts`	output	array of size `_c`$\times$`_nconst` that stores the weights of each cell. In multi-constraint partitioning, each cell v_i has `_nconst` weights, and they are store in `cwghts[i*_nconst]` through `cwghts[(i+1)*_nconst-1]`
`nwghts`	output	array of size `_n` that stores the cost of each net.
`xpins`	output	array of size `_n+1` that stores the beginning index of pins (cells) connected to nets.
`pins`	output	array that stores the pin-lists of nets. Cells connected to net n_j are stored in `pins[xpins[j]]` through `pins[xpins[j+1]-1]`.

```
int PaToH_Write_Hypergraph(char *filename, int numbering, int _c, int _n, int _nconst,
                      int *cwghts, int *nwghts, int *xpins, int *pins);
```

Description:
Writes a hypergraph into a file. See Section **??** for details of the file format.

Parameters:

`filename`	input	name of the file that will contain the hypergraph.
`numbering`	input	the index base for the indices, see Section **??**.
`_c`	output	number of cells of the hypergraph.
`_n`	output	number of nets of the hypergraph.
`_nconst`	output	number of constraints.
`cwghts`	output	array of size `_c`×`_nconst` that stores the weights of each cell. In multi-constraint partitioning, each cell v_i has `_nconst` weights, and they are store in `cwghts[i*_nconst]` through `cwghts[(i+1)*_nconst-1]`
`nwghts`	output	array of size `_n` that stores the cost of each net.
`xpins`	output	array of size `_n+1` that stores the beginning index of pins (cells) connected to nets.
`pins`	output	array that stores the pin-lists of nets. Cells connected to net n_j are stored in `pins[xpins[j]]` through `pins[xpins[j+1]-1]`.

```
int PaToH_Compute_Cut(int _k, int cuttype, int _c, int _n, int *nwghts,
                      int *xpins, int *pins, int *partvec);
```

Description:
Given number of parts, cuttype, hypergraph and part vector, computes and returns cut.

Parameters:

`_k`	input	number of parts.
`cuttype`	input	must be either `PATOH_CUTPART` for cutnet metric (Equation **??**(a)) or `PATOH_CONPART` for "Connectivity-1" metric (Equation **??**(b)).
`_c`	input	number of cells of the hypergraph.
`_n`	input	number of nets of the hypergraph.
`nwghts`	input	array of size `_n` that stores the cost of each net.
`xpins`	input	array of size `_n+1` that stores the beginning index of pins (cells) connected to nets.
`pins`	input	array that stores the pin-lists of nets. Cells connected to net n_j are stored in `pins[xpins[j]]` through `pins[xpins[j+1]-1]`.
`partvec`	input	array of size `_c` that returns the part number of each cell.

```
int PaToH_Compute_Part_Weights(int _k, int _c, int _nconst,
                               int *cwghts, int *partvec, int *partweights);
```

Description:
Given number of parts, number of cells, number of constraints, cell weights and a part vector, computes the part weights.

Parameters:

`_k`	input	number of parts.
`_c`	input	number of cells of the hypergraph.
`_nconst`	input	number of constraints.
`cwghts`	input	array of size `_c`×`_nconst` that stores the weights of each cell. In multi-constraint partitioning, each cell v_i has `_nconst` weights, and they are store in `cwghts[i*_nconst]` through `cwghts[(i+1)*_nconst-1]`
`partvec`	input	array of size `_c` that stores the part number of each cell belongs to.
`partweights`	output	array of size `pargs->_k`×`_nconst` that returns the total part weight of each part for each constraint.

4.3 Data Structures

User controls the execution of the multilevel bisection algorithm by setting appropriate parameters. First argument of PaToH partitioning functions is a pointer to a structure of type `PaToH_Parameters`. This structure is defined in file `patoh.h`. We have categorized the parameters in to four groups. Following subsections briefly describes the each parameter.

4.3.1 Miscellaneous Parameters

- `cuttype`: determines the cost function for partitioning. Must be either `PATOH_CUTPART` for cutnet metric (Equation ??(a)) or `PATOH_CONPART` for "Connectivity-1" metric (Equation ??(b)).
- `_k`: number of parts.
- `outputdetail`: detail of verbose output. Use `PATOH_OD_<X>` constants, where `<X>` should be `NONE`, `LOW`, `MEDIUM` and `HIGH`, for none, low, medium and high output detail.
- `seed`: seed of the random generator. Set to -1 for using current time as the seed for random generator, set to 0 for using partitioning count (number of times PaToH partitioners called) as the seed. Set to any other non-zero value to for fixing random generator seed.
- `doinitperm`: if set to a non-zero value, PaToH shuffles the pins and nets lists of the hypergraph prior to partitioning.
- `bisec_fixednetsizetrsh`: During the each bisection nets with size larger than this value will be discarded. Please note that, if such a larger net is split during the recursive bisection it may be considered in the further partitionings.
- `bisec_netsizetrsh`: Nets with size larger than `bisec_netsizetrsh`$\times s_{avg}$ are discarded during the each bisections step, where s_{avg} is the average net size.
- `bisec_partmultnetsizetrsh`: Nets with size larger than `bisec_partmultnetsizetrsh`$\times K$ are discarded during the each bisections step of a K-way partitioning.
- `bigVcycle`: the maximum number of big V-cycles (default is 1 for `PATOH_SUGPARAM_SPEED` and `PATOH_SUGPARAM_DEFAULT`, and higher for `PATOH_SUGPARAM_QUALITY`).
- `smallVcycle`: the maximum number of small V-cycles.
- `usesamematchinginVcycles`: if set to a non-zero value PaToH will use the same coarsening algorithm during the V-cycles. If it is zero, PaToH will automatically selects a coarsening algorithm for each V-cycle.
- `usebucket`: PaToH can use both the heap and the bucket data structures as priority queue. If this parameter is 1, the bucket data structure is always used, if it is 0 the heap data structure is always used, if it is -1 PaToH determines when to use the heap data structure using the parameters `maxcellinheap`, `heapchk_mul`, and `heapchk_div` defined below. However,

since PaToH includes multiple coarsening, initial partitioning and refinement algorithms with different properties and needs, all combinations using both type of priority queue have not been implemented. PaToH may ignore this parameter for some certain algorithms.

- `maxcellinheap`: Heap will not be used if the current hypergraph has more cell than this number.
- `heapchk_mul`: Heap will be used if $bs \times$ `heapchk_mul`/`heapchk_div` $< |\mathcal{V}_i|$ at the level i, where bs is required bucket size.
- `heapchk_div`: Heap will be used if $bs \times$ `heapchk_mul`/`heapchk_div` $< |\mathcal{V}_i|$ at the level i, where bs is required bucket size.
- `MemMul_CellNet`: PaToH allocates three large continues arrays to be able to run Multilevel Partitioning. The first array holds the internal cell and net structures. This parameter tells to PaToH to allocate `MemMul_CellNet` times much memory that is required to hold the cell and net structures of the original hypergraph.
- `MemMul_Pins`: The second large array is used to store net-lists of cells (nets array) and pin-lists of nets (pins array). This parameter tells to PaToH to allocate `MemMul_Pins` times much memory that is required to hold pins and nets arrays of the original hypergraph.
- `MemMul_General`: The last large array is used to store temporary working arrays required during the multilevel partitioning. This parameter tells to PaToH to allocate `MemMul_Pins` times much memory that is required to hold pins array of the original hypergraph.

4.3.2 Coarsening Parameters

- `crs_VisitOrder`: cell visit order for coarsening algorithms
  - `PATOH_VO_CONT=0`: Continuous/Sequential (increasing vertex ID order),
  - `PATOH_VO_RAND=1`: Random (default),
  - `PATOH_VO_SCD=2`: Non-decreasing cell degree order,
  - `PATOH_VO_SMAXNS=3`: Non-decreasing maximum net size order,
  - `PATOH_VO_SMINNS=4`: Non-decreasing minimum net size order,
  - `PATOH_VO_SMINNSSUM=5`: Non-decreasing minimum of net size sum sorted,
  - `PATOH_VO_SWEEP=6`: Sweep: increasing vertex ID order in even levels, decreasing vertex ID order in odd levels.
- `crs_alg`: coarsening algorithm choices:

  In matching-based clustering schemes listed below (first eight), vertex u denotes the unmatched vertex visited according the order determined by the `crs_VisitOrder` parameter. Vertex u is the source of the current matching process and an unmatched vertex v is selected according to a criterion among all unmatched vertices adjacent to u. Recall that two vertices u and v are said to be adjacent if they share at least one net, i.e., $nets[u] \cap nets[v] \neq \emptyset$. Here, $\mathcal{N}_{uv}$ denotes the set of nets shared by vertices u and v, and $N_{uv} = |\mathcal{N}_{uv}|$ denotes the number of nets shared between u and v.

- PATOH_CRS_HCM=1: Heavy connectivity matching. Vertex v has *maximum* connectivity value N_{uv}.
- PATOH_CRS_PHCM=2: Probabilistic heavy connectivity matching.
- PATOH_CRS_MANDIS=3: Manhattan distance. Vertex v has *minimum* Manhattan Distance $M_{uv} = d_u + d_v - 2N_{uv}$.
- PATOH_CRS_AVEDIS=4: Average distance. Vertex v has *minimum* average Manhattan Distance $M_{uv}/(d_u - N_{uv})$.
- PATOH_CRS_CANBERRA=5: Canberra metric. Vertex v has *minimum* $M_{uv}/(d_u + d_v)$ ratio.
- PATOH_CRS_ABS=6: Absorption Matching. Vertex v has *maximum* sum $\sum_{n \in N_{uv}} 1/(s_n - 1)$. This similarity metric favors matching vertex pairs connected via nets of small sizes.
- PATOH_CRS_GCM=7: Greedy Cut Matching. Vertex v has *minimum* $d_u + d_v - N_{uv}$ value.
- PATOH_CRS_SHCM=8: Scaled Heavy Connectivity Matching. Vertex v has *maximum* $N_{uv}/(d_u + d_v - N_{uv})$ ratio.

In agglomerative clustering schemes listed below, vertex u denotes the unclustered vertex visited according the order determined by the `crs_VisitOrder` parameter. Vertex u is the source of the current clustering process and all vertices adjacent to vertex u are considered for selection according to a criterion. The selection of a vertex v adjacent to u corresponds to including vertex u to singleton or multinode cluster C_v that contains vertex v to grow a new multinode cluster $C_{uv} = \{u\} \cup C_v$.

- PATOH_CRS_HCC=9: Heavy Connectivity Clustering. This metric is the agglomerative version of PATOH_CRS_HCM. That is, v has maximum N_{u,C_v}, which denotes the number of nets shared between vertex u and cluster C_v.
- PATOH_CRS_HPC=10: Heavy Pin Clustering. Cluster v has maximum $\sum_{n \in \mathcal{N}_{u,C_v}} |pins[n] \cap C_v|$.
- PATOH_CRS_ABSHCC=11: Absorption Clustering using Nets, This metric is the agglomerative version of PATOH_CRS_ABS. That is, v has maximum $\sum_{n \in N_{u,C_v}} 1/(s_n - 1)$.
- PATOH_CRS_ABSHPC=12: Absorption Clustering using Pins, Similar to PATOH_CRS_ABSHCC, but this accumulates absorption metric for every pin that connects u to C_v in this level. This is the default coarsening scheme.
- PATOH_CRS_CONC=13: Connectivity Clustering,
- PATOH_CRS_GCC=13: Greedy Cut Clustering. This is agglomerative version of PATOH_CRS_GCM. With this metric v_j is chosen to form a cluster with v_i that has minimum $d_i + d_j - N_{i,j}$.
- PATOH_CRS_SHCC=15: Scaled Heavy Connectivity Clustering,

In the net-based clustering algorithms listed below, nets are visited in random order and their pins are considered for clustering. We only recommend to use these schemes when average net size of the hypergraph is very small.

- PATOH_CRS_NC=16: Net Clustering. All pins of a net are clustered if none of them has been clustered yet. If at least one of them has been clustered, net is skipped.

  - `PATOH_CRS_MNC=17`: Modified Net Clustering. All pins of a net that are not currently clustered gathered to form a cluster.

- `crs_coarsento`: limits the number of cells in the coarsest hypergraph.
- `crs_coarsentokmult`: Number of cells in the coarsest hypergraph is set to maximum of $K \times$`crs_coarsentokmult`.
- `crs_coarsenper`: Stops coarsening when number of cells is not reduced more than `crs_coarsenper`% (default is 9%).
- `crs_maxallowedcellwmult`: limits the construction of large cells. Maximum weight of a cell can be at most `crs_maxallowedcellwmult`$\times W_{avg}$.
- `crs_idenafter`: starting level of identical net detection in coarsening. Supplying negative values results in automatic computation of the parameter.
- `crs_iden_netsizetrh`: Threshold net size for identical net detection. Nets whose sizes are equal or less than this values will be checked.
- `crs_useafter`: Changes the coarsening algorithm after that level to `crs_useafteralg`.
- `crs_useafteralg`: Coarsening algorithm that will be used after level `crs_useafter`.

4.3.3 Initial Partitioning Parameter

- `nofinstances`: PaToH can refine multiple partitions during the uncoarsening phase. This parameter sets the number of partitioning instance to be constructed in initial partitioning phase. Each of these instances will be refined during the uncoarsening phase.
- `initp_alg`: Determines the initial partitioning algorithm, here is the list of the implemented algorithms:
  - `PATOH_IPA_GHGP=1`: Greedy Hypergraph Growing Partition (GHGP). In this algorithm we grow a cluster around a randomly selected vertex. During the course of the algorithm, the selected and unselected vertices induce a bipartitioning on the coarsest hypergraph. The unselected vertices connected to the growing cluster are inserted into a priority queue according to their FM move gains. Here, the gain of an unselected vertex corresponds to the decrease in the cutsize of the current bipartition if the vertex moves to the growing cluster. Then, a vertex with the highest gain is selected from the priority queue. After a vertex moves to the growing cluster, the gains of its unselected adjacent vertices which are currently in the priority queue are updated and those not in the priority queue are inserted into the queue. This cluster growing operation continues until a predetermined bipartition balance criterion is reached. Since the coarsest graph is small, GHGP algorithm is run multiple times starting from different random vertices and select the best bipartition for refinement during the uncoarsening phase.
  - `PATOH_IPA_AGG_MATCH=2`: Agglomerative Match and Bin Packing. This methods uses agglomerative matching to further create bigger clusters then, simple bin packing is used to assign those clusters to parts.

  - `PATOH_IPA_BINPACK=3`: Bin Packing uses best fit decreasing bin packing heuristic to assign vertices (cells) of the coarsest graph to parts.
  - `PATOH_IPA_BF=4`: Breadth-First Partitioning. This is very similar to GHGP. It starts with assigning randomly selected vertex to part 0 and the others to part 1. It then traverses the hypergraph in breadth-first manner and moves the visited vertices to from part 1 to part 0. This cluster growing operation continues until a predetermined bipartition balance criterion is reached.
  - `PATOH_IPA_RANDOM1=5`: A random initial partitioning. Vertices are visited in a random order and a random part assignment is choosen for each vertex. Vertex is placed in that part unless the assignment violates the imblance requirement.
  - `PATOH_IPA_RANDOM2=6`: Another random initial partitiong. Vertices are visited in a random order and are assigned to the part with minimum weight.
  - `PATOH_IPA_RANDOM3=7`: Yet another random initial partitioning. A random vertex visit order is created and this list into partitioned into two for best balance.
  - `PATOH_IPA_GHG_MAXPIN=8`: Greedy hypergraph Growing with Max Pin. A variant of GHGP that prioritizes the moves of the vertices to the growing cluster by number of pins connected to the growing cluster.
  - `PATOH_IPA_GHG_MAXNET=9`: GreedyHypergraph Growing with Max Net. A variant of GHGP that prioritizes the moves of the vertices to the growing cluster by number of nets connected to the growing cluster. That is it discards the connections to the other nets/cells.
  - `PATOH_IPA_GHG_MAXPOSGAIN=10`: GreedyHypergraph Growing with Max only-Pos FM Gain. A variant of GHGP that only moves vertices that has positive FM gains.
  - `PATOH_IPA_COMP_GHGP=11`: Component bin-pack and Greedy Hypergraph Growing Partition. This partitioning method first finds connected components of the coarse graph then assigns those to parts using bin-packing.
  - `PATOH_IPA_GREEDY_COMP_GHGP=12`: Greedy Component bin-pack and Greedy Hypergraph Growing Partition. This partitioning method first finds connected components of the coarse graph then generates a coarser graph from those, and then uses GHGP to partition that.
  - `PATOH_IPA_ALL=13`: use one of the above at each instance and/or run of initial partitioning.

- `initp_runno`: the number of initial partititioning runs for each instance.
- `initp_ghg_trybalance`: if it is set to a non-zero value, PaToH tries to find better balanced partitions during greedy hypergraph growing partitioning.
- `initp_refalg`: refinement algorithm that will be used after each initial partitioning, please refer to `ref_alg` in the next section, for a list of available algorithms.

4.3.4 Uncoarsening Parameters

- `ref_alg`: Determines the refinement algorithm that will be used during the uncoarsening. Current version of PaToH contains 18 KLFM-based refinement algorithms:

    * `PATOH_REFALG_NONE=0`: No refinement (`NONE`).
    * `PATOH_REFALG_T_BFM=1`: Boundary FM (BFM) with tight balance. This algorithm only moves vertices, once in each pass, that are in the boundary. It always moves vertices from heavily loaded part to underloaded part.
    * `PATOH_REFALG_T_FM=2`: FM with tight balance. Similar to previous algorithm this algorithm always moves vertices from heavily loaded part to underloaded part, but every vertex is eligible for a move once in each pass of the algorithm.
    * `PATOH_REFALG_D_BFM=3`: BFM with dynamic locking. Similar to T_BFM, moves boundary vertices but allows a vertex to be moved more than once in each pass.
    * `PATOH_REFALG_D_FM=4`: FM with dynamic locking. This is a generalized implementation of FM algorithm that allows vertices to be moved more than once in each pass.
    * `PATOH_REFALG_BKL=5`: Boundary Kernighan-Lin (BKL). This algorithms swaps the vertices in the boundary, instead of moving a single vertex.
    * `PATOH_REFALG_KL=6`: Kernihgan-Lin (KL). Swap-based refinment algorithm. Swap code is not optimized agressively hence it should be avoided for larger hypergraphs. However, due to balance constraints, it might be preferable especially in the higher levels of uncoarsening (when graph is small).
    * `PATOH_REFALG_MLG_BFM=7`: BFM with Krishnamurthy's multilevel gain improvement [?].
    * `PATOH_REFALG_MLG_FM=8`: FM with Krishnamurty's multilevel gain.
    * `PATOH_REFALG_BFMKL=9`: One pass BFM followed by one pass BKL.
    * `PATOH_REFALG_FMKL=10`: One pass FM followed by one pass KL.
- `ref_useafter`: After that level of coarsening refinement algorithm `ref_useafter_alg` will be used.
- `ref_useafteralg`: Refinement algorithm that will be used after level `ref_useafter`.
- `ref_passcnt`: Limits the number of passes at each level of uncoarsening.
- `ref_maxnegmove`: Limits the number of consecutive negative-gain moves, for early termination.
- `ref_maxnegmovemult`: Limits the number of consecutive negative-gain moves to `ref_maxnegmovemult`$\times|\mathcal{V}_i|$ at the i-th level of uncoarsening.
- `ref_dynamiclockcnt`: Limits the maximum number of moves of a cell in a pass. Setting this to one will be equivelant of running the clasic FM algorithm.
- `ref_slow_uncoarsening`: PaToH switches to "faster" refinement method (`PATOH_REFALG_D_BFM`) if uncoarsening is `ref_slow_uncoarsening` times slower than coarsening. To get repeatable results no matter what, this number should be very large.
- `balance`: enforces/relaxes PaToH's balance constraint. 0: Strictly forces balance to be $\varepsilon/\log K$ in each bisection, 1: Dynamically adjusts imbalance in each recursion aiming ε imbalance at the end, 2: each bisection in recursion will have maximum imbalance ε.
- `init_imbal`: imbalance ratio of the coarsest hypergraph, i.e., maximum part weight in the coarsest hypergraph can be at most $W_{avg} \times$ (1+`init_imbal`).
- `final_imbal`: imbalance ratio of the final partition, i.e., ε in Eq (**??**).

- `fast_initbal_mult`: To give some room to refinement algorithm in terms of imbalance, at the beginning of each uncoarsening level partition is forced to have maximum imbalance ratio of `fast_initbal_mult`$\times\varepsilon$, then selected refinement algorithm is executed.
- `init_sol_discard_mult`: At the coarsest level, instances which have `init_sol_discard_mult` times worse cutsize than the partition with minimum cutsize are discarded.
- `final_sol_discard_mult`: At the final partition, instances which have `init_sol_discard_mult` times worse cutsize than the partition with minimum cutsize are discarded. Note that, PaToH linearly interpolates this discard multiplier at each level of uncoarsening.

5 Stand-Alone Program

Distribution includes a stand-alone program, called `patoh`, for single constraint partitioning[1]. `patoh` gets its paramaters from command line arguments. You can run the PaToH from command line as follows:

```
> patoh <hypergraph-file> <number-of-parts> [[parameter1] [parameter2] ....]
```

You can tune the parameters using optional `[parameter]` arguments. The syntax of these optional parameters is as follows; two-letter abbreviation of a parameter is followed by an equal sign and a value. For example, parameter `ref_alg` is abbreviated as "RA" and to select "Boundary FM with dynamic locking" (3rd algorithm) you should use "RA=3". For a complete example, lets say we would like to partition the sample hypergraph `ken-11.u` (part of the distribution) into 4 parts using the Kernighan-Lin refinement algorithm with cutnet metric (the default is "Connectivity-1" metric (Equation **??**(b)). Below is the command you need to execute and a sample output:

```
> patoh ken-11.u 4 RA=6 UM=U



+++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++ PaToH v3.2 (c) Nov 1999-, by  Umit V. Catalyurek
+++ Build Date: Sun Mar 13 17:41:19 2011 -0400
+++++++++++++++++++++++++++++++++++++++++++++++++++++++

********************************************************************************
Hypergraph : ken-11.u   #Cells : 14694   #Nets : 14694   #Pins : 82454
********************************************************************************
 4-way partitioning results of PaToH:

 Cut Cost:  4775
 Part Weights   : Min=      20408 (0.010) Max=      20947 (0.016)
--------------------------------------------------------------------------------
I/O          :       0.007 sec
I.Perm/Cons.H:       0.002 sec  ( 0.7%)
Coarsening   :       0.087 sec  (37.9%)
Partitioning :       0.011 sec  ( 4.8%)
Uncoarsening :       0.128 sec  (55.8%)
Total        :       0.229 sec
Total (w I/O):       0.236 sec
--------------------------------------------------------------------------------
```

This output shows that cut cost according to cutnet metric is 4775. Final imbalance ratios for least loaded and most loaded parts are 1.6% and -1.0%, and partitioning (without I/O) only took 0.229 seconds.

Tables **??** and **??** display the command-line abbreviation of each parameter, the value-types of the parameters and the valid ranges of the values.

[1]Please note that this executable will not work with multiple vertex weights. For multi-constraint partioning use the provided library interface. Also see sample source codes for use of multi-constraint partitioning.

Table 1: Stand-alone program parameters

Parameter	Abbreviation	Type	Range
Miscellaneous Parameters			
`outputdetail`	OD	int	0, 1, 2, 3
`seed`	SD	int	-1: random, otherwise sets seed
`doinitperm`	DP	int	0, 1
`bisec_fixednetsizetrsh`	float	int	$[1, max_{int})$
`bisec_netsizetrsh`	NT	float	$[0.5, max_{float})$
`bisec_partmultnetsizetrsh`	NM	int	$[1, max_{int})$
`bigVcycle`	BV	int	$[1, max_{int})$
`smallVcycle`	SV	int	$[1, max_{int})$
`usesamematchinginVcycles`	SM	int	0, 1
`usebucket`	UB	int	-1, 0, 1
`maxcellinheap`	HC	int	$[0, max_{int})$
`heapchk_mul`	HM	int	$[1, max_{int})$
`heapchk_div`	HD	int	$[1, max_{int})$
`MemMul_CellNet`	A0	int	$[1, max_{int})$
`MemMul_Pins`	A1	int	$[1, max_{int})$
`MemMul_General`	A2	int	$[1, max_{int})$
Coarsening Parameters			
`crs_VisitOrder`	VO	int	[0, 6]
`crs_alg`	MT	int	[1, 17]
`crs_coarsento`	CT	int	$[10, max_{int})$
`crs_coarsentokmult`	CK	int	$[1, max_{int})$
`crs_coarsenper`	CP	int	[1, 100]
`crs_maxallowedcellwmult`	CM	float	[0.01-1.0]
`crs_idenafter`	ID	int	$[-1, max_{int})$
`crs_iden_netsizetrh`	IT	int	$[2, max_{int})$
`crs_useafter`	FL	int	$[0, max_{int})$
`crs_useafteralg`	FM	int	[1, 17]
Initial Partitioning Parameter			
`nofinstances`	NI	int	$[1, max_{int})$
`initp_alg`	PA	int	[1, 13]
`initp_runno`	IR	int	$[1, max_{int})$
`initp_ghg_trybalance`	TB	int	0, 1
`initp_refalg`	IA	int	[0, 10]

Table 2: Stand-alone program parameters (continued)

Parameter	Abbreviation	Type	Range
Uncoarsening Parameters			
`ref_alg`	RA	int	[0, 10]
`ref_useafter`	RL	int	$[0, max_{int})$
`ref_useafteralg`	RF	int	[0, 10]
`ref_passcnt`	RP	int	$[1, max_{int})$
`ref_maxnegmove`	RN	int	$[5, max_{int})$
`ref_maxnegmovemult`	RU	float	[0.0001, 1.0]
`ref_dynamiclockcnt`	LC	int	-1, 1, 2, 3
`balance`	BA	int	0, 1, 2
`init_imbal` & `final_imbal`	IB	float	[0.00, 0.50]
`init_imbal`	II	float	[0.00, 0.50]
`final_imbal`	FI	float	[0.00, 0.50]
`fast_initbal_mult`	FB	float	[0.5, 2.0]
`init_sol_discard_mult`	DI	float	[0.01, 1.00]
`final_sol_discard_mult`	DF	float	[0.01, 1.00]
Parameter for Stand-alone program			
total # of runs	NR	i	$[1, max_{int})$

5.1 Input File Format

The input hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is stored in a plain text file. The first line after the possible comment lines describes the size of the hypergraph, the index base (0 or 1) and the weighting scheme. The rest of the file contains information for each net, and possibly for each vertex– depending on the weighting scheme. Any line beginning with '%' is a comment line and skipped.

The first line contains 4, optionally 6 integers. The first one–either 1, or 0– shows the base value used in indexing the elements of $\mathcal{V}$ and $\mathcal{N}$. Next the sizes of the sets $|\mathcal{V}|$, $|\mathcal{N}|$, and *pins* should be present. The fifth integer is optional and describes the weighting scheme of the hypergraph, if present. The hypergraph can have weights associated with cells, nets, or both, 1, 2, 3 respectively. The sixth integer also optional and denotes the number of constraints, in other words number of weights for each cells. If it is omitted it is assumed to be 1.

The next $|\mathcal{N}|$ lines contain the information about the nets. i^{th} line (excluding the comments) contains the cell list of net n_i. In the case of the weighted nets, each line begins with an integer representing the weight of n_i.

If cells are weighted, following the net lines, each cell's weight must be supplied. If there are more than one weight constraints then each cell should have number of constraint weights.

Input file corresponding to the sample hypergraph of Section **??** is displayed in Figure **??**.

5.2 Output File Format

When standalone PaToH executable `patoh` completes K-way partitioning of an hypergraph, $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, stored in input file `input.hygr`, it generates an output file named `input.hygr.part.`K. This file contains $|\mathcal{V}|$ integers in range $[0, K-1]$. i^{th} entry in this file represents the part number

```
1 8 9 28
8 6 3 5 2
4 5 1 7
4 2 5 7
4 7
3 5
8 2 4
6 5 2
5 7 2
8 4
```

(a)

```
1 8 9 28 2
10 8 6 3 5 2
15 4 5 1 7
13 4 2 5 7
18 4 7
25 3 5
20 8 2 4
14 6 5 2
27 5 7 2
29 8 4
```

(b)

```
1 8 9 28 1
8 6 3 5 2
4 5 1 7
4 2 5 7
4 7
3 5
8 2 4
6 5 2
5 7 2
8 4
80 85 30 55 42 39 90 102
```

(c)

```
1 8 9 28 3
10 8 6 3 5 2
15 4 5 1 7
13 4 2 5 7
18 4 7
25 3 5
20 8 2 4
14 6 5 2
27 5 7 2
29 8 4
80 85 30 55 42 39 90 102
```

(d)

Figure 4: (a) Hypergraph file without weights, (b) Hypergraph file with net weights (10, 15, 13, 18, 25, 20, 14, 27, and 29), (c) Hypergraph file with cell weights (80, 85, 30, 55, 42, 39, 90, and 102), (d) Hypergraph file with weights on both nets and cells.

that i^{th} cell is assigned to.

6 License

PaToH binary distribution is available free of charge for non-commercial, research use by individuals, academic or research institutions and corporations. Commercial use of PaToH software requires commercial license. Please direct commercial-use license inquiries to umit@gatech.edu.