

# An Efficient Parallel Implementation of a Perfect Hashing Method for Hypergraphs

Somesh Singh, INRIA and LIP, France.

Bora Uçar, CNRS and LIP, France.



GrAPL 2022: Workshop on  
Graphs, Architectures,  
Programming, and Learning

# Problem of Interest

Given: A  $d$ -dimensional sparse tensor  $\mathcal{T}$

Goal: To answer queries of the form — “Is  $\mathcal{T}[i_1, \dots, i_d]$  zero or nonzero?”

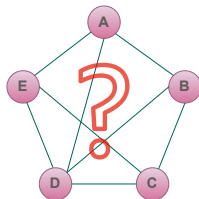
A desirable solution should have:

- $O(d)$  query response time
- *Small* memory overhead
- *Fast* preprocessing

Our focus: Hashing methods with **worst-case optimal** lookups

# Motivating Applications

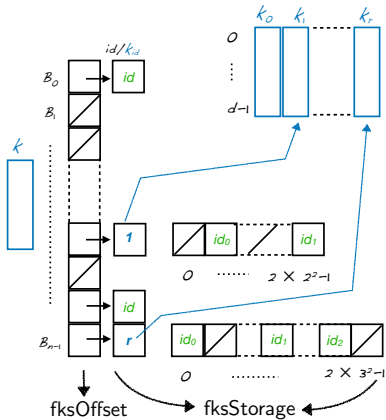
- 1 Kolda and Hong\* propose an efficient algorithm for the decomposition of sparse tensors.
  - Sample the zeros and nonzeros of the given tensor.
  - For sampling zeros, a random set of indices is created, and those positions in the given tensor are checked for zero.
- 2 Checking for the presence of edges in a dense graph or subgraph (e.g. a quasi-clique).



---

\*T. G. Kolda and D. Hong, "Stochastic gradients for large-scale tensor decomposition," SIAM Journal on Mathematics of Data Science, vol. 2, no. 4, pp. 1066–1095, 2020.

# FKSLean<sup>§</sup> — A Perfect Hashing Method



FKSLean data-structure

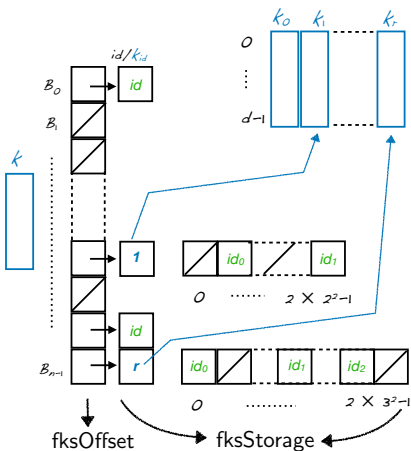
- FKSLean employs a two-level structure to obtain a perfect hashing.
- First level hash function:  

$$h(k, x, p, n) := (k^T x \bmod p) \bmod n$$
- Second level hash function:  

$$h(k_i, x, p, 2b_i^2) := (k_i^T x \bmod p) \bmod 2b_i^2$$

<sup>§</sup>Bertrand et al., "Algorithms and data structures for hyperedge queries," Inria Grenoble Rhône-Alpes, Research Report RR-9390v4, Apr. 2022.

## FKSLearn — Storage Requirements



## FKSLean data-structure

- $K$  is a set of  $d$ -tuples.

At each bucket  $B_i$ :

- $b_i$  is the number of hyperedges.
- If  $b_i = 0$ , nothing is stored.
- If  $b_i = 1$ , a reference to the only hyperedge in  $B_i$  is stored.
- If  $b_i \geq 2$ ,
  - A  $k_i \in \mathbb{K}$  which defines a perfect hashing for  $B_i$  is stored.
  - Storage space of size  $2b_i^2$ , which holds the references to the  $b_i$  hyperedges in  $B_i$ .

## A few theoretical results [Bertrand *et al.*]

- 1 One can quickly find a random  $k \in U$  guaranteeing  $O(n)$  total storage space.
- 2 For every bucket  $B_i$ , one can quickly find a random  $k_i \in U$  that defines a second level perfect hash function for that bucket.
- 3  $O(\log n)$  different  $d$ -tuples in  $K$  are enough, in expectation, to supply each bucket with a suitable hash function.

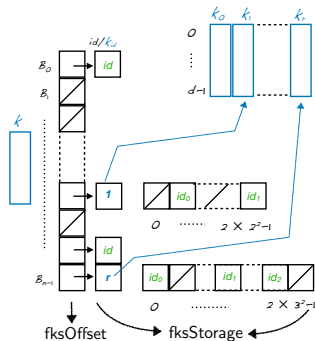
## In practice [Bertrand *et al.*]

- 1 Total storage space required for the buckets is less than  $5n$ .
- 2 Less than  $0.5 \log_2 n$   $d$ -tuples in  $K$  suffice.

# PARFKSLEAN: Parallelize the Construction of FKSLean

Parallel construction proceeds in two steps:

- 1 Setting up `fksOffset`, in parallel
  - 1 Bucketing
  - 2 Build hyperedge-lists for buckets
  - 3 Populate `fksOffset`
- 2 Populating `fksStorage`, in parallel



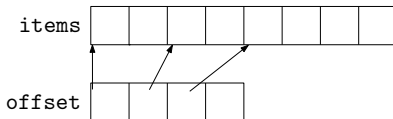
# 1 Setting-up fksOffset

## 1 Bucketing

- Compute in parallel  $h(\mathbf{k}, \mathbf{e}, \mathbf{p}, \mathbf{n})$  for every hyperedge  $\mathbf{e}$  and store it in *bucket\_ids* array.

## 2 Building hyperedge-lists for buckets

- Maintain two arrays — *items* and *offset*.



- Populate *offset* array with histogram of *bucket\_ids* array.
- Parallel prefix-sum on *offset* array using a two-pass algorithm.
- Populate the *items* array in parallel.



# 1 Setting-up fksOffset

## 3 Populating fksOffset

- Populate fksOffset in parallel, using the following relation:

$$\text{fksOffset}[i] := \begin{cases} b_i & \text{if } b_i \in \{0, 1\}, \\ 1 + 2b_i^2 & \text{otherwise.} \end{cases}$$

where,  $b_i := \text{offset}[i + 1] - \text{offset}[i]$ .

- Parallel prefix-sum on fksOffset array.
- Examine fksOffset[n] for checking the storage requirement.

## 2 Populating fksStorage

- 1 Populate  $\mathbf{K}$  with  $2 \log_2 n$  keys.
- 2 Coarse-grained parallelization for populating fksStorage — handle every bucket independently.
  - If  $b_i = 0$ , do nothing.
  - If  $b_i = 1$ , store the id of the hyperedge in fksStorage at position  $\text{fksOffset}[i]$ .
  - If  $b_i \geq 2$ ,
    - Pick a  $\mathbf{k}_i$  from  $\mathbf{K}$  to effect a perfect hashing of the hyperedges mapped to  $B_i$ .
    - Place hyperedge  $\mathbf{e}$  in the hyperedge-list of  $B_i$  at position  $(h(\mathbf{k}_i, \mathbf{e}, p, 2b_i^2) + \text{fksOffset}[i])$  in fksStorage.

# Experimental Evaluation

---

<b>CPU</b>	Intel Xeon Gold 5218 (64 cores, 2.3 GHz, 384 GB RAM)
------------	--

<b>Software</b>	Debian GNU/Linux 10 (64 bit), GCC 8.3.0, OpenMP
-----------------	---

---

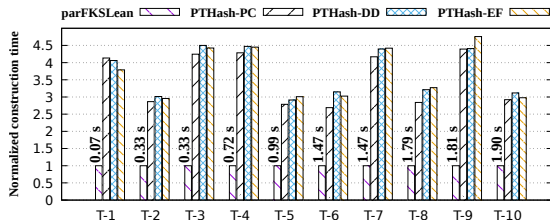
**State-of-the-art:** [PTHash](#)<sup>†</sup> — nonminimal and minimal perfect hash function for static sets, with support for parallel construction.

**Inputs:** Tensors from the [FROSTT](#) (<http://frostdt.io/>) dataset.

---

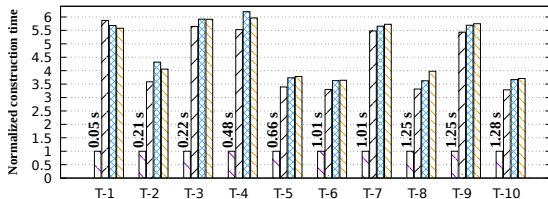
<sup>†</sup>G. E. Pibiri and R. Trani, “PTHash: Revisiting FCH minimal perfect hashing,” in 44th SIGIR, International Conference on Research and Development in Information Retrieval. ACM, 2021, pp. 1339–1348

# Construction Time



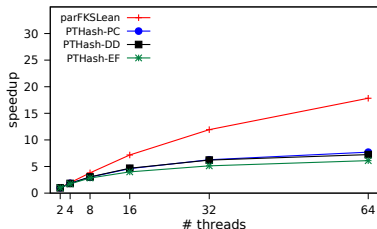
# threads = 32

# threads = 64

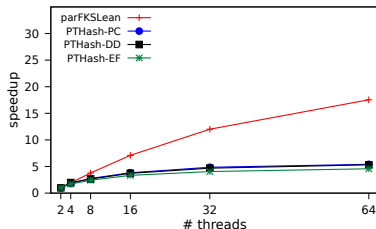


**Takeaway:** In the construction phase, **parFKSLean** is **always** faster than all the three variants of PTHash for all thread configurations.

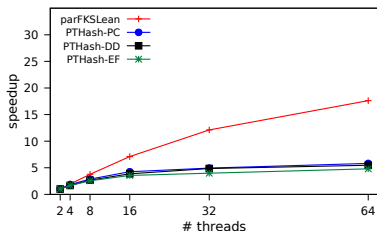
# Scalability of Construction Phase



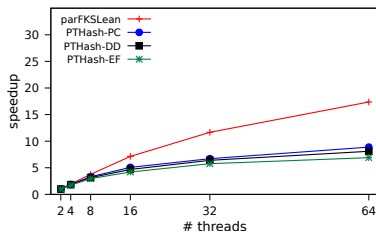
nell-2



flickr-4d



delicious-4d



nell-1

**Takeaway:** **PARFKSLEAN** exhibits better parallel scaling.

# Query Response Time

Tensor	#Threads	PThash			PARFKSLEAN
		-PC	-DD	-EF	
nell-2	2	2.01	1.64	2.10	0.97
	4	1.01	0.95	1.06	0.53
	8	0.46	0.49	0.54	0.27
	16	0.25	0.27	0.29	0.15
	32	0.14	0.15	0.16	0.11
	64	0.08	0.09	0.11	0.07
flickr-4d	2	2.51	2.04	2.20	1.07
	64	0.11	0.09	0.09	0.08
delicious-4d	2	2.30	2.02	2.25	1.11
	64	0.10	0.09	0.15	0.08
nell-1	2	2.01	1.64	2.10	0.97
	64	0.11	0.10	0.08	0.08

Execution time (in seconds) for  $10^7$  queries on four large tensors.

**Takeaway:** PARFKSLEAN is at least as fast as the best performing variant of PThash in all thread configurations for all inputs.

# Conclusions

- 1 **PARFKSLEAN** parallelizes the construction phase of FKSLean.
- 2 The construction phase of **PARFKSLEAN** exhibits good parallel scaling.
- 3 **PARFKSLEAN** outperforms the state-of-the-art both in construction, and query response.

- 1 **PARFKSLEAN** parallelizes the construction phase of FKSLean.
- 2 The construction phase of **PARFKSLEAN** exhibits good parallel scaling.
- 3 **PARFKSLEAN** outperforms the state-of-the-art both in construction, and query response.

## Thank You

(<https://perso.ens-lyon.fr/somesh.singh/>)



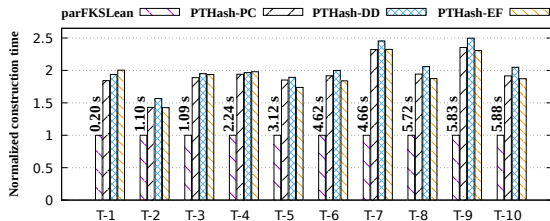
# Backup Slides

# Input Tensors

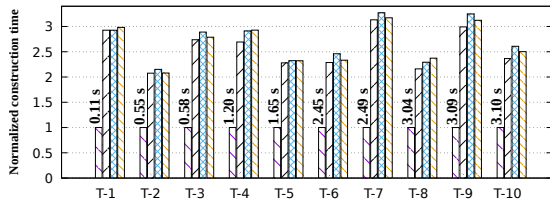
Tensor	$d$	Dimensions	$n$
chicago_crime (T-1)	4	$6,186 \times 24 \times 77 \times 32$	5,330,673
vast-2015-mc1-3d (T-2)	3	$165,427 \times 11,374 \times 2$	26,021,854
vast-2015-mc1-5d (T-3)	5	$165,427 \times 11,374 \times 2 \times 100 \times 89$	26,021,945
enron (T-4)	4	$6,066 \times 5,699 \times 244,268 \times 1,176$	54,202,099
nell-2 (T-5)	3	$12,092 \times 9,184 \times 28,818$	76,879,419
flickr-3d (T-6)	3	$319,686 \times 28,153,045 \times 1,607,191$	112,890,310
flickr-4d (T-7)	4	$319,686 \times 28,153,045 \times 1,607,191 \times 731$	112,890,310
delicious-3d (T-8)	3	$532,924 \times 17,262,471 \times 2,480,308$	140,126,181
delicious-4d (T-9)	4	$532,924 \times 17,262,471 \times 2,480,308 \times 1,443$	140,126,181
nell-1 (T-10)	3	$2,902,330 \times 2,143,368 \times 25,495,389$	143,599,552

Input tensors from FROSTT dataset

# Construction Time



# threads = 16



**Takeaway:** In the construction phase, **parFKSLean** is **always** faster than all the three variants of PTHash for all thread configurations.