HAL
open science

An Efficient Parallel Implementation of a Perfect Hashing Method for Hypergraphs

Somesh Singh, Bora Uçar

To cite this version:

Somesh Singh, Bora Uçar. An Efficient Parallel Implementation of a Perfect Hashing Method for Hypergraphs. GrAPL 2022 - Workshop on Graphs, Architectures, Programming, and Learning, IEEE, May 2022, Lyon, France. pp.265–274. hal-03612360

HAL Id: hal-03612360

https://inria.hal.science/hal-03612360

Submitted on 17 Mar 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

An Efficient Parallel Implementation of a Perfect Hashing Method for Hypergraphs

Somesh Singh
INRIA and LIP (CNRS - Université de Lyon - INRIA - ENS Lyon), France
somesh.singh@ens-lyon.fr

Bora Uçar
CNRS and LIP (CNRS - Université de Lyon - INRIA - ENS Lyon), France
bora.ucar@ens-lyon.fr

***Abstract*—Querying the existence of an edge in a given graph or hypergraph is a building block in several algorithms. Hashing-based methods can be used for this purpose, where the given edges are stored in a hash table in a preprocessing step, and then the queries are answered using the lookup operations. While the general hashing methods have fast lookup times in the average case, the worst case run time is much higher. Perfect hashing methods take advantage of the fact that the items to be stored are all available and construct a collision free hash function for the given input, resulting in an optimal lookup time even in the worst case. We investigate an efficient shared-memory parallel implementation of a recently proposed perfect hashing method for hypergraphs. We experimentally compare the resulting parallel algorithms with the state-of-the-art and demonstrate better run time and scalability on a set of hypergraphs corresponding to real-life sparse tensors.**

I. Introduction

We investigate parallel algorithms for constructing data structures to answer queries about the existence of edges in a given graph or hypergraph. More precisely, given a hypergraph $H = (V, E)$ with the vertex set V and the hyperedge set E, a query consists of a subset q of vertices V and asks if q is a hyperedge in E. Our focus is on d-uniform, d-partite hypergraphs where the vertex set is the union of d disjoint sets, $V = \bigcup_{i=1}^{d} V^{(i)}$, and each hyperedge contains exactly one vertex from each set $V^{(i)}$. A special case is $d = 2$, which concerns bipartite graphs.

Much like bipartite graphs naturally model matrices, d-uniform, d-partite hypergraphs model d-dimensional tensors (or multidimensional arrays). Let $\mathcal{T}$ be a d-dimensional tensor of size $s_1 \times \cdots \times s_d$. Each nonzero entry in $\mathcal{T}$ is addressed with an ordered set of d indices, or a d-tuple, of the form $[i_1, \ldots, i_d]$ where $i_j \in \{1, \ldots, s_j\}$, for $j = 1, \ldots, d$. The d-uniform, d-partite hypergraph $H = (V, E)$ associated with $\mathcal{T}$ can then be defined as follows. The vertex set $V = \bigcup_{i=1}^{d} V^{(i)}$, where $V^{(i)} = \{v_1^{(i)}, \ldots, v_{s_i}^{(i)}\}$. The hyperedge set E contains a hyperedge $h = [v_{i_1}^{(1)}, \ldots, v_{i_d}^{(d)}]$ for each nonzero $\mathcal{T}[i_1, \ldots, i_d]$. With this correspondence in mind, our aim is thus developing efficient parallel algorithms to answer queries asking if a given position in matrix or a tensor is nonzero.

A particular use case arises in a recent tensor decomposition method proposed by Kolda and Hong [10]. They develop a sparse tensor decomposition algorithm where zeros and nonzeros of a given tensor need to be sampled. For sampling zeros, a random set of indices is created, and those positions in the given tensor are checked to see if they are zero or not. Modeling the given sparse tensor as a hypergraph thus leads to the problem at hand. In a more general setting, a data structure that quickly answers edge queries can be used to detect if a given set of vertices form a clique or a dense enough vertex set, in time quadratically proportional to the size of the vertex set—independent of the sum of degrees of the vertices in the set—which can be much larger.

A query in the mentioned scenarios has d indices. Since, a query cannot be answered without reading its indices, an answer can only be delivered in $\Omega(d)$ time. An efficient data structure to answer a query

should therefore take $\Theta(d)$ time. If we consider d a constant, a query should be answered in constant time. One can use classical hashing methods to answer the queries. While in the average case the query response time will be $O(d)$, the worst case is much higher. Since the hypergraph (or the tensor) is available to preprocess before answering queries and does not change, one can do better. *Perfect hashing methods* work for a given set of static items, and find a hash function such that when used on the given set there is no collision. Such methods thus enable a $\Theta(d)$ query time in the worst case. One perfect hashing method, called FKSLean, is recently developed by Bertrand *et al.* [2] for the problem at hand. We propose PARFKSLEAN—efficient parallelization of the methods used to construct the data structures of FKSLean, on shared-memory parallel systems.

The organization of the rest of this paper is as follows. Since we parallelize FKSLean, we present its detailed summary in the next section. We briefly survey some recent studies on perfect hashing in the same section. In Section III, we describe our parallel implementation: PARFKSLEAN. We evaluate PARFKSLEAN in Section IV on a set of large problem instances, and compare it with the state-of-the-art. Section V concludes the paper with a summary and planned research.

II. Background

In a d-partite d-uniform hypergraph $H = (V, E)$, the hyperedge set E is a set of d-tuples. These hypergraphs are generalizations of bipartite graphs and model sparse tensors and matrices.

Throughout the paper, n denotes the number of hyperedges, and p is a prime number larger than n. Without loss of generality we assume that $n \geq |V^{(i)}|$ for $i = 1, \ldots, d$, as otherwise there are vertices which are not in any of the hyperedges. The universe of all d-tuples of the form $[x_0, \ldots, x_{d-1}]$ where x_i is between 0 and $p-1$ is denoted by U. In other words, $U = \{0, \ldots, p-1\}^d$ where $E \subseteq U$. Each query will be an element of the universe U.

A. A summary of FKSLean

The FKSLean method [2] uses a two-level structure to obtain a perfect hashing for a given static set of hyperedges. It is an adaptation of a well-known method [7] and handles the hypergraphs efficiently. A first level hash function is used to split the given hyperedges into buckets. Then for each bucket a perfect hashing of hyperedges mapped to that bucket is found and the hyperedges, or pointers to the hyperedges, are stored in a space associated with that bucket.

Let $\mathbf{x}, \mathbf{y}$ be two elements of the universe U. We use $\mathbf{x}^T\mathbf{y} := \sum_{i=1}^{d} x_i y_i$ to denote the inner product of the vectors corresponding to d-tuples $\mathbf{x}$ and $\mathbf{y}$. In the FKSLean method, a $\mathbf{k} \in U$ is chosen for the first level, and the hash function $h : U \rightarrow \{0, \ldots, n-1\}$ is defined as $h(\mathbf{k}, \mathbf{x}, p, n) := (\mathbf{k}^T\mathbf{x} \bmod p) \bmod n$. Then, each hyperedge $\mathbf{x} \in E$ is assigned to the bucket B_i where $i := h(\mathbf{x}, \mathbf{k}, p, n)$. Let b_i denote the number of hyperedges from E that are mapped to B_i. A space of size $2b_i^2$ is associated with the bucket B_i to store the references to the hyperedges mapped to B_i, each at a unique place in this space. This is achieved by choosing a $\mathbf{k}_i \in U$ for each bucket B_i such that $h(\mathbf{k}_i, \mathbf{x}, p, 2b_i^2) := ({\mathbf{k}_i}^T\mathbf{x} \bmod p) \bmod 2b_i^2$ is an injection for all $\mathbf{x}$ mapped to B_i by the first level hash function. In other words, each hyperedge $\mathbf{x}$ in B_i is mapped to a unique number in $\{0, \ldots, 2b_i^2 - 1\}$ with $h(\mathbf{k}_i, \mathbf{x}, p, 2b_i^2)$. Then a reference to $\mathbf{x}$ is stored in the space associated with B_i at location $h(\mathbf{k}_i, \mathbf{x}, p, 2b_i^2)$. Note that as there are b_i hyperedges in the bucket B_i, only $\frac{1}{2b_i}$ of the storage space contains references to the hyperedges; other entries are empty.

Given this structure, testing if a given hyperedge $\mathbf{q}$ is in the hyperedge set E can be carried out in two steps. First, the bucket number $i = h(\mathbf{k}, \mathbf{q}, p, n)$ is computed. Then, B_i's storage at $h(\mathbf{k}_i, \mathbf{q}, p, 2b_i^2)$ is checked. If no reference is stored there, then $\mathbf{q}$ is not in E; else the hyperedge whose reference is stored at that position is compared with $\mathbf{q}$, and the answer is returned.

FKSLean keeps a set $\mathbf{K}$ of d-tuples so that each bucket B_i uses one of the d-tuples in $\mathbf{K}$ as $\mathbf{k}_i$. The idea is to store only a few different d-tuples and use them at different buckets for the second level hash functions. A bucket thus stores the id of a d-tuple in $\mathbf{K}$ instead of a d-tuple itself.

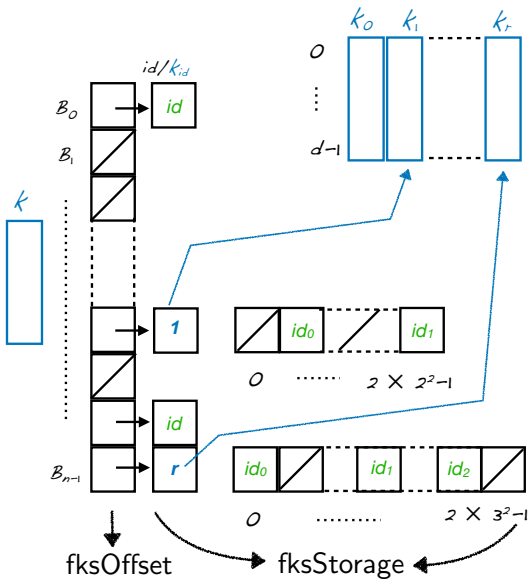


Fig. 1: The FKSLean data structure [2].

Figure 1 depicts the data structure of FKSLean [2]. As seen in the figure, there is a d-tuple $\mathbf{k}$ used for the first level hash function, there is a set $\mathbf{K}$ of d-tuples, and a storage space for each bucket. For each bucket B_i, one needs to know the number b_i of hyperedges mapped to B_i by the first level hash function. Then, (i) if $b_i = 0$, then nothing else is needed for that bucket; (ii) if $b_i = 1$, then a reference to the only hyperedge mapped to B_i is stored (no second level hash function is needed); (iii) otherwise, the index of a d-tuple from $\mathbf{K}$ which defines a perfect hashing for B_i is stored, along with a storage space of size $2b_i^2$, which holds the references to b_i hyperedges.

The effectiveness of FKSLean stems from three theoretical results [2]. First, for a randomly chosen $\mathbf{k} \in U$ the probability that $\sum b_i^2 < 7n$ is more than $1/2$. Thanks to this result, one can, in expectation, find a $\mathbf{k}$ such $\sum b_i^2 < 7n$ in a few trials. Such a $\mathbf{k}$ guarantees a total of $O(n)$ storage space for the buckets. The second theoretical result is that for a randomly chosen $\mathbf{k}_i \in U$ the probability that $\mathbf{k}_i$ defines a perfect hashing $h(\mathbf{k}_i, \mathbf{x}, p, 2b_i^2)$ for the hyperedges of B_i is more than $1/2$. This conveys that such a $\mathbf{k}_i$ can be found in a constant number of trials in expectation. These two properties, based on the original source [7], guarantee that in expectation (or in the average case), the structure can be constructed in $O(dn)$ time. The third theoretical result, unique to FKSLean, is that $O(\log n)$ different d-tuples in $\mathbf{K}$ are enough, in expectation, to supply each bucket with a suitable hash function.

As discussed above, FKSLean uses a total of $O(n)$ storage space for the buckets, and another $O(d \log n)$ space for $\mathbf{K}$. Theoretical results and practical experiments [2] show that the constants in the big-oh formulas are small. In practice, less than $5n$ storage space, and less than $0.5 \log_2 n$ tuples in $\mathbf{K}$ are needed.

B. Other recent works

There are other recent studies focusing on perfect hash functions for static sets. Among them, the most recent ones are PTHash [14], RecSplit [6] and BBHash [12]. Both PTHash and BBHash have parallel implementations of their construction phases. The current PTHash implementation [15] compares PTHash with BBHash and reports better results with PTHash, thus forming the state-of-the-art. We therefore use PTHash for our experiments.

PTHash and other recent perfect hash functions cited above create minimally perfect hash functions for static sets. These functions map each element of a given set of n elements to the non-negative integers $\{0, \ldots, n-1\}$ without any collisions. Therefore they can be used in the target problem. In this approach, the hyperedges are processed to find a minimally perfect hash function f, bijectively mapping the hyperedge set to $\{0, \ldots, n-1\}$. Then an array of size n is allocated, and the id of a hyperedge $\mathbf{e}$ is stored at the location $f(\mathbf{e})$ of this array (the indices start from zero). In order to check if a given hyperedge $\mathbf{q}$ is in the original set, one thus needs to fetch the id of the hyperedge $\mathbf{e}$ stored at $f(\mathbf{q})$ and compare $\mathbf{q}$ against $\mathbf{e}$.

The recent construction algorithms implemented in PTHash, FKSLean, RecSplit and BBHash are very efficient, and enable worst case $\Theta(d)$ time for queries. Among these, only FKSLean supports hyperedges natively; for others an initial mapping of the input hyperedges to the keys of 64 or 128 bits is necessary.

While the original implementation of PTHash [14] creates minimal perfect hash functions, the recent one [15] also constructs non-minimal hash functions. This relaxation of the minimality constraint enables faster construction time. Our experiments use this latter implementation [15].

We can think of two other alternative approaches to answer hyperedge queries. First, one can use variants of Bloom filters [3], which are designed for approximate set membership tests. Designing memory- and run time-efficient Bloom filter variants is a very active area [4], [8], [18]. In general, these filters have a small false positive probability. Therefore, using a variant of Bloom filter in the targeted application contexts necessitates constructing another data structure to verify every positive answer returned by the filter. While this can reduce the query response time, one has to invest in constructing two data structures. The second alternative is to use Cuckoo hashing [13] and its variants. These are perfect hashing approaches with $O(1)$ lookup time in the worst case; for our case it is $O(d)$. The construction is much more involved. In our case, where the items to be stored are available at the outset, constructing a Cuckoo hash table amounts to constructing a random bipartite graph on items and memory slots, and testing for a perfect matching of items to slots. While there are efficient algorithms for matching [1], [5], finding suitable hash functions, constructing the bipartite graphs implicitly or explicitly, and finding maximum cardinality matchings in parallel will be much more complicated than the algorithms that we parallelize.

III. Parallel construction

parFKSLean implements a parallel version of the construction phase of FKSLean. For efficient memory access, the storage space associated with the buckets is kept as a contiguous array. This leads us to use a data structure similar to the well-known compressed sparse row (CSR) format for representing the buckets and the associated storage space. In Figure 1, fksStorage is a contiguous array and holds the storage space associated with buckets one after another; fksOffset contains the start address of each bucket in fksStorage. In order to build these arrays efficiently, another matrix in CSR format is built in an intermediate step. In other words, a large body of the work carried out by parFKSLean is formulated as building two matrices, one after another. The first one is an $n \times n$ matrix and is built columnwise by setting a single nonzero per column randomly (the first level hash function). A CSR representation of this matrix is built to create a hyperedge list per bucket. The second matrix is of size $n \times S$, where S is the size of the fksStorage array. The row pointers of this second matrix are set up using those of the first one. The column indices are the keys in the second level hash function, shifted according to the start of the buckets, and are not stored. Instead, the hyperedge ids are stored along with zeros—that is why the storage is CSR-like.

The construction is done in three stages. In the first stage, a prime number is chosen. In most practical cases, where $n < 2^{31} - 1$, one can chose p as the Mersenne prime $2^{31} - 1$. If $n \geq 2^{31} - 1$, then a larger prime can be found. There is always a prime number between n and $2n$ for $n > 1$ by Bertrand's postulate [9, p. 455], hence at most n numbers need to be tested for finding p. The second stage finds a suitable $\mathbf{k}$ for the first level hashing and prepares data structures for the next stage. The third stage carries out the second level hashing and finalizes the data structure of Figure 1. We now describe the second and third stages.

A. Setting up fksOffset

The main objective here is to find a $\mathbf{k}$ guaranteeing a bound on the size of fksStorage. It is reported earlier [2] that any randomly chosen $\mathbf{k}$ satisfies the bound $\sum b_i^2 < 7n$, with a much higher probability than 1/2. With this in mind, parFKSLean computes the size of fksStorage while building buckets for a randomly chosen $\mathbf{k}$. This results in this stage carrying out the first level hashing efficiently, and enables coarse grained parallelism in the next stage for the second level hashing. In this phase, thus (i) a suitable $\mathbf{k}$ is found; (ii) fksOffset pointers in Figure 1 are set to the correct value; and (iii) for each bucket a list of hyperedges that map to it are

constructed. The overall computation is carried out in three super-steps outlined below.

1) Bucketing: PARFKSLEAN computes $h(\mathbf{k}, \mathbf{e}, p, n)$ for each hyperedge $\mathbf{e}$ in parallel and stores them in an array bucket_ids. Since these computations are independent, bucket_ids can be populated fully in parallel. This step is parallelized using OpenMP's *parallel for* construct with *static* schedule and a fixed chunk size of $\lceil \frac{n}{T} \rceil$, where T is the number of threads. This achieves good load balancing.

2) Building hyperedge-lists for buckets: Next, the hyperedge-lists of the buckets are built in the CSR format with arrays items and offset. The items array stores the concatenated hyperedge-lists of all buckets contiguously. The offset array stores each bucket's hyperedge-list's starting position in items. The items and offset arrays both have $O(n)$ space complexity. These two arrays are not part of the PARFKSLEAN's final data structure; they are created for efficient parallelization in the next stage.

PARFKSLEAN builds offset in a parallel histogram computation using the bucket_ids array followed by a prefix sum. In the parallel histogram computation, the hyperedges are distributed uniformly among the threads. A thread visits its hyperedges, and for a hyperedge $\mathbf{e}$, it atomically increments offset[bucket_ids[$\mathbf{e}$]] (we use *atomic add*). The histogram computation requires performing $O(n)$ atomic operations. Then the prefix-sum on offset is computed using a well-known, two-pass algorithm. In this algorithm, first a parallel segmented reduction is carried out on segments of size $O(\frac{n}{T})$ by T threads. Then, a prefix-sum of the partial segment-wise sums is computed; this is only $O(T)$ work and is done sequentially. At last, a parallel segmented scan of the offset array is carried out, taking the prefix-sum of segment-wise sums into account, to finalize the computation.

Once offset is computed, PARFKSLEAN then constructs the items array in parallel. We assign $\lceil \frac{n}{T} \rceil$ hyperedges to each thread. A thread processes its hyperedges one by one. For each processed hyperedge $\mathbf{e}$, a thread gets an offset value by atomically decrementing offset[bucket_ids[$\mathbf{e}$]], and writes the id of $\mathbf{e}$ at the corresponding place in items. This requires $O(n)$ atomic operations.

3) Setting up the storage space: The third super-step in this stage is to prepare the fksOffset array shown in Figure 1 for the sake of efficiency in the second level hashing. By observing that

$$b_i = \mathsf{offset}[i+1] - \mathsf{offset}[i] \ ,$$

this array is initialized in parallel by setting

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

A static schedule assigning $\lceil \frac{n}{T} \rceil$ buckets to each thread will effect good load balancing during this initialization. Then, a parallel prefix-sum operation is carried out on fksOffset, as discussed for offset before. At the end, the total size of fksStorage is available at fksOffset[n], and can be checked to see if the bounds on memory are satisfied.

B. Constructing *fksStorage*

At this stage, PARFKSLEAN first allocates fksStorage. Then, the pool of keys, $\mathbf{K}$ in Figure 1, is populated. We put $2\log_2 n$ keys in $\mathbf{K}$, which is small and does not require parallelization. These many keys are sufficient to supply each bucket with a suitable hash function. In the *remote event* that these keys prove inadequate, additional keys may be generated and added to $\mathbf{K}$. Then PARFKSLEAN follows a coarse-grained parallelization based on assigning buckets to threads to find a perfect hashing for each bucket. Thanks to the work done in the previous stage, the arrays offset, items, and fksOffset are available, and the whole computation is embarrassingly parallel.

If a bucket is empty, there is nothing to be done. If a bucket has only a single hyperedge, the id of that hyperedge is stored in fksStorage. For a bucket B_i having more than one hyperedge, the d-tuples from $\mathbf{K}$ are tested one by one to find a perfect hashing. For this purpose, each hyperedge $\mathbf{e}$ in the hyperedge-list of B_i is placed at the location $h(\mathbf{k}_i, \mathbf{e}, p, 2b_i^2) + \mathsf{fksOffset}[i]$ for a tested $\mathbf{k}_i$, until we process all $\mathbf{e}$ (in which case $\mathbf{k}_i$ defines a perfect hashing), or a collision occurs (in which case another d-tuple is tried). Work per bucket is non-uniform. In order to avoid potential workload

imbalance, PARFKSLEAN makes use of a dynamic scheduler. The loop over buckets is parallelized among T threads using OpenMP's *parallel for* construct using the *dynamic* schedule with a chunk size of $\left\lceil \frac{n}{2\times T} \right\rceil$. As the number of buckets is much more than the number of threads, such a parallelization scheme is expected to achieve good load balancing.

IV. EXPERIMENTS

A. Setup

All experiments are carried out on an Intel Xeon Gold 5218 CPU with 64 cores (two sockets, 32 cores each), 2.3 GHz clock speed, 22 MB L3-cache, and 384 GB DDR4-2667 memory, which runs Debian GNU/Linux 10 (64 bit). All codes are in C++ and are compiled with GCC 8.3.0, with optimization flag `-O3` and `-march=native`. We use OpenMP for parallelization. For all $\bmod n$ and $\bmod p$ operations in PARFKSLEAN, we use fastmod library [11], and fast modulo operations with Mersenne primes [17], respectively; while $\bmod 2b_i^2$ operations are carried out with the standard modulo operator `%` in C++.

Input Tensors. We present experiments with ten sparse tensors derived from real-world applications for evaluating the performance of PARFKSLEAN. All the tensors are available in FROSTT [16], and are described in Table I, in the increasing order of number of nonzeros. We also use T-1 through T-10 to refer to these tensors in the given order.

Comparisons. We compare the proposed PARFKSLEAN with the state-of-the-art implementation of PTHash (version 1.0.1 available at https://github.com/jermp/pthash/). We run PTHash using three different parameter setups as recommended in the original source [15] denoted by PTHash-PC, PTHash-DD, and PTHash-EF. The construction phase of PTHash is parallel, but not the query phase. For parallelizing the query phase, we followed the same loop-parallelism both in PARFKSLEAN and PTHash, where each query is assigned to a thread; we used the *static schedule* for parallel *for-loops* from OpenMP. We thus principally investigate the run time and scalability of the construction phase of PARFKSLEAN and PTHash.

Tensor	d	**Dimensions**	n
chicago_crime (T-1)	4	$6{,}186\times24\times77\times32$	5,330,673
vast-2015-mc1-3d (T-2)	3	$165{,}427\times11{,}374\times2$	26,021,854
vast-2015-mc1-5d (T-3)	5	$165{,}427\times11{,}374\times2\times100\times89$	26,021,945
enron (T-4)	4	$6{,}066\times5{,}699\times244{,}268\times1{,}176$	54,202,099
nell-2 (T-5)	3	$12{,}092\times9{,}184\times28{,}818$	76,879,419
flickr-3d (T-6)	3	$319{,}686\times28{,}153{,}045\times1{,}607{,}191$	112,890,310
flickr-4d (T-7)	4	$319{,}686\times28{,}153{,}045\times1{,}607{,}191\times731$	112,890,310
delicious-3d (T-8)	3	$532{,}924\times17{,}262{,}471\times2{,}480{,}308$	140,126,181
delicious-4d (T-9)	4	$532{,}924\times17{,}262{,}471\times2{,}480{,}308\times1{,}443$	140,126,181
nell-1 (T-10)	3	$2{,}902{,}330\times2{,}143{,}368\times25{,}495{,}389$	143,599,552

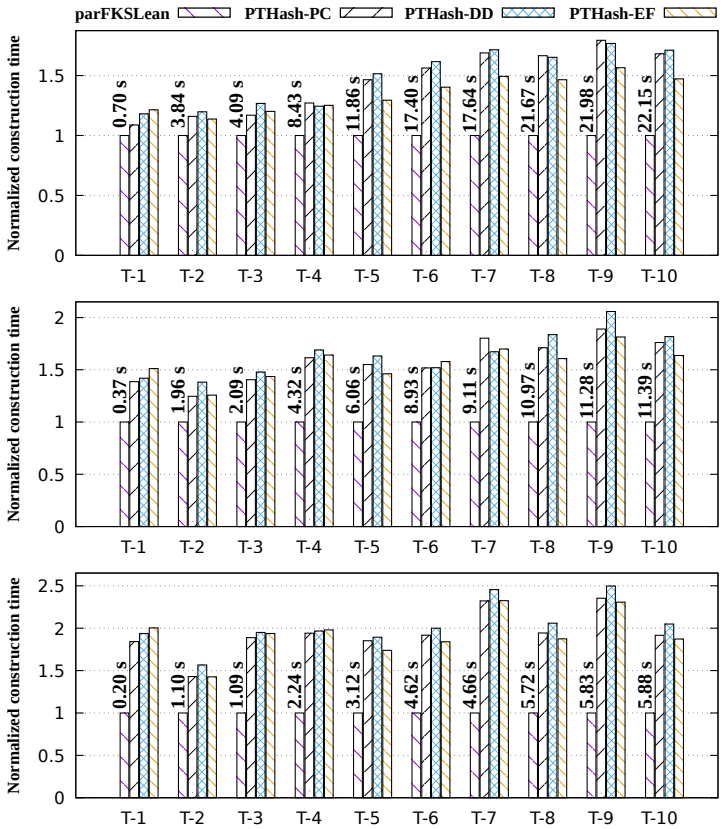
TABLE I: Input tensors.

We also briefly investigate the query response time of both tools for the sake of completeness. We report the execution times of the construction- and the query-phase of PARFKSLEAN and PTHash averaged over 10 independent runs.

PTHash implementation works on 64-bit items and the interface suggests converting other type of inputs to 64-bit keys with a technique called universe reduction [17]. We do that conversion offline and do not include the timings of the conversion in the construction phase of the PTHash variants.

B. Analysis of the construction phase

Figure 2 presents a comparison of the run time of PARFKSLEAN and PTHash in the construction phase, for six different thread configurations $\{2, 4, 8, 16, 32, 64\}$. In each of the figures, the construction time is normalized with respect to the construction time of PARFKSLEAN for every tensor. Thus, for every input, the bar corresponding to PARFKSLEAN is of unit height. The run time of the PARFKSLEAN is mentioned, in seconds, over the bar corresponding to PARFKSLEAN for



(a) # threads = 2

(b) # threads = 4

(c) # threads = 8

Fig. 2: Construction time of PARFKSLEAN and PTHash with 2, 4, and 8 threads. Y-axis is the construction time normalized with respect to PARFKSLEAN. The absolute construction time (in seconds) of PARFKSLEAN is given on top of its bar for every tensor.

each tensor. For the bars corresponding to the three PTHash variants, if the height of the bar is greater than one unit, then the higher the bar, slower it is in comparison to PARFKSLEAN.

As seen in Figures 2a–2f, the construction phase of PARFKSLEAN is always faster than all three variants of PTHash. As the number of threads increases, the relative performance of PARFKSLEAN with respect to PTHash improves, across all input tensors, as evidenced by the higher bars corresponding to PTHash variants for higher thread counts. PARFKSLEAN is up to 5.6 times faster than the best performing variant of PTHash in the construction phase; the former achieves the maximum speedup with respect to PTHash-PC on vast-2015-mc1-5d (T-3) when run using 64 threads (Figure 2f). In the light of this discussion, we conclude that the construction phase of PARFKSLEAN is **always faster** than that of the fastest PTHash variant.

Figure 3 presents the parallel scaling of the construction phase of PARFKSLEAN and PTHash variants on nell-2, flickr-4d, delicious-4d, nell-1. We took the four largest tensors from the data set with different number of nonzeros. In the figure, the speedup of PARFKSLEAN and PTHash variants is with respect to their respective run time with two threads. This comparison reveals the efficacy of the parallelization of PARFKSLEAN. We observe that PARFKSLEAN scales better than PTHash for all

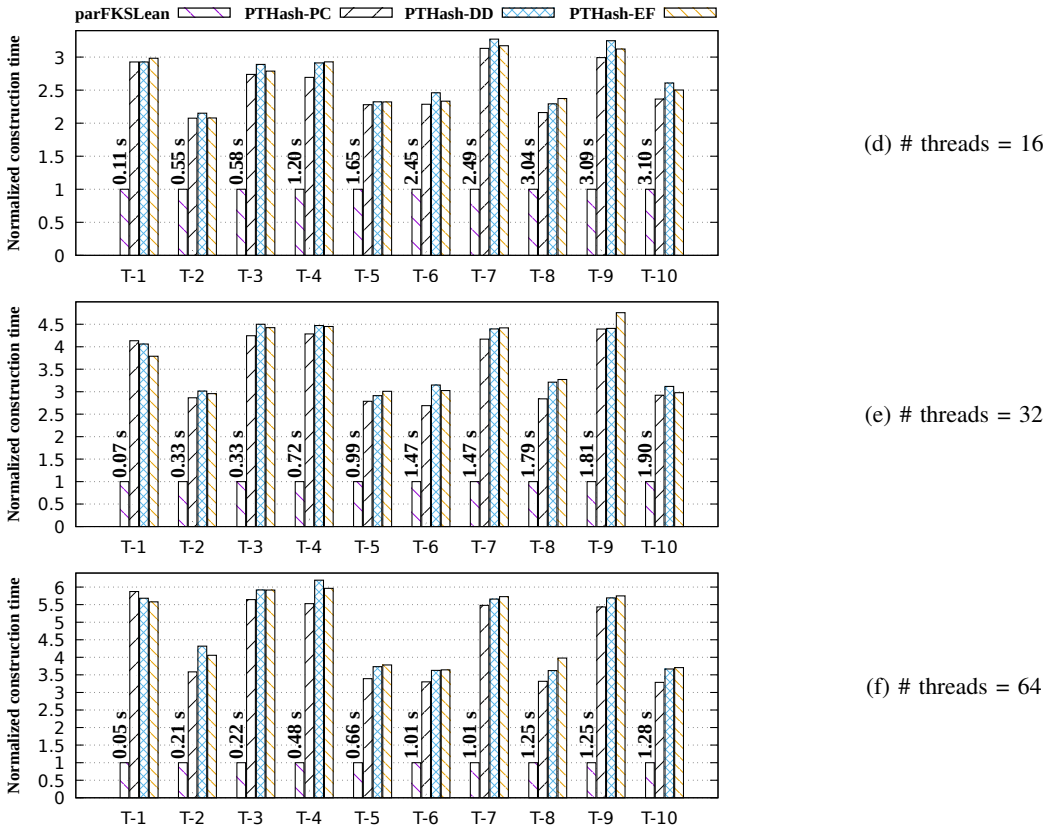
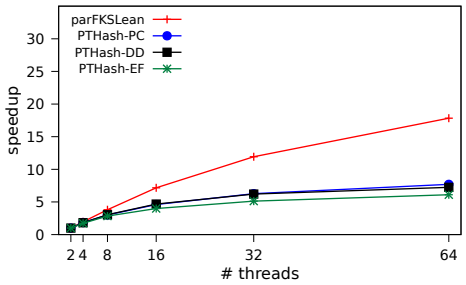


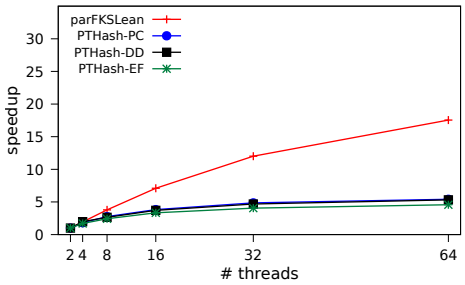
Fig. 2: *continued*: Construction time of PARFKSLEAN and PTHash with 16, 32, and 64 threads. Y-axis is the construction time normalized with respect to PARFKSLEAN. The absolute construction time (in seconds) of PARFKSLEAN is given on top of its bar for every tensor.

the four inputs. For PARFKSLEAN, the speedup consistently increases with the increasing number of threads. Its speedup at 64 threads over 2 threads is higher than 17, for all the four tensors. The speedup curves of PARFKSLEAN are similar across different tensors, which suggests that PARFKSLEAN is robust and not too sensitive to the input. On the other hand, the speedup of PTHash variants nearly plateaus after 32 threads, for all the inputs.

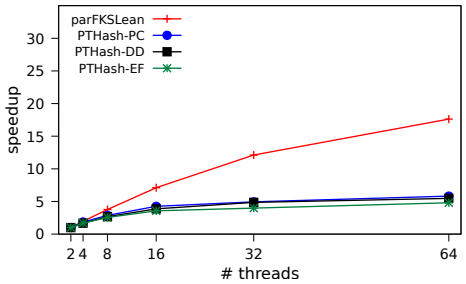
Scalability study. We now comment on the scalability of PARFKSLEAN with respect to an optimized sequential implementation, FKSLean. Table II presents the construction time of FKSLean and PARFKSLEAN with one thread for the four large tensors mentioned before. The construction time of PARFKSLEAN with 2 threads is 1.9 times faster than that with 1 thread—seen by comparing Table II and the absolute run time given in Figure 2a. In the table, we observe that FKSLean is, on average, 2.66 times faster than PARFKSLEAN executed using 1 thread. Therefore, FKSLean is faster than PARFKSLEAN executed using 2 threads in the construction phase, across all inputs; the former is 1.37 times faster than the latter on average. This is because PARFKSLEAN does more work than FKSLean, in the interest of parallelization, as discussed in Section III. Furthermore, there are parallelization-related overheads. The benefits of parallelization outweigh the extra work done and other overheads with 4 threads and more in



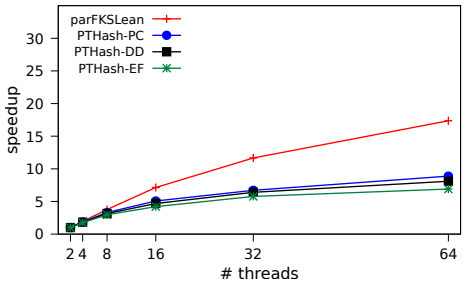
(a) nell-2



(b) flickr-4d



(c) delicious-4d



(d) nell-1

Fig. 3: Scalability of the construction phase of PARFKSLEAN and PTHash on four large tensors.

Tensor	FKSLean	PARFKSLEAN with #threads = 1
nell-2 (T-5)	8.78	23.48
flickr-4d (T-7)	12.93	34.92
delicious-4d (T-9)	16.12	42.86
nell-1 (T-10)	16.37	43.23

TABLE II: Construction time (in seconds) of FKSLean, and PARFKSLEAN with 1 thread.

PARFKSLEAN, and PARFKSLEAN always gets speedups with respect to FKSLean in this setting. The scalability in all thread settings (4 and more) is observable by comparing the run times in Figure 2 to that of FKSLean in Table II. In particular at 64 threads, PARFKSLEAN obtains a mean speedup of 12.94 with respect to FKSLean for the tensors T-5, T-7, T-9 and T-10.

Combining the observations made for Figure 2, Figure 3, and Table II, we conclude that the construction phase of PARFKSLEAN scales well; and that it obtains speedups with respect to FKSLean with four and more threads. Furthermore its parallel scaling is better than PTHash's.

C. Analysis of the query response time

We assume that the queries are available at the start of the query phase. The query processing is parallelized using a coarse-grained strategy. A query is assigned to a thread. All the queries are processed independently, in parallel. There is no parallelism in processing a single query.

Table III compares the run time of the query phase of PARFKSLEAN and PTHash variants to answer 10^7 queries for the four large tensors mentioned before. As seen in this table, PARFKSLEAN is at least as fast as the best performing variant of PTHash in all thread configurations and for all the inputs, except nell-1 with 64 threads, for which both the tools have the same response time. PARFKSLEAN achieves a maximum speedup of 1.94 with respect to PTHash-DD on flickr-4d with 4 threads. Thus, PARFKSLEAN is generally faster than the fastest PTHash variant for answering the queries. As in both the cases each query is answered by a single thread, the difference is due to the original performance of the respective algorithms.

Tensor	#Threads	PTHash -PC	PTHash -DD	PTHash -EF	PARFKSLEAN
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
	4	1.40	1.07	1.19	0.55
	8	0.78	0.61	0.62	0.28
	16	0.40	0.34	0.32	0.15
	32	0.21	0.16	0.17	0.11
	64	0.11	0.09	0.09	0.08
delicious-4d	2	2.30	2.02	2.25	1.11
	4	1.24	1.12	1.18	0.57
	8	0.66	0.59	0.61	0.30
	16	0.36	0.32	0.31	0.16
	32	0.19	0.17	0.16	0.11
	64	0.10	0.09	0.15	0.08
nell-1	2	2.43	2.11	2.11	1.06
	4	1.25	1.11	1.14	0.55
	8	0.64	0.60	0.58	0.30
	16	0.33	0.31	0.30	0.17
	32	0.18	0.16	0.17	0.11
	64	0.11	0.10	0.08	0.08

TABLE III: Execution time (in seconds) of the query phase of PARFKSLEAN and PTHash for 10^7 queries on four large tensors.

V. Conclusion

This paper proposes an efficient shared-memory parallel implementation of a recently developed perfect hashing method. This method is used for efficiently querying the existence of hyperedges in a given hypergraph. Put differently, the method is designed for querying the nonzeros of a sparse matrix or a tensor. Experiments are carried out on large sparse tensors and better run time and scalability are demonstrated with respect to another state-of-the-art perfect hashing method.

The focus of the paper is on d-uniform, d-partite hypergraphs. Luckily, the method easily extends to arbitrary hypergraphs. Care must be taken in order not to waste too much space in storing the second level hash functions. Another generalization concerns dynamic settings, where the hyperedges may be inserted or deleted; this generalization can find applications in diverse settings. The foregoing two points form the future work.

References

[1] A. Azad, A. Buluç, and A. Pothen, "Computing maximum cardinality matchings in parallel on bipartite graphs via tree-grafting," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 44–59, 2017.

[2] J. Bertrand, F. Dufossé, and B. Uçar, "Algorithms and data structures for hyperedge queries," Inria Grenoble Rhône-Alpes, Research Report RR-9390, Feb. 2021.

[3] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commununication of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[4] P. C. Dillinger, L. Hübschle-Schneider, P. Sanders, and S. Walzer, "Fast succinct retrieval and approximate membership using ribbon," *arxiv:2109.01892*, 2021.

[5] F. Dufossé, K. Kaya, and B. Uçar, "Two approximation algorithms for bipartite matching on multicore architectures," *Journal of Parallel and Distributed Computing*, vol. 85, pp. 62–78, 2015.

[6] E. Esposito, T. Mueller Graf, and S. Vigna, "RecSplit: Minimal perfect hashing via recursive splitting," in *Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2020, pp. 175–185.

[7] M. L. Fredman, J. Komlós, and E. Szemerédi, "Storing a sparse table with $O(1)$ worst case access time," *J. ACM*, vol. 31, no. 3, pp. 538–544, 1984.

[8] T. M. Graf and D. Lemire, "Binary fuse filters: Fast and smaller than xor filters," *arXiv:2201.01174*, 2022.

[9] G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*, 6th ed. Oxford Univ. Press, 2008.

[10] 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.

[11] D. Lemire, O. Kaser, and N. Kurz, "Faster remainder by direct computation: Applications to compilers and software libraries," *Softw. Pract. Exp.*, no. 6, pp. 953–970, 2019.

[12] A. Limasset, G. Rizk, R. Chikhi, and P. Peterlongo, "Fast and Scalable Minimal Perfect Hashing for Massive Key Sets," in *16th International Symposium on Experimental Algorithms (SEA)*, 2017, pp. 25:1–25:16.

[13] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *Algorithms — ESA 2001*, F. M. auf der Heide, Ed. Springer Berlin Heidelberg, 2001, pp. 121–133.

[14] 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.

[15] G. E. Pibiri and R. Trani, "Parallel and external-memory construction of minimal perfect hash functions with PTHash," *arxiv:2106.02350*, 2021.

[16] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis, "FROSTT: The formidable repository of open sparse tensors and tools," http://frostt.io/, 2017.

[17] M. Thorup, "High speed hashing for integers and strings," *arxiv:1504.06804*, 2015.

[18] S. Walzer, "Peeling close to the orientability threshold–spatial coupling in hashing-based data structures," in *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2021, pp. 2194–2211.